Ein Buch zum Mitmachen und Verstehen

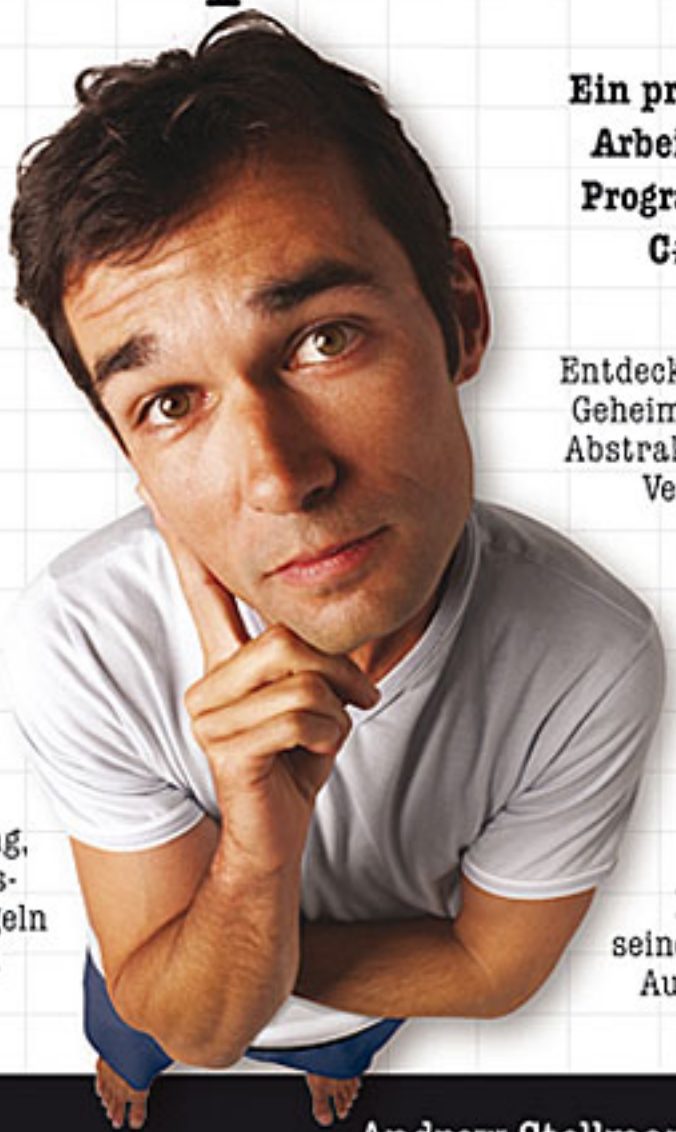
2. Auflage
Behandelt C# 4.0 und Visual Studio 2010

C#

von Kopf bis Fuß



Zeigen Sie Ihren Daten mit LINQ, wer der Boss ist



Ein praxisorientiertes Arbeitsbuch für die Programmierung mit C# und .NET

Entdecken Sie die Geheimnisse von Abstraktion und Vererbung



Reanimieren Sie einen alten Arcade-Klassiker





Erfahren Sie, wie es Susi gelang, mit Erweiterungsmethoden die Regeln von Objekthausen zu beugen



Erleben Sie, wie Tim seine Daten mit generischen Auflistungen in den Griff bekommt

O'REILLY®

Andrew Stellman & Jennifer Greene
Deutsche Übersetzung von Lars Schulten

Über den Übersetzer dieses Buchs

Lars Schulten ist freier Übersetzer für IT-Fachliteratur und hat für den O'Reilly Verlag schon unzählige Bücher zu ungefähr allem übersetzt, was man mit Computern so anstellen kann. Eigentlich hat er mal Philosophie studiert, aber mit Computern schlägt er sich schon seit den Zeiten herum, da Windows laufen lernte. Die Liste der Dinge, mit denen er sich beschäftigt, ist ungefähr so lang, launenhaft und heterogen wie die seiner Lieblingsessen und Lieblingsbücher.

Allein tritt er eigentlich nur auf, wenn er mal wieder versucht, den körperlichen Verfall mit sportlicher Betätigung aufzuhalten. Sonst ist er immer in Begleitung eines Buchs, seines Laptops oder Frederics unterwegs. Frederic ist sechs Jahre alt und setzt gern eine sehr kritische Miene auf, wenn Papa die Spielerei mit dem Computer als Arbeit bezeichnet.

Weitere Bücher von O'Reilly zu C#

Programmieren mit C# 3.0

C# 3.0 – kurz & gut

LINQ – kurz & gut

C# 3.0 Entwurfsmuster

C# 3.0 in a Nutshell

C# 3.0 Cookbook™

Weitere Bücher in O'Reillys *Von Kopf bis Fuß*-Reihe

Ajax von Kopf bis Fuß

Entwurfsmuster von Kopf bis Fuß

HTML mit CSS und XHTML von Kopf bis Fuß

Java™ von Kopf bis Fuß

JavaScript von Kopf bis Fuß

Objektorientierte Analyse und Design von Kopf bis Fuß

Softwareentwicklung von Kopf bis Fuß

SQL von Kopf bis Fuß

Head First EJB™

Head First PMP

Head First Servlets and JSP

Der Inhalt (im Überblick)

	Einführung	xxix
1	Produktiv mit C#: *Grafische Anwendungen in 10 Minuten und weniger*	1
2	Es ist alles bloß Code: *Unter der Motorhaube*	43
3	Objekte: Orientieren Sie sich!: *Aufgabenspezifischer Code*	85
4	Typen und Referenzen: *Es ist 10:00. Wissen Sie, wo Ihre Daten sind?*	123
	C#-Workshop 1: *Ein Tag beim Rennen*	163
5	Kapselung: *Alles zu zeigen ist nicht immer richtig*	173
6	Vererbung: *Der Stammbaum Ihres Objekts*	205
7	Schnittstellen und abstrakte Klassen: *Klassen, die Versprechen halten*	251
8	Enums und Auflistungen: *Daten in Massen speichern*	309
	C#-Workshop 2: *Die Suche*	363
9	Dateien lesen und schreiben: *Speichere das Array, rette die Welt*	385
10	Exception-Handling: *Fehler-Prävention*	439
11	Events und Delegates: *Was Ihr Code macht, wenn Sie nicht gucken*	483
12	Rückblick und Vorschau: *Wissen, Macht und cooles Zeug erstellen*	515
13	Steuerelemente und Grafik: *Machen Sie es ansehnlich*	563
14	Captain Amazing: *Der Tod des Objekts*	621
15	LINQ: *Bekommen Sie Ihre Daten in den Griff*	653
	C#-Workshop 3: *Invaders*	681
A	Anhang A: *Die Top-5 der Themen, die es nicht ins Buch geschafft haben*	703

Der Inhalt (jetzt ausführlich)

Einführung

Ihr Gehirn und C#. *Sie* versuchen, etwas zu *lernen*, und Ihr *Hirn* tut sein Bestes, damit das Gelernte nicht *hängen bleibt*. Es denkt nämlich: »Wir sollten lieber ordentlich Platz für wichtigere Dinge lassen, z.B. für das Wissen darüber, welche Tiere einem gefährlich werden könnten oder dass es eine ganz schlechte Idee ist, nackt Snowboard zu fahren.« Tja, *wie* schaffen wir es nun, Ihr Gehirn davon zu überzeugen, dass Ihr Leben davon abhängt, etwas über C# zu wissen?

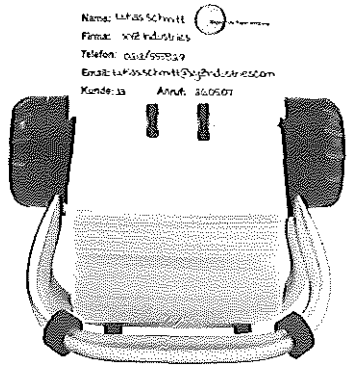
Für wen ist dieses Buch?	xxviii
Wir wissen, was Ihr Gehirn denkt	xxix
Metakognition	xxxi
Machen Sie sich Ihr Hirn untertan	xxxiii
Was Sie für dieses Buch brauchen	xxxiv
Lies mich	xxxv
Die Fachgutachter	xxxvi
Danksagungen	xxxvii

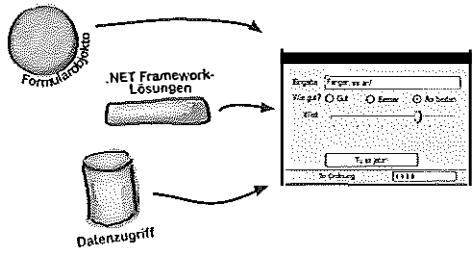
Produktiv mit C#

1 Grafische Anwendungen in 10 Minuten und weniger

Sie möchten Programme so richtig schnell erstellen können?

Mit C# halten Sie eine **mächtige Programmiersprache** und ein **wertvolles Werkzeug** in Ihren Händen. Dank der **Visual Studio IDE** müssen Sie nie wieder Stunden damit vergeuden, obskuren Code zu schreiben, damit ein Button schließlich tut, was er soll. Noch besser ist, dass Sie sich endlich nur noch darauf konzentrieren müssen, **Ihre Arbeit zu erledigen**, statt sich zu merken, welcher der Methodenparameter den *Namen* eines Buttons angibt und welcher sein *Label*. Klingt das vielversprechend? Blättern Sie um, damit wir mit dem Programmieren beginnen können.





Warum Sie C# lernen sollten	2
C# und Visual Studio IDE erleichtern Ihnen eine Menge Dinge	3
Verhelfen Sie Ihrem Chef zu einem papierlosen Büro	4
Bringen Sie die Bedürfnisse Ihres Kunden in Erfahrung, bevor Sie beginnen, Ihr Programm zu erstellen	5
Das werden Sie aufbauen	6
Was Sie in Visual Studio tun …	8
Was Visual Studio für Sie macht …	8
Die Benutzerschnittstelle entwickeln	12
Visual Studio hinter den Kulissen	14
Den automatisch generierten Code ergänzen	15
Sie können Ihre Anwendung bereits ausführen	16
Wir brauchen eine Datenbank zur Speicherung der Informationen	18
Die Tabelle für die Kontaktliste erstellen	20
Die Felder auf einer Adresskarte sind die Spalten in der Tabelle Personen	22
Den Tabellenbau abschließen	25
Erstellen Sie ein Diagramm für Ihre Daten, damit Ihre Anwendung darauf zugreifen kann	26
Ihre Kartendaten in die Datenbank einfügen	28
Formular und Datenobjekte über eine Datenquelle verbinden	30
Dem Formular datenbankgesteuerte Steuerelemente hinzufügen	32
Gute Programme sind intuitiv zu verwenden	34
IHRE Anwendung in eine Anwendung für ALLE verwandeln	37
Ihren Nutzern die Anwendung geben	38
Sie sind noch NICHT fertig: Testen Sie Ihre Anwendung	39
Sie haben eine vollständige, datengesteuerte Anwendung erstellt	40

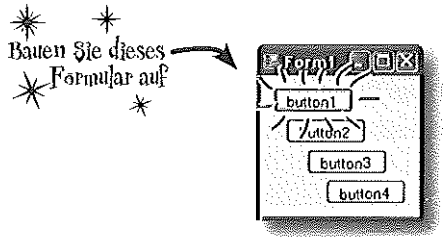
Es ist alles bloß Code

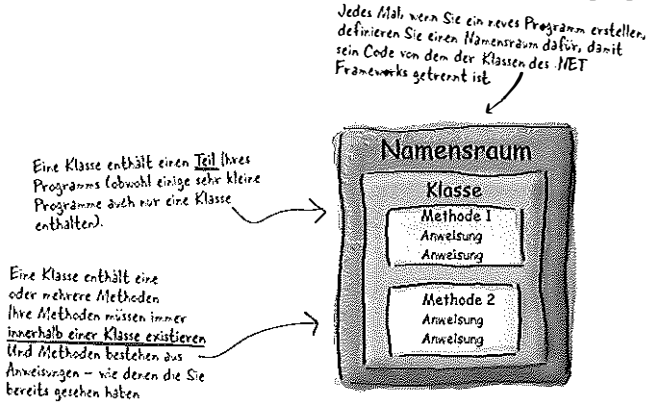
2 Unter der Motorhaube

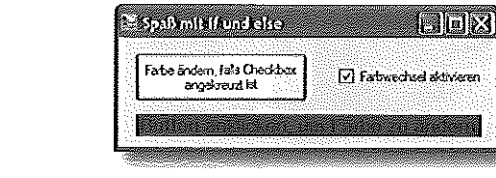
Sie sind Programmierer, nicht bloß IDE-Benutzer.

Mit der IDE können Sie eine Menge Dinge erledigen. Aber sie kann Sie nur ein bestimmtes Stück Ihres Wegs begleiten. Sicher gibt es eine Menge **sich wiederholender Aufgaben**, die Sie erledigen müssen, wenn Sie eine Anwendung aufbauen. Diese Aufgaben kann Ihnen die IDE wunderbar abnehmen. Aber die Arbeit mit der IDE ist *erst der Anfang.* Sie können Ihre Programme dazu bringen, so viel mehr zu machen – und das geht nur, indem Sie **C#-Code schreiben**. Haben Sie das Programmieren einmal im Griff, gibt es *nichts mehr*, was Ihre Programme nicht tun könnten.

Wenn Sie das tun …	44
… macht die IDE das	45
Wo die Programme herkommen	46
Die IDE hilft Ihnen beim Programmieren	48
Wenn Sie Dinge in der IDE ändern, ändern Sie auch Ihren Code	50
Anatomie eines Programms	52
Ihr Programm weiß, wo es anfangen muss	54
Sie können den Einstiegspunkt Ihres Programms ändern	56
Zwei Klassen können im gleichen Namensraum sein	61
Ihre Programme nutzen Variablen, um mit Daten zu arbeiten	62
C# nutzt vertraute mathematische Symbole	64
Schleifen führen eine Aktion immer wieder aus	65
Zeit, mit dem Programmieren zu beginnen	66
if/else-Anweisungen fällen Entscheidungen	67
Bedingungen aufstellen und prüfen	68





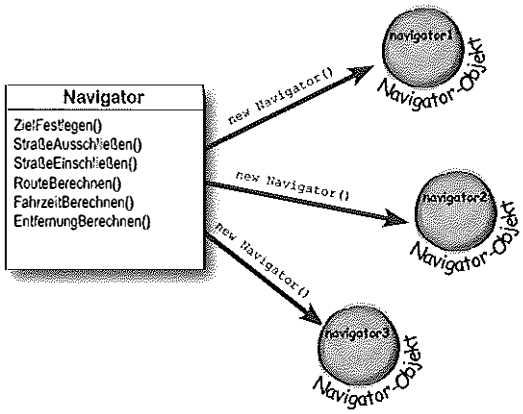


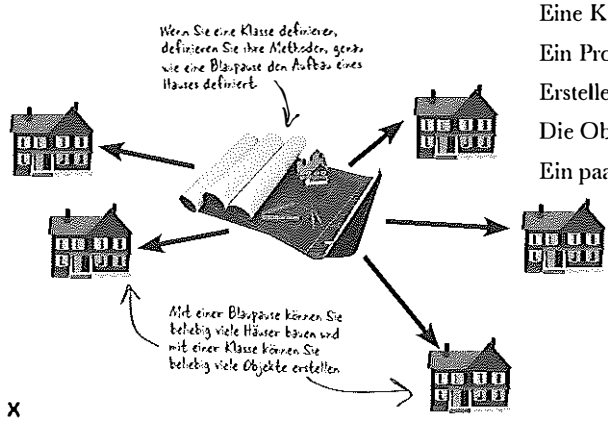
Objekte: Orientieren Sie sich!

3 Aufgabenspezifischer Code

Jedes Programm, das Sie schreiben, löst ein Problem.

Bevor Sie ein Programm erstellen, überlegen Sie sich am besten, welches *Problem* Ihr Programm lösen soll. Deswegen sind ***Objekte*** so ungemein hilfreich. Mit diesen können Sie Ihren Code auf Basis des Problems strukturieren, das er lösen soll. Und dann können Sie Ihre Zeit damit verbringen, *über das Problem nachzudenken*, an dem Sie arbeiten müssen, anstatt gleich beim Schreiben des Codes im Sumpf der Implementierungsanforderungen zu versinken. Wenn Sie Objekte richtig einsetzen, erhalten Sie Code, der *intuitiv* zu schreiben sowie leicht zu lesen und zu ändern ist.





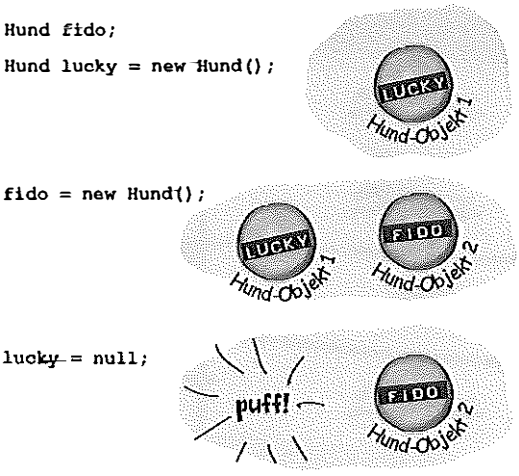
Wie Mark über sein Problem nachdenkt	86
Wie Marks Navi über das Problem nachdenkt	87
Die Methoden von Marks Navigator-Klasse	88
Nutzen Sie das Gelernte, um eine einfache Anwendung zu erstellen	89
Mark hat eine Idee	90
Mark kann sein Problem mit Objekten lösen	91
Sie nutzen eine Klasse, um ein Objekt zu erstellen	92
Wenn Sie ein neues Objekt einer Klasse erstellen, bezeichnet man dieses als Instanz der Klasse	93
Eine bessere Lösung … dank Objekten!	94
Eine Instanz nutzt Felder, um Dinge festzuhalten	98
Erstellen wir mal ein paar Instanzen!	99
Objekte auf dem Speicher	100
Was Ihr Programm im Kopf hat	101
Gute Klassen- und Methodennamen machen Code intuitiver	102
Geben Sie Ihren Klassen eine natürliche Struktur	104
Klassendiagramme helfen Ihnen, Ihre Klassen zu organisieren	106
Eine Klasse für die Arbeit mit ein paar Typen erstellen	110
Ein Projekt für die Jungs erstellen	111
Erstellen Sie ein Formular zur Interaktion mit den Jungs	112
Die Objektinitialisierung kann noch einfacher sein	115
Ein paar Ideen zum Entwurf intuitiver Klassen	116

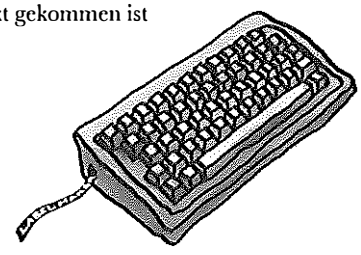
4 Typen und Referenzen

Es ist 10:00. Wissen Sie, wo Ihre Daten sind?

Datentyp, Datenbank, Lieutenant Commander Data ... all das ist wichtiges Zeug. Ohne Daten sind Ihre Programme nutzlos. Sie brauchen **Informationen** von Ihren Anwendern und nutzen diese, um andere Informationen nachzuschlagen oder neue Informationen zu produzieren, die Sie dann wieder an Ihre User zurückliefern. Eigentlich schließt fast alles, was Sie beim Programmieren tun, auf die eine oder andere Weise **Arbeit mit Daten** ein. In diesem Kapitel werden Sie alles über die **Datentypen** von C# lernen, erfahren, wie Sie in Ihrem Programm mit Daten arbeiten, und Sie werden sogar auf einige schmutzige Geheimnisse über **Objekte** stoßen (*psst ... auch Objekte sind Daten*).

Der Typ einer Variablen bestimmt, welche Art Daten sie speichert 124
Eine Variable ist wie ein Daten-To-Go-Becher 126
10 Kilo Daten in einem 5-Kilo-Sack 127
Auch eine Zahl der richtigen Größe können Sie nicht einfach irgendeiner Variablen zuweisen 128
Wandeln Sie einen zu großen Wert um, passt C# ihn automatisch an 129
Einige Umwandlungen macht C# automatisch 130
Beim Aufruf einer Methode müssen die Variablen den Typen der Parameter entsprechen 131
Einen Operator mit = kombinieren 136
Auch für Objekte braucht man Variablen 137
Mit Referenzvariablen auf Objekte verweisen 138
Referenzen sind wie Etiketten für Ihr Objekt 139
Gibt es keine Referenzen mehr, holt die Müllabfuhr Ihr Objekt 140
Mehrere Referenzen und ihre Nebenwirkungen 142
Zwei Referenzen bedeuten ZWEI Möglichkeiten, die Daten eines Objekts zu ändern 147
Ein Sonderfall: Arrays 148
Arrays können auch einen Haufen Referenzvariablen speichern 149
Willkommen beim Sandwich-Discount vom Strammen Max 150
Objekte nutzen Referenzen, um miteinander zu reden 152
Wohin noch kein Objekt gekommen ist 153



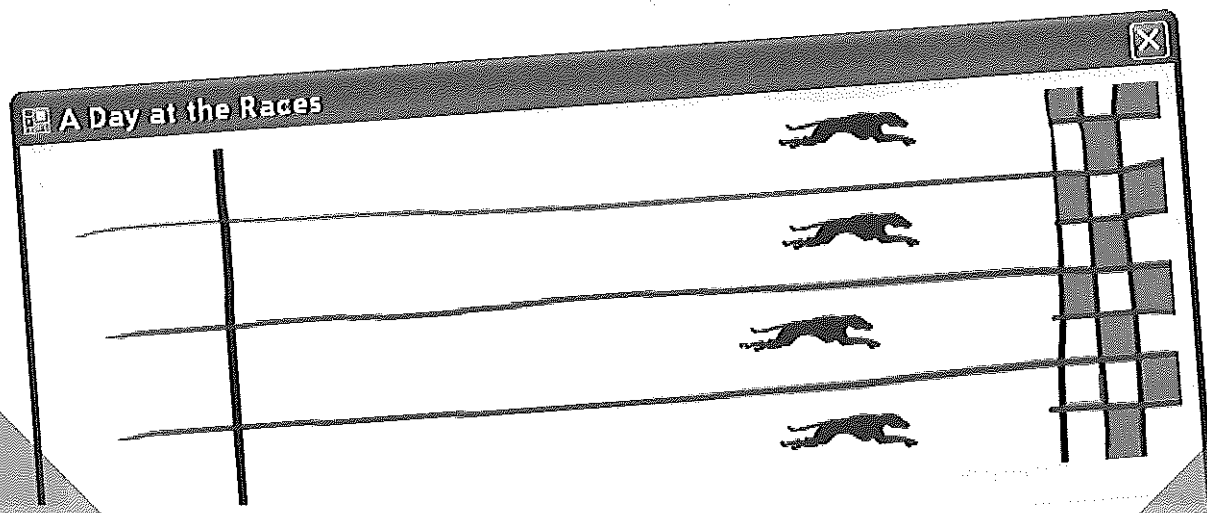


C#-Workshop 1

Ein Tag beim Rennen

Tim, Tom und Jan verbringen ihre Zeit mit Vorliebe auf der Rennbahn, sind es allerdings leid, dabei ständig ihr gesamtes Geld zu verlieren. Sie sollen den Dreien einen Simulator aufbauen, der ihnen hilft, den Sieger zu bestimmen, bevor sie ihr Geld setzen. Und wenn Sie dabei gute Arbeit leisten, erhalten Sie Ihren Anteil an ihren Profiten.

Die Spezifikation: Erstellen Sie eine Rennbahn-Simulation 164
Das fertige Produkt 172



Kapselung

5 Alles zu zeigen ist nicht immer richtig

Denken Sie auch, dass man manches nicht gleich allen zeigen muss?

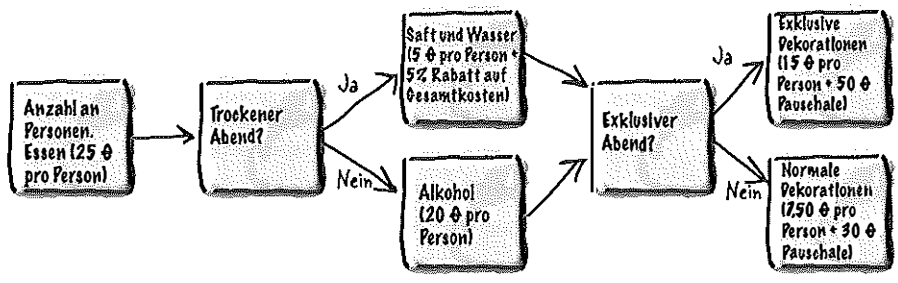
Manchmal geht es Ihren Objekten ebenfalls so. Genau so, wie Sie nicht jeden Ihr Tagebuch oder Ihre Kontoauszüge lesen lassen, lassen gute Objekte ***andere*** Objekte nicht in ihre Felder blicken. In diesem Kapitel werden Sie die Macht der **Kapselung** kennenlernen. Sie **werden die Daten Ihrer Objekte privat machen** und Methoden hinzufügen, **die schützen, wie auf diese Daten zugegriffen wird.**







Kathrin ist Event-Managerin	174
Was der Kostenrechner macht	175
Kathrins Testlauf	180
Jede Option sollte einzeln berechnet werden	182
Objekte werden leicht unbeabsichtigt missbraucht	184
Kapselung bedeutet, dass man Daten in einer Klasse privat hält	185
Steuern Sie den Zugriff auf Felder und Methoden mit Kapselung	186
Aber ist das Feld WahrerName WIRKLICH geschützt?	187
Auf private Felder und Methoden kann nur aus der Klasse selbst heraus zugegriffen werden	188
Ein paar Gedanken zur Kapselung von Klassen	191
Kapselung hält Ihre Daten rein	192
Eigenschaften vereinfachen die Kapselung	193
Eine Anwendung zum Testen der Klasse Bauer erstellen	194
Automatische Eigenschaften	195
Was ist, wenn wir den Futtermultiplikator ändern möchten?	196
Initialisieren Sie private Felder mit einem Konstruktor	197



Vererbung

6 Der Stammbaum Ihres Objekts

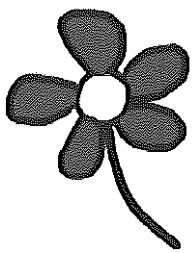
Es gibt Momente, da *WILL man wie die eigenen Eltern sein.*

Ist Ihnen schon einmal ein Objekt über den Weg gelaufen, das ***fast*** genau das macht, was ***Ihr*** Objekt machen soll? Und haben Sie sich gewünscht, Sie ***bräuchten einfach nur ein paar Dinge zu ändern*** und das Objekt wäre perfekt? Genau das ist der Grund dafür, dass **Vererbung** eins der mächtigsten und wichtigsten Konzepte in C# ist. Bevor Sie mit diesem Kapitel durch sind, werden Sie gelernt haben, wie Sie von einer Klasse **ableiten**, um ihr Verhalten zu bekommen, dabei aber die **Flexibilität** bewahren, Änderungen an diesem Verhalten vorzunehmen. Sie **vermeiden doppelten Code, modellieren die wahre Welt** präziser nach und erhalten Code, der **leichter zu warten ist.**

Kathrin plant auch Geburtstagsfeiern 206
Wir brauchen eine Geburtstagsfeier-Klasse 207
Eine Sache noch ... eine Gebühr von 100 € hinzufügen 213
Nutzen Ihre Klassen Vererbung, müssen Sie Ihren Code nur einmal schreiben 214
Erstellen Sie Ihr Klassenmodell, indem Sie allgemein beginnen und immer spezifischer werden 215
Wie würden Sie einen Zoo-Simulator entwerfen? 216
Vermeiden Sie mit Vererbung doppelten Code in Unterklassen 217
Unterschiedliche Tiere machen unterschiedliche Geräusche 218
Denken Sie daran, wie die Tiere gruppiert sind 219
Die Klassenhierarchie erstellen 220
Jede Unterklasse erweitert Ihre Basisklasse 221
Erweiterung geben Sie mit einem Doppelpunkt an 222
Wir wissen, dass Vererbung den Unterklassen die Felder, Eigenschaften und Methoden der Basisklasse hinzufügt 225
Eine Unterklasse kann Methoden überschreiben, um geerbte Methoden zu ändern oder zu ersetzen 226
Überall dort, wo Sie die Basisklasse verwenden können, können Sie stattdessen eine Unterklasse verwenden 227
Basisklassenzugriff über das Schlüsselwort base 232
Hat die Basisklasse einen Konstruktor, braucht ihn die Unterklasse auch 233
Jetzt können Sie die Arbeit für Kathrin abschließen! 234
Bauen Sie eine Bienenstockverwaltung 239
Erst bauen Sie das Grundsystem 240
Erweitern Sie die Bienenverwaltung mit Vererbung 245





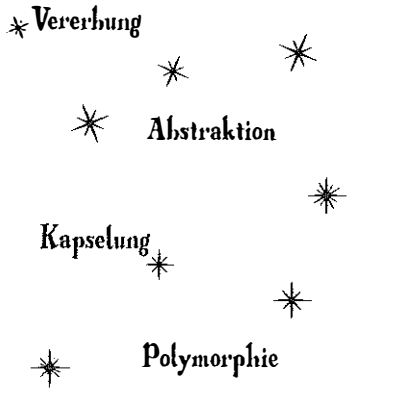


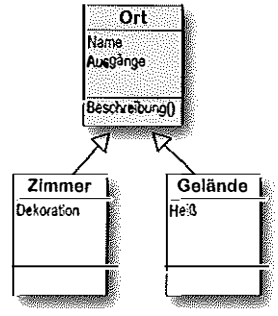
Schnittstellen und abstrakte Klassen

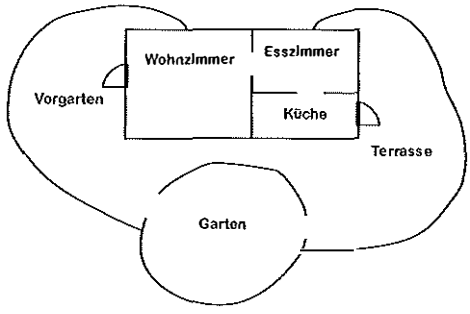
7 Klassen, die Versprechen halten

Taten sagen mehr als Worte.

Gelegentlich müssen Sie Ihre Objekte auf Basis dessen gruppieren, **was sie tun können, und** nicht auf Basis der Klassen, von denen sie erben. Dann kommen **Schnittstellen** ins Spiel – diese ermöglichen Ihnen, mit jeder Klasse zu arbeiten, die bestimmte Aufgaben erledigen kann. Aber mit **großer Macht geht eine große Verantwortung einher**: Jede Klasse, die eine Schnittstelle implementiert, muss versprechen, **all ihre Verpflichtungen zu erfüllen** ... oder der Compiler tritt Ihnen vors Schienbein, Verstanden()?







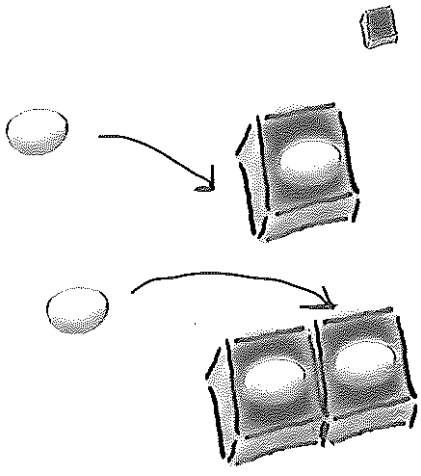
Kehren wir zu den Bienen zurück	252
Mit Vererbung Klassen für unterschiedliche Arten von Bienen erstellen	253
Eine Schnittstelle sagt einer Klasse, dass sie bestimmte Methoden und Eigenschaften implementieren muss	254
Schnittstellen definieren Sie mit dem Schlüsselwort interface	255
Jetzt die Klasse NektarWache erstellen, die beide Aufgaben erfüllt	256
Klassen, die Schnittstellen implementieren, müssen ALLE Methoden der Schnittstelle einschließen	257
Sammeln Sie Erfahrung im Umgang mit Schnittstellen	258
Eine Schnittstelle referenzieren	260
Schnittstellenreferenzen funktionieren wie Objektreferenzen	261
Mit »is« ermitteln, ob eine Klasse eine Schnittstelle implementiert	262
Eine Schnittstelle kann von einer anderen erben	263
Eine RoboBiene kann arbeiten und braucht keinen Honig	264
»is« sagt, welchen Typ Ihr Objekt hat, »as«, wie es behandelt werden soll	265
Eine KaffeeMaschine ist auch ein Gerät	266
Umwandeln funktioniert mit Objekten und Schnittstellen	267
Mit Downcasting ein Gerät wieder zu einer Kaffeemaschine machen	268
Upcasting und Downcasting funktionieren auch mit Schnittstellen	269
Es gibt nicht nur public und private	273
Zugriffsmodifizierer ändern Geltungsbereiche	274
Manche Klassen sollten nicht instantiiert werden	277
Eine Kreuzung zwischen einer normalen Klasse und einer Schnittstelle	278
Manche Klassen sollten nicht instantiiert werden	280
Eine abstrakte Methode hat keinen Rumpf	281
Der Deadly Diamond of Death!	286
Polymorphie: ein Objekt in vielen Formen	289

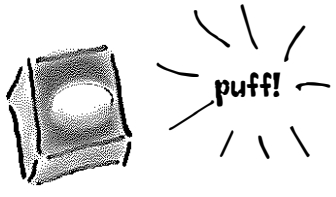
Enums und Auflistungen

8 Daten in Massen speichern

Eigentlich tritt alles immer in Massen auf.

Im wahren Leben wird man mit Daten nie in kleinen Fragmenten konfrontiert. Nein, sie begegnen Ihnen immer in **Massen, Bergen und Haufen**. Und Sie brauchen ziemlich leistungsfähige Werkzeuge, um sie zu organisieren. Genau das sind **Auflistungen**. Mit diesen können Sie alle Daten **speichern, sortieren und verwalten**, die Ihr Programm durchforsten muss. Sie können sich darauf konzentrieren, wie Ihr Programm mit den Daten arbeitet, und es den Auflistungen überlassen, sich für Sie um die Daten selbst zu kümmern.





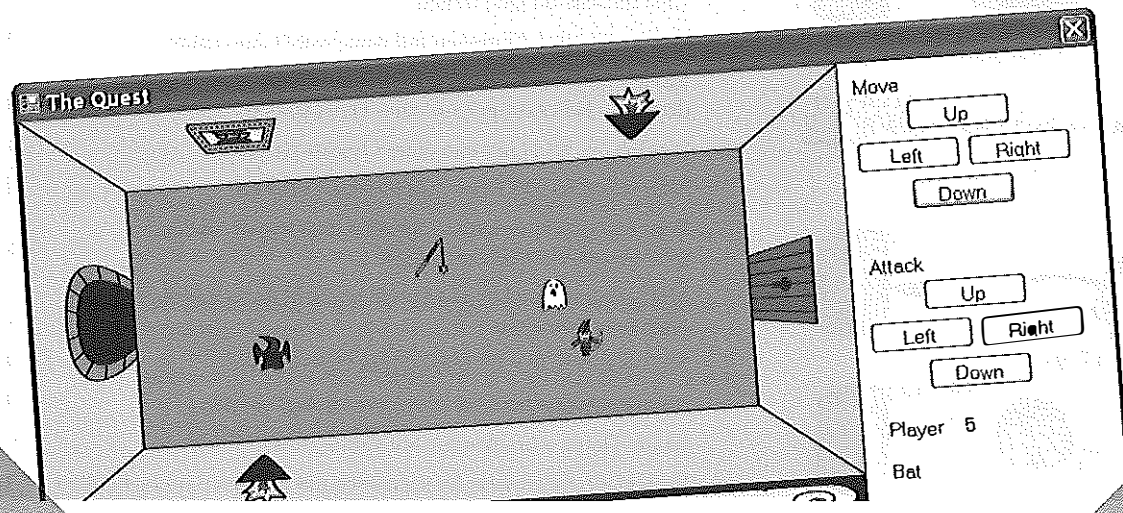
Gelegentlich sind Strings nicht gut genug, wenn es darum geht, Kategorien von Daten zu speichern 310
Mit Enumerationen können Sie erlaubte Werte aufzählen 311
Mit Enums können Zahlen über Namen repräsentiert werden 312
Wir könnten einen Kartenstapel über ein Array repräsentieren 315
Die Arbeit mit Arrays ist nicht immer einfach 316
Listen erleichtern das Speichern von Sammlungen ... jeder Art 317
Listen sind flexibler als Arrays 318
Listen schrumpfen und wachsen dynamisch 321
List-Objekte können beliebige Typen speichern 322
Auflistungsinitialisierer funktionieren wie Objektinitialisierer 326
Erstellen wir eine Liste mit Enten 327
Listen sind leicht, aber SORTIEREN kann verzwickt sein 328
Zwei Wege, Ihre Enten zu sortieren 329
Mit IComparer sagen Sie Ihren Listen, wie sie sortieren sollen 330
Erstellen Sie eine Instanz Ihrer Comparer-Klasse 331
IComparer kann komplexe Vergleiche durchführen 332
Schlüssel und Werte speichern Sie in Wörterbüchern 335
Die Dictionary-Funktionalität im Überblick 336
Schlüssel und Wert können auch unterschiedliche Typen haben 337
Auch Sie können Methoden überladen 343
Noch MEHR Auflistungstypen ... 355
Eine Queue ist FIFO – First-in-First-out 356
Ein Stack ist LIFO – Last-in-First-out 357

C#-Workshop 2

Die Suche

Sie haben die Aufgabe, ein Adventure zu erstellen, in dem ein mächtiger Abenteurer von Level zu Level die tödlichsten Feinde besiegt. Sie werden ein rundenbasiertes Spiel erstellen. Das bedeutet, dass erst der Spieler und dann die Gegner ihren Zug machen. Der Spieler kann sich bewegen oder angreifen, und dann erhält jeder Gegner die Möglichkeit, sich zu bewegen und anzugreifen. Das Spiel geht so lange, bis der Spieler auf allen sieben Leveln seine Gegner besiegt hat oder stirbt.

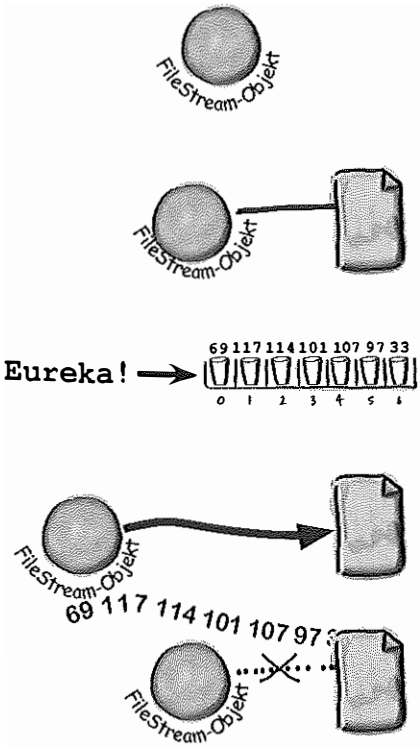
Die Spezifikation: Erstellen Sie ein Adventure -Spiel 364
Der Spaß fängt gerade erst an! 484



Dateien lesen und schreiben

9 Speichere das Array, rette die Welt

Gelegentlich zahlt es sich aus, wenn man Dinge festhält. Bisher hatten alle Ihre Programme nur ein Kurzzeitgedächtnis. Sie starten, laufen eine Weile und enden dann. Manchmal reicht das nicht aus, insbesondere wenn Sie mit wichtigen Daten arbeiten. Sie müssen **Ihre Arbeit speichern können.** In diesem Kapitel werden wir uns ansehen, wie man **Daten in einer Datei speichert** und diese **Informationen dann wieder aus einer Datei einliest.** Sie werden etwas über die **.NET-Stream-Klassen** lernen und sich mit den Mysterien **Hexadezimal** und **Binär** auseinandersetzen.



C# nutzt Streams, um Daten zu lesen und zu schreiben	386
Unterschiedliche Streams lesen und schreiben andere Dinge	387
Ein FileStream schreibt Bytes in eine Datei	388
Zum Lesen und Schreiben braucht man zwei Objekte	393
Daten können durch mehrere Streams laufen	394
Öffnen Sie mit eingebauten Objekten Standarddialoge	397
Dialoge sind auch Objekte	399
Mit den eingebauten Klassen File und Directory können Sie mit Dateien und Verzeichnissen arbeiten	400
Öffnen und speichern Sie Dateien mit Dateidialogen	403
IDisposable sorgt dafür, dass Ihre Objekte entsorgt werden	405
Vermeiden Sie Dateisystemfehler mit using-Anweisungen	406
Beim Schreiben von Dateien muss man in der Regel viele Entscheidungen treffen	412
Wählen Sie Ihre Optionen mit einer switch-Anweisung	413
Ein überladener Kartenstapel()-Konstruktor, der einen Kartenstapel aus einer Datei einliest	415
Was bei der Serialisierung eines Objekts passiert	417
Aber was genau IST der Zustand eines Objekts?	418
Wird ein Objekt serialisiert, werden die Objekte, die es referenziert, ebenfalls serialisiert ...	419
Mit Serialisierung können Sie ein ganzes Objekt auf einmal lesen oder schreiben	420
.NET wandelt Text automatisch in Unicode um	425
Mit byte-Arrays können Daten verschoben werden	426
Binäre Daten schreiben Sie mit einem BinaryWriter	427
Sie können serialisierte Dateien auch manuell lesen und schreiben	429
StreamReader und StreamWriter reichen vollkommen aus	433

Exception-Handling

10 Fehler-Prävention

Programmierer sind keine Feuerlöscher. Sie haben sich auf die Hinterbeine gesetzt, einen Haufen trockener Handbücher und ein paar ansprechende Von Kopf bis Fuß-Bücher durchgearbeitet und haben den Gipfelpunkt Ihres Berufsstands erreicht: **Meisterprogrammierer**. Aber Sie erhalten immer noch Beschwerden, weil **ihr Programm abstürzt** oder **sich nicht so verhält, wie es sich verhalten soll**. Nichts kann einem die Programmierstimmung so verhageln wie die Forderung, einen seltsamen Fehler zu beheben ... aber mit **Exception-Handling** kann sich Ihr Code **um Probleme kümmern**, die aufkommen. Und Sie können auf diese Probleme sogar reagieren und dafür sorgen, **dass das Programm weiterläuft**.



Brians Entschuldigungen müssen mobil werden	440
Löst Ihr Programm eine Exception aus, erzeugt .NET ein Exception-Objekt	444
Brians Code machte etwas Unerwartetes	446
Alle Exception-Objekte erben von Exception	448
Der Debugger hilft Ihnen, Exceptions in Ihrem Code aufzuspüren und zu verhindern	449
Nutzen Sie den Debugger, um herauszufinden, was genau das Problem in der Entschuldigungsverwaltung ist	450
Och nee, der Code hat immer noch Probleme	453
Exceptions mit try und catch behandeln	455
Was passiert, wenn eine Methode gefährlich ist?	456
Folgen Sie try/catch mit dem Debugger	458
Nutzen Sie finally, wenn Sie Code haben, der IMMER ausgeführt werden soll	460
Nutzen Sie das Exception-Objekt, um Informationen zum Problem zu erhalten	465
Mehrere Typen von Exceptions mit mehreren Catch-Blöcken abfangen	466
Eine Klasse löst eine Exception aus, eine andere fängt sie ab	467
Bienen brauchen eine HonigLeerException	468
using gibt Ihnen try und finally kostenlos	471
Implementieren Sie IDisposable für eigene Aufräumarbeiten	472
Der SCHLIMMSTE catch-Block: Kommentare	474
Vorübergehende Lösungen sind okay (vorübergehend)	475
Einige einfache Gedanken zum Exception-Handling	476
finally verschafft Brian seinen Urlaub ...	481

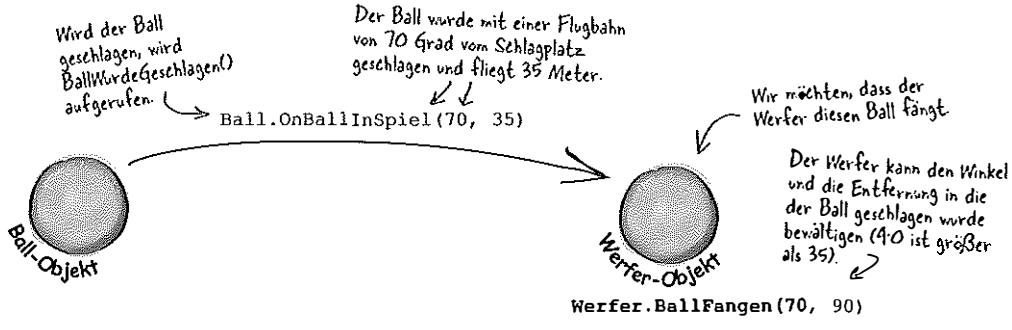
Events und Delegates

11 Was Ihr Code macht, wenn Sie nicht gucken

Ihre Objekte beginnen, für sich selbst zu denken.

Sie können nicht immer kontrollieren, was Ihre Objekte machen. Manche Dinge passieren einfach. Und wenn sie passieren, sollten Ihre Objekte schlau genug sein, **auf alles eine Antwort zu wissen**, was eintreten könnte. Darum geht es bei Events (oder Ereignissen). Ein Objekt *veröffentlicht* ein Event, andere Objekte *abonnieren* es, und alle arbeiten zusammen, damit die Dinge im Fluss bleiben. Das geht so lange gut, bis zu viele Objekte auf dasselbe Event reagieren. Dann werden sich **Callbacks** als praktisch erweisen.

Möchten auch Sie, dass Ihre Objekte für sich selbst denken?	484
Aber woher weiß ein Objekt, dass es reagieren soll?	484
Wenn Events eintreten ... lauschen Objekte	485
Ein Objekt setzt ein Event ab, andere lauschen darauf ...	486
Dann behandeln andere Objekte das Event	487
Die Punkte verbinden	488
Die IDE erzeugt Event-Handler automatisch	492
Alle Formulare, die Sie bisher erstellt haben, nutzen Events	498
Event-Sender mit Event-Empfängern verbinden	500
Ein Delegate VERTRITT die eigentliche Methode	501
Delegates in Aktion	502
Jedes Objekt kann ein öffentliches Event abonnieren ...	505
Nutzen Sie Callbacks statt Events, um genau ein Objekt mit einem Delegate zu verbinden	507
Callbacks nutzen Delegates, aber KEINE Events	50



Rückblick und Vorschau

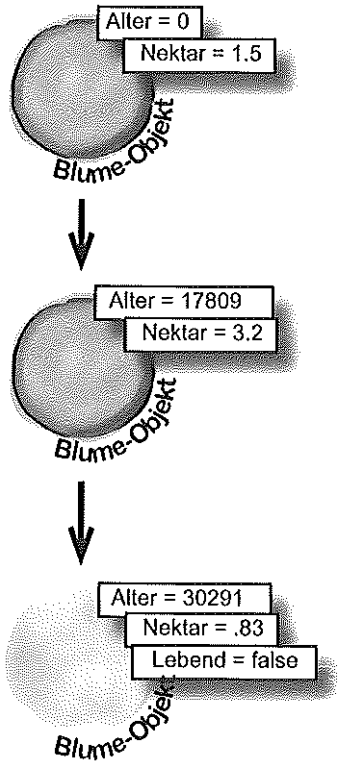
12 Wissen, Macht und cooles Zeug erstellen

Nur Lernen bringt nichts, man muss das Gelernte auch ANWENDEN.

Erst wenn Sie tatsächlich Code schreiben, wissen Sie, ob Sie die schwereren C#-Konzepte wirklich *verstanden* haben. In diesem Kapitel werden wir einige neue Feinheiten kennenlernen: **Timer** und die Verarbeitung von Collections mit **LINQ** (um nur ein paar zu nennen). Außerdem werden wir eine **richtig komplexe Anwendung** aufbauen und damit prüfen, ob Sie alles, was Sie in den vorangegangenen Kapiteln gelernt haben, auch wirklich beherrschen. Schnallen Sie sich an ... jetzt stürzen wir uns in den Aufbau **cooler Software.**

Du hast einen langen Weg hinter dir, Baby 516
Wir sind Imker geworden 517
Die Architektur des Bienenstock-Simulators 518
Den Bienenstock-Simulator erstellen 519
Leben und Sterben einer Blume 523
Nun brauchen wir die Klasse Biene 524
Geben wir den Bienen ein Zuhause 528
Die Klasse Stock ergänzen 532
Die Laufen()-Methode des Stocks 533
Wir sind bereit für die Welt 534
Wir bauen ein rundenbasiertes System 535
Den Bienen ein Verhalten geben 542
Das Hauptformular bringt die Welt ans Laufen() 544
Wir können die Welt nutzen, um Informationen zu erhalten 545
Timer lösen Events immer wieder aus 546
Im Hintergrund nutzt Timer ein Delegate 547
Geben Sie dem Simulator einen Timer 548
Arbeiten wir mit Gruppen von Bienen 554
Eine Auflistung sammelt ... DATEN 555
LINQ vereinfacht die Arbeit mit den Daten in Auflistungen und Datenbanken 557

Leben und Sterben einer Blume

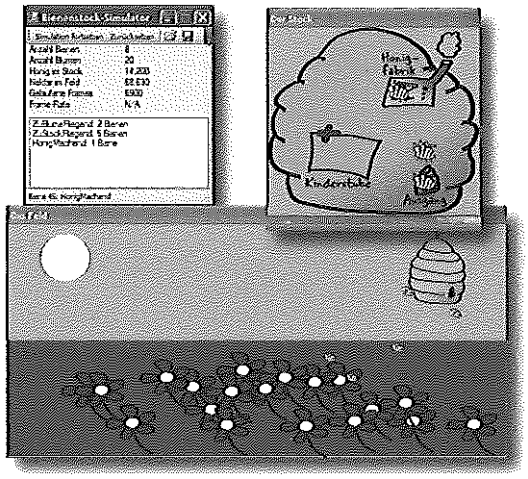


Steuerelemente und Grafik

13 Machen Sie es ansehnlich

Manchmal müssen Sie die Grafik in die eigenen Hände nehmen.

Lange haben wir uns darauf verlassen, dass sich Steuerelemente um alles Sichtbare in unseren Anwendungen kümmern. Aber manchmal reicht das nicht – beispielsweise wenn Sie ein **Bild animieren** möchten. Und wenn Sie einmal mit Animationen beginnen, werden Sie bald **eigene Steuerelemente** für Ihre .NET-Programme erstellen, denen Sie vielleicht etwas **Double-Buffering** hinzufügen, oder sogar **direkt auf Ihr Formular malen**. Alles beginnt mit dem **Graphics**-Objekt, **Bitmaps** und dem Willen, sich mit dem grafischen Status quo nicht zufriedenzugeben.



Sie haben die ganze Zeit Steuerelemente für die Interaktion mit Ihren Programmen verwendet 564
Formular-Steuerelemente sind auch nur Objekte 565
Ein Renderer für Ihre Architektur 568
Controls sind für grafisch angezeigte Elemente gut geeignet 570
Ihr erstes animiertes Steuerelement erstellen 573
Ihre Steuerelemente müssen dispose() anbieten 577
Mit UserControl können Steuerelemente leicht erstellt werden 578
Fügen Sie dem Projekt die neuen Formulare hinzu 582
Schauen wir uns zunächst die Leistungsprobleme an 590
Bitmaps skalieren Sie über ein Graphics-Objekt 592
Ihre Grafikressourcen werden als Bitmap-Objekt gespeichert 593
Nutzen Sie System.Drawing, um Ihre Grafiken SELBST zu STEUERN 594
Ein GDI+-Schnellkurs 595
Mit Graphics ein Bild auf ein Formular zeichnen 596
Graphics kann unser Transparenzproblem lösen 601
Paint macht Ihre Zeichnungen dauerhaft 602
Das Neuzeichnen von Formularen und Steuerelementen aus der Nähe betrachtet 605
Double-Buffering macht Ihre Animation glatter 608
Double-Buffering ist in Formulare und Steuerelemente eingebaut 609
Mit einem Graphics-Objekt und einem Event-Handler drucken 614
PrintDocument nutzt den Drucken-Dialog und die Druckvorschau 615

14 CAPTAIN AMAZING
DER TOD DES OBJEKTS

Captain Amazing, das Super-Objekt von Objekthausen, verfolgt seinen Erzfeind ... 622

Ihre letzte Chance, etwas zu TUN ... der Destruktor Ihres Objekts 628

Wann GENAU läuft ein Destruktor? 629

Dispose() arbeitet mit using, Destruktoren mit der Garbage Collection 630

Destruktoren können von der Stabilität abhängen 632

Lassen Sie ein Objekt sich selbst in Dispose() serialisieren 633

Ein Struct sieht aus wie ein Objekt ... 637

... ist aber nicht auf dem Heap 637

Werte werden kopiert, Referenzen werden zugewiesen 638

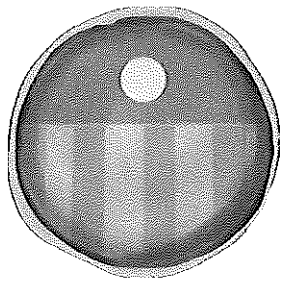
Structs sind Werttypen, Objekte sind Referenztypen 639

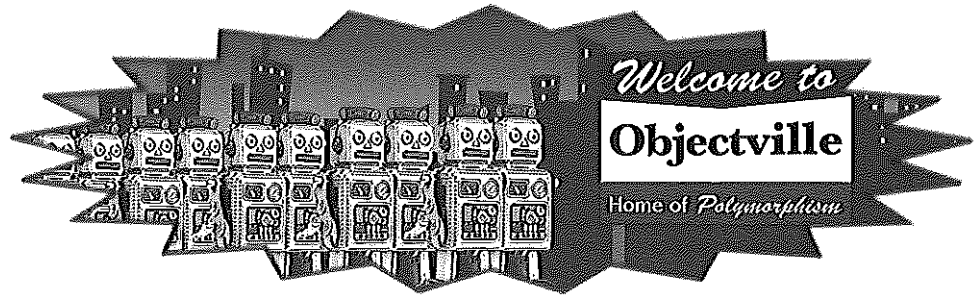
Stack vs. Heap: mehr zum Speicher 641

Captain Amazing ... nicht ganz 645

Erweiterungsmethoden fügen BESTEHENDEN Klassen neue Verhalten hinzu 646

Einen elementaren Typ erweitern: string 648





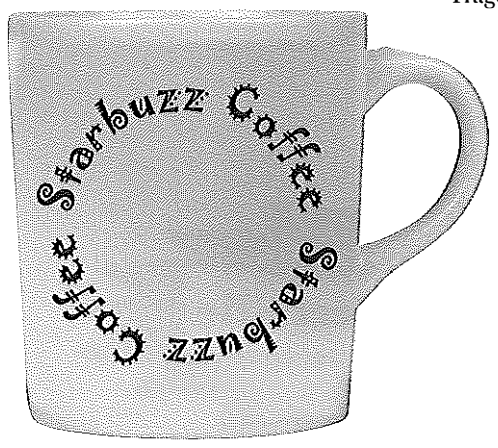
LINQ

15 Bekommen Sie Ihre Daten in den Griff

Die Welt ist datengesteuert ... Sie sollten damit zu leben wissen.

Vorbei sind die Zeiten, in denen Sie Tage, sogar Wochen programmieren konnten, ohne sich mit **Massen von Daten** befassen zu müssen. Heute ***dreht sich alles um Daten***. Und häufig muss man sich sogar mit Daten **aus unterschiedlichen Quellen** herumschlagen ... und in unterschiedlichen Formaten. Datenbanken, XML, Auflistungen aus anderen Programmen ... all das ist heute die Aufgabe eines guten C#-Programmierers. Und mit **LINQ** lässt sich all das bewältigen. Mit LINQ können Sie **Ihre Daten** nicht bloß auf leichte, intuitive Weise **abfragen**, sondern auch **gruppieren** und **Daten von unterschiedlichen Quellen zusammenführen**.

Ein leichtes Projekt ...	654
... aber die Daten sind völlig zerstreut	655
LINQ kann Daten aus mehreren Quellen ziehen	656
.NET-Auflistungen sind bereits für LINQ eingerichtet	657
LINQ vereinfacht Abfragen	658
LINQ ist einfach, Ihre Abfragen sind das nicht immer	659
LINQ ist vielseitig	662
LINQ kann Ergebnisse zu Gruppen zusammenfassen	667
Tims Werte gruppieren	668
Zwei Auflistungen in einer Abfrage mit join kombinieren	671
Tim hat eine Menge Kohle gespart	672
LINQ mit einer SQL-Datenbank verbinden	674
Fragen Sie Sternback und die Papierfabrik mit einem Join ab	678

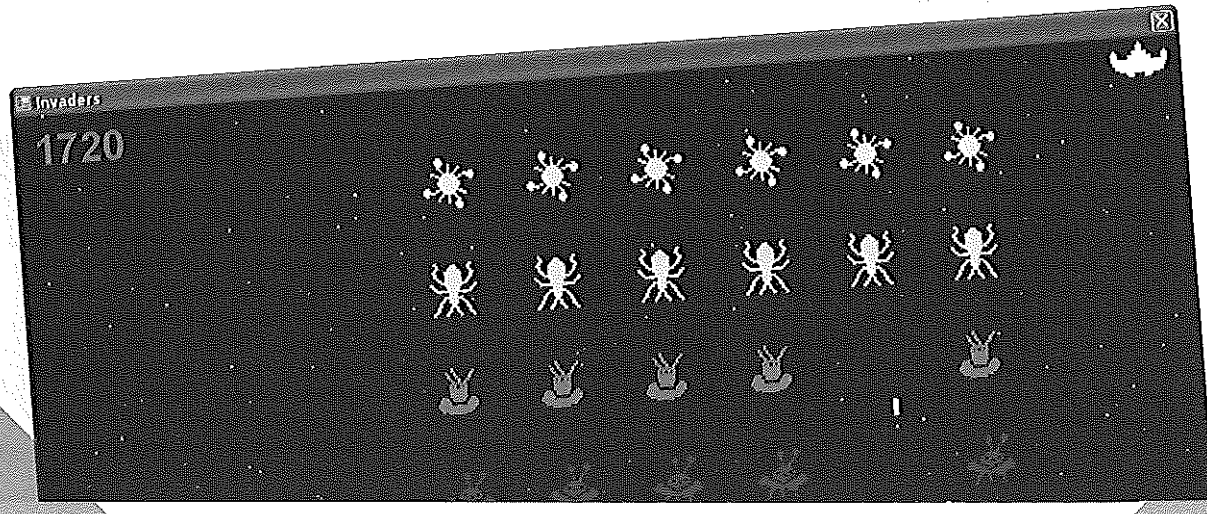


C#-Workshop 3

Invaders

In diesem Workshop werden Sie einer der beliebtesten, am meisten bewunderten und am häufigsten kopierten Ikonen der Videospielgeschichte die Ehre erweisen, einem Spiel, das keiner weiteren Vorstellung bedarf. Es ist Zeit, ein Invaders-Spiel zu schreiben.

Der Urvater aller Videospiele 682
Und es ist immer noch mehr zu tun ... 701



Was übrig bleibt

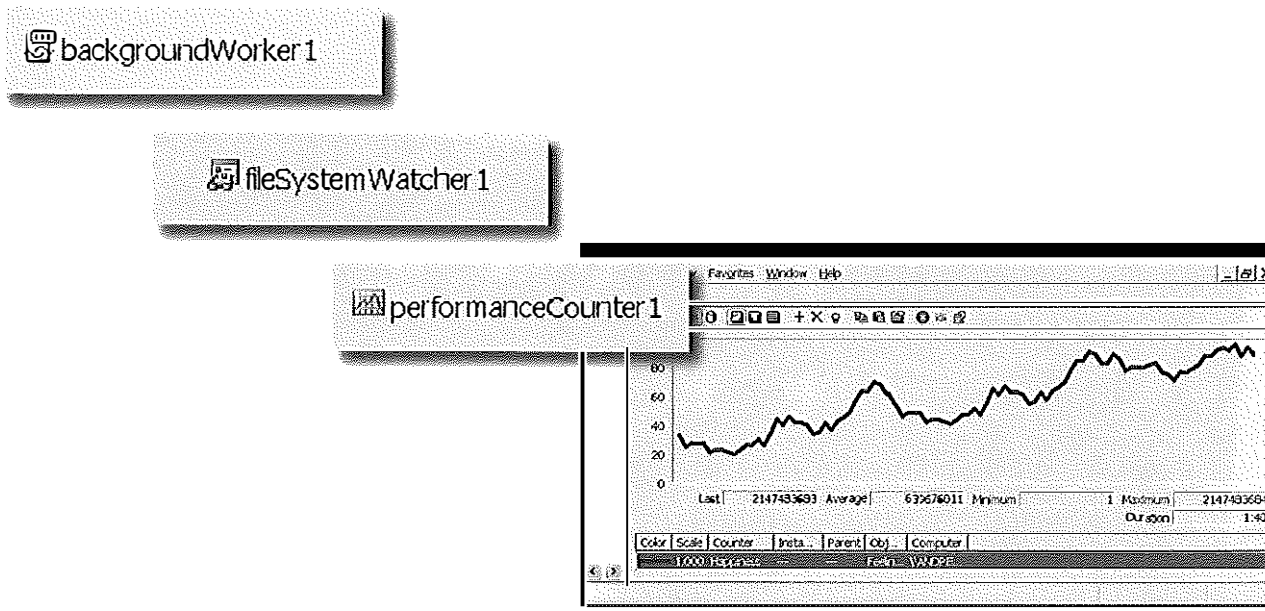
Die Top-5 der Themen, die es nicht ins Buch geschafft haben

Der Spaß fängt gerade erst an!

Wir haben Ihnen viele wunderbare Werkzeuge gezeigt, mit denen Sie mit C# richtig **mächtige Software** aufbauen können. Aber es war unmöglich, **alle Werkzeuge, Technologien und Techniken** in dieses Buch einzuschließen – so viele Seiten hat es einfach nicht. Wir mussten einige *sehr harte Entscheidungen* in Bezug darauf treffen, was wir mit aufnehmen und was wir weglassen. Hier sind einige Themen, die es nicht geschafft haben. Aber auch wenn wir zu ihnen nicht gekommen sind, denken wir trotzdem, dass sie **wichtig und nützlich** sind, und möchten Ihnen einen kleinen Ausblick auf sie bieten.

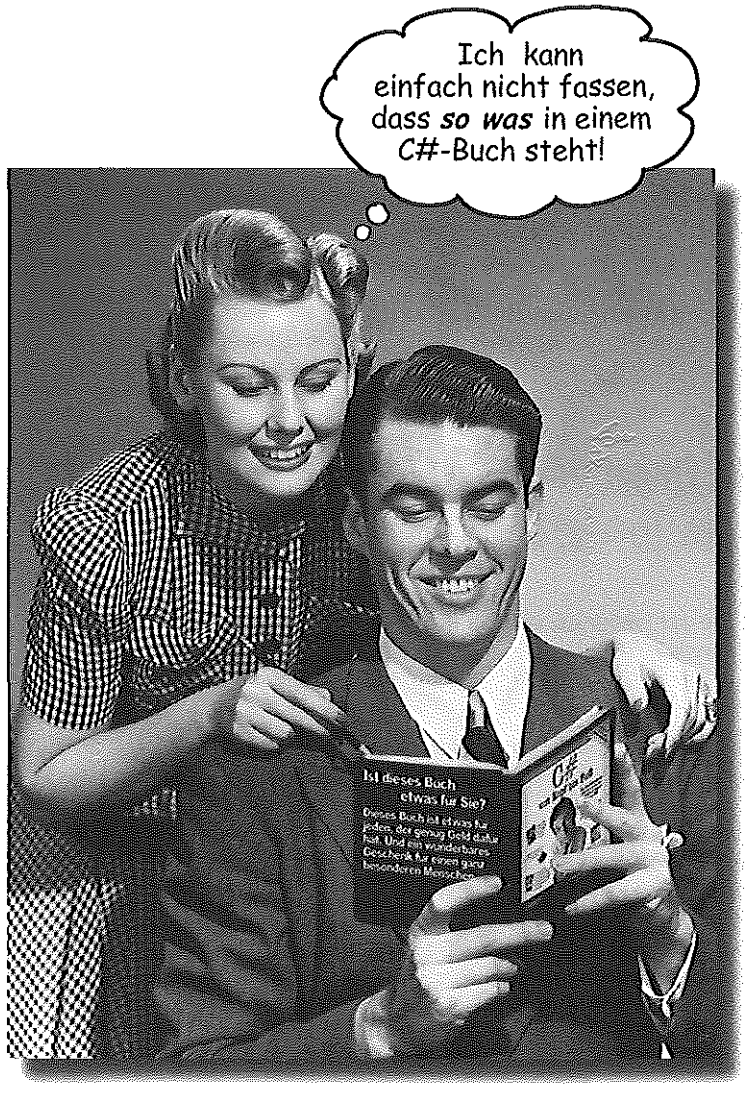
1. LINQ to XML 704
2. Umgestalten (Refactoring) 706
3. Einige unserer Lieblingskomponenten in der Toolbox 708
4. Konsolenanwendungen 710
5. Windows Presentation Foundation 712

Wussten Sie, dass Sie mit C# und dem .NET Framework ... 714



Wie man dieses Buch benutzt

Einführung



In diesem Abschnitt beantworten wir die brennende Frage:
»Und? Warum STEHT so was in einem C#-Buch?«

Für wen ist dieses Buch?

Wenn Sie alle folgenden Fragen mit »Ja« beantworten können ...

1. Möchten Sie C# **lernen**?

2. Basteln Sie gern herum – lernen Sie, indem Sie etwas anwenden, statt einfach nur zu lesen?

3. Ziehen Sie **anregende Partyunterhaltungen** trockenen, öden, akademischen Vorlesungen vor?

... dann ist dieses Buch etwas für Sie.

Wer sollte eher die Finger von diesem Buch lassen?

Wenn Sie ***eine*** dieser Fragen mit »Ja« beantworten müssen ...

1. Macht Sie der Gedanke, viel Code schreiben zu müssen, etwas nervös?

2. Sind Sie ein ausgebuffter C++- oder Java-Programmierer und nur auf der Suche nach einem Referenzbuch?

3. Sie haben **Angst, etwas Neues auszuprobieren?** Sie unterziehen sich lieber einer Wurzelbehandlung, als in einer Streifen-Karo-Kombination auf die Straße zu gehen? Sie sind überzeugt davon, dass ein Fachbuch nicht seriös sein kann, wenn C# vermenschlicht wird?



... dann ist dieses Buch nicht das richtige für Sie.

[Anmerkung aus dem Marketing: Dieses Buch ist etwas für jeden, der eine Kreditkarte besitzt.]

Wir wissen, was Sie gerade denken.

»Wie kann *das* ein ernsthaftes C#-Programmierbuch sein?«

»Was sollen all die Abbildungen?«

»Kann ich auf diese Weise wirklich *lernen*?«

Und wir wissen, was Ihr *Gehirn* gerade denkt.

Ihr Gehirn lechzt nach Neuem. Es ist ständig dabei, Ihre Umgebung abzusuchen, und es *wartet* auf etwas Ungewöhnliches. So ist es nun einmal gebaut, und es hilft Ihnen zu überleben.

Heutzutage ist es weniger wahrscheinlich, dass Sie von einem Tiger verputzt werden. Aber Ihr Gehirn hält immer noch Ausschau. Man weiß ja nie.

Also, was macht Ihr Gehirn mit all den gewöhnlichen, normalen Routinesachen, denen Sie begegnen? Es tut alles in seiner Macht Stehende, damit es dadurch nicht bei seiner *eigentlichen* Arbeit gestört wird: Dinge zu erfassen, die wirklich *wichtig* sind. Es gibt sich nicht damit ab, die langweiligen Sachen zu speichern, sondern lässt diese gar nicht erst durch den »Dies-ist-offensichtlich-nicht-wichtig«-Filter.

Woher *weiß* Ihr Gehirn denn, was wichtig ist? Nehmen Sie an, Sie machen einen Tagesausflug und ein Tiger springt vor Ihnen aus dem Gebüsch: Was passiert dabei in Ihrem Kopf und Ihrem Körper?

Neuronen feuern. Gefühle werden angekurbelt. *Chemische Substanzen durchfluten Sie.*

Und so weiß Ihr Gehirn:

Dies muss wichtig sein! Vergiss es nicht!

Aber nun stellen Sie sich vor, Sie sind zu Hause oder in einer Bibliothek. In einer sicheren, warmen, tigerfreien Zone. Sie lernen. Bereiten sich auf eine Prüfung vor. Oder Sie versuchen, irgendein schwieriges Thema zu lernen, von dem Ihr Chef glaubt, Sie bräuchten dafür eine Woche oder höchstens zehn Tage.

Da ist nur ein Problem: Ihr Gehirn versucht Ihnen einen großen Gefallen zu tun. Es versucht dafür zu sorgen, dass diese *offensichtlich* unwichtigen Inhalte nicht knappe Ressourcen verstopfen. Ressourcen, die besser dafür verwendet würden, die wirklich *wichtigen* Dinge zu speichern. Wie Tiger. Wie die Gefahren des Feuers. Oder dass Sie nie wieder in Shorts snowboarden sollten.

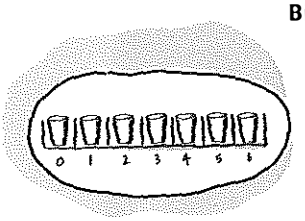
Und es gibt keine einfache Möglichkeit, Ihrem Gehirn zu sagen: »Hey, Gehirn, vielen Dank, aber egal, wie langweilig dieses Buch auch ist und wie klein der Ausschlag auf meiner emotionalen Richterskala gerade ist, ich *will* wirklich, dass du diesen Kram behältst.«



Wir stellen uns unseren Leser als einen aktiv Lernenden vor.

Also, was ist nötig, damit Sie etwas *lernen*? Erst einmal müssen Sie es *aufnehmen* und dann dafür sorgen, dass Sie es nicht wieder *vergessen*. Es geht nicht darum, Fakten in Ihren Kopf zu schieben. Nach den neuesten Forschungsergebnissen der Kognitionswissenschaft, der Neurobiologie und der Lernpsychologie gehört zum *Lernen* viel mehr als nur Text auf einer Seite. Wir wissen, was Ihr Gehirn anmacht.

Einige der Lernprinzipien dieser Buchreihe:



Bilder einsetzen. An Bilder kann man sich viel besser erinnern als an Worte allein und lernt so viel effektiver (bis zu 89% Verbesserung bei Abrufbarkeits- und Lerntransferstudien). Außerdem werden die Dinge dadurch verständlicher. **Text in oder neben die Grafiken setzen,** auf die sie sich beziehen, anstatt darunter oder auf eine andere Seite. Die Leser werden auf den Bildinhalt bezogene Probleme dann mit *doppelt* so hoher Wahrscheinlichkeit lösen können.

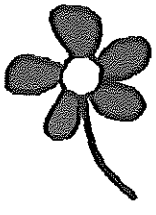
Verwenden Sie einen gesprächsorientierten Stil mit persönlicher Ansprache. Nach neueren Untersuchungen haben Studenten nach dem Lernen bei Tests bis zu 40% besser abgeschnitten, wenn der Inhalt den Leser direkt in der ersten Person und im lockeren Stil angesprochen hat statt in einem formalen Ton. Halten Sie keinen Vortrag, sondern erzählen Sie Geschichten. Benutzen Sie eine zwanglose Sprache. Nehmen Sie sich selbst nicht zu ernst. Würden *Sie* einer anregenden Unterhaltung beim Abendessen mehr Aufmerksamkeit schenken oder einem Vortrag?



Bringen Sie den Lernenden dazu, intensiver nachzudenken. Mit anderen Worten: Falls Sie nicht aktiv Ihre Neuronen strapazieren, passiert in Ihrem Gehirn nicht viel. Ein Leser muss motiviert, begeistert und neugierig sein und dazu angeregt werden, Probleme zu lösen, Schlüsse zu ziehen und sich neues Wissen anzueignen. Und dafür brauchen Sie Herausforderungen, Übungen, zum Nachdenken anregende Fragen und Tätigkeiten, die beide Seiten des Gehirns und mehrere Sinne einbeziehen.

Ziehen Sie die Aufmerksamkeit des Lesers auf sich – und behalten Sie sie. Wir alle haben schon Erfahrungen dieser Art gemacht: »Ich will das wirklich lernen, aber ich kann einfach nicht über Seite 1 hinaus wach bleiben.« Ihr Gehirn passt auf, wenn Dinge ungewöhnlich, interessant, merkwürdig, auffällig, unerwartet sind. Ein neues, schwieriges, technisches Thema zu lernen muss nicht langweilig sein. Wenn es das nicht ist, lernt Ihr Gehirn viel schneller.





Sprechen Sie Gefühle an. Wir wissen, dass Ihre Fähigkeit, sich an etwas zu erinnern, wesentlich von dessen emotionalem Gehalt abhängt. Sie erinnern sich an das, was Sie *bewegt*. Sie erinnern sich, wenn Sie etwas *fühlen*. Nein, wir erzählen keine herzzerreißenden Geschichten über einen Jungen und seinen Hund. Was wir erzählen, ruft Überraschungs-, Neugier-, Spaß- und Was-soll-das?-Emotionen hervor und dieses Hochgefühl, das Sie beim Lösen eines Puzzles empfinden oder wenn Sie etwas lernen, was alle anderen schwierig finden. Oder wenn Sie merken, dass Sie etwas können, was dieser »Ich-bin-ein-besserer-Techniker-als-du«-Typ aus der Technikabteilung *nicht kann*.



Metakognition: Nachdenken übers Denken

Wenn Sie wirklich lernen möchten, und zwar schneller und nachhaltiger, dann schenken Sie Ihrer Aufmerksamkeit Aufmerksamkeit. Denken Sie darüber nach, wie Sie denken. Lernen Sie, wie Sie lernen.



Die meisten von uns haben in ihrer Jugend keine Kurse in Metakognition oder Lerntheorie gehabt. Es wurde von uns *erwartet,* dass wir lernen, aber nur selten wurde uns auch *beigebracht,* wie man lernt.

Wir nehmen aber an, dass Sie wirklich etwas über C# lernen möchten, wenn Sie dieses Buch in den Händen halten. Und wahrscheinlich möchten Sie nicht viel Zeit aufwenden. Und Sie wollen sich an das *erinnern,* was Sie lesen, und es anwenden können. Und deshalb müssen Sie es *verstehen.* Wenn Sie so viel wie möglich von diesem Buch profitieren wollen oder von irgendeinem anderen Buch oder einer anderen Lernerfahrung, übernehmen Sie Verantwortung für Ihr Gehirn. Ihr Gehirn im Zusammenhang mit diesem Lernstoff.

Der Trick besteht darin, Ihr Gehirn dazu zu bringen, neuen Lernstoff als etwas wirklich Wichtiges anzusehen. Als entscheidend für Ihr Wohlbefinden. So wichtig wie ein Tiger. Andernfalls stecken Sie in einem dauernden Kampf, in dem Ihr Gehirn sein Bestes gibt, um die neuen Inhalte davon abzuhalten, hängen zu bleiben.

Wie bringen Sie also Ihr Gehirn dazu, C# für so wichtig zu halten wie einen Tiger?

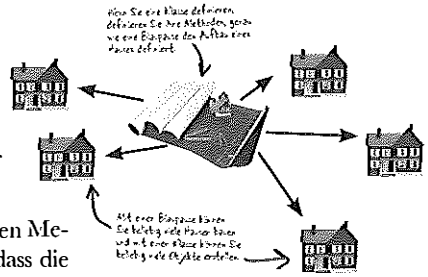
Da gibt es den langsamen, ermüdenden Weg oder den schnelleren, effektiveren Weg. Der langsame Weg geht über bloße Wiederholung. Natürlich ist Ihnen klar, dass Sie lernen und sich sogar an die langweiligsten Themen erinnern *können,* wenn Sie sich die gleiche Sache immer wieder einhämmern. Wenn Sie nur oft genug wiederholen, sagt Ihr Gehirn: »Er hat zwar nicht das *Gefühl,* dass das wichtig ist, aber er sieht sich dieselbe Sache *immer und immer wieder* an – dann muss sie wohl wichtig sein.«

Der schnellere Weg besteht darin, ***alles zu tun, was die Gehirnaktivität erhöht***, vor allem verschiedene *Arten* von Gehirnaktivität. Eine wichtige Rolle dabei spielen die auf der vorhergehenden Seite erwähnten Dinge – alles Dinge, die nachweislich helfen, dass Ihr Gehirn *für* Sie arbeitet. So hat sich z.B. in Untersuchungen gezeigt: Wenn Wörter *in* den Abbildungen stehen, die sie beschreiben (und nicht irgendwo anders auf der Seite, z.B. in einer Bildunterschrift oder im Text), versucht Ihr Gehirn herauszufinden, wie die Wörter und das Bild zusammenhängen, und dadurch feuern mehr Neuronen. Und je mehr Neuronen feuern, umso größer ist die Chance, dass Ihr Gehirn mitbekommt: Bei dieser Sache *lohnt* es sich aufzupassen und vielleicht auch, sich daran zu erinnern.

Ein lockerer Sprachstil hilft, denn Menschen tendieren zu höherer Aufmerksamkeit, wenn ihnen bewusst ist, dass sie ein Gespräch führen – man erwartet dann ja von ihnen, dass sie dem Gespräch folgen und sich beteiligen. Das Erstaunliche daran ist: Es ist Ihrem Gehirn ziemlich *egal*, dass die »Unterhaltung« zwischen Ihnen und einem Buch stattfindet! Wenn der Schreibstil dagegen formal und trocken ist, hat Ihr Gehirn den gleichen Eindruck wie bei einem Vortrag, bei dem in einem Raum passive Zuhörer sitzen. Nicht nötig, wach zu bleiben.

Aber Abbildungen und ein lockerer Sprachstil sind erst der Anfang.

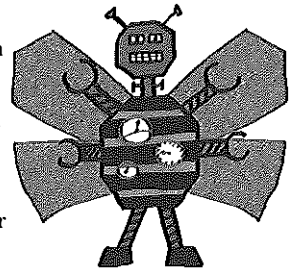
Das haben WIR getan:



Wir haben ***Bilder*** verwendet, weil Ihr Gehirn auf visuelle Eindrücke eingestellt ist, nicht auf Text. Soweit es Ihr Gehirn betrifft, sagt ein Bild *wirklich* mehr als 1.024 Worte. Und dort, wo Text und Abbildungen zusammenwirken, haben wir den Text *in* die Bilder eingebettet, denn Ihr Gehirn arbeitet besser, wenn der Text *innerhalb* der Sache steht, auf die er sich bezieht, und nicht in einer Bildunterschrift oder irgendwo vergraben im Text.

Wir haben ***Redundanz*** eingesetzt, d.h. dasselbe auf *unterschiedliche* Art und mit verschiedenen Medientypen ausgedrückt, damit Sie es über *mehrere Sinne* aufnehmen. Das erhöht die Chance, dass die Inhalte an mehr als nur einer Stelle in Ihrem Gehirn verankert werden.

Wir haben Konzepte und Bilder in ***unerwarteter*** Weise eingesetzt, weil Ihr Gehirn auf Neuigkeiten programmiert ist. Und wir haben Bilder und Ideen mit zumindest *etwas* ***emotionalem*** *Charakter* verwendet, weil Ihr Gehirn darauf eingestellt ist, auf die Biochemie von Gefühlen zu achten. An alles, was ein *Gefühl* in Ihnen auslöst, können Sie sich mit höherer Wahrscheinlichkeit erinnern, selbst wenn dieses Gefühl nicht mehr ist als ein bisschen ***Belustigung, Überraschung oder Interesse.***



Wir haben einen ***umgangssprachlichen Stil*** mit direkter Anrede benutzt, denn Ihr Gehirn ist von Natur aus aufmerksamer, wenn es Sie in einer Unterhaltung wähnt als wenn es davon ausgeht, dass Sie passiv einer Präsentation zuhören – sogar dann, wenn Sie *lesen.*

Wir haben mehr als 80 ***Aktivitäten*** für Sie vorgesehen, denn Ihr Gehirn lernt und behält von Natur aus besser, wenn Sie Dinge ***tun,*** als wenn Sie nur darüber *lesen.* Und wir haben die Übungen zwar anspruchsvoll, aber doch lösbar gemacht, denn so ist es den meisten Lesern am liebsten.

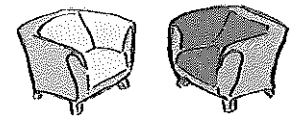
Wir haben ***mehrere unterschiedliche Lernstile*** eingesetzt, denn vielleicht bevorzugen *Sie* ein Schritt-für-Schritt-Vorgehen, während jemand anders erst einmal den groben Zusammenhang verstehen und ein Dritter einfach nur ein Codebeispiel sehen möchte. Aber ganz abgesehen von den jeweiligen Lernvorlieben profitiert *jeder* davon, wenn er die gleichen Inhalte in unterschiedlicher Form präsentiert bekommt.

Punkt für Punkt

Wir liefern Inhalte für ***beide Seiten Ihres Gehirns,*** denn je mehr Sie von Ihrem Gehirn einsetzen, umso wahrscheinlicher werden Sie lernen und behalten und umso länger bleiben Sie konzentriert. Wenn Sie mit einer Seite des Gehirns arbeiten, bedeutet das häufig, dass sich die andere Seite des Gehirns ausruhen kann; so können Sie über einen längeren Zeitraum produktiver lernen.

Und wir haben ***Geschichten*** und Übungen aufgenommen, die ***mehr als einen Blickwinkel repräsentieren,*** denn Ihr Gehirn lernt von Natur aus intensiver, wenn es gezwungen ist, selbst zu analysieren und zu beurteilen.

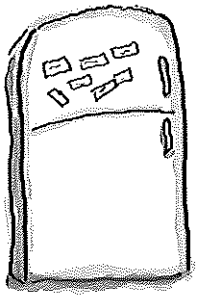
Kamingespräche



Wir haben ***Herausforderungen*** eingefügt: in Form von Übungen und indem wir ***Fragen*** stellen, auf die es nicht immer eine eindeutige Antwort gibt, denn Ihr Gehirn ist darauf eingestellt, zu lernen und sich zu erinnern, wenn es an etwas *arbeiten* muss. Überlegen Sie: Ihren *Körper* bekommen Sie ja auch nicht in Form, wenn Sie nur die Leute auf dem Sportplatz *beobachten.* Aber wir haben unser Bestes getan, um dafür zu sorgen, dass Sie – wenn Sie schon hart arbeiten – an den *richtigen* Dingen arbeiten. Dass Sie ***nicht einen einzigen Dendriten darauf verschwenden,*** ein schwer verständliches Beispiel zu verarbeiten oder einen schwierigen, mit Fachbegriffen gespickten oder übermäßig gedrängten Text zu analysieren.



Wir haben ***Menschen*** eingesetzt. In Geschichten, Beispielen, Bildern usw. – denn *Sie sind* ein Mensch. Und Ihr Gehirn schenkt *Menschen* mehr Aufmerksamkeit als *Dingen.*



Schneiden Sie dies aus und heften Sie es an Ihren Kühlschrank.

Und das können SIE tun, um sich Ihr Gehirn untertan zu machen

So, wir haben unseren Teil der Arbeit geleistet. Der Rest liegt bei Ihnen. Diese Tipps sind ein Anfang; hören Sie auf Ihr Gehirn und finden Sie heraus, was bei Ihnen funktioniert und was nicht. Probieren Sie neue Wege aus.

1. **Immer langsam. Je mehr Sie verstehen, umso weniger müssen Sie auswendig lernen.**
   Lesen Sie nicht nur. Halten Sie inne und denken Sie nach. Wenn das Buch Sie etwas fragt, springen Sie nicht einfach zur Antwort. Stellen Sie sich vor, dass Sie das wirklich jemand *fragt*. Je gründlicher Sie Ihr Gehirn zum Nachdenken zwingen, umso größer ist die Chance, dass Sie lernen und behalten.

2. **Bearbeiten Sie die Übungen. Machen Sie selbst Notizen.**
   Wir haben sie entworfen, aber wenn wir sie auch für Sie lösen würden, wäre das, als würde jemand anderes Ihr Training für Sie absolvieren. Und *sehen* Sie sich die Übungen *nicht einfach nur an.* **Benutzen Sie einen Bleistift.** Es deutet vieles darauf hin, dass körperliche Aktivität *beim* Lernen den Lernerfolg erhöhen kann.

3. **Lesen Sie die Abschnitte »Es gibt keine dummen Fragen«.**
   Und zwar alle. Das sind keine Zusatzanmerkungen – ***sie gehören zum Kerninhalt!*** Überspringen Sie sie nicht.

4. **Lesen Sie dies als Letztes vor dem Schlafengehen. Oder lesen Sie danach zumindest nichts Anspruchsvolles mehr.**
   Ein Teil des Lernprozesses (vor allem die Übertragung in das Langzeitgedächtnis) findet erst statt, *nachdem* Sie das Buch zur Seite gelegt haben. Ihr Gehirn braucht Zeit für sich, um weitere Verarbeitung zu leisten. Wenn Sie in dieser Zeit etwas Neues aufnehmen, geht ein Teil dessen, was Sie gerade gelernt haben, verloren.

5. **Trinken Sie Wasser. Viel.**
   Ihr Gehirn arbeitet am besten in einem schönen Flüssigkeitsbad. Austrocknung (zu der es schon kommen kann, bevor Sie überhaupt Durst verspüren) beeinträchtigt die kognitive Funktion.

6. **Reden Sie drüber. Laut.**
   Sprechen aktiviert einen anderen Teil des Gehirns. Wenn Sie etwas verstehen oder Ihre Chancen verbessern wollen, sich später daran zu erinnern, sagen Sie es laut. Noch besser: Versuchen Sie es jemand laut zu erklären. Sie lernen dann schneller und haben vielleicht Ideen, auf die Sie beim bloßen Lesen nie gekommen wären.

7. **Hören Sie auf Ihr Gehirn.**
   Achten Sie darauf, Ihr Gehirn nicht zu überladen. Wenn Sie merken, dass Sie etwas nur noch überfliegen oder dass Sie das gerade erst Gelesene vergessen haben, ist es Zeit für eine Pause. Ab einem bestimmten Punkt lernen Sie nicht mehr schneller, indem Sie mehr hineinzustopfen versuchen; das kann sogar den Lernprozess stören.

8. **Aber bitte mit Gefühl!**
   Ihr Gehirn muss wissen, dass es *um etwas Wichtiges geht.* Lassen Sie sich in die Geschichten hineinziehen. Erfinden Sie eigene Bildunterschriften für die Fotos. Über einen schlechten Scherz zu stöhnen ist *immer noch* besser, als gar nichts zu fühlen.

9. **Schreiben Sie Software, massenhaft!**
   Programmieren lernt man nur auf eine Weise: **indem man viel, viel Code schreibt**. Das werden Sie das ganze Buch hindurch machen. Programmieren ist ein Handwerk. Gut darin wird man nur durch Übung. Und die werden wir Ihnen verschaffen: Alle Kapitel bieten Übungen, die Ihnen ein Problem zur Lösung aufgeben. Überspringen Sie diese nicht einfach – ein Großteil des Lernprozesses tritt beim Lösen der Übungen ein. Wir geben zu allen Übungen Lösungen an – haben Sie keine Angst, **in die Lösung zu schauen**, wenn Sie hängen bleiben! (Man bleibt leicht mal an etwas ganz Kleinem hängen.) Aber probieren Sie, das Problem zu lösen, bevor Sie sich die Lösung ansehen.

Was Sie für dieses Buch brauchen

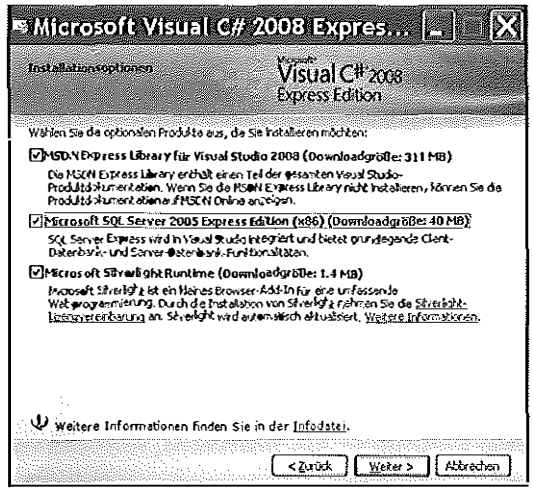
Wir haben dieses Buch unter Verwendung von Visual Studio 2008 Express Edition geschrieben. Das heißt, dass wir C# 3.0 und das .NET Framework 3.5 verwenden. Als wir dieses Buch geschrieben haben, befand sich Visual Studio 2008 noch in der Beta-Phase. Veröffentlicht wurde es erst nach dem Druck der amerkanischen Ausgabe. Alle Übungen und Codefragmente funktionieren mit den Beta-Versionen von Visual Studio 2008 Express. Sie sollten auch mit der endgültigen Version funktionieren. Wir empfehlen Ihnen, sie von der Microsoft-Website herunterzuladen – Download und Gebrauch sind kostenlos.

VISUAL STUDIO 2008 EXPRESS EINRICHTEN

- Das Herunterladen und Installieren von Visual C# 2008 Express ist ein Kinderspiel. Hier ist der Link zur Download-Seite für die Visual Studio 2008 Express-Versionen:

  `http://www.microsoft.com/germany/express/download/default.aspx`

  Achten Sie bei der Installation darauf, dass Sie alle Optionen auswählen.

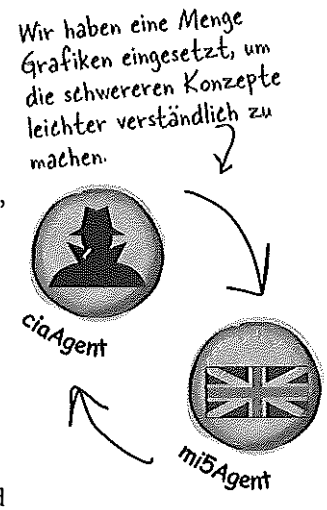


Bedenken Sie bitte, dass Sie, wenn Sie eine ältere Version von Visual Studio, C# oder dem .NET Framework verwenden müssen, in diesem Buch auf Themen stoßen werden, die mit Ihrer Version nicht kompatibel sind. Das C#-Team bei Microsoft hat der Sprache einige ziemlich coole Features hinzugefügt. Wir werden Sie darauf hinweisen, wenn wir über eins dieser Themen sprechen. Wenn Sie eine ältere Version verwenden, müssen Sie sich also darüber im Klaren sein, dass es in diesem Buch Code gibt, der bei Ihnen nicht funktioniert.

- Laden Sie das Installationspaket für Visual C# 2008 Express Edition. Achten Sie darauf, dass Sie eine vollständige Installation durchführen. Das sollte Ihnen alles liefern, was Sie benötigen: die IDE (die wir Ihnen im Folgenden erklären), SQL Server Express und das .NET Framework 3.5.
- Nach der Installation haben Sie im Start-Menü eine neue Option: ***Microsoft Visual C# 2008 Express Edition***. Klicken Sie darauf, um die IDE zu öffnen. Jetzt sind Sie für dieses Buch bereit.

Lies mich

Dies ist ein Lehrbuch, keine Referenz. Wir haben mit Absicht alles weggelassen, was Ihnen dabei in die Quere kommen könnte, das zu lernen, was auch immer wir an einem bestimmten Punkt gerade behandeln. Und wenn Sie das Buch das erste Mal durchhaben, müssen Sie wieder am Anfang beginnen, weil das Buch Annahmen darüber macht, was Sie bereits gesehen und gelernt haben.

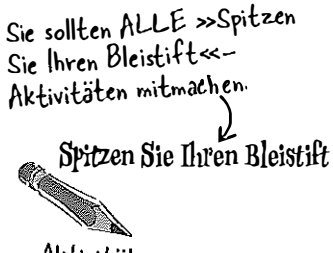


Die Aktivitäten sind NICHT optional.

Die Übungen und Aktivitäten sind keine Zusätze. Sie sind ein wesentlicher Bestandteil des Buchs. Einige sollen Ihr Gedächtnis unterstützen, andere Ihr Verständnis wecken und wieder andere werden Ihnen helfen, das Gelernte anzuwenden. ***Überspringen Sie die Übungen nicht.*** Die Kreuzworträtsel sind das Einzige, was Sie nicht tun *müssen*, sind aber eine gute Möglichkeit, Ihrem Hirn eine Chance zu geben, über die Wörter und Begriffe, die Sie gelernt haben, in einem anderen Kontext nachzudenken.

Die Redundanz ist beabsichtigt und wichtig.

Eins der Dinge, das ein Von Kopf bis Fuß-Buch so anders macht, ist, dass wir möchten, dass Sie die Sache *wirklich* verstehen. Und wir möchten, dass Sie, wenn Sie das Buch durchgearbeitet haben, behalten, was Sie gelernt haben. Die meisten Referenzbücher zielen nicht auf Behalten und Erinnern ab, aber in diesem Buch geht es ums *Lernen*, deswegen werden Ihnen manche Konzepte auch mehrfach begegnen.



Machen Sie alle Übungen!

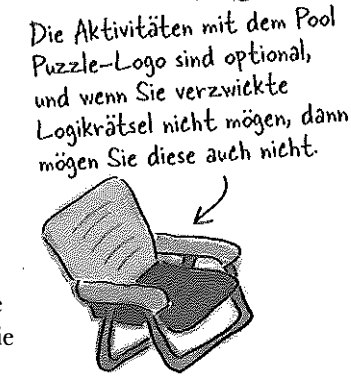
Als wir dieses Buch schrieben, sind wir von einer wichtigen Voraussetzung ausgegangen: dass Sie wirklich lernen möchten, wie man in C# programmiert. Wir wissen also, dass Sie sich die Hände von Anfang an beim Wühlen in Code schmutzig machen wollen. Wir bieten Ihnen eine Menge Möglichkeiten, Ihre Fähigkeiten zu schärfen, indem wir in jedes Kapitel Übungen einschließen. Einige haben wir mit »*Tun Sie das!*« beschriftet – wenn Sie darauf stoßen, bedeutet es, dass wir Sie durch alle Schritte zur Lösung eines bestimmten Problems leiten. Stoßen Sie aber auf das Übung-Logo mit den Turnschuhen, haben wir Ihnen die Lösung eines großen Happens des Problems überlassen und präsentieren Ihnen anschließend die Lösung, die wir gefunden haben. Haben Sie keine Angst, einen Blick auf die Lösung zu werfen – **das ist kein Mogeln**! Aber am meisten lernen Sie, wenn Sie versuchen, das Problem erst selbst zu lösen.



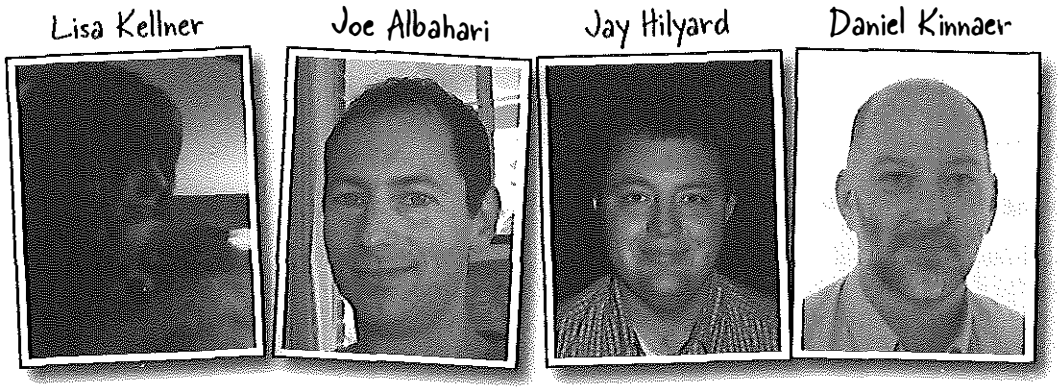
Außerdem haben wir den Quellcode für die Lösungen aller Übungen ins Web gestellt, damit Sie ihn herunterladen können. Sie finden ihn unter:
`http://examples.oreilly.de/german_examples/hfcsharpger/`

Zu den Kopfnuss-Übungen gibt es keine Lösungen.

Für manche dieser Übungen gibt es keine richtige Lösung, und bei anderen gehört es zum Lernprozess der Kopfnuss-Aktivitäten, dass Sie selbst überlegen, ob und wann Ihre Lösungen richtig sind. Bei einigen Kopfnuss-Übungen finden Sie Hinweise, die Sie in die richtige Richtung lenken.



Die Fachgutachter



Nicht abgebildet (und trotzdem ebenso umwerfend): Wayne Bradney, Dave Murdoch und Bridgette Julie Landers.





Als wir dieses Buch schrieben, wies es einen Haufen Fehler, Haken, Probleme, Tippfehler und schreckliche arithmetische Fehler auf. Na gut, ganz so schlimm war es nicht. Trotzdem sind wir dankbar für die Arbeit, die unsere Fachgutachter für dieses Buch geleistet haben. Ohne dieses ausgezeichnete Gutachterteam wäre das Buch mit Fehlern gedruckt worden (einschließlich ein bis zwei großer).

Vor allem möchten wir **Joe Albahari** für die ausführliche technische Beratung danken. Er hat uns bei einigen wichtigen Dingen berichtigt. Wäre er nicht gewesen, müssten Sie falsche Dinge lernen. Außerdem wollten wir **Lisa Kellner** danken – das ist das dritte Buch, das sie für uns korrigiert hat, und das hatte entscheidenden Einfluss auf die Lesbarkeit des endgültigen Produkts. Danke Lisa! Besonderen Dank auch an **Jay Hilyard** und **Daniel Kinnaer**, die viele unserer Fehler aufgespürt und korrigiert haben, und an **Aayam Singh**, der tatsächlich alle Übungen gemacht hat, **bevor** wir sie repariert und ihre Fehler behoben haben. Aayam, du bist wirklich engagiert. Danke!



Danksagungen

Unserem Lektor:

Wir möchten unserem Lektor, **Brett McLaughlin**, für seine Arbeit an diesem Buch danken. Beim Aufbau hat er uns sehr geholfen, und die Comic-Idee in Kapitel 14 war ganz allein seine Idee. Wir meinen, dass das richtig gut geworden ist. Danke Brett!



Brett McLaughlin

Dem O'Reilly-Team:



Lou Barr

Lou Barr ist eine wunderbare Grafikerin, die sich bei diesem Buch selbst übertroffen hat. Sie hat unglaublich viele Stunden aufgewandt und einige ziemlich umwerfende Darstellungen gestaltet. Wenn Sie in diesem Buch auf etwas stoßen, das fantastisch aussieht, sollten Sie ihr (und ihren verrückten InDesign-Künsten) danken. Sie hat alle Monster- und Alien-Grafiken für die Workshops sowie den vollständigen Comic gemacht. Vielen Dank Lou! Du bist unsere Heldin, und es ist großartig, mit dir zu arbeiten.



Catherine Nolan

Es gibt so viele Menschen bei O'Reilly, denen wir danken wollen – hoffentlich vergessen wir niemanden. Das Head First-Team ist klasse – **Laurie Petrycki**, **Catherine Nolan**, **Sanders Kleinfeld** (der beste Hersteller aller Zeiten!), **Caitrin McCullough** und **Keith McNamara**. Besonderen Dank an **Colleen Gorman** für ihr aufmerksames Sprachkorrektorat, **Ron Bilodeau**, der seine Zeit und Preflighting-Expertise zur Verfügung gestellt hat, sowie **Adam Witwer**, der das Buch einer letzten Qualitätssicherung unterzog. Sie alle halfen, das Buch in Rekordzeit aus der Produktion in den Druck zu bekommen. Und wie immer: Wir lieben **Mary Treseler** und können es nicht erwarten, wieder mit ihr zu arbeiten! Und ein großes Hallo an all unsere anderen Freunde und Lektoren, **Andy Oram**, **Isabel Kunkle** und **Mike Hendrickson**. Und wenn Sie dieses Buch jetzt lesen, können Sie dem besten Marketing-Team der Branche danken: **Marsee Henon**, **Sara Peyton**, **Mary Rotman**, **Jessica Boyd** und den anderen Leuten in Sebastopol.

1 Produktiv mit C#

Grafische Anwendungen in 10 Minuten und weniger



Sie möchten Programme so richtig schnell erstellen können?

Mit C# halten Sie eine **mächtige Programmiersprache** und ein **wertvolles Werkzeug** in Ihren Händen. Dank der **Visual Studio IDE** müssen Sie nie wieder Stunden damit vergeuden, obskuren Code zu schreiben, damit ein Button schließlich tut, was er soll. Noch besser ist, dass Sie sich endlich nur noch darauf konzentrieren müssen, **Ihre Arbeit zu erledigen**, statt sich zu merken, welcher der Methodenparameter den *Namen* eines Buttons angibt und welcher sein *Label*. Klingt das vielversprechend? Blättern Sie um, damit wir mit dem Programmieren beginnen können.

Warum Sie C# lernen sollten

C# und die Visual Studio IDE machen es Ihnen leicht, beim Schreiben von Code zur Sache zu kommen und Ihren Code schnell zu schreiben. Wenn Sie mit C# arbeiten, ist die IDE Ihr bester Freund und ständiger Begleiter.

Die IDE – oder Visual Studio Integrated Development Environment – ist ein wichtiger Bestandteil der Arbeit mit C#. Sie ist ein Programm, das Ihnen hilft, Ihren Code zu bearbeiten, Ihre Dateien zu verwalten und Ihre Projekte zu veröffentlichen.

Das sind die Dinge, die die IDE für Sie automatisiert ...

Jedes Mal, wenn Sie anfangen, ein Programm zu schreiben, oder einfach nur einen Button in ein Formular platzieren wollen, verlangt Ihr Programm eine Masse sich stets wiederholenden Codes.

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

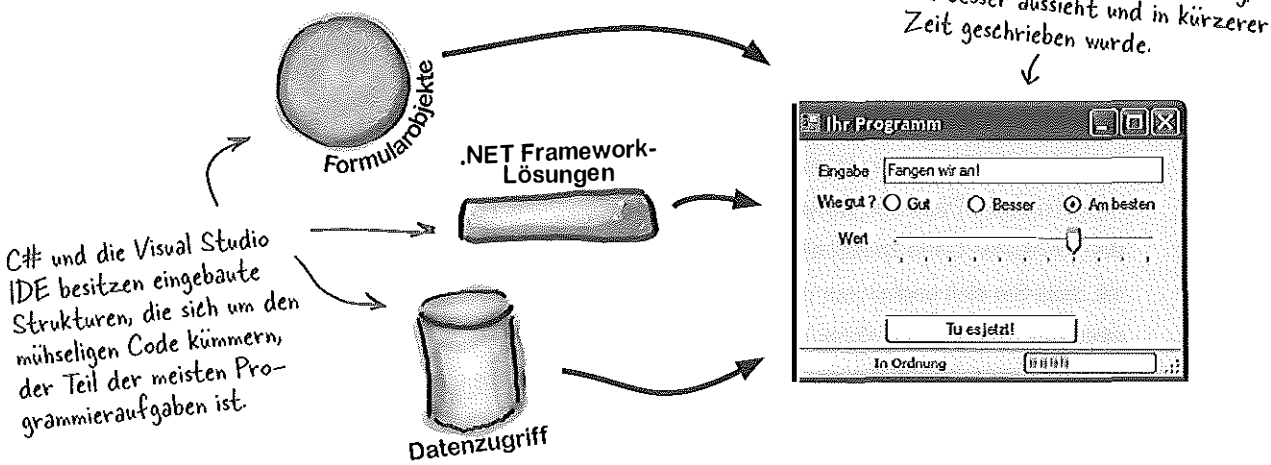
namespace WindowsApplication1
{
    static class Program
    {
        /// <summary>
        /// Der Haupteinstiegspunkt für die Anwendung.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

```
Private void InitializeComponent()
{
  this.button1 = new System.Windows.Forms.Button();
  this.SuspendLayout();
  //
  // button1
  //
  this.button1.Location = new System.Drawing.Point(105, 56);
  this.button1.Name = "button1";
  this.button1.Size = new System.Drawing.Size(75, 23);
  this.button1.TabIndex = 0;
  this.button1.Text = "button1";
  this.button1.UseVisualStyleBackColor = true;
  this.button1.Click += new System.EventHandler(this.button1_Click);
  //
  // Form1
  //
  this.AutoScaleDimensions = new System.Drawing.SizeF(8F, 16F);
  this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
  this.ClientSize = new System.Drawing.Size(292, 267);
  this.Controls.Add(this.button1);
  this.Name = "Form1";
  this.Text = "Form1";
  this.ResumeLayout(false);
}
```

Dieser ganze Code ist erforderlich, um einen Button in ein Formular zu zeichnen. Fügen Sie dem Formular noch ein paar weitere grafische Elemente hinzu, führt das unter Umständen zu zehnmal so viel Code.

Was Sie mit Visual Studio und C# erhalten ...

Mit einer Sprache wie C#, die speziell auf die Windows-Programmierung ausgerichtet ist, und der Visual Studio IDE können Sie sich sofort mit dem befassen, was Ihr Programm **tun** soll:



C# und die Visual Studio IDE besitzen eingebaute Strukturen, die sich um den mühseligen Code kümmern, der Teil der meisten Programmieraufgaben ist.

Das Ergebnis ist eine Anwendung, die besser aussieht und in kürzerer Zeit geschrieben wurde.

C# und Visual Studio IDE erleichtern Ihnen eine Menge Dinge

Wenn Sie C# und Visual Studio einsetzen, erhalten Sie all diese wunderbaren Features, ohne dass Sie dazu zusätzliche Arbeit leisten müssen. Beide zusammen ermöglichen Ihnen:

1. **Anwendungen SCHNELL aufzubauen.** Mit C# ist das Erstellen von Programmen ein Kinderspiel. Die Sprache ist mächtig und leicht zu erlernen, und die Visual Studio IDE erledigt eine Menge Arbeit automatisch für Sie. Die stupiden Programmieraufgaben können Sie der IDE überlassen und sich selbst auf das konzentrieren, was Ihr Code bewirken soll.

2. **Gut aussehende Benutzerschnittstellen zu gestalten.** Der Form-Designer in der Visual Studio IDE ist eins der am einfachsten zu benutzenden Design-Werkzeuge, die es gibt. Er macht so viel für Sie, dass Sie die Entwicklung umwerfend aussehender Benutzerschnittstellen als einen der erfüllendsten Teile der Entwicklung von C#-Anwendungen empfinden werden. Sie werden voll ausgestattete professionelle Programme aufbauen, ohne Stunden damit zuzubringen, eine grafische Benutzerschnittstelle vollständig von null an aufzubauen.

3. **Datenbanken zu erstellen und mit ihnen zu interagieren.** Die IDE schließt eine einfache Schnittstelle zum Aufbau von Datenbanken ein und arbeitet nahtlos mit SQL Server Express sowie einer Reihe weiterer beliebter Datenbanksysteme zusammen.

4. **Sich darauf zu konzentrieren, Ihre TATSÄCHLICHEN Probleme zu lösen.** Die IDE macht eine Menge für Sie, aber ***Sie*** steuern immer noch, was Sie mit C# aufbauen. Die IDE ermöglicht Ihnen einfach, sich auf Ihr Programm, Ihre Arbeit (oder Ihr Vergnügen) und Ihre Kunden zu konzentrieren. Sie selbst erledigt dagegen die gesamte Kernarbeit wie:

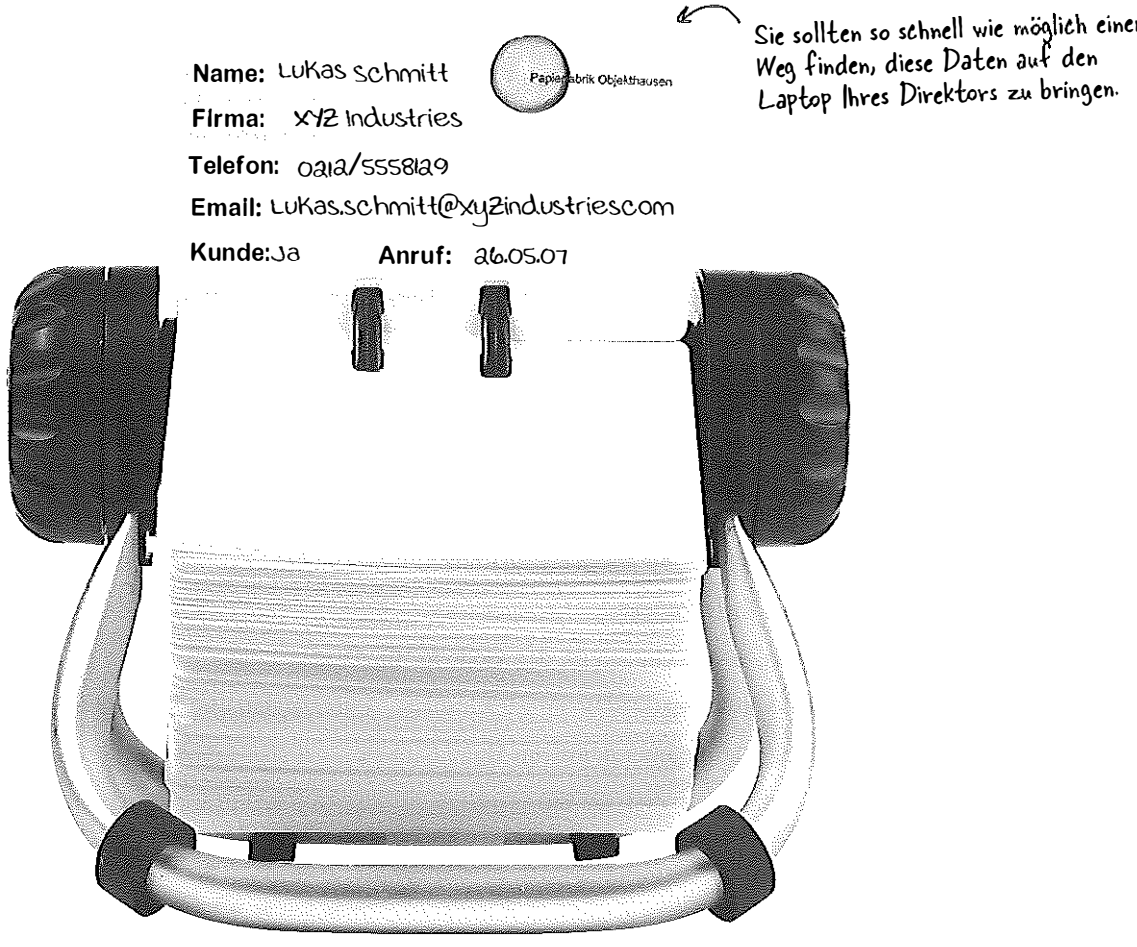
   - ★ das Nachhalten Ihrer Projekte,
   - ★ dafür zu sorgen, dass es einfach ist, den Code für Ihr Projekt zu bearbeiten,
   - ★ das Festhalten von Projektressourcen wie Grafiken, Audiodateien und Symbolen,
   - ★ die Verwaltung und die Interaktion mit Datenbanken.

   All das heißt, dass Sie die ganze Zeit, die Sie ansonsten mit diesen Routineaufgaben verbringen müssten, in die **Entwicklung von Killeranwendungen** stecken können.

In Kürze werden Sie sehen, was genau wir damit meinen.

Verhelfen Sie Ihrem Chef zu einem papierlosen Büro

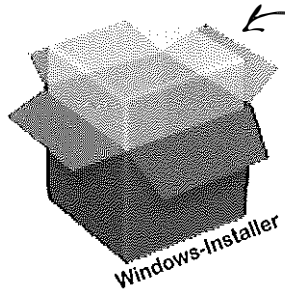
Die Papierfabrik Objekthausen hat gerade einen neuen Direktor eingestellt. Er liebt das Wandern, Kaffee und die Natur ... und er ist entschlossen, zur Rettung des Regenwalds beizutragen. Er will sein Büro papierlos machen. Den Anfang will er bei seinem Adressbuch machen. Er ist gerade auf dem Weg nach Davos, wo er das Wochenende beim Skifahren ausspannen will. Aber er erwartet, dass bei seiner Rückkehr ein neues Adressbuchprogramm fertig ist. Andernfalls ... sagen wir es so ... der alte Direktor wird bei der Jobsuche nicht mehr so einsam sein.



Bringen Sie die Bedürfnisse Ihres Kunden in Erfahrung, bevor Sie beginnen, Ihr Programm zu erstellen

Bevor wir anfangen können, die Adressbuchanwendung – oder *überhaupt irgendeine* Anwendung -- zu schreiben, müssen wir uns einen Augenblick Zeit nehmen und darüber nachdenken, **wer sie verwenden wird** und **was die Anwendung für den Nutzer machen muss**.

① Der Direktor muss das Adressbuchprogramm im Büro, aber auch auf seinem Laptop ausführen können. Er braucht ein Installationsprogramm, das sicherstellt, dass alle erforderlichen Dateien auf beide Rechner kommen.

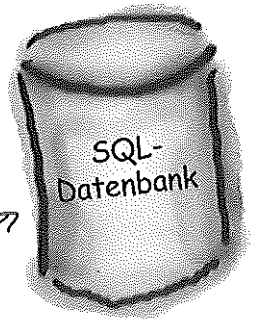


Der Direktor möchte das Programm an seinem Arbeitsplatz und seinem Laptop ausführen. Ein Installationsprogramm ist also unumgänglich.

② Der Vertrieb der Papierfabrik Objekthausen hätte ebenfalls gern Zugriff auf das Adressbuch des Direktors. Dort können seine Daten dann eingesetzt werden, um Rundschreiben zu versenden und Hinweise auf Kunden abzufragen, um den Papierverkauf anzukurbeln.

Nach Meinung des Direktors wäre eine Datenbank der beste Weg, um allen in der Firma Zugriff auf die Daten zu geben und dennoch nur eine Kopie aller Kontaktinformationen pflegen zu müssen.

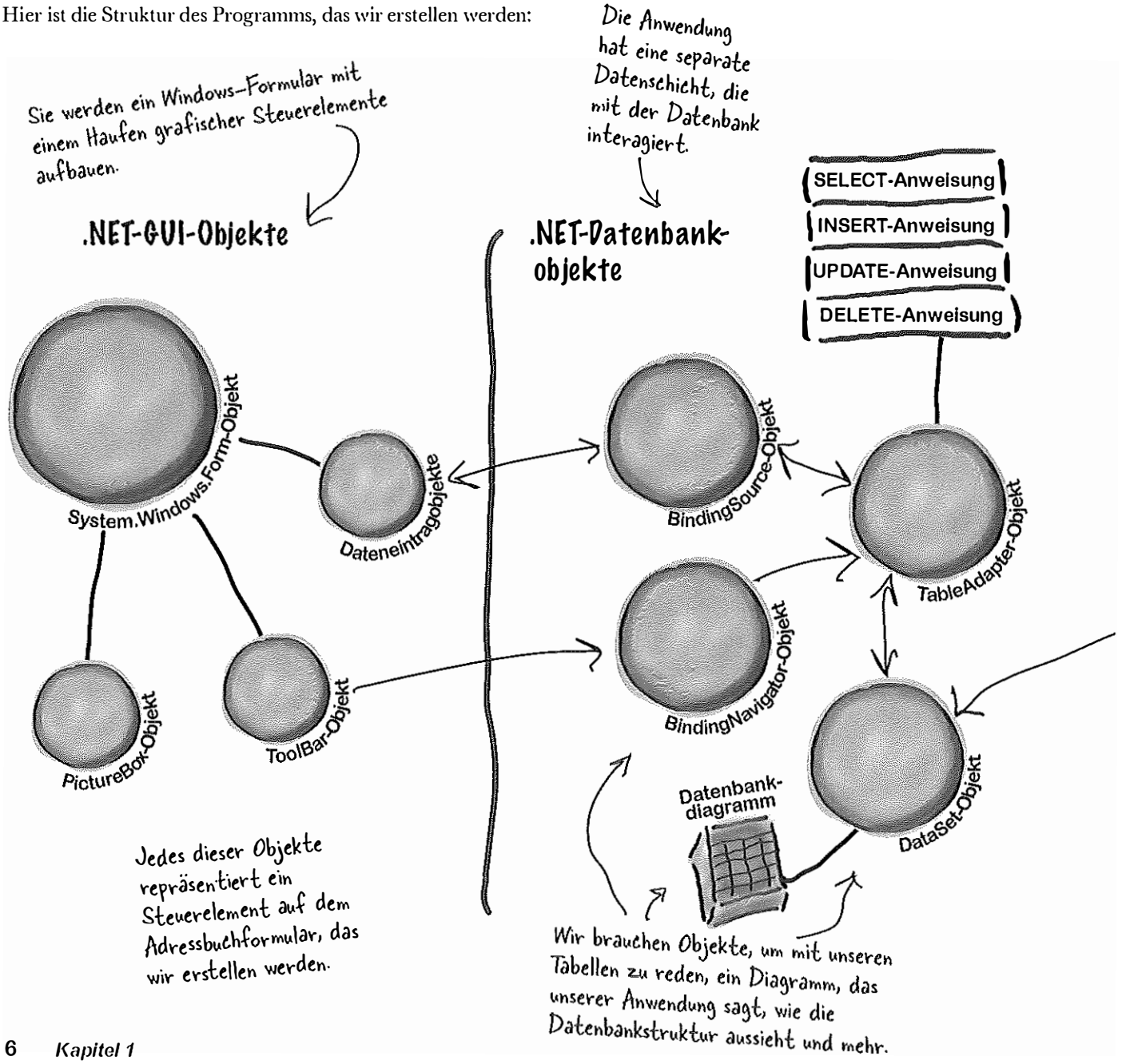
Wir wissen bereits, das Visual C# die Arbeit mit Datenbanken erleichtert. Sind die Kontakte in einer Datenbank, können der Direktor und das Vertriebsteam auf die Informationen zugreifen, obwohl es nur eine Kopie der Daten gibt.



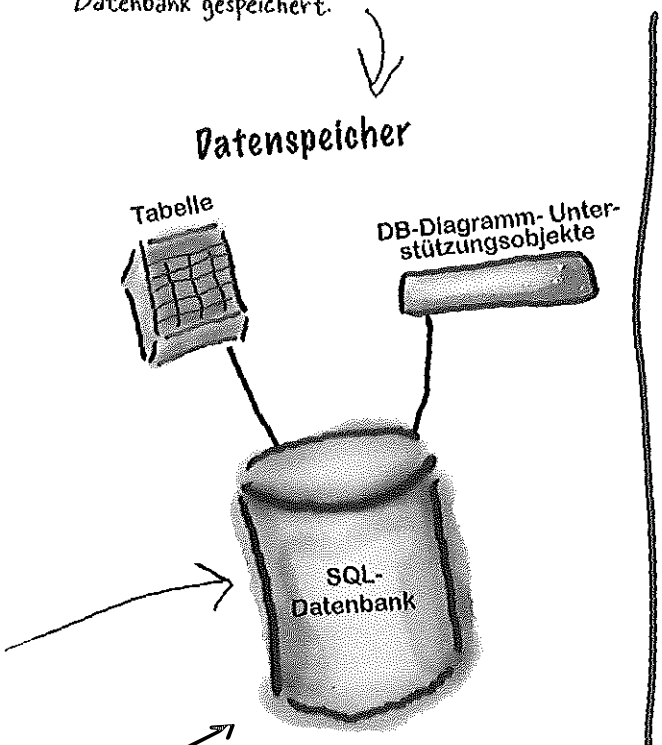
Das werden Sie aufbauen

Sie brauchen eine Anwendung mit einer grafischen Benutzerschnittstelle, Objekte, die mit einer Datenbank reden können, die Datenbank selbst und einen Installer. Das klingt nach einer Menge Arbeit, aber all das werden Sie auf den folgenden paar Seiten aufbauen.

Hier ist die Struktur des Programms, das wir erstellen werden:

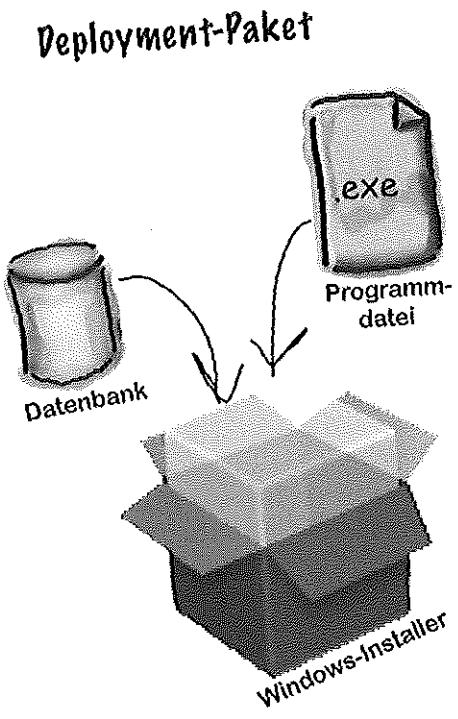


Alle Daten werden in einer Tabelle einer SQL Server Express-Datenbank gespeichert.



Hier ist die Datenbank selbst, die zu erstellen und warten Visual Studio uns helfen wird.

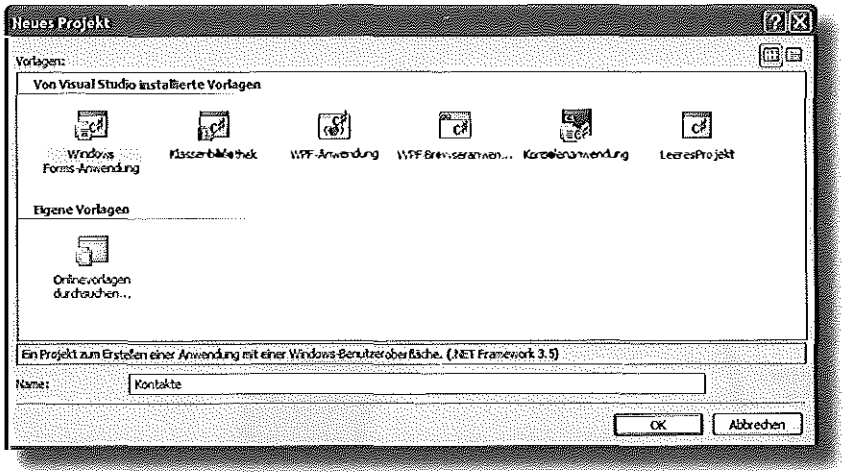
Ist das Programm erstellt, wird es zu einem Windows-Installer verpackt.



Die Vertriebsabteilung muss nur auf das Paket klicken, um das Programm zu installieren und dann zu verwenden.

Was Sie in Visual Studio tun ...

Legen Sie los und starten Sie Visual Studio, wenn Sie das nicht schon gemacht haben. Überspringen Sie die Startseite und wählen Sie im Menü **Datei** den Eintrag Neues Projekt. Geben Sie Ihrem Projekt den Namen »Kontakte« und klicken Sie auf OK.





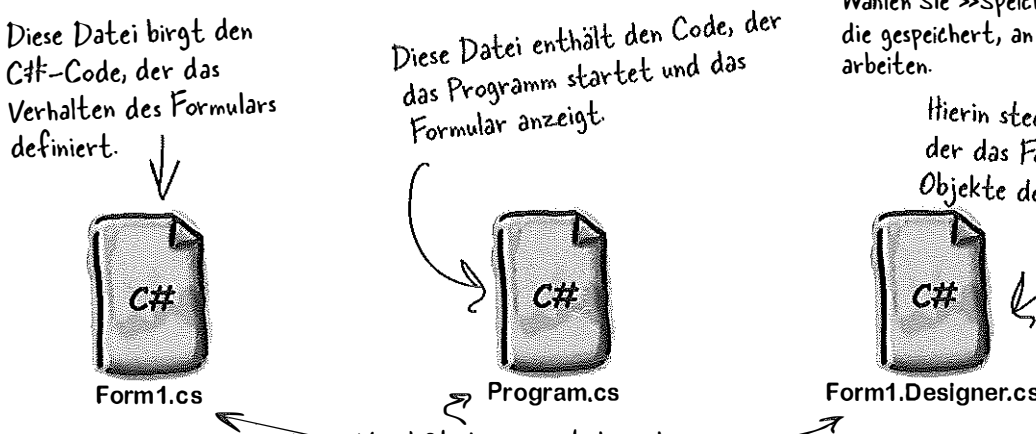
Bei Ihrer IDE kann das etwas anders aussehen.

So sieht das Fenster »Neues Projekt« in Visual Studio 2008 Express Edition aus. Arbeiten Sie mit der Professional- oder Team Foundation-Version, kann das etwas anders sein. Aber machen Sie sich keine Gedanken, alles funktioniert gleichermaßen.

Was Visual Studio für Sie macht ...

Wenn Sie ein neues Projekt erstellen, erzeugt die IDE die Dateien Form1.cs, Form1.Designer.cs und Program.cs. Diese fügt sie dem Projektmappen-Explorer-Fenster hinzu. Gespeichert werden sie standardmäßig in `Eigene Dateien\Visual Studio 2008\Projects\Kontakte\`.

Achten Sie darauf, dass Sie Ihr Projekt unmittelbar nach der Erstellung speichern, indem Sie im Datei-Menü »Alles Speichern« wählen – das speichert alle Projektdateien im Ordner. Wählen Sie »Speichern«, wird nur die gespeichert, an der Sie gerade arbeiten.



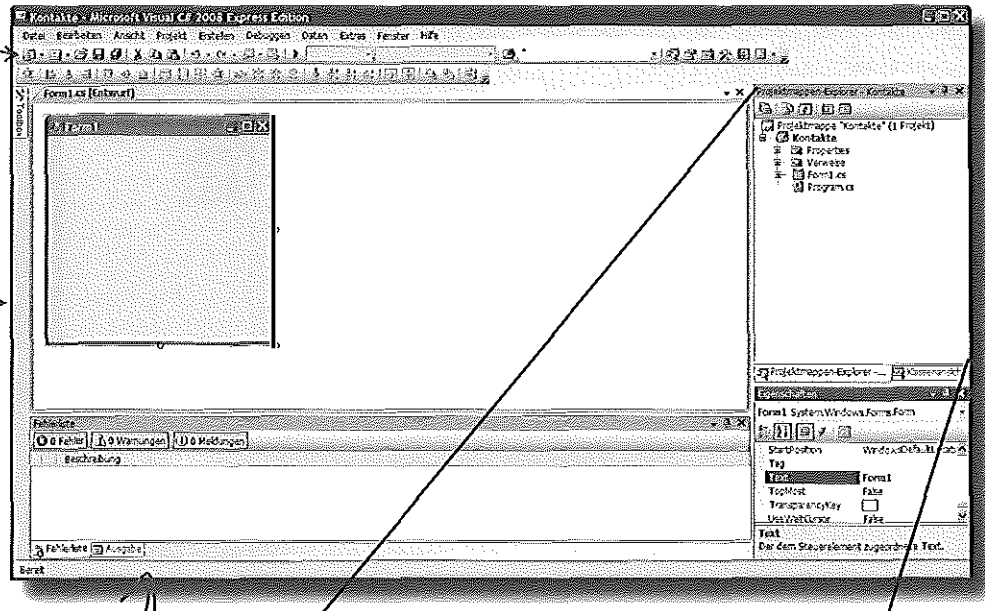
Spitzen Sie Ihren Bleistift

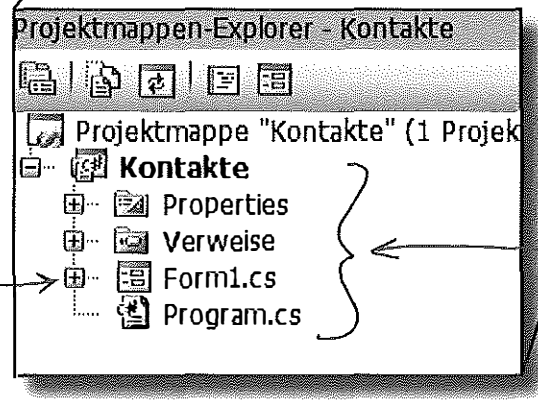
Unten sehen Sie, wie Ihr Bildschirm jetzt wahrscheinlich aussieht. Auf Basis dessen, was Sie bereits wissen, sollten Sie erkennen können, was die meisten dieser Fenster und Dateien bedeuten. Versuchen Sie, in die Zellen unten jeweils eine Anmerkung einzutragen, die beschreibt, was der entsprechende Teil der IDE macht. Einmal haben wir dies erledigt, um Ihnen auf die Sprünge zu helfen.

Diese Symbolleiste enthält Schaltflächen für die Aufgabe, die Sie gerade in der IDE erledigen.

Sollte Ihre IDE diesem Bild nicht genau entsprechen, können Sie im Fenster-/Menü >>Fensterlayout zurücksetzen<< wählen.

Wir haben das Fenster aufgeblasen, damit Sie mehr Platz haben.







Spitzen Sie Ihren Bleistift
Lösung

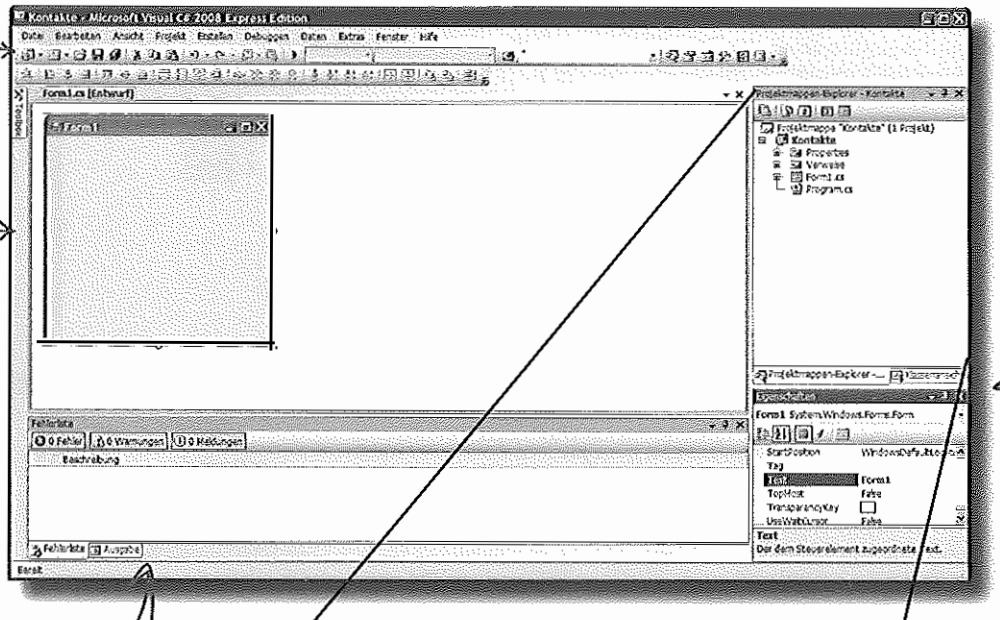
Wir haben unsere Anmerkungen zu den einzelnen Bereichen der Visual Studio C# IDE eingetragen. Sie haben eventuell abweichende Dinge notiert, sollten aber im Wesentlichen herausgefunden haben, wozu die einzelnen Fenster und Bereiche der IDE verwendet werden.

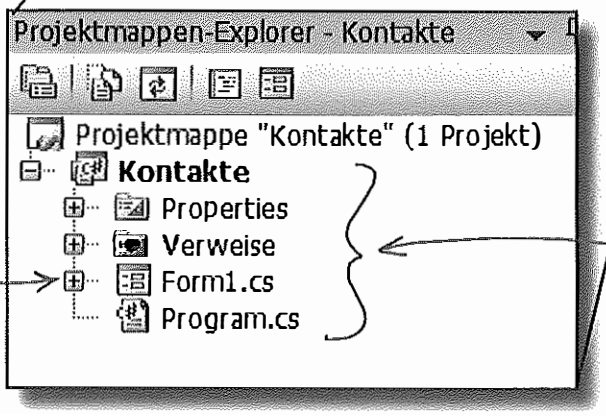
Diese Symbolleiste enthält Schaltflächen für die Aufgabe, die Sie gerade in der IDE erledigen.

Das ist die Toolbox. Sie enthält einen Haufen grafischer Steuerelemente, die Sie auf Ihr Formular ziehen können.

Der untere Bereich ist für das Debuggen zuständig. Er zeigt Ihnen, wenn Ihr Code Fehler hat.

Dieses Fenster zeigt Ihnen alle Eigenschaften der Steuerelemente in Ihrem Formular.





Die Dateien Form1.cs und Program.cs, die die IDE für Sie erstellt hat, als Sie das neue Projekt hinzugefügt haben, erscheinen im Projektmappen-Explorer.

Über den Projektmappen-Explorer in der IDE können Sie zwischen den Dateien wechseln.

Es gibt keine Dummen Fragen

F: Wenn die IDE diesen ganzen Code für mich schreibt, heißt C# lernen also zu lernen, wie man die IDE verwendet?

A: Nein. Die IDE ist sehr gut, wenn es darum geht, bestimmten Code für Sie zu generieren, aber ihre Reichweite ist beschränkt. Sie ist gut, wenn es darum geht, eine gute Ausgangsbasis für Sie zu schaffen oder um automatisch Eigenschaften der Steuerelemente in Ihrem Formular zu ändern. Aber die schweren Dinge beim Programmieren – herauszufinden, was Ihr Programm tun muss, und es dazu zu bringen, genau das auch zu tun –, das ist etwas, das Ihnen keine IDE abnehmen kann. Obgleich die Visual Studio IDE eine der fortschrittlichsten Entwicklungsumgebungen ist, kann sie nicht alles. ***Sie*** – nicht die IDE – schreiben den Code, der die eigentliche Arbeit erledigt.

F: Ich habe ein neues Projekt in Visual Studio angelegt, aber als ich zum Ordner »Projekte« unter Meine Dateien gegangen bin, habe ich es dort nicht gefunden. Woran liegt das?

A: Das heißt, dass Sie auf alle Fälle Visual Studio 2008 verwenden – mit 2005 passiert das nicht. Wenn Sie in Visual Studio 2008 ein neues Projekt erstellen, erstellt die IDE es zunächst im Ordner `Lokale Einstellungen\Anwendungsdaten\Temporary Projects`. Speichern Sie das Projekt dann das erste Mal, werden Sie zur Eingabe eines neuen Dateinamens aufgefordert, und als Speicherort wird der Ordner `Meine Dateien\Visual Studio 2008\Projects` vorgeschlagen. Versuchen Sie, ein neues Projekt zu öffnen oder das temporäre zu schließen, werden Sie aufgefordert, dass temporäre Projekt entweder zu speichern oder zu verwerfen.

F: Was ist, wenn die IDE Code erstellt, den ich in meinem Projekt nicht haben möchte?

A: Sie können ihn ändern. Die IDE ist so eingerichtet, dass sie Code auf Basis der Weise erzeugt, auf die das Element, das Sie gezogen oder hinzugefügt haben, am häufigsten verwendet wird. Aber es kann passieren, dass das nicht das ist, was Sie sich vorgestellt haben. Alles, was die IDE für Sie macht – jede Zeile Code, die sie für Sie erstellt, jede Datei, die sie hinzufügt –, kann geändert werden – entweder manuell, indem Sie die Dateien direkt bearbeiten, oder über eine leicht zu verwendende Schnittstelle der IDE.

F: Ist es in Ordnung, dass ich Visual Studio Express heruntergeladen und installiert habe? Oder brauche ich für bestimmte Dinge in diesem Buch eine der kostenpflichtigen Versionen von Visual Studio?

A: In diesem Buch gibt es nichts, dass Sie mit der kostenlosen Version von Visual Studio (die Sie von der Microsoft-Website herunterladen können) nicht machen können. Die Unterschiede zwischen der Express-Version und den anderen Versionen (Professional und Team Foundation) werden Ihnen beim Schreiben von C#-Code und dem Aufbau vollwertiger funktionierender Anwendungen nicht im Weg stehen.

F: Kann ich die Namen der Dateien ändern, die die IDE für mich generiert?

A: Natürlich. Wenn Sie ein neues Projekt erstellen, gibt die IDE Ihnen ein Standard-Formular mit dem Namen Form1 (für das es die Dateien Form1.cs, Form1.Designer.cs und Form1.resx gibt). Die Namen dieser Dateien können Sie über den Projektmappenexplorer beliebig ändern. Standardmäßig haben diese Dateien den gleichen Namen wie das Formular. Im Eigenschaften-Fenster können Sie sehen, dass das Formular selbst aber weiterhin den Namen Form1 behält, wenn Sie die Namen der Dateien ändern. Den Namen des Formulars können Sie ändern, indem Sie die "(Name)"-Zeile im Fenster Eigenschaften ändern. Die Änderung des Formularnamens hat ihrerseits keine Auswirkungen auf die Namen der Dateien.

C# ist ganz egal, wie Sie Dateien oder Formulare (oder andere Teile des Programms) nennen. Aber wenn Sie aussagekräftige Namen wählen, kann das die Arbeit mit Ihren Programmen erleichtern. Noch kümmern wir uns um die Namen nicht – wie man aussagekräftige Namen für Teile von Programmen wählt, werden wir später noch viel ausführlicher besprechen.

F: Ich habe die IDE gerade vor mir, aber mein Bildschirm sieht ganz anders aus als Ihrer! Darin fehlen einige der Fenster, und andere sind am falschen Platz. Woran liegt das?

A: Wenn Sie im Menü »Fenster« den Befehl »Fensterlayout zurücksetzen« wählen, stellt die IDE für Sie das Standardlayout wieder her. Dann sieht Ihr Bildschirm genauso aus wie die in diesem Kapitel.

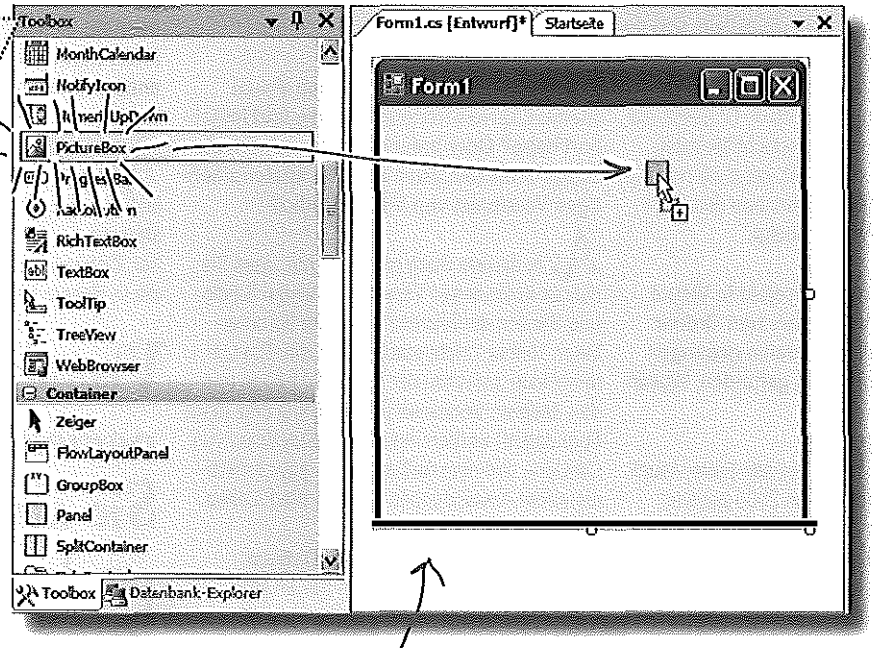
Visual Studio generiert Code, den Sie als Ausgangspunkt für Ihre Anwendungen einsetzen können. Dass die Anwendung tut, was sie tun soll, bleibt weiterhin Ihre Aufgabe.

Die Benutzerschnittstelle entwickeln

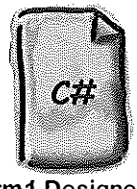
Das Hinzufügen von Steuerelementen und die Gestaltung der Benutzerschnittstelle lässt sich in der Visual Studio IDE mit Drag-and-Drop erledigen. Fügen wir dem Formular also ein Logo hinzu.

1 **Nutzen Sie das PictureBox-Steuerelement, um ein Bild hinzuzufügen.** Klicken Sie auf das PictureBox-Steuerelement in der Toolbox und ziehen Sie es auf Ihr Formular. Im Hintergrund fügt die IDE `Form1.Designer.cs` dann Code für ein neues PictureBox-Steuerelement hinzu.

Sollten Sie die Toolbox nicht sehen, probieren Sie, den Mauszeiger über dem Wort »Toolbox« schweben zu lassen, das oben links in der IDE zu sehen ist. Sehen Sie es dort nicht, wählen Sie im Menü Ansicht »Toolbox«, damit es erscheint.



Jedes Mal, wenn Sie eine Änderung an den Eigenschaften eines Steuerelements im Formular vornehmen, wird der Code in Form1.Designer.cs von der IDE geändert.

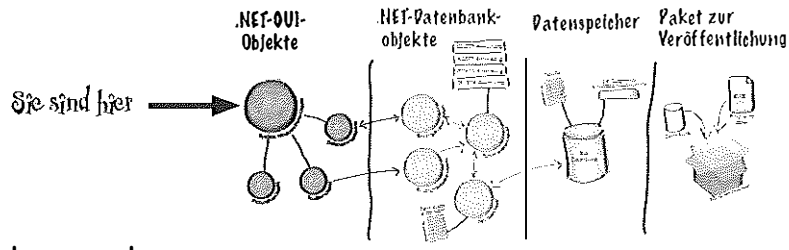


Form1.Designer.cs

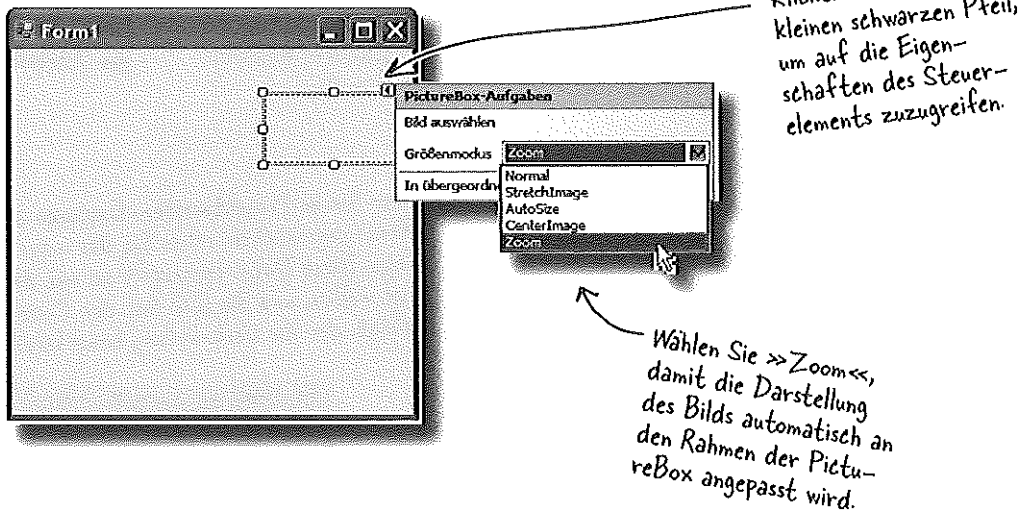


Es ist kein Problem, wenn Sie kein Profi in der Benutzerschnittstellengestaltung sind.

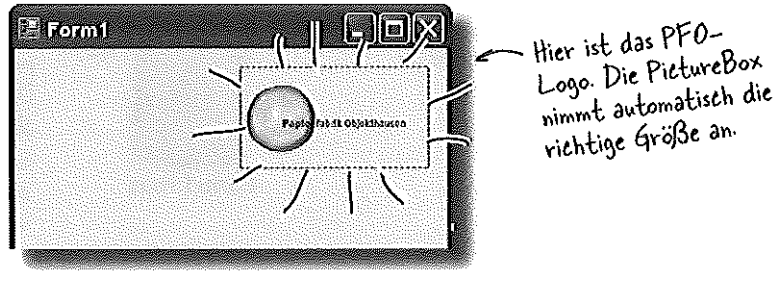
Wir werden später noch eine Menge zur Gestaltung guter Benutzerschnittstellen sagen. Im Augenblick sollten Sie sich nur darauf konzentrieren, das Logo und die anderen Steuerelemente in Ihr Formular zu bekommen, und sich Gedanken über ***Verhalten*** machen. Etwas Style werden wir später hinzufügen.



2 **Die PictureBox in den Zoom-Modus versetzen.**
Jedes Steuerelement in Ihrem Formular hat Eigenschaften, die Sie setzen können. Klicken Sie auf den kleinen schwarzen Pfeil für das Steuerelement, um auf diese Eigenschaften zuzugreifen. Ändern Sie die Größenmodus-Eigenschaft der PictureBox in »Zoom«, damit Sie sehen, wie das funktioniert:

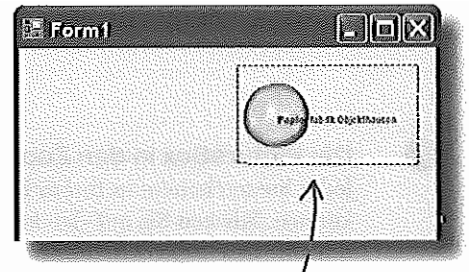


3 **Laden Sie das Logo der Papierfabrik Objekthausen herunter.**
Holen Sie sich das Logo der Papierfabrik Objekthausen von der O'Reilly-Website (**`http://examples.oreilly.de/german_examples/hfcsharpger`**), und speichern Sie es auf Ihrer Festplatte. Klicken Sie dann auf den Eigenschaftenpfeil der PictureBox und wählen Sie »Image«. Klicken Sie auf »Import«, suchen Sie nach dem Logo, und schon sind Sie so weit:



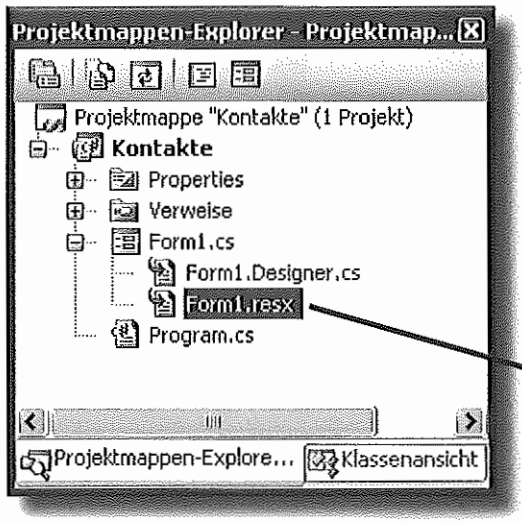
Visual Studio hinter den Kulissen

Jedes Mal, wenn Sie etwas in der Visual Studio IDE tun, ***schreibt*** die IDE ***Code für Sie***. Als Sie das Logo erstellt und Visual Studio gesagt haben, dass es das heruntergeladene Bild verwenden soll, hat Visual Studio eine Ressource erstellt und diese mit Ihrer Anwendung verknüpft. Eine **Ressource** ist eine Grafikdatei, eine Audiodatei, ein Icon oder eine andere Art von Datendatei, die mit Ihrer Anwendung gebündelt wird. Die Grafikdatei wird in das Programm integriert, damit sie, wenn das Programm auf einem anderen Rechner installiert wird, mitinstalliert wird und von der PictureBox verwendet werden kann.

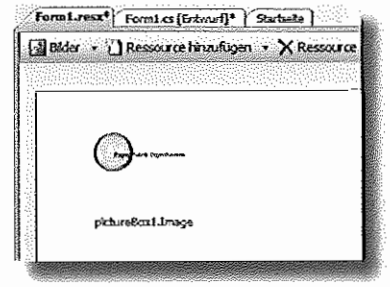


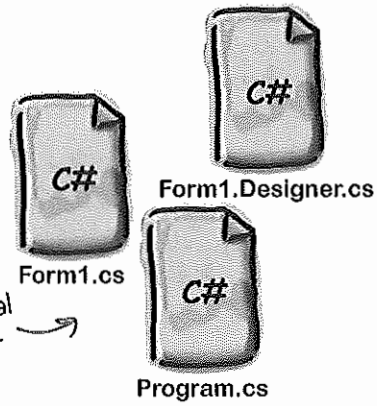
Dieses Bild ist jetzt eine Ressource der Adressbuchanwendung.

Als Sie das PictureBox-Steuerelement auf Ihr Formular gezogen haben, hat die IDE automatisch für Sie eine Ressourcendatei namens Form1.resx erstellt, um die Ressource zu speichern und im Projekt festzuhalten. Klicken Sie doppelt auf diese Datei, sehen Sie die frisch importierte Grafik.

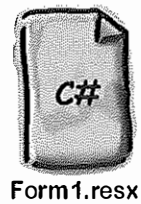


Wenn Sie im Projektmappen-Explorer auf Form1.resx klicken, sehen Sie das importierte Logo. Diese Datei verknüpft sie mit der PictureBox, und die IDE hat den Code hinzugefügt, der die Verknüpfung bewerkstelligt.





Hier sind die Dateien, die Visual Studio schon erstellt hatte.

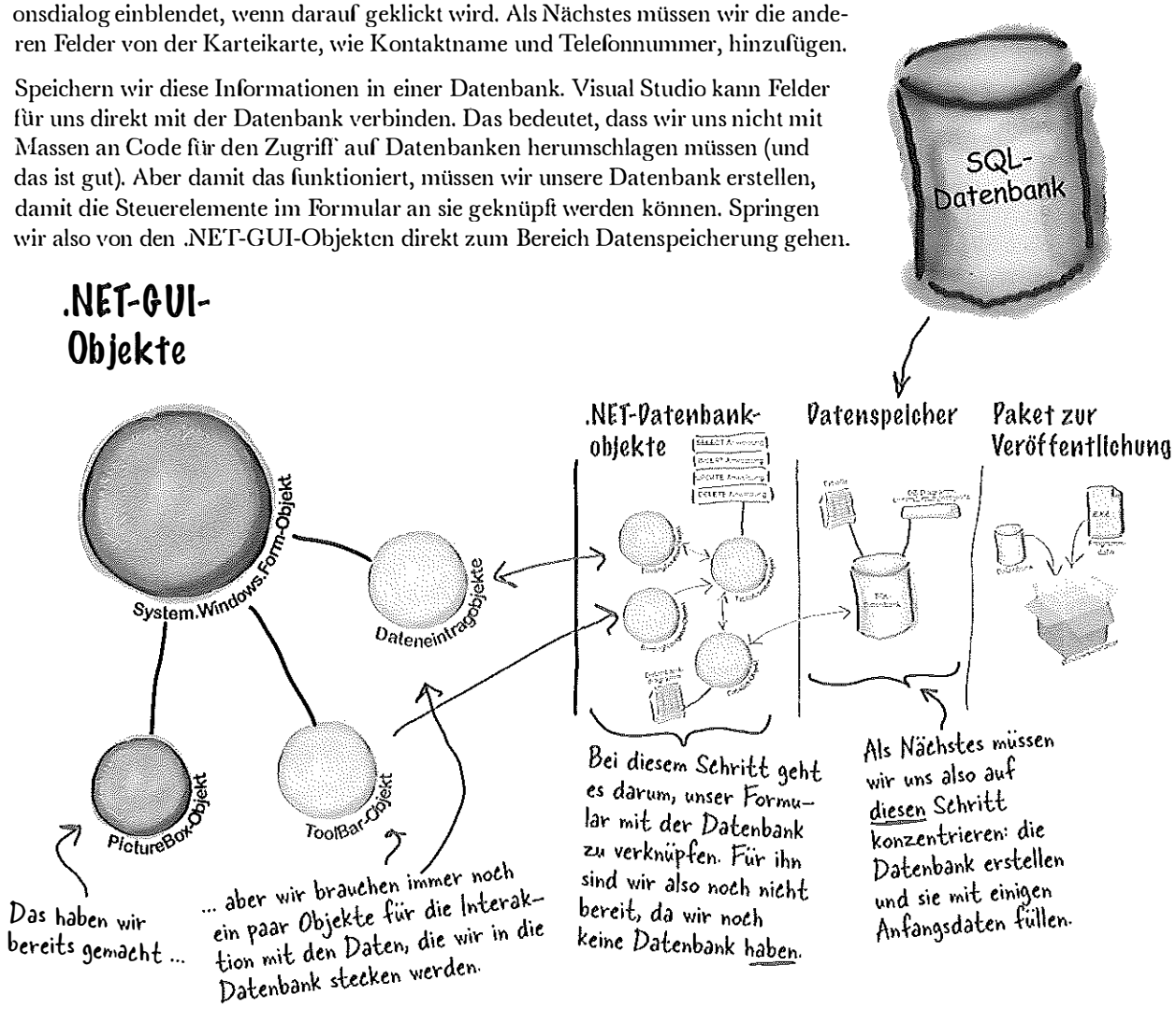


Dadurch dass Sie das Bild importiert haben, hat die IDE diese Datei für Sie erstellt. Sie enthält alle Ressourcen (Bilder, Videos, Sounddateien und andere gespeicherte Daten), die mit Form1 verknüpft sind.

Das haben wir bisher gemacht

Wir haben ein Formular und ein PictureBox-Objekt erstellt, das einen Informationsdialog einblendet, wenn darauf geklickt wird. Als Nächstes müssen wir die anderen Felder von der Karteikarte, wie Kontaktname und Telefonnummer, hinzufügen.

Speichern wir diese Informationen in einer Datenbank. Visual Studio kann Felder für uns direkt mit der Datenbank verbinden. Das bedeutet, dass wir uns nicht mit Massen an Code für den Zugriff auf Datenbanken herumschlagen müssen (und das ist gut). Aber damit das funktioniert, müssen wir unsere Datenbank erstellen, damit die Steuerelemente im Formular an sie geknüpft werden können. Springen wir also von den .NET-GUI-Objekten direkt zum Bereich Datenspeicherung gehen.



Visual Studio kann Code erstellen, der Ihr Formular mit einer Datenbank verknüpft, aber Sie müssen ZUVOR die Datenbank eingerichtet haben, damit dieser Code generiert wird.

Wir brauchen eine Datenbank zur Speicherung der Informationen

Bevor wir dem Formular die restlichen Felder hinzufügen, müssen wir eine Datenbank erstellen, an die wir das Formular binden. Die IDE kann eine Menge Code für die Verbindung von Formular und Datenbank erstellen, aber erst müssen wir die Datenbank selbst definieren.

Bevor Sie weitermachen, sollten Sie prüfen, ob Sie das Debuggen beendet haben.

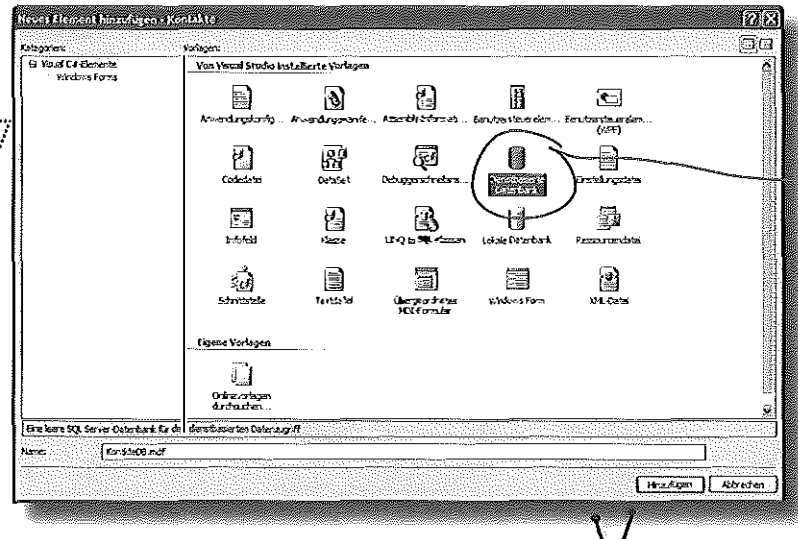
1 **Fügen Sie Ihrem Projekt eine SQL-Datenbank hinzu.** **Klicken Sie** im Projektmappen-Explorer **rechts auf das Projekt Kontakte**, wählen Sie Hinzufügen und dann Neues Element. Wählen Sie das Symbol Dienstbasierte Datenbank und geben Sie ihr den Namen KontakteDB.mdf.

Diese Datei ist unsere neue Datenbank.



KontakteDB.mdf

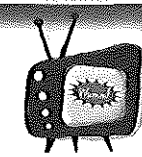
Wählen Sie das richtige Symbol für die Version, die Sie nutzen. Wählen Sie **SQL-Datenbank**, falls Sie Visual Studio Express 2005 nutzen, und **Dienstbasierte Datenbank**, sollten Sie 2008 nutzen.



Das Symbol für die SQL-Datenbank funktioniert nur, wenn Sie SQL Server Express installiert haben. Gehen Sie zum README-Abschnitt der Einleitung, wenn Sie nicht wissen, wie das geht.

2 **Brechen Sie den »Assistent zum Konfigurieren von Datenquellen« ab.**
Die Konfiguration einer Datenquelle wollen wir im Augenblick noch überspringen, klicken Sie also auf den Button Abbrechen. Dazu werden wir später zurückkehren, nachdem wir unsere Datenbankstruktur eingerichtet haben.

3 **Betrachten Sie Ihre Datenbank im Projektmappen-Explorer.**
Gehen Sie zum Projektmappen-Explorer, werden Sie sehen, dass der Dateiliste KontakteDB hinzugefügt wurde. Doppelklicken Sie auf KontakteDB.mdf im Projektmappen-Explorer und schauen Sie auf die linke Seite des Bildschirms. Die Toolbox hat sich in den Datenbank-Explorer verwandelt.

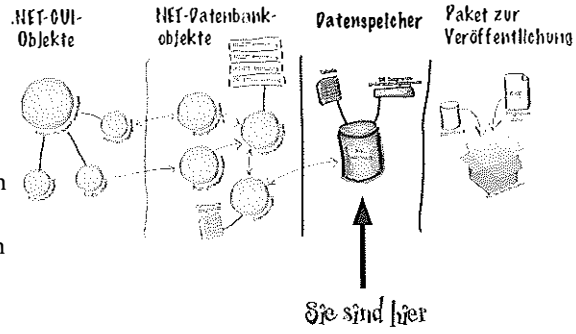


Aufgepasst

Sollten Sie nicht die Express-Version verwenden, sehen Sie den »Server-Explorer« anstelle des »Datenbank-Explorers«.

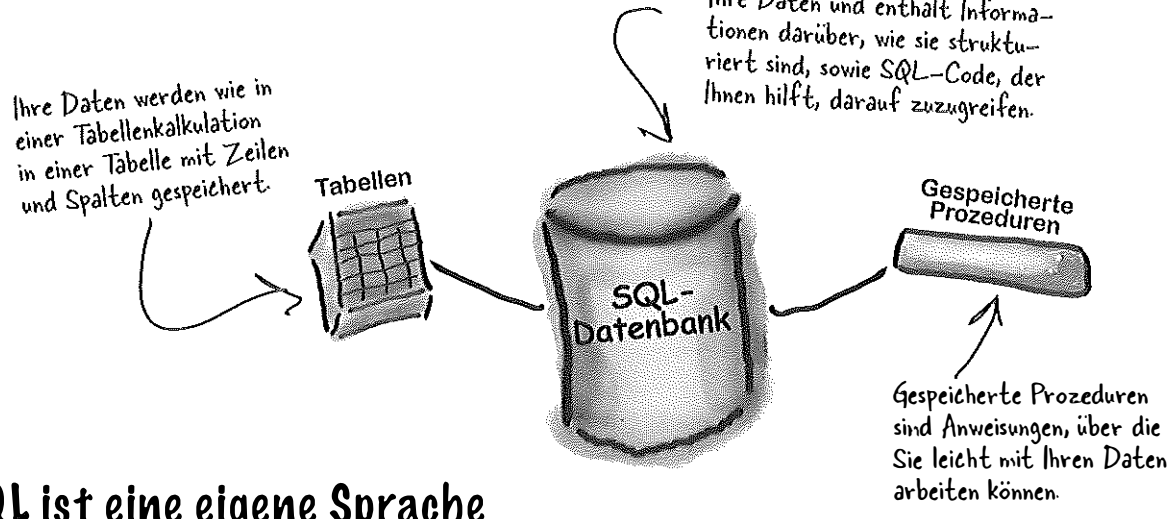
Die Versionen Visual Studio 2008 Professional und Team Foundation haben kein Datenbank-Explorer-Fenster. Stattdessen haben sie ein Server-Explorer-Fenster, das alles macht, was der Datenbank-Explorer macht, Ihnen aber auch ermöglicht, Daten in Ihrem Netzwerk zu untersuchen.

Die IDE hat eine Datenbank erstellt



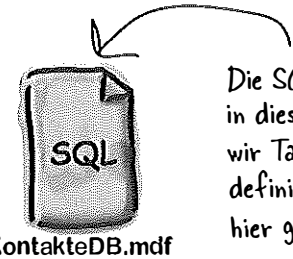
Als Sie der IDE gesagt haben, dass sie Ihrem Projekt eine neue SQL-Datenbank hinzufügen soll, hat sie diese neue Datenbank für Sie erstellt. Eine **SQL-Datenbank** ist ein System, das für Sie Daten organisiert und aufeinander bezogen speichert. Die IDE gibt Ihnen alle Werkzeuge, die Sie benötigen, um Ihre Daten und Datenbanken zu verwalten.

Die Daten existieren in einer SQL-Datenbank in Tabellen. Für den Augenblick können Sie sich diese Tabellen wie die in einer Tabellenkalkulation vorstellen. Die Spalten sind die Datenkategorien, wie der Name und die Telefonnummer eines Kontakts, und die Zeilen stehen jeweils für die Daten auf einer Kontaktkarte.



SQL ist eine eigene Sprache

SQL steht für **Structured Query Language**. Das ist eine Programmiersprache für den Zugriff auf die Daten in einer Datenbank. Sie hat eine eigene Syntax, eigene Schlüsselwörter und eine eigene Struktur. SQL-Code gibt es in Form von **Anweisungen** und **Abfragen**, über die auf Daten zugegriffen und Daten abgerufen werden. Eine SQL-Datenbank kann **gespeicherte Prozeduren** enthalten. Das sind Sammlungen von SQL-Anweisungen und -Abfragen, die Sie in der Datenbank gespeichert haben und jederzeit ausführen können. Die IDE generiert SQL-Anweisungen und gespeicherte Prozeduren automatisch für Sie, damit Ihr Programm auf die Daten in der Datenbank zugreifen kann.



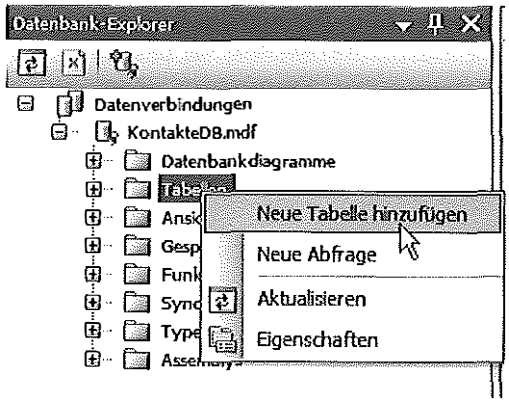
Die SQL-Datenbank steckt in dieser Datei. Gleich werden wir Tabellen und Daten dafür definieren, und all das wird auch hier gespeichert.

[Notiz aus dem Marketing: Können wir hier nicht einen Verweis auf »SQL von Kopf bis Fuß« einbauen?]

Die Tabelle für die Kontaktliste erstellen

Wir haben eine Datenbank und müssen jetzt Informationen darin speichern. Aber unsere Informationen müssen eigentlich in eine Tabelle. Das ist die Datenstruktur, die die Datenbank nutzt, um einzelne Datenteile festzuhalten. Erstellen wir für unsere Anwendung eine Tabelle namens »Personen« zur Speicherung der Kontaktinformationen:

1. **Der Datenbank KontakteDB eine Tabelle hinzufügen.** Klicken Sie im Datenbank-Explorer rechts auf Tabellen und wählen Sie Neue Tabelle hinzufügen. Dadurch öffnet sich ein Fenster, in dem Sie die Spalten für die gerade erstellte Tabelle definieren können.



Jetzt müssen wir in unserer Tabelle Spalten anlegen. Fügen wir der neuen Tabelle Personen zunächst eine Spalte namens KontaktID hinzu, damit jeder Kontakt-Datensatz eine eindeutige ID hat.

2. **Der Tabelle Personen die Spalte KontaktID hinzufügen.** Geben Sie in das Feld Spaltenname »KontaktID« ein und wählen Sie im Aufklappmenü Datentyp den Typ int aus. Achten Sie darauf, dass Sie das Auswahlfeld NULL zulassen abwählen.

   Dann machen wir diese Spalte zum Primärschlüssel unserer Tabelle. Markieren Sie die gerade erstellte Spalte KontaktID und klicken Sie auf den Button Primärschlüssel festlegen. Das sagt der Datenbank, dass jeder Datensatz einen eigenen eindeutigen Primärschlüsselwert hat.



Das ist der Primärschlüssel-Button. Ein Primärschlüssel hilft Ihrer Datenbank, Datensätze schnell zu finden.

Es gibt keine Dummen Fragen

F: Was war noch einmal eine Spalte?

A: Eine Spalte ist ein Feld in einer Tabelle. In einer Tabelle Personen könnten Sie also Spalten wie Vorname und Nachname haben. Außerdem hat eine Spalte immer einen Datentyp wie String, Date oder Bool.

F: Wozu brauchen wir die Spalte KontaktID?

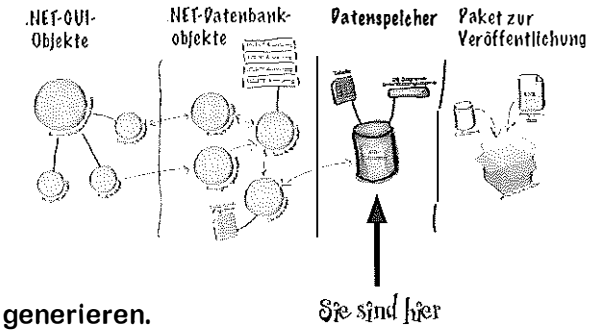
A: In den meisten Datenbanken ist es nützlich, wenn jeder Datensatz eine eindeutige ID hat. Da wir Kontaktinformationen für einzelne Personen speichern, haben wir uns entschieden, dafür eine Spalte zu erstellen, und haben sie KontaktID genannt.

F: Was bedeutet das Int im Feld Datentyp?

A: Der Datentyp sagt der Datenbank, was für Daten in einer Spalte zu erwarten sind. int steht für Integer, d.h. ganze Zahlen. Die Spalte KontaktID enthält also ganze Zahlen.

F: Das ist eine Menge Zeug. Muss ich das alles verstehen?

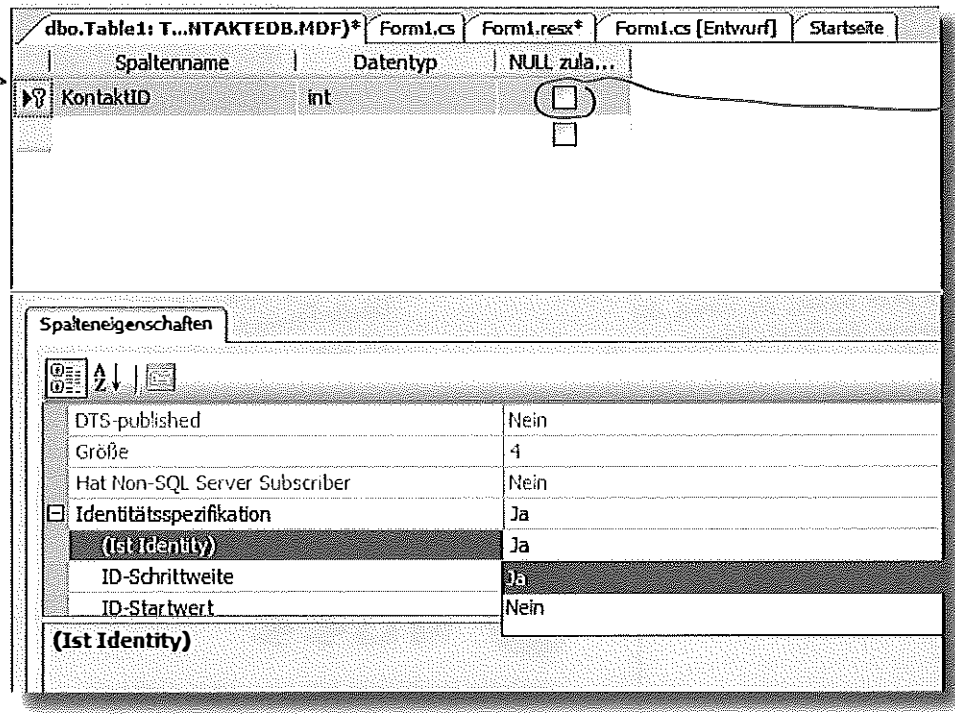
A: Nein. Im Augenblick müssen Sie noch nicht alles verstehen. Konzentrieren Sie sich auf die grundlegenden Schritte. In späteren Kapiteln des Buchs werden wir noch eine Menge Zeit mit Datenbanken verbringen. Und wenn Sie unbedingt sofort mehr erfahren müssen, können Sie sich natürlich ***SQL von Kopf bis Fuß*** zulegen und es parallel zu diesem Buch lesen.



3 **Die Datenbank anweisen, IDs automatisch zu generieren.**
Da KontaktID eine Zahl für die Datenbank und nicht unsere Nutzer ist, können wir die Datenbank anweisen, sich darum zu kümmern, dass die IDs automatisch erzeugt und zugewiesen werden. So müssen wir uns nicht darum kümmern, den Code dafür zu schreiben.

Scrollen Sie in den Eigenschaften unter Ihrer Tabelle zum Eintrag Identitätsspezifikation, klicken Sie auf den Button + und wählen Sie Ja neben der Eigenschaft (Ist Identity).

Dieses Fenster nutzen Sie, um Ihre Tabelle und die Daten zu definieren, die sie speichern wird.

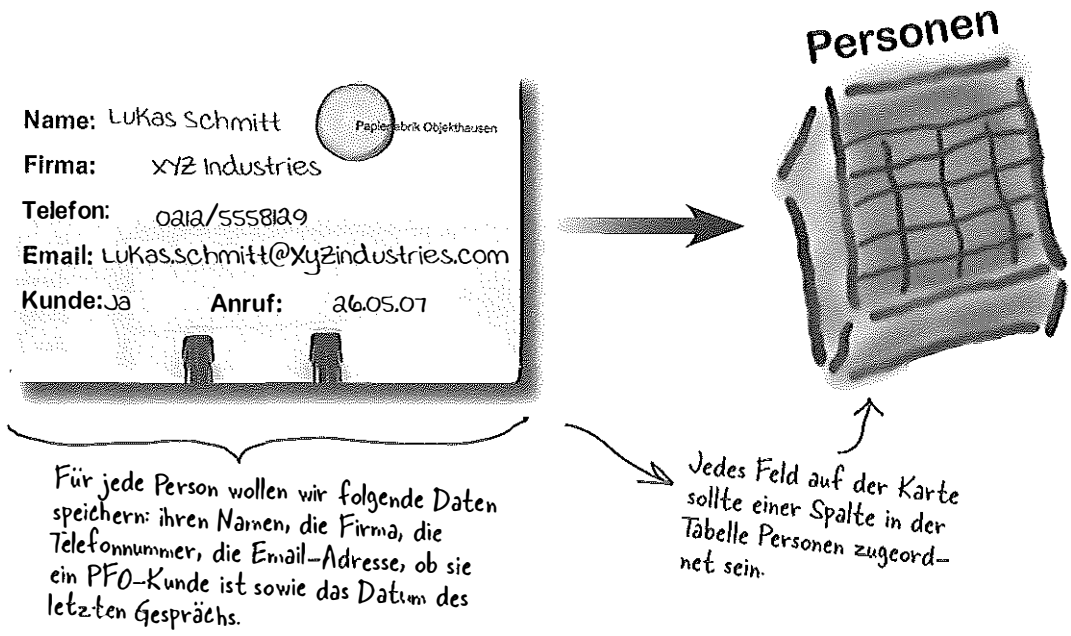


Es ist wichtig, dass dieser Punkt nicht ausgewählt ist. Da der Primärschlüssel das wichtigste Mittel ist, über das Ihr Programm Datensätze sucht, muss die Spalte immer einen Wert enthalten.

Das sorgt dafür, dass das Feld KontaktID automatisch aktualisiert wird, wenn ein neuer Datensatz hinzukommt.

Die Felder auf einer Adresskarte sind die Spalten in der Tabelle Personen

Nachdem Sie den Primärschlüssel für die Tabelle erstellt haben, müssen Sie alle Felder definieren, die Sie in der Datenbank festhalten möchten. Jedes Feld auf unserer handgeschriebenen Adresskarte sollte zu einer Spalte in der Tabelle Personen werden.





KOPF-NUSS

Welche Probleme könnten sich ergeben, wenn mehrere Zeilen für die gleiche Person gespeichert werden?

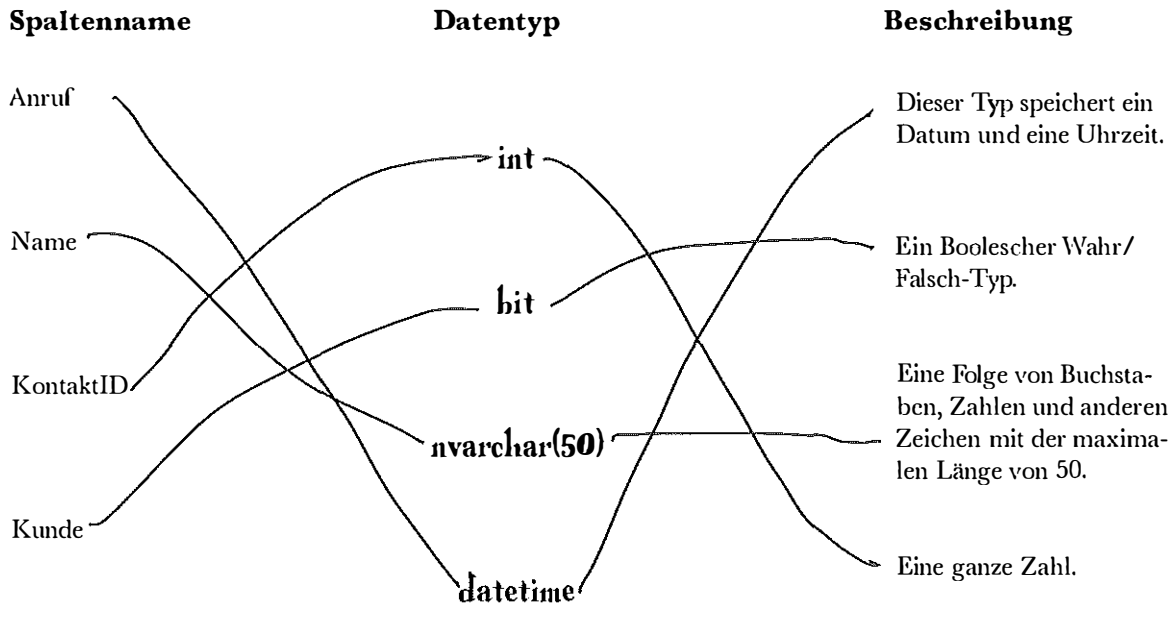


Nachdem Sie die Tabelle Personen und eine Primärschlüsselspalte erstellt haben, müssen Sie die Spalten für all diese Datenfelder hinzufügen. Schauen Sie, ob Sie herausfinden, welcher Datentyp zu welcher Tabellenspalte gehört, und ordnen Sie außerdem den einzelnen Datentypen die passende Beschreibung zu.

Spaltenname	Datentyp	Beschreibung
Anruf		Dieser Typ speichert ein Datum und eine Uhrzeit.
	int	
Name		Ein Boolescher Wahr/Falsch-Typ.
	bit	
KontaktID		Eine Folge von Buchstaben, Zahlen und anderen Zeichen mit der maximalen Länge von 50.
	nvarchar(50)	
Kunde		Eine ganze Zahl.
	datetime	

WER MACHT WAS?

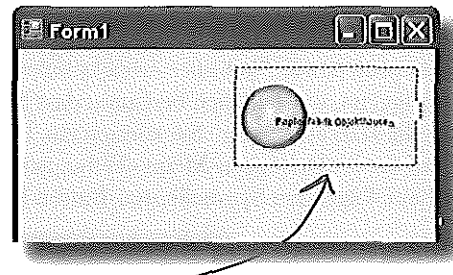
Nachdem Sie die Tabelle Personen und eine Primärschlüsselspalte erstellt haben, müssen Sie die Spalten für all diese Datenfelder hinzufügen. Schauen Sie, ob Sie herausfinden, welcher Datentyp zu welcher Tabellenspalte gehört, und ordnen Sie außerdem den einzelnen Datentypen die passende Beschreibung zu.



Den automatisch generierten Code ergänzen

Die IDE erstellt eine Menge Code für Sie, aber Sie müssen diesen Code trotzdem noch anpacken und ergänzen. Richten wir das Logo so ein, dass es eine Informationsmeldung anzeigt, wenn darauf geklickt wird.

Prüfen Sie, ob Ihr Formular in der IDE angezeigt wird, und klicken Sie doppelt auf das PictureBox-Steuerelement. Sie sollten dann Code sehen, der so ähnlich wie dieser aussieht:




```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
    private void pictureBox1_Click(object sender, EventArgs e)
    {
        MessageBox.Show("Adressbuch 1.0.\nGeschrieben von: Ihr Name", "Info");
    }
}
```

Als Sie doppelt auf das PictureBox-Steuerelement geklickt haben, hat die IDE diese Methode erstellt. Sie wird jedes Mal ausgeführt, wenn der Benutzer in der laufenden Anwendung auf das Logo klickt.

Dieser Methodenname verschafft Ihnen eine Vorstellung davon, wann sie ausgeführt wird: wenn nämlich jemand auf dieses PictureBox-Steuerelement klickt.

Klicken Sie doppelt auf die PictureBox, öffnet sie diesen Code mit einem blinkenden Cursor, der sich genau hier befindet. Ignorieren Sie Fenster, die die IDE eventuell öffnet, während Sie tippen. Sie versucht, Ihnen zu helfen, aber diese Hilfe brauchen wir im Augenblick noch nicht.

Geben Sie diese Codezeile ein. Sie bewirkt, dass ein Meldungsfenster mit dem angegebenen Text eingeblendet wird. Der Titel des Fensters lautet »Info«.

Nachdem Sie diese Codezeile eingegeben haben, sollten Sie den Code über das Speichern-Symbol in der IDE-Symbolleiste speichern oder indem Sie im Datei-Menü »Speichern« wählen. Machen Sie es sich zur Gewohnheit, regelmäßig »Alles speichern« einzusetzen!

Es gibt keine Dummen Fragen

F: Was ist eine Methode?

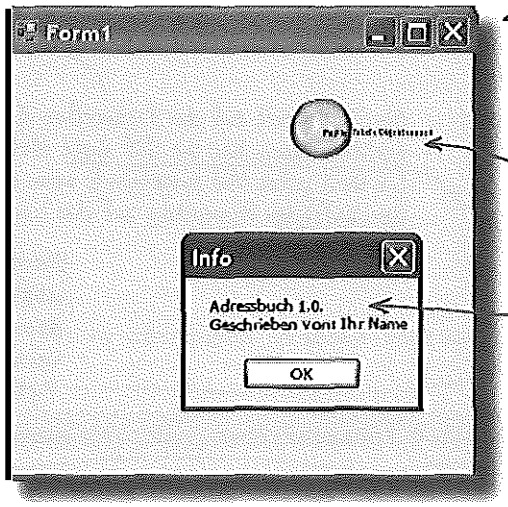
A: Eine Methode ist bloß ein *benannter Codeblock*. Mit Methoden werden wir uns in Kapitel 2 ausführlicher befassen.

F: Was macht das \n-Ding?

A: Das ist ein Zeilenumbruch. Er sagt C#, dass es »Adressbuch 1.0.« auf eine Zeile schreiben und dann für »Geschrieben von:« eine neue Zeile beginnen soll.

Sie können Ihre Anwendung bereits ausführen

Drücken Sie die F5-Taste auf Ihrer Tastatur oder klicken Sie auf den kleinen grünen Pfeil-Button ( ) in der Symbolleiste, um zu prüfen, was Sie bisher erreicht haben. (Das nennt man »Debuggen«, was lediglich bedeutet, dass Sie Ihr Programm über die IDE ausführen.) Sie können das Debuggen beenden, indem Sie im Debuggen-Menü »Debuggen beenden« auswählen oder auf folgenden Symbolleisten-Button klicken: .



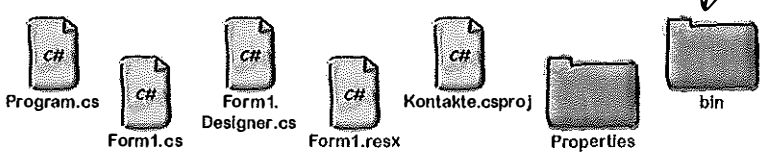
Diese drei Buttons funktionieren alle – und Sie mussten keinen Code schreiben, um sie zum Funktionieren zu bringen.

Klicken Sie auf das PFO-Logo, wird das Info-Fenster eingeblendet, das Sie gerade programmiert haben.

Wo sind meine Dateien?

Führen Sie Ihr Programm aus, kopiert Visual Studio Ihre Dateien alle in `Meine Dateien\Visual Studio 2008\Projects\Kontakte\Kontakte\bin\debug`. Sie können sogar in dieses Verzeichnis gehen und Ihr Programm direkt ausführen, indem Sie doppelt auf die von der IDE erstellte .exe-Datei klicken.

C# wandelt Ihr Programm in eine ausführbare Datei um, die als Programmdatei bezeichnet wird. Sie finden sie hier, im Ordner Debug.



Das ist kein Fehler. Es gibt zwei Ordnerebenen. Der innere Ordner enthält die eigentlichem C#-Codedateien.

Es gibt keine Dummen Fragen

F: In meiner IDE steht beim grünen Pfeil »Debuggen«. Ist das eine Problem?

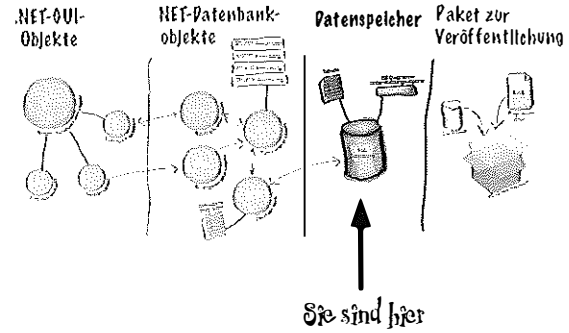
A: Nein. Debuggen bedeutet, zumindest für unsere momentanen Absichten, nur, dass Sie Ihre Anwendung in der IDE ausführen. Später werden wir uns noch ausführlicher mit dem Debuggen befassen, aber im Augenblick können Sie es einfach als eine Methode betrachten, Ihr Programm auszuführen.

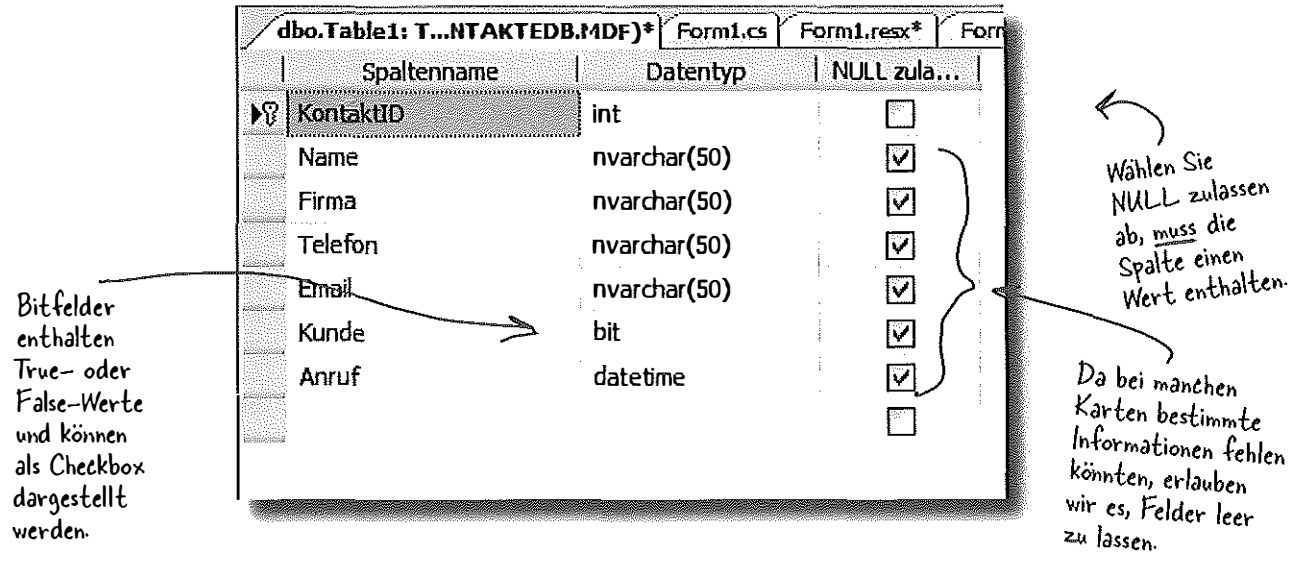
F: Ich sehe In meiner IDE In der Symbolleiste keinen Debuggen beenden-Button. Warum nicht?

A: Der Button Debuggen beenden befindet sich in einer speziellen Symbolleiste, die **nur erscheint**, wenn ein Programm läuft. Starten Sie die Anwendung erneut und schauen Sie, ob er dann da ist.

Den Tabellenbau abschließen

Kehren Sie dorthin zurück, wo Sie die Spalte KontaktID eingegeben haben, und geben Sie die anderen fünf Spalten von der Adresskarte ein. So sollte Ihre Datenbanktabelle aussehen, wenn Sie fertig sind:



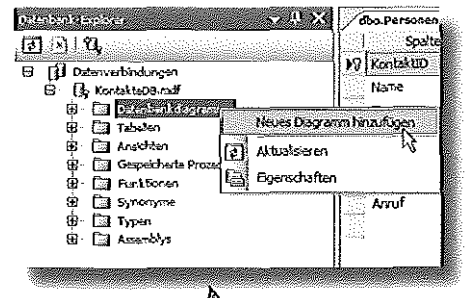


Klicken Sie in der Symbolleiste auf den Button Speichern, um Ihre neue Tabelle zu speichern. Sie werden nach einem Namen gefragt. Geben Sie den Namen »Personen« an und klicken Sie auf OK.



Erstellen Sie ein Diagramm für Ihre Daten, damit Ihre Anwendung darauf zugreifen kann

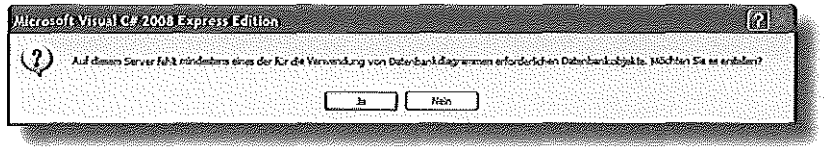
Nachdem Sie Ihre Datenbank und Ihre Tabellen erstellt haben, müssen Sie Ihre Anwendung darüber informieren. Das ist der Punkt, an dem ein Datenbankdiagramm ins Spiel kommt. Ein **Datenbankdiagramm** ist eine einfache Beschreibung Ihrer Tabelle, die die Visual Studio IDE verwenden kann, um mit der Tabelle zu arbeiten. Es sorgt dafür, dass die IDE automatisch SQL-Anweisungen zum Hinzufügen, Ändern und Löschen von Zeilen in der Tabelle generiert.



Denken Sie daran, dass sich diese Optionen alle unter KontakteDB befinden und deswegen für diese spezielle Datenbank gelten.

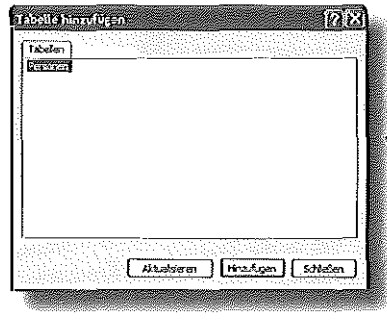
1 **Ein neues Datenbankdiagramm erstellen.**
Gehen Sie ins Fenster Datenbank-Explorer und klicken Sie rechts auf den Knoten Datenbankdiagramme. Wählen Sie Neues Diagramm hinzufügen.

2 **Die IDE den Zugriffscode generieren lassen.**
Bevor Sie die IDE über Ihre spezielle Tabelle informieren, muss sie einige grundlegende gespeicherte Routinen für die Interaktion mit der Datenbank erstellen. Klicken Sie hier einfach auf Ja, damit die IDE an die Arbeit geht.

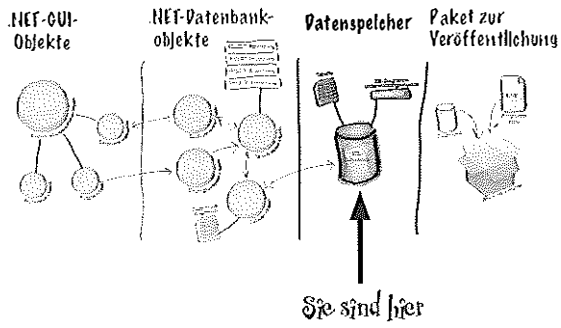


Die IDE erzeugt einige gespeicherte Prozeduren, die Ihrem Code ermöglichen, mit der von Ihnen erstellten Datenbank zu interagieren.

3 **Wählen Sie die Tabellen, mit denen Sie arbeiten möchten.**
Wählen Sie in dem dann erscheinenden Fenster die Tabelle Personen und klicken Sie auf Hinzufügen. Jetzt ist die IDE bereit, spezifischen Code für Ihre Tabelle zu erstellen.



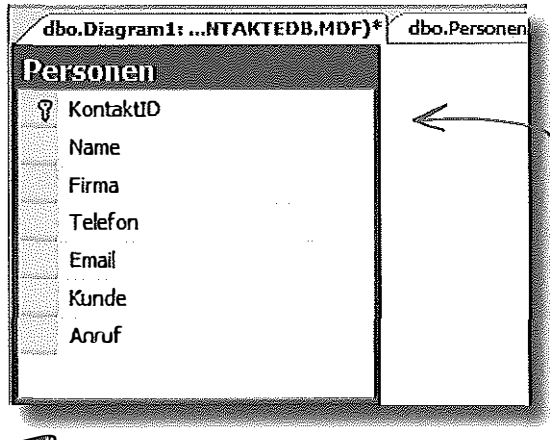
Haben Sie Datenbanken mit mehreren Tabellen, taucht hier ein Eintrag für jede Tabelle auf.



4 **Geben Sie dem Diagramm den Name PersonenDiagramm.**
Wählen Sie Datei→Diagram1 speichern. Sie werden aufgefordert, einen Namen für das neue Datenbankdiagramm einzugeben. Wählen Sie dafür den Namen PersonenDiagramm. Jetzt sind Sie so weit.

Wählen Sie stattdessen Speichern→Alles speichern, wenn Sie Visual Studio 2005 verwenden.

Hier wird das Datenbankdiagramm grafisch dargestellt. Es ist eine sehr einfache Darstellung Ihrer Tabelle.



Das ist nur ein Bild des Datenbankentwurfs, den Sie gerade erstellt haben. Es markiert das Feld KontaktID als Ihren Primärschlüssel und führt alle weiteren Spalten der Tabelle auf.

Hätten Sie in der Datenbank weitere Tabellen, für die Diagramme erstellt werden sollen, erschienen auch diese hier.

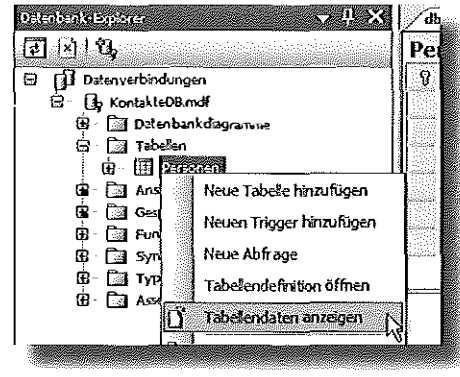
Ein Datenbankdiagramm beschreibt Ihre Tabellen für die Visual Studio IDE. Die IDE nutzt das Datenbankdiagramm dann, um automatisch generierten Code für die Arbeit mit Ihrer Datenbank zu erstellen.

Ihre Kartendaten in die Datenbank einfügen

Jetzt sind Sie bereit, mit der Eingabe der Karteninformationen in die Datenbank zu beginnen. Hier sind einige der Kontakte Ihres Chef – diese werden wir nutzen, um die Datenbank mit ein paar Datensätzen einzurichten.

1. Erweitern Sie im Datenbank-Explorer (oder im Server-Explorer) den Knoten Tabellen, klicken Sie rechts auf die Tabelle Personen und wählen Sie dann Tabellendaten anzeigen.

2. Erscheint im Hauptfenster das Tabellenraster, können Sie mit der Eingabe der unten stehenden Daten loslegen. (Zunächst werden Sie nur NULL-Werte sehen – überschreiben Sie diese einfach, wenn Sie Ihre erste Zeile einfügen. Und ignorieren Sie die Ausrufezeichen, die neben den Daten erscheinen.) Die Spalte KontaktID müssen Sie nicht ausfüllen, da das automatisch erfolgt.



Ihre Aufgabe ist es, die Daten von diesen sechs Karten in die Tabelle Personen einzufügen.

Geben Sie in die Spalte Kunde >>True<< oder >>False<< ein. So speichert SQL Ja/Nein-Informationen.

Papierfabrik Objekthausen

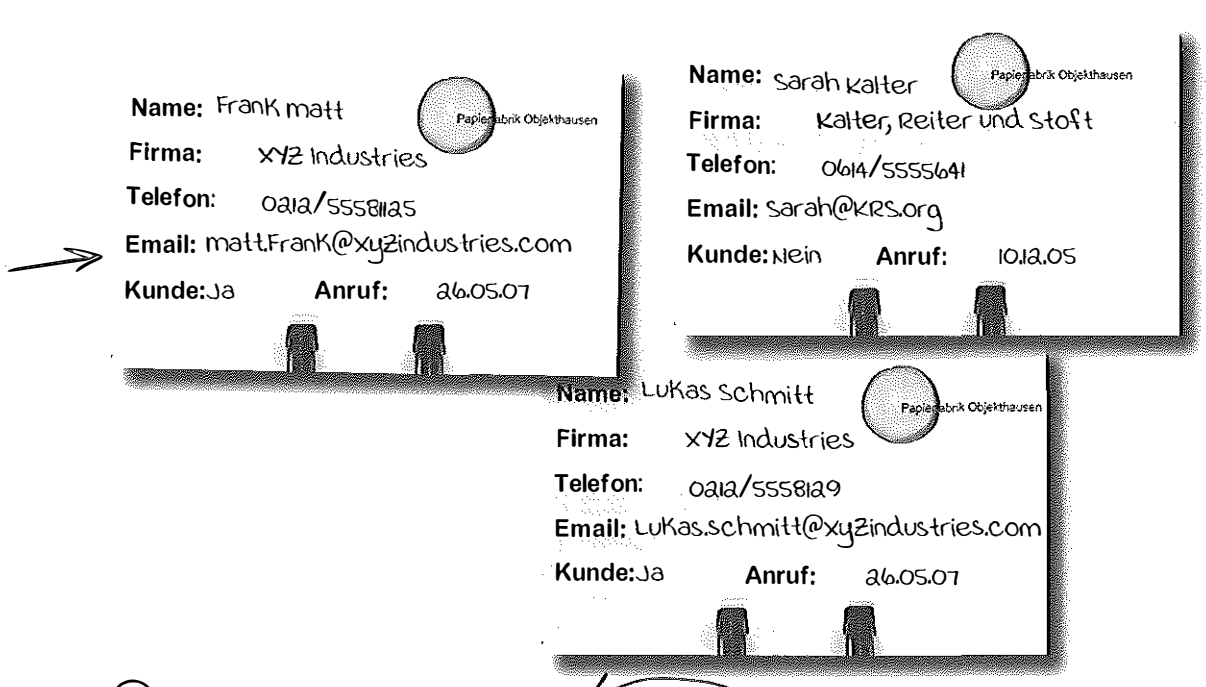
Name: Lisa Nilson
Firma: JTP
Telefon: 0419/5552578
Email: LisaNilson@JTP.org
Kunde: Ja **Anruf:** 04.03.06

Papierfabrik Objekthausen

Name: Marie Erichson
Firma: Erichson Events
Telefon: 0212/5559523
Email: marie@erichsonevents.info
Kunde: Nein **Anruf:** 17.05.07

Papierfabrik Objekthausen

Name: Lukas Löhrer
Firma: Black Box Inc.
Telefon: 0718/5555638
Email: lloehrer@xblackboxinc.de
Kunde: Ja **Anruf:** 26.05.07



3 Wählen Sie im Datei-Menü erneut Alles speichern, wenn Sie alle sechs Datensätze eingegeben haben. Das sollte die Datensätze in der Datenbank speichern.

»Alles speichern« sagt der IDE, dass sie alles in Ihrer Anwendung speichern soll. Das ist was anderes als »Speichern«, das nur die Datei speichert, an der Sie arbeiten.

Es gibt keine Dummen Fragen

F: Was passiert mit den Daten, nachdem Ich sie eingegeben habe? Wo sind sie hingewandert?

A: Die IDE hat die von Ihnen eingegebenen Daten automatisch in der Personen-Tabelle Ihrer Datenbank gespeichert. Die Tabelle, ihre Spalten, die Datentypen und alle Daten darin wurden in der SQL Server Express-Datei KontakteDB.mdf gespeichert. Diese Datei wird als Teil ihres Projekts gespeichert und wird genau wie Ihre Codedateien von der IDE aktualisiert, wenn Sie etwas ändern.

F: So ... Ich habe diese sechs Datensätze jetzt eingegeben. Werden sie immer Teil meines Programms bleiben?

A: Ja, sie sind ebenso sehr Teil des Programm wie der Code, den Sie schreiben, und das Formular, das Sie aufbauen. Der Unterschied ist, dass die Datei KontakteDB.mdf nicht zu einer Programmdatei kompiliert wird, sondern kopiert und mit der Programmdatei gespeichert wird. Wenn Ihre Anwendung auf Daten zugreifen muss, liest sie und schreibt sie in KontakteDB.mdf im Ausgabeverzeichnis Ihres Programms.

Diese Datei ist eigentlich eine SQL-Datenbank, die Ihr Programm über den Code nutzen kann, den die IDE für Sie erstellt hat.



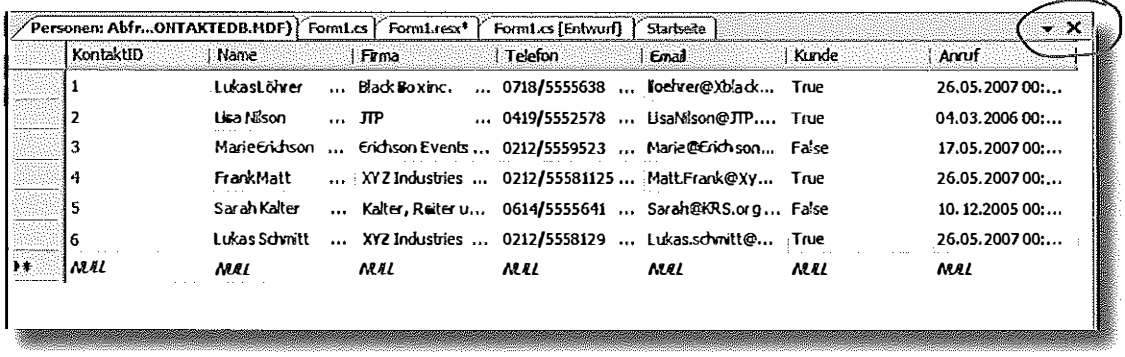
KontakteDB.mdf

Ihr Formular über eine Datenquelle mit Ihren Datenobjekten verbinden

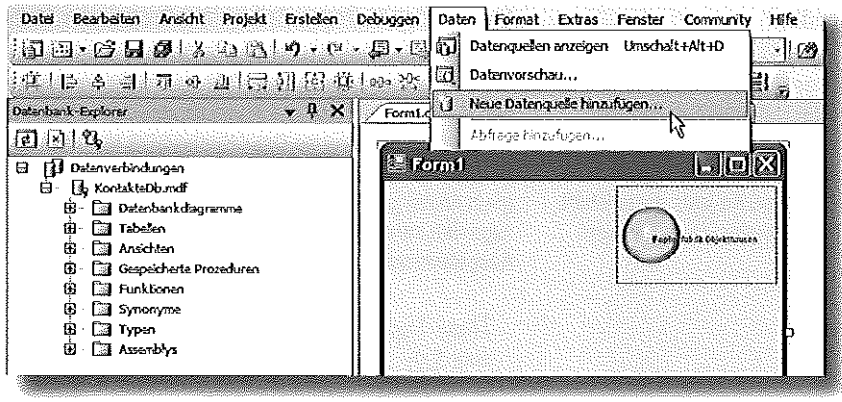
Endlich sind wir bereit, die .NET-Datenbankobjekte zu erstellen, die unser Formular nutzen wird, um mit der Datenbank zu reden. Wir brauchen eine **Datenquelle**, und das ist eigentlich nur eine Sammlung von SQL-Anweisungen, die Ihr Programm nutzen wird, um mit der Datenbank KontakteDB zu kommunizieren.

Sie müssen das Datengitter und das Diagramm schließen, um zu Ihrem Formular zurückzugelangen.

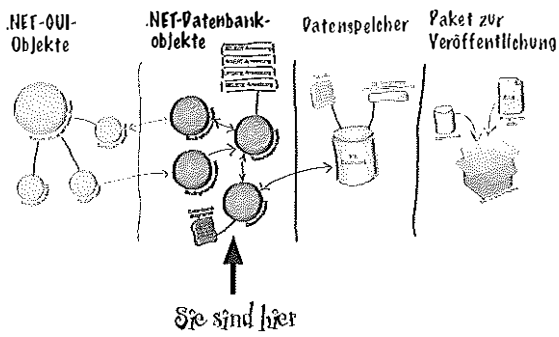
1 **Kehren Sie zum Formular Ihrer Anwendung zurück.**
Schließen Sie die Tabelle Personen und das Datenbankdiagramm für KontakteDB. Jetzt sollte die Registerkarte Form1.cs [Entwurf] sichtbar sein.



2 **Fügen Sie Ihrer Anwendung eine neue Datenquelle hinzu.**
Das sollte jetzt einfach sein. Gehen Sie ins Menü Daten und wählen Sie Neue Datenquelle hinzufügen...



Die Datenquelle, die Sie erstellen, wird die Interaktion zwischen Ihrem Formular und der Datenbank abwickeln.

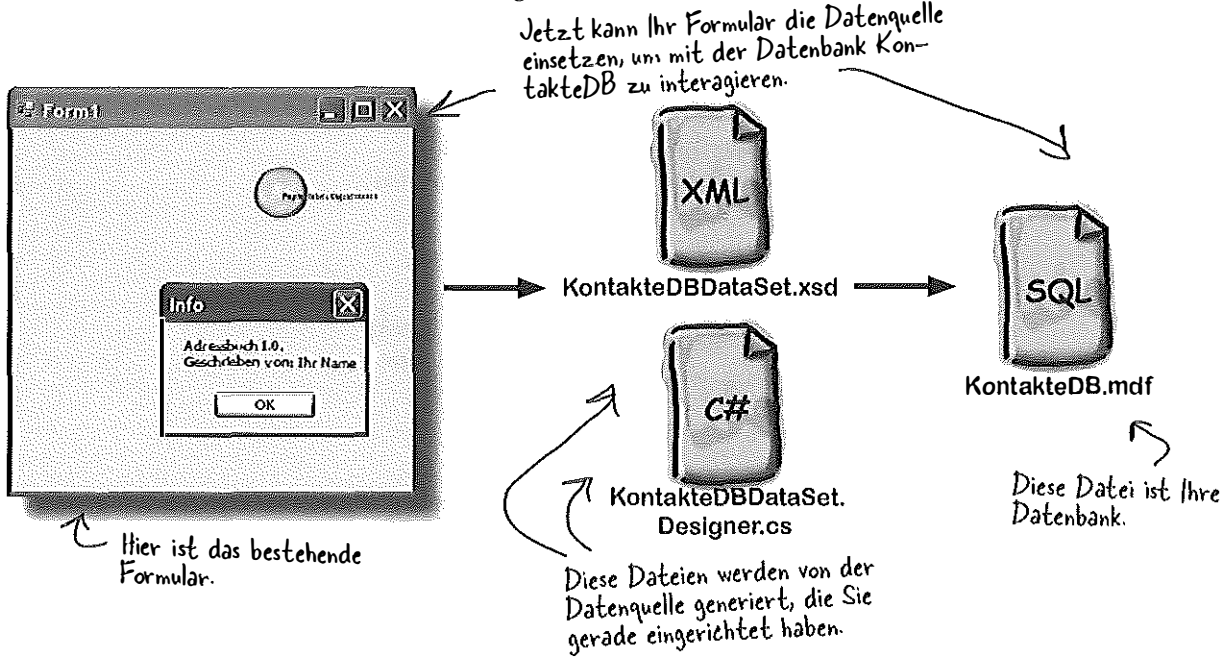


3 **Die neue Datenquelle konfigurieren.**

Jetzt müssen Sie Ihre Datenquelle so einrichten, dass sie die Datenbank KontakteDB nutzt. Das machen Sie so:

- ★ Wählen Sie Datenbank und klicken Sie auf den Button Weiter.
- ★ Klicken Sie wiederum auf Weiter im Fenster »Wählen Sie Ihre Datenverbindung«.
- ★ Prüfen Sie, ob im folgenden Fenster »Verbindung speichern« die Checkbox Ja, Verbindung speichern unter angekreuzt ist, und klicken Sie Weiter.
- ★ Aktivieren Sie im Fenster »Datenbankobjekte auswählen« die Checkbox Tabellen.
- ★ Prüfen Sie, ob im Feld DataSet-Name »KontakteDBDataSet« steht, und klicken Sie auf Fertig stellen.

Diese Schritte verbinden Ihre neue Datenquelle mit der Tabelle Personen in der Datenbank KontakteDB.



Dem Formular datenbankgesteuerte Steuerelemente hinzufügen

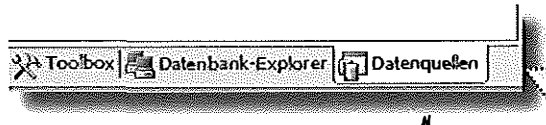
Jetzt können wir zu unserem Formular zurückkehren und ihm weitere Steuerelemente hinzufügen. Aber das sind nicht einfach irgendwelche Steuerelemente, es sind Steuerelemente, die an unsere Datenbank und die Spalten in der Tabelle Personen *gebunden* sind. Das heißt einfach, dass eine Änderung an den Daten in einem der Steuerelemente im Formular automatisch eine Änderung der Daten in der entsprechenden Datenbankspalte bewirkt.

Es war etwas Arbeit erforderlich, aber jetzt sind wir wieder zurück bei der Erstellung von Formularobjekten, die mit unserem Datenspeicher interagieren.

Folgendermaßen erzeugt man mehrere datenbankgesteuerte Steuerelemente:

1. **Die Datenquelle wählen, die Sie verwenden möchten.**
Wählen Sie im Menü Daten »Datenquellen anzeigen«. Damit rufen Sie das Fenster Datenquellen auf, in dem die Datenquellen angezeigt werden, die Sie für die Anwendung eingerichtet haben.

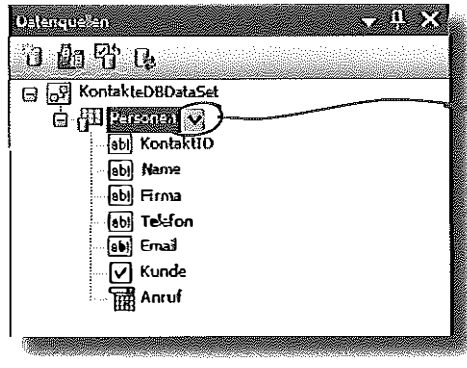
Sollten Sie diese Registerkarte nicht sehen, wählen Sie »Datenquellen anzeigen« im Menü Daten.



Dieses Fenster zeigt Ihnen alle Datenquellen. Wir haben nur eine eingerichtet, aber Sie könnten weitere für andere Tabellen oder Datenbanken haben.

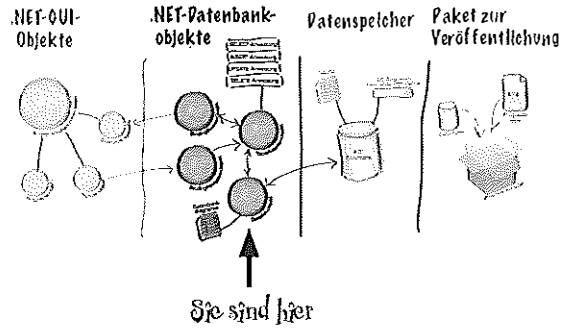
Nach der Registerkarte Datenquelle können Sie unten im Fenster Datenbank-Explorer suchen und dann darauf klicken.

2. **Wählen Sie die Tabelle Personen.**
Unter KontakteDBDataSet sollten Sie die Tabelle Personen und alle Spalten darin sehen. Falls Sie die Spalten nicht sehen, klicken Sie auf den Pfeil für das Aufklappmenü und wählen dann Details.



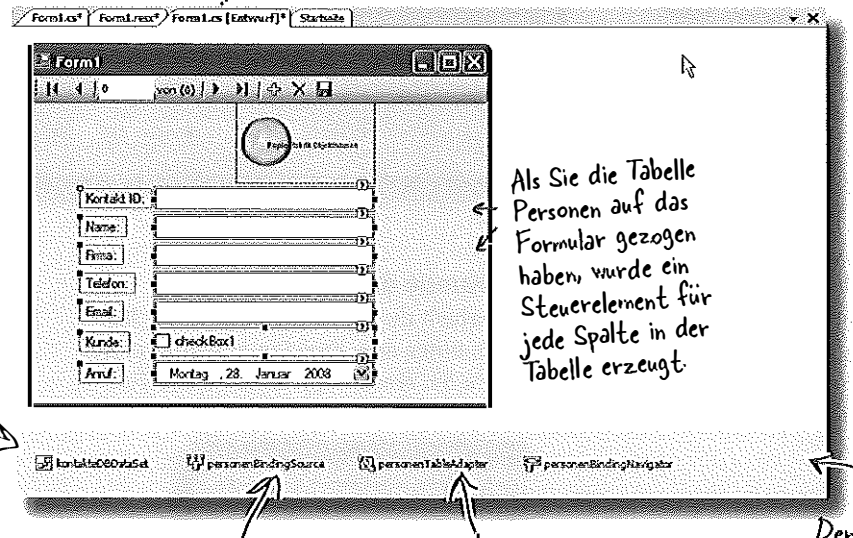
Das ist der kleine Pfeil, auf den Sie klicken müssen.

Hier sollten alle Spalten erscheinen, die Sie erstellt haben.



3 **An die Tabelle Personen gebundene Steuerelemente erstellen.**
Ziehen Sie die Tabelle Personen auf Ihr Formular und lassen Sie sie dort fallen. Auf dem Formular sollten jetzt Steuerelemente für alle Spalten in der Datenbank erscheinen. Zerbrechen Sie sich jetzt noch nicht darüber den Kopf, wie diese aussehen. Prüfen Sie nur, ob alle im Formular vorhanden sind.

Klicken Sie sich aus Versehen aus dem Formular heraus, an dem wir arbeiten, kommen Sie jederzeit dorthin zurück, indem Sie auf die Registerkarte »Form1.cs [Entwurf]« klicken oder über den Projektmappen-Explorer Form1.cs öffnen.



Die IDE erzeugt diese Symbolleiste zur Navigation durch die Tabelle Personen.

Als Sie die Tabelle Personen auf das Formular gezogen haben, wurde ein Steuerelement für jede Spalte in der Tabelle erzeugt.

Diese Objekte erscheinen in Ihrem Formular nicht, repräsentieren aber das DataSet, das die IDE für die Interaktion mit der Tabelle Personen und der Datenbank KontakteDB erstellt hat.

Dieses Objekt verknüpft das Formular mit der Tabelle Personen.

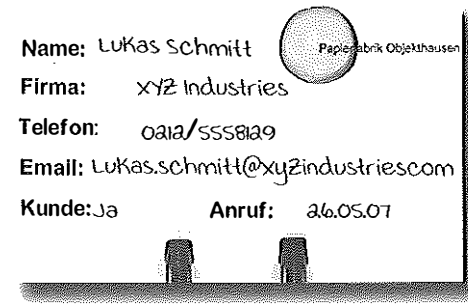
Dieser Adapter ermöglicht Ihren Steuerelementen, über SQL-Anweisungen, die die IDE und die Datenquelle für Sie generiert haben, mit der Datenbank zu interagieren.

Der BindingNavigator verbindet die Symbolleistensteuerelemente mit der Tabelle.

Gute Programme sind intuitiv zu verwenden

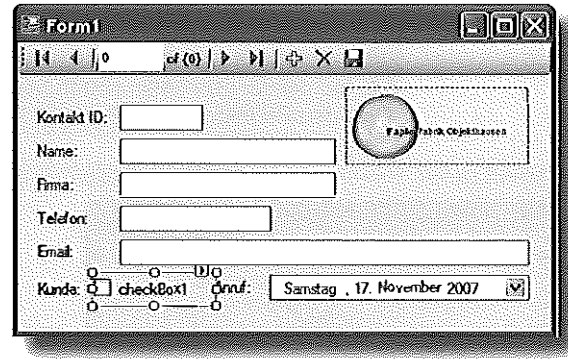
Im Augenblick funktioniert das Programm nur. Aber toll aussschen tut es noch nicht. Ihre Anwendung muss nicht nur funktionieren, sie sollte auch leicht zu verwenden sein. Mit nur ein paar einfachen Schritten können Sie das Formular dazu bringen, dass es den Papierzetteln, die wir zu Anfang des Kapitels verwendet haben, deutlich ähnlicher sieht.

Unser Formular wäre intuitiver, würde es den Kontaktkarten ähneln.

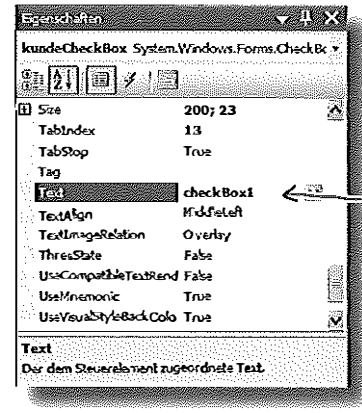


1 **Felder und Beschriftungen ausrichten.**
Richten Sie die Felder und Beschriftungen am linken Rand des Formulars aus. Ihr Formular sieht dann wie jede andere Anwendung aus, und das bewirkt, dass sich Ihre Benutzer bei der Arbeit damit wohler fühlen.

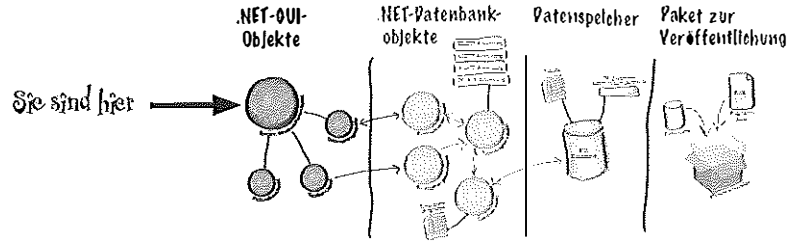
Auf dem Formular erscheinen blaue Linien, wenn Sie Steuerelemente verschieben. Sie sollen Ihnen bei der Ausrichtung der Felder helfen.



2 **Die Text-Eigenschaft der Checkbox Kunde ändern.**
Als Sie die Felder auf Ihr Formular zogen, erhielt die Checkbox rechts eine Beschriftung, die gelöscht werden muss. Rechts unter dem Projektmappen-Explorer sehen Sie das Eigenschaften-Fenster. Scrollen Sie abwärts zur Eigenschaft Text und löschen Sie die Beschriftung »checkbox1«.



Löschen Sie dieses Wort, damit die Beschriftung verschwindet.



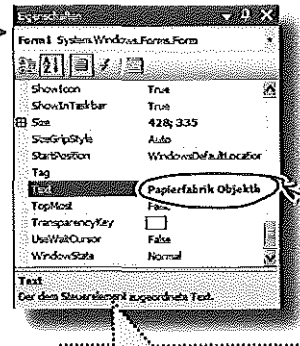
③ Der Anwendung einen professionellen Anstrich verleihen.

Den Namen des Formulars können Sie ändern, indem Sie an eine leere Stelle im Formular klicken und im Eigenschaften-Fenster der IDE nach der Eigenschaft Text suchen. Ändern Sie den Namen des Formulars in »Papierfabrik Objekthausen - Adressliste«.

Außerdem können Sie die Maximieren- und Minimieren-Buttons im gleichen Fenster abschalten, indem Sie nach den Eigenschaften MaximizeBox und MinimizeBox suchen. Setzen Sie beide auf False.

Den Maximieren-Button wollen wir abschalten, weil das Maximieren des Formulars die Position der Steuerelemente nicht ändert und das Formular dann seltsam aussieht.

Das Fenster Eigenschaften sollte sich in der unteren rechten Ecke der IDE unter dem Projektmappen-Explorer befinden.



Die Eigenschaft Text steuert den Titel Ihres Formulars.

Sollten Sie kein Eigenschaften-Fenster haben, können Sie es anschalten, indem Sie es im Ansicht-Menü auswählen.

Eine gute Anwendung funktioniert nicht nur, sondern ist auch leicht zu verwenden. Man sollte immer prüfen, ob sie sich so verhält, wie es ein typischer Nutzer erwarten würde.

Testlauf

Gut. Jetzt müssen wir nur noch eine Sache tun ... das Programm ausführen und prüfen, ob es alles tut, was es tun soll! Machen Sie es genau so wie eben – drücken Sie auf Ihrer Tastatur die Taste F5 oder klicken Sie auf den grünen Pfeil-Button ▶ in der Symbolleiste (oder wählen Sie im Debuggen-Menü »Ausführen«).

Sie können Ihr Programm jederzeit ausführen, selbst wenn Sie noch nicht fertig sind – sollte Ihr Code allerdings einen Fehler haben, teilt die IDE Ihnen das mit und verhindert, dass Sie das Programm ausführen.

Klicken Sie auf den X-Button in der Ecke, um das Programm anzuhalten, damit Sie zum nächsten Schritt übergehen können.

Über diese Steuerelemente können Sie die verschiedenen Datensätze in der Datenbank durchblättern.



Damit werden wir im nächsten Kapitel noch mehr Zeit verbringen.

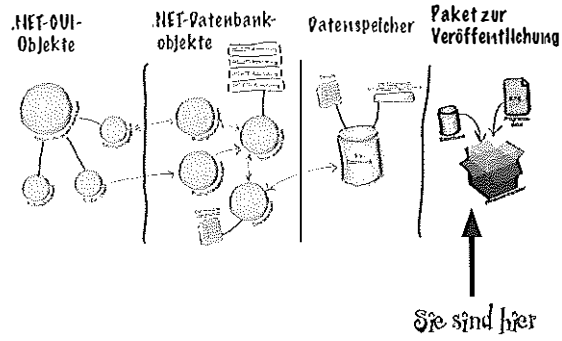
Die IDE erstellt das Programm erst und führt es dann aus.

Führen Sie Ihr Programm in der IDE aus, macht sie eigentlich zwei Dinge. Erst **erstellt** sie Ihr Programm, dann **führt** sie es **aus**. Dabei sind mehrere Schritte erforderlich. Sie **kompiliert** den Code, d.h. wandelt ihn in eine ausführbare Programmdatei um. Dann platziert sie den kompilierten Code gemeinsam mit allen Ressourcen und anderen Dateien in einem Unterverzeichnis des bin-Ordners.

Die Programmdatei und die SQL-Datenbankdatei finden Sie nun in `bin/debug`. Da die Datenbank jedes Mal neu kopiert wird, gehen alle von Ihnen vorgenommenen Änderungen verloren, wenn Sie das Programm das nächste Mal in der IDE ausführen. Aber wenn Sie die Programmdatei aus Windows heraus ausführen, werden Ihre Daten gespeichert – bis Sie das Programm erneut erstellen. Dann überschreibt die IDE die SQL-Datenbank wieder mit einer neuen Kopie, die die Daten enthält, die Sie im Datenbank-Explorer eingerichtet haben.

Beim Erstellen des Programms werden die Daten in der Datenbank überschrieben.

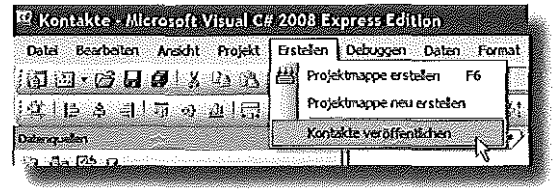
Wie Sie IHRE Anwendung in eine Anwendung für ALLE verwandeln



Sie haben jetzt ein gutes Programm. Aber es läuft nur auf Ihrem Computer. Das bedeutet, dass kein anderer die Anwendung verwenden oder Sie dafür bezahlen kann, sieht, wie gut Sie sind, und Ihnen einen Job geben möchte ... und Ihr Chef und Ihre Kunden können keine der Berichte sehen, die Sie auf Basis der Datenbank erstellen.

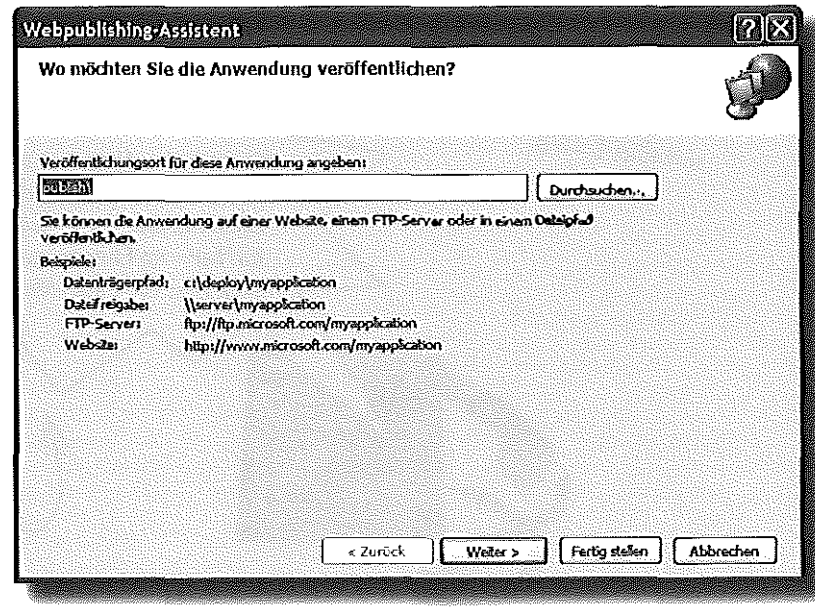
C# macht es leicht, eine erstellte Anwendung zu **veröffentlichen**. Die Veröffentlichung nimmt eine Anwendung und installiert sie auf anderen Rechnern. Mit der Visual C# IDE können Sie die Verteilung in nur zwei Schritten einrichten.

1. Wählen Sie im Menü Erstellen *Kontakte veröffentlichen*.



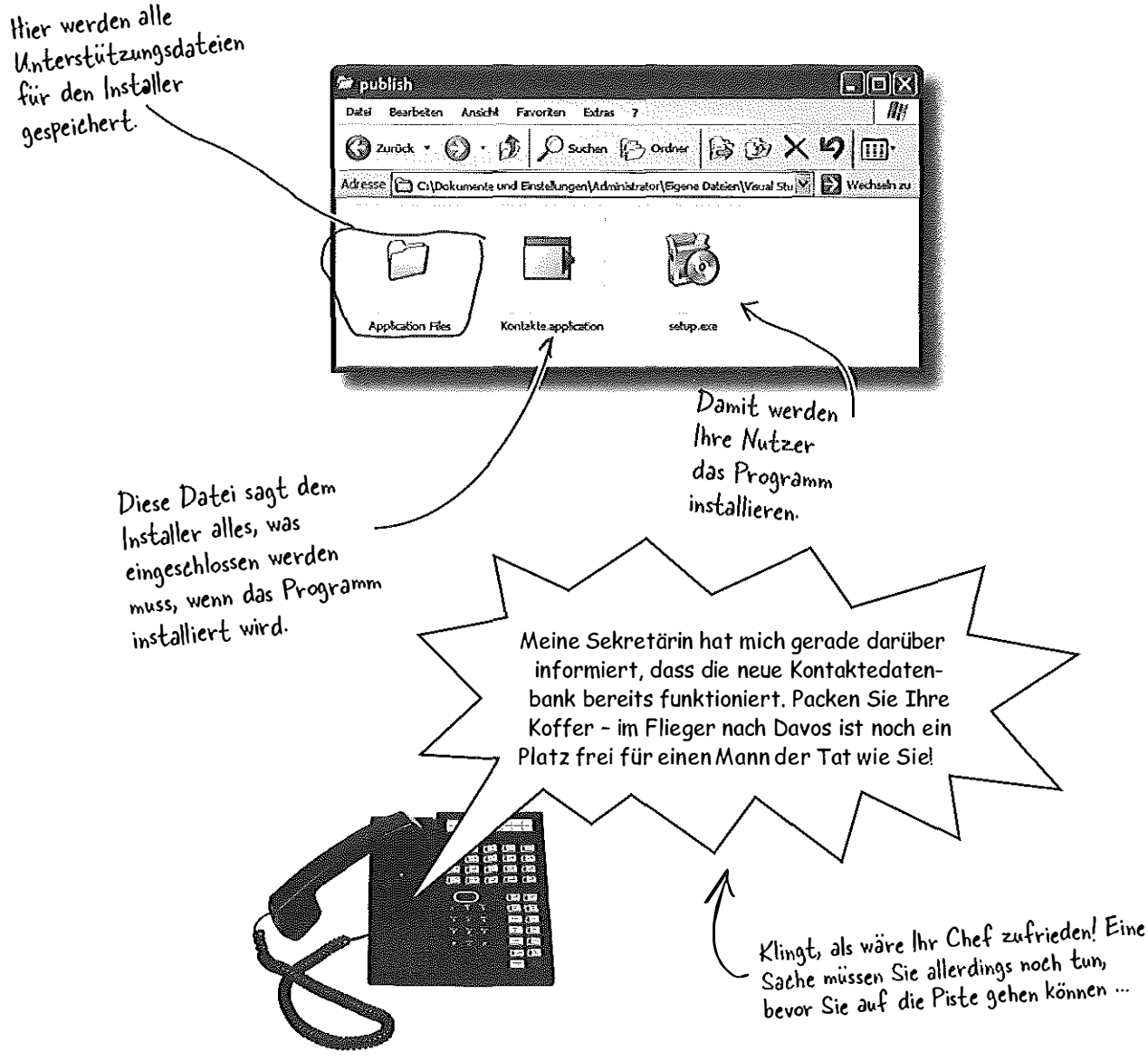
Beim Erstellen werden die Dateien nur auf den lokalen Rechner kopiert. Durch die Veröffentlichung werden eine Setup-Programmdatei und eine Konfigurationsdatei erstellt, damit Ihr Programm auf beliebigen Rechnern installiert werden kann.

2. Akzeptieren Sie im Publishing-Assistent einfach die Defaults, indem Sie auf »Fertig stellen« klicken. Ihre Anwendung wird verpackt, und dann wird Ihnen ein Ordner gezeigt, der Ihre Setup.exe enthält.



Ihren Nutzern die Anwendung geben

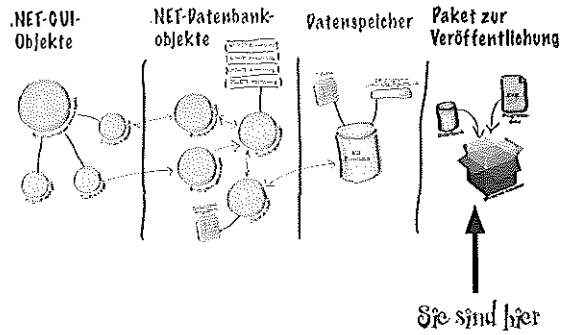
Nachdem Sie ein Veröffentlichungspaket erstellt haben, erhalten Sie einen neuen Ordner namens `publish/`. Dieser Ordner enthält verschiedene Dinge, die alle für die Installation verwendet werden. Für Ihre Nutzer am wichtigsten ist `setup`, ein Programm, mit dem sie Ihr Programm auf ihren eigenen Rechnern installieren können.



Sie sind noch NICHT fertig: Testen Sie Ihre Anwendung

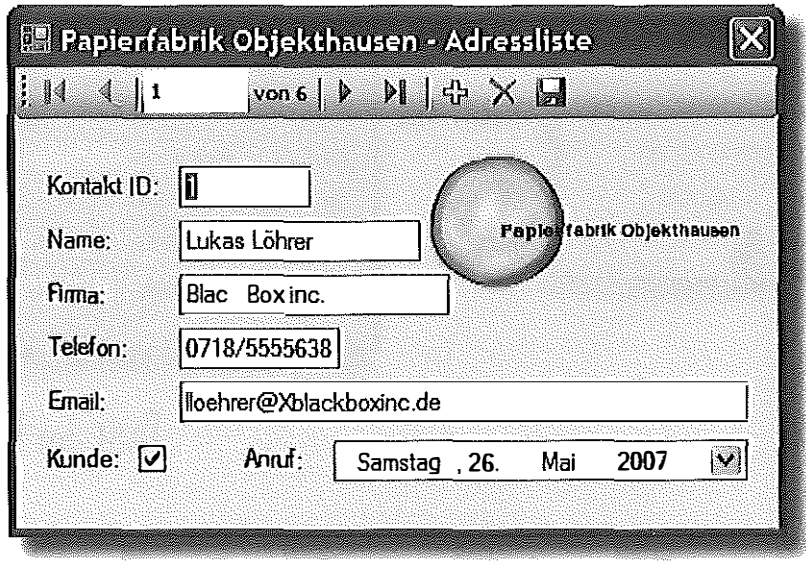
Bevor Sie die Korken irgendwelcher Champagnerflaschen knallen lassen, müssen Sie die Verteilung und die Installation noch testen. Sie würden doch niemandem Ihr Programm geben, bevor Sie es selbst ausgeführt haben, oder?

Schließen Sie Visual Studio IDE. Klicken Sie auf das Setup-Programm und wählen Sie einen Ort auf dem Rechner, an dem das Programm installiert werden soll. Nun können Sie es von dort aus ausführen und prüfen, ob alles wie erwartet funktioniert. Sie können Datensätze jetzt auch so hinzufügen und ändern, dass die Änderungen in der Datenbank gespeichert werden.



Jetzt werden die Datensätze, die Sie ändern, hinzufügen oder löschen, auch in der Datenbank gespeichert.

Sie können die Pfeile und die Textfelder einsetzen, um zwischen den Datensätzen zu wechseln.



Legen Sie los ... nehmen Sie ein paar Änderungen vor. Sie haben die Anwendung verteilt, also werden sie diesmal gespeichert.

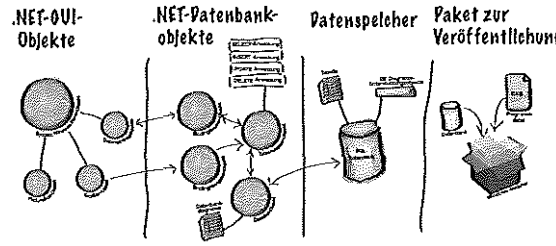
Die sechs Datensätze, die Sie anfänglich eingegeben haben, sind alle da.

TESTEN SIE ALLES!

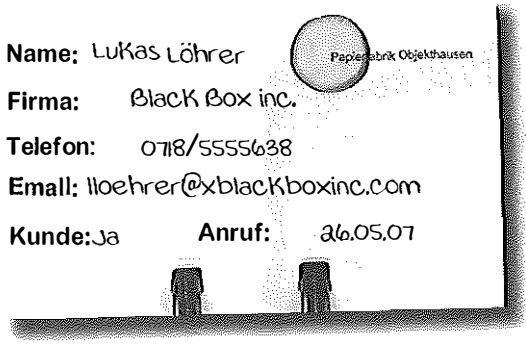
Testen Sie Ihr Programm, Ihr Veröffentlichungspaket und auch die Daten in Ihrer Anwendung.

Sie haben eine vollständige, datengesteuerte Anwendung erstellt

Die Visual Studio IDE hat es ziemlich leicht gemacht, eine Windows-Anwendung zu erstellen, eine Datenbank zu entwerfen und zu erstellen und beides schließlich zu verknüpfen. Mit ein paar Klicks zusätzlich konnten Sie sogar einen Installer bauen.



Von hier



nach da



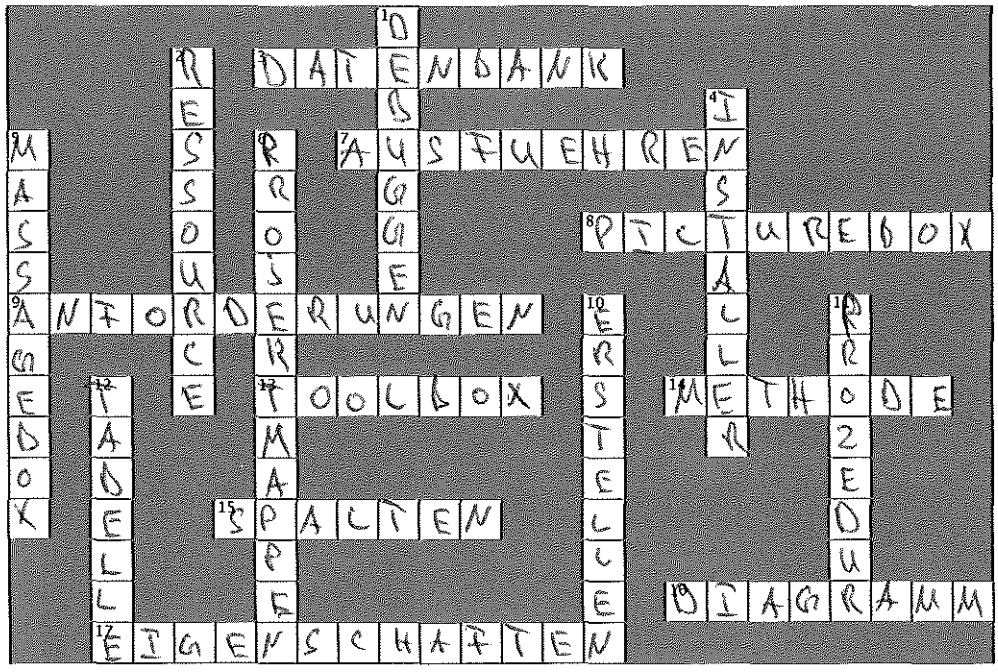
fast im Handumdrehen!

Die Stärke von Visual C# ist, dass Sie Anwendungen schnell eingerichtet und laufen haben und sich dann darauf konzentrieren können, was Ihr Programm tun soll, statt sich um massenhaft Code für Fenster, Buttons und den SQL-Zugriff zu kümmern.



C#-Kreuzworträtsel

Lehnen Sie sich einen Moment zurück und testen Sie Ihr C#-Vokabular mit diesem Kreuzworträtsel. Alle Wörter stammen aus diesem Kapitel.

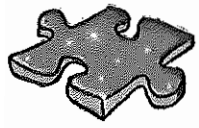


Waagerecht

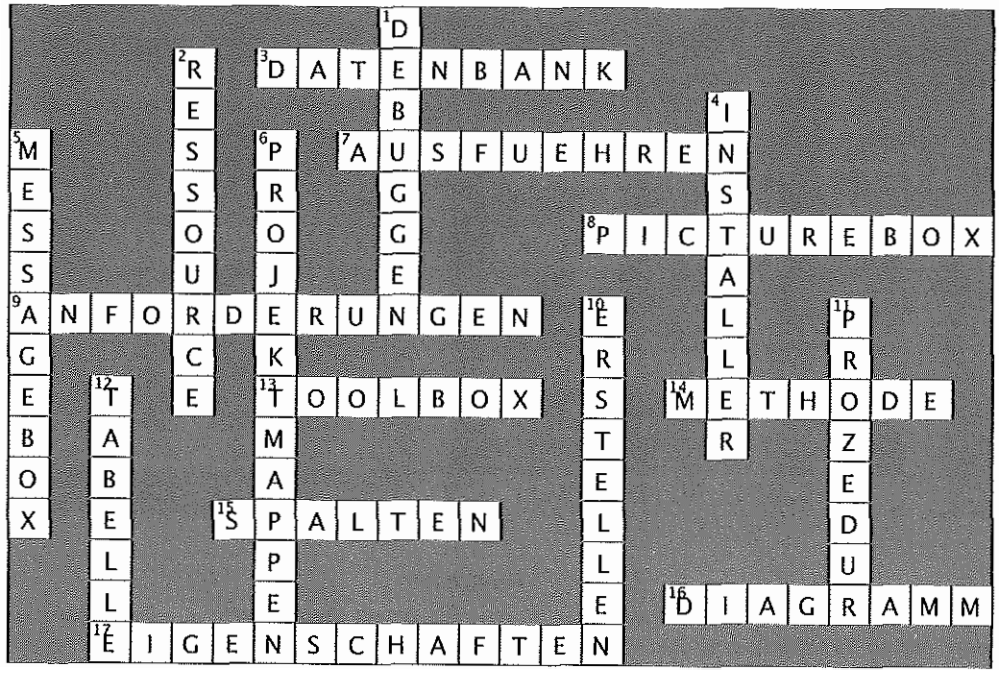
3. Im _____-Explorer können Sie den Inhalt Ihrer SQL-Tabellen bearbeiten und an Ihr Programm binden.
7. Wenn Sie das aus der IDE heraus tun, nennt man es »debuggen«.
8. Ziehen Sie eins dieser Objekte auf Ihr Formular, um ein Bild anzuzeigen.
9. Bevor Sie mit dem Aufbau einer Anwendung beginnen, sollten Sie zuerst immer über Ihre Benutzer und deren _____ nachdenken.
13. Daraus ziehen Sie Objekte auf Ihr Formular.
14. Klicken Sie in der IDE doppelt auf ein sichtbares Steuerelement, erzeugt die IDE für Sie etwas, das Sie dann mit Code füllen.
15. Davon kann jede Zelle Ihrer Datenbank mehrere haben, und alle können unterschiedliche Datentypen haben.
16. Das Datenbank-_____ liefert der IDE Informationen zu Ihrer Datenbank, damit sie automatisch SQL-Anweisungen erstellen kann.
17. Das ändern Sie, um das Erscheinungsbild oder das Verhalten der Objekte auf Ihrem Formular zu ändern.

Senkrecht

1. Das tun Sie mit Ihrem Programm, wenn Sie es aus der IDE heraus ausführen.
2. Ein Bild, eine Sounddatei, ein Icon oder eine Datei, die so mit Ihrem Projekt verknüpft ist, dass Ihre Objekte leicht auf sie zugreifen können.
4. Ein solches Programm erstellen Sie, damit Sie Ihre Anwendung auf andere Rechner verteilen können.
5. Das »Info«-Fenster im Kontaktlistenprogramm war ein derartiges Objekt.
6. Im _____-Explorer der IDE finden Sie alle Dateien in Ihrem Projekt.
10. Bevor Sie ein Programm ausführen können, muss die IDE das tun, um die Programmdatei zu generieren und die Dateien ins Ausgabeverzeichnis zu verschieben.
11. Mit einer gespeicherten _____ kann eine SQL-Datenbank Abfragen und Anweisungen speichern, die Sie später wiederverwenden können.
12. Eine SQL-Datenbank kann viele davon verwenden, um Ihre Daten zu speichern.

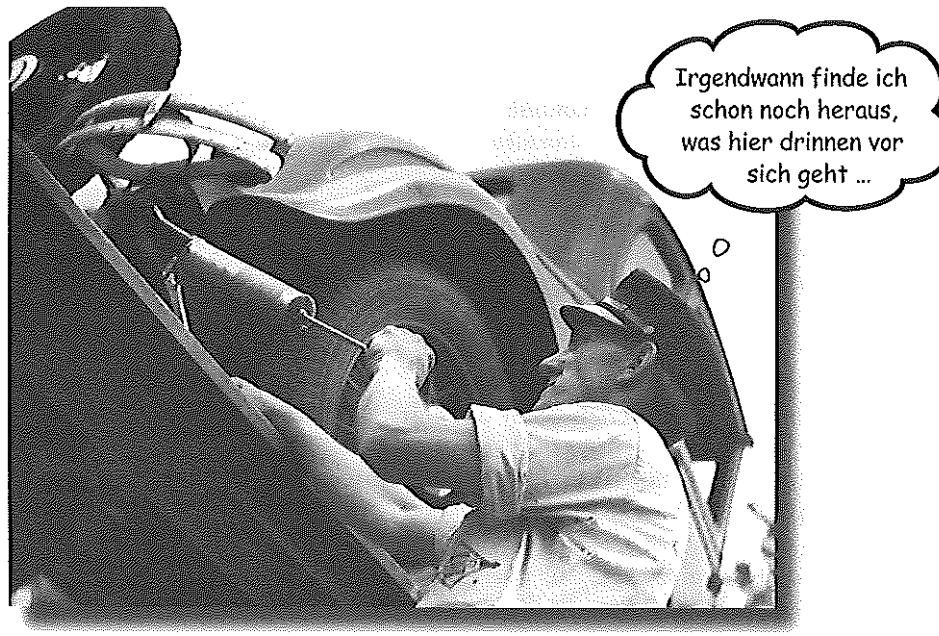


C#-Kreuzworträtsel, Lösung



2 Es ist alles bloß Code

Unter der Motorhaube



Sie sind Programmierer, nicht bloß IDE-Benutzer.

Mit der IDE können Sie eine Menge Dinge erledigen. Aber sie kann Sie nur ein bestimmtes Stück Ihres Wegs begleiten. Sicher gibt es eine Menge **sich wiederholender Aufgaben**, die Sie erledigen müssen, wenn Sie eine Anwendung aufbauen. Diese Aufgaben kann Ihnen die IDE wunderbar abnehmen. Aber die Arbeit mit der IDE ist *erst der Anfang*. Sie können Ihre Programme dazu bringen, so viel mehr zu machen – und das geht nur, indem Sie **C#-Code schreiben**. Haben Sie das Programmieren einmal im Griff, gibt es *nichts mehr*, was Ihre Programme nicht tun könnten.

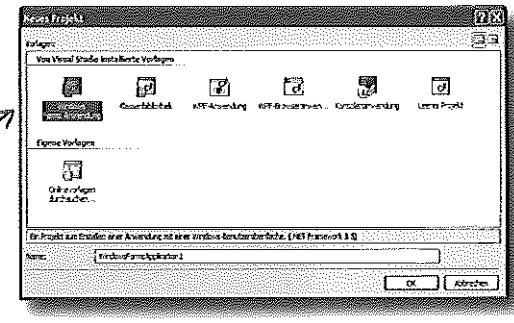
Wenn Sie das tun ...

Alle diese Aufgaben drehen Sie um Standardaktionen und Codebausteine. Bei diesen Dingen kann Ihnen die IDE gut helfen.

Die IDE ist ein mächtiges Werkzeug – aber das ist alles, was sie ist, ein *Werkzeug*, das Sie verwenden können. Jedes Mal, wenn Sie Ihr Projekt ändern oder etwas in die IDE ziehen, erstellt sie automatisch Code. Sie ist richtig gut, wenn es darum geht, **Codebausteine** oder Code zu schreiben, der ohne große Anpassung wiederverwendet werden kann.

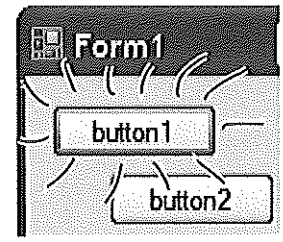
Sehen wir uns an, was die IDE während der Entwicklung einer typischen Anwendung macht ...

1. **Erstellen Sie eine Projektmappe für eine neue Windows-Anwendung ...**
   Mit der IDE können Sie verschiedene Arten von Anwendungen erstellen, aber im Augenblick werden wir uns auf Windows-Anwendungen konzentrieren. Das sind Programme mit grafischen Elementen wie Formularen und Buttons.

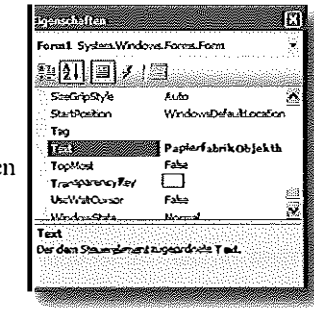


Achten Sie darauf, dass Sie immer ein Windows-Anwendung-Projekt erstellen – das sagt der IDE, dass sie ein leeres Formular erstellen und Ihrem neuen Projekt hinzufügen soll.

2. **Ziehen Sie einen Button aus der Toolbox auf Ihr Formular und klicken dann doppelt darauf ...**
   Mit Buttons sorgen Sie dafür, dass in Ihrem Formular Dinge passieren. Wir werden viele Buttons einsetzen, um uns die verschiedenen Teile der Programmiersprache C# anzusehen. Außerdem sind sie Bestandteil fast jeder C#-Anwendung, die Sie schreiben werden.



3. **Setzen Sie eine Eigenschaft Ihres Formulars ...**
   Das **Eigenschaften-Fenster** der IDE ist ein sehr mächtiges Werkzeug, das Sie einsetzen können, um die Attribute von fast allem in Ihrem Formular zu ändern: aller grafischen und funktionellen Eigenschaften der Steuerelemente Ihres Formulars, Attribute Ihrer Datenbanken und sogar Optionen Ihres Projekts selbst.



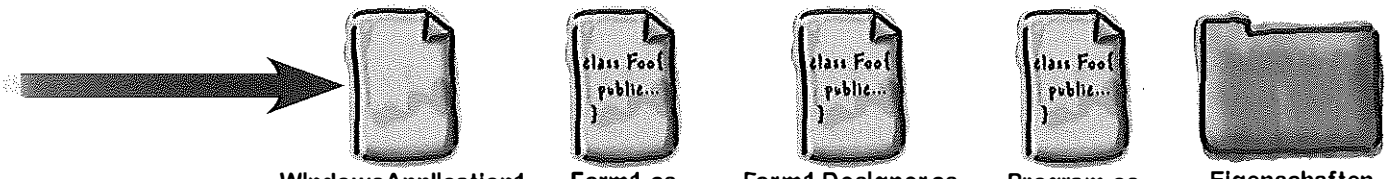
Das Eigenschaften-Fenster der IDE ist ein sehr leicht zu handhabendes Mittel, spezifische Codeteil ein Form1.Designer.cs automatisch zu bearbeiten. Manuell würden Sie dazu erheblich länger brauchen.

... macht die IDE das

Jedes Mal, wenn Sie in der IDE eine Änderung vornehmen, ändert diese den Code. Das bedeutet, dass sie die Dateien ändert, die den Code enthalten. Manchmal verändert sie ein paar Zeilen, manchmal fügt sie Ihrem Projekt aber auch ganze Dateien hinzu.

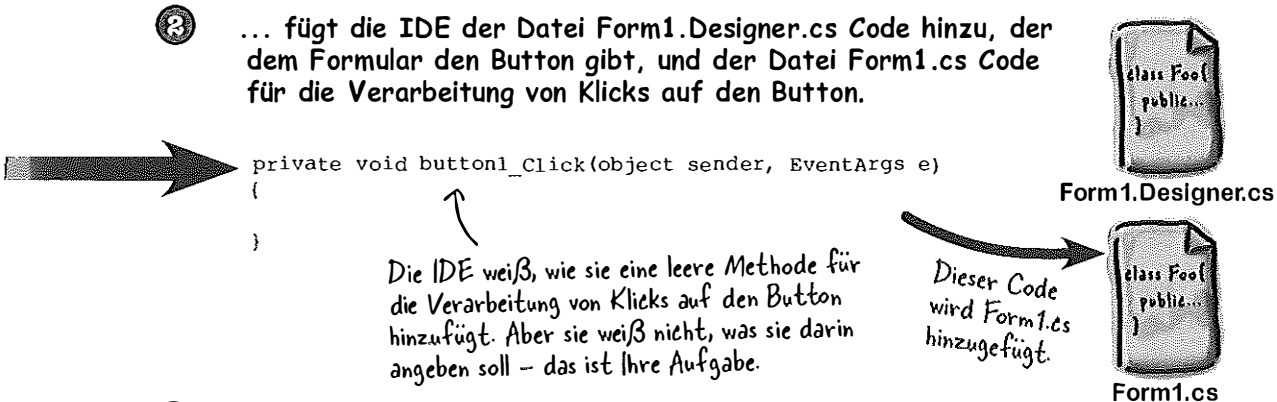
1 **... erstellt die IDE Dateien und Ordner für das Projekt.**

Diese Dateien werden auf Basis einer vordefinierten Schablone erstellt, die den grundlegenden Code für die Erstellung und Anzeige eines Formulars enthält.



WindowsApplication1.csproj | Form1.cs | Form1.Designer.cs | Program.cs | Eigenschaften

2 **... fügt die IDE der Datei Form1.Designer.cs Code hinzu, der dem Formular den Button gibt, und der Datei Form1.cs Code für die Verarbeitung von Klicks auf den Button.**




```
private void button1_Click(object sender, EventArgs e)
{

}
```

Form1.Designer.cs

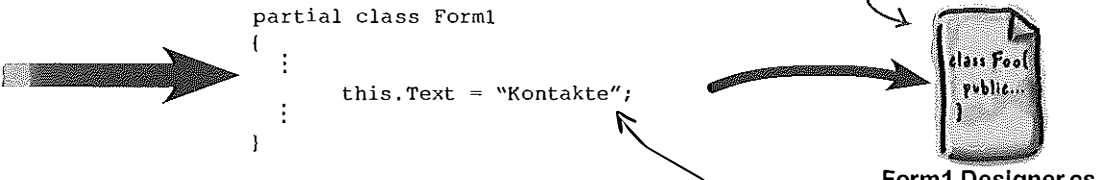
Die IDE weiß, wie sie eine leere Methode für die Verarbeitung von Klicks auf den Button hinzufügt. Aber sie weiß nicht, was sie darin angeben soll – das ist Ihre Aufgabe.

Dieser Code wird Form1.cs hinzugefügt.

Form1.cs

3 **... öffnet die IDE die Datei Form1.Designer.cs und aktualisiert eine Zeile Code.**

Die IDE geht zu dieser Datei ...




```
partial class Form1
{
  ⋮
      this.Text = "Kontakte";
  ⋮
}
```

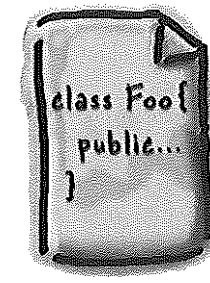
Form1.Designer.cs

... und aktualisiert diese Zeile Code.

Wo die Programme herkommen

Ein C#-Programm mag als Anweisungen in einem Haufen von Dateien beginnen, endet aber als ein Programm, das auf Ihrem Rechner läuft. Hier erfahren Sie, wie es dorthin kommt.

Jedes Programm beginnt als Sammlung von Quellcodedateien.



Sie haben bereits gesehen, wie man ein Programm bearbeitet und wie die IDE Ihr Programm in Dateien in einem Ordner speichert. Diese Dateien **sind** Ihr Programm – Sie können sie in einen neuen Ordner kopieren und von dort öffnen und werden alles darin finden: Formulare, Ressourcen, Code und alles andere, das Sie Ihrem Projekt hinzugefügt haben.

Sie können sich die IDE als eine Art ausgefallenen Datei-Editor vorstellen. Sie kümmert sich für Sie automatisch um Einrückungen, ändert die Farben von Schlüsselwörtern, ergänzt schließende Klammern und schlägt sogar vor, welche Wörter als Nächstes kommen könnten. Aber eigentlich macht sie nicht mehr, als die Dateien zu bearbeiten, die Ihr Programm enthalten.

Sie können Ihr Programm auch im Editor erstellen, aber das würde erheblich mehr Zeit in Anspruch nehmen.

Die IDE bündelt alle Dateien für Ihr Programm zu einer **Projektmappe**, indem sie eine Projektmappendatei (.sln) und einen Order erstellt, der die anderen Dateien für das Programm enthält. Die Projektmappendatei enthält eine Liste der Projektdateien in der Projektmappe (die mit .csproj enden), und die Projektdateien enthalten eine Liste aller anderen Dateien, die mit dem Programm verknüpft sind. In diesem Buch werden Sie Projektmappen erstellen, die nur ein Projekt enthalten, aber Sie können einer Projektmappe über den Projektmappen-Explorer der IDE leicht weitere Projekte hinzufügen.

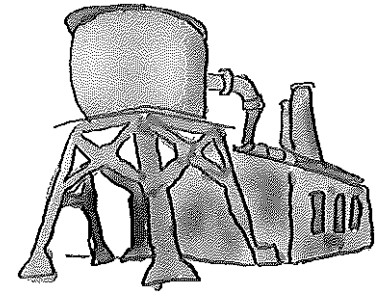
Das .NET Framework gibt Ihnen die erforderlichen Werkzeuge.



C# ist nur eine Sprache – für sich selbst kann sie eigentlich ***nichts*** tun. Und hier kommt das **.NET Framework** ins Spiel. Erinnern Sie sich an diesen Maximieren-Button, den Sie im Kontakte-Formular abgeschaltet haben? Wenn Sie auf den Maximieren-Button eines Fensters klicken, ist da Code, der dem Fenster sagt, wie es sich selbst maximieren und den ganzen Bildschirm einnehmen soll. Dieser Code ist Teil des .NET Framework. Buttons, Checkboxen, Listen ... all das sind Teile des .NET Framework. Gleiches gilt für die internen Teile, die Ihr Formular mit der Datenbank verknüpfen. Es hat Werkzeuge zum Zeichnen von Grafiken, zum Lesen und Schreiben von Dateien, zur Verwaltung von Auflistungen ... alle möglichen Arten von Werkzeugen für viele Aufgaben, die Programmierer täglich erledigen müssen.

Die Werkzeuge im .NET Framework sind in **Namensräume** eingeteilt. Diese Namensräume haben Sie bereits in den »using«-Zeilen am Anfang Ihres Codes gesehen. Ein Namensraum heißt System.Windows.Forms – aus ihm stammen Ihre Buttons, Checkboxen und Formulare. Jedes Mal, wenn Sie ein neues Windows-Anwendung-Projekt erstellen, fügt die IDE die erforderlichen Dateien hinzu, damit Ihr Projekt ein Formular enthält und diese Dateien am Anfang die Zeile »`using System.Windows.Forms;`« enthalten.

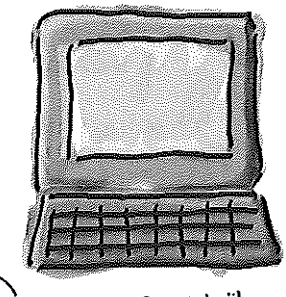
Erstellen Sie das Programm, um eine ausführbare Programmdatei zu erzeugen.



Wählen Sie im Erstellen-Menü »Projektmappe erstellen«, **kompiliert** die IDE Ihr Programm. Das macht sie, indem sie den **Compiler** aufruft. Dieser ist ein Werkzeug, das den Quellcode Ihres Programms liest und in eine **ausführbare Programmdatei** verwandelt. Die Programmdatei ist eine Datei auf Ihrer Festplatte, die mit `.exe` endet – auf diese klicken Sie doppelt, um Ihr Programm auszuführen. Wenn Sie das Programm kompilieren, wird die Programmdatei im Ordner `bin` gespeichert, der sich im Projektordner befindet. Wenn Sie Ihre Projektmappe veröffentlichen, wird die Programmdatei (gemeinsam mit allen anderen erforderlichen Dateien) in den Ordner kopiert, in dem Sie es veröffentlichen.

Wählen Sie »Debuggen starten« im Debuggen-Menü, kompiliert die IDE Ihr Programm und führt die Programmdatei aus. Die IDE bietet noch fortgeschrittenere Werkzeuge für das **Debuggen** Ihres Programms. Diese ermöglichen Ihnen, das Programm auszuführen und es dabei anzuhalten (zu »unterbrechen«), damit Sie herausfinden können, was vor sich geht.

Ihr Programm läuft in der CLR.



Momentan müssen Sie sich über die CLR noch nicht so viele Gedanken machen. Es reicht, wenn Sie wissen, dass sie da ist und sich für Sie darum kümmert, Ihr Programm automatisch auszuführen. Während wir weitergehen, werden Sie noch viel mehr über die CLR lernen.

Ein Doppelklick auf die Programmdatei bewirkt, das Windows Ihr Programm ausführt. Aber es gibt eine zusätzliche »Schicht« zwischen Windows und Ihrem Programm, die als **Common Language Runtime** oder CLR bezeichnet wird. Früher, und so lange ist das noch gar nicht her (aber bevor es C# gab), war das Programmieren schwerer, weil man sich mit Hardware- und anderen elementaren Rechnerfragen herumschlagen musste. Man wusste nie so genau, wie die Leute ihre Rechner konfiguriert hatten. Die CLR – die häufig als eine **virtuelle Maschine** bezeichnet wird – kümmert sich für Sie um all diese Dinge, indem sie eine Art »Übersetzung« zwischen Ihrem Programm und dem Computer vornimmt, auf dem es läuft.

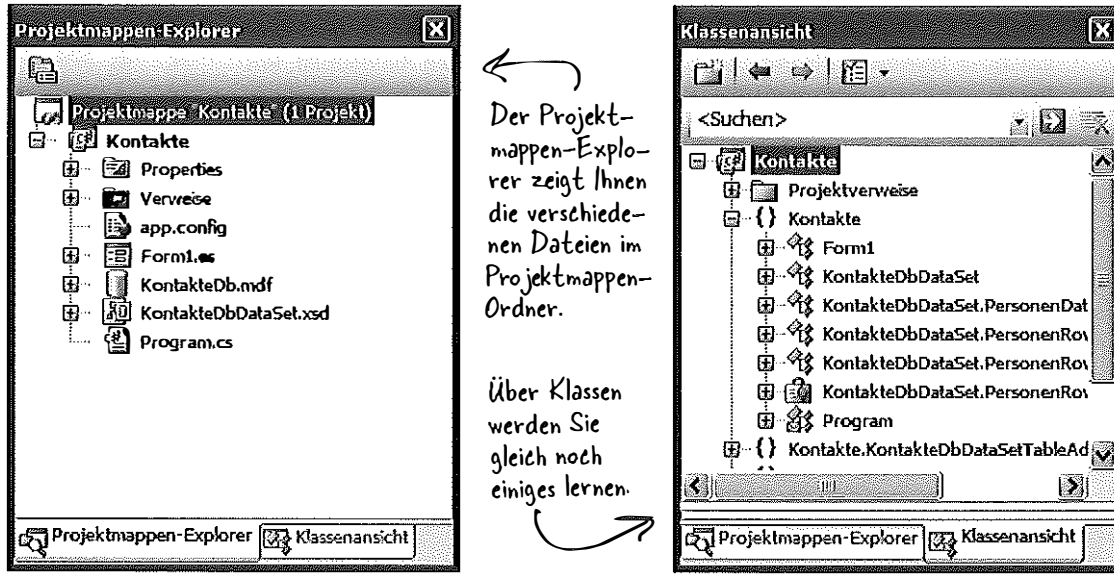
Sie werden all die Dinge kennenlernen, die die CLR für Sie macht. Beispielsweise bietet sie eine präzise Verwaltung des Speichers Ihres Computers, indem sie ermittelt, welche Daten Ihr Programm nicht mehr benötigt, und diese dann für Sie entfernt. Das gehört zu den Dingen, die Programmierer früher selbst erledigen mussten, mit denen Sie sich jetzt aber nicht mehr herumschlagen müssen. Sie werden es nicht immer wissen, aber die CLR wird Ihnen das Erlernen von C# erheblich erleichtern.

Die IDE hilft Ihnen beim Programmieren

Sie haben bereits ein paar der Dinge gesehen, die die IDE tun kann.
Schauen wir uns einige der Werkzeuge, die sie uns gibt, genauer an.

Der Projektmappen-Explorer zeigt Ihnen alles in Ihrem Projekt.

Sie werden eine Menge Zeit damit verbringen, zwischen Klassen hin- und herzuwechseln. Am einfachsten geht das, indem Sie den Projektmappen-Explorer verwenden. Er bietet zwei Ansichten: eine Projektmappen-Explorer-Ansicht (die Ihnen die Dateien in Ihrem Projekt zeigt) und ein Klassenansicht (die Ihnen zeigt, wie Ihr Code logisch in Klassen gegliedert ist).



Nutzen Sie die Registerkarten, um zwischen geöffneten Dateien zu wechseln.

Da Ihr Programm auf mehrere Dateien verteilt ist, haben Sie meist mehrere Codedateien gleichzeitig geöffnet. Jede wird dann in einer eigenen Registerkarte des Codeeditors angezeigt. Die IDE zeigt neben dem Dateinamen einen Asterisk (*) an, wenn die Datei noch nicht gespeichert wurde.

Hier ist die Ressourcendatei, der Sie das Logo der Papierfabrik Objekthausen hinzugefügt haben.

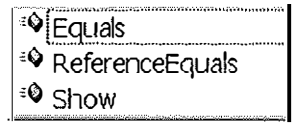


Beim Arbeiten an einem Formular können Sie gleichzeitig zwei Dateien dafür geöffnet haben – eine für den Formularentwurf und eine für den Formularcode.

Die IDE hilft Ihnen, Code zu schreiben.

Haben Sie das kleine Fenster bemerkt, das aufgesprungen ist, als Sie in der IDE Code eingegeben haben? Dieses Feature heißt IntelliSense, und es ist wirklich praktisch. Beispielsweise zeigt es Ihnen mögliche Vervollständigungen der aktuellen Codezeile. Geben Sie `MessageBox` und einen Punkt ein, weiß es, dass es drei zulässige Möglichkeiten gibt, diese Zeile zu vervollständigen:

```
MessageBox.
```

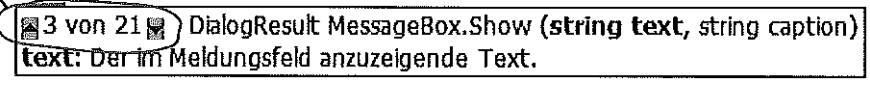


Die IDE weiß, dass MessageBox drei Methoden namens Equals, ReferenceEquals und Show hat. Geben Sie S ein, wählt sie Show. Drücken Sie die Tabulator- oder Enter-Taste, um den Vorschlag auszuwählen. Das kann Ihnen viel Zeit sparen, wenn Sie diverse lange Methodennamen eingeben müssen.

Wählen Sie Show und tippen dann `(`, zeigt IntelliSense Ihnen Informationen zu den Möglichkeiten, die Zeile zu vervollständigen:

Das heißt, dass es 21 verschiedene Möglichkeiten gibt, die Show-Methode von MessageBox aufzurufen (also Wege, unterschiedliche Buttons und Symbole anzuzeigen).

```
MessageBox.Show(
```



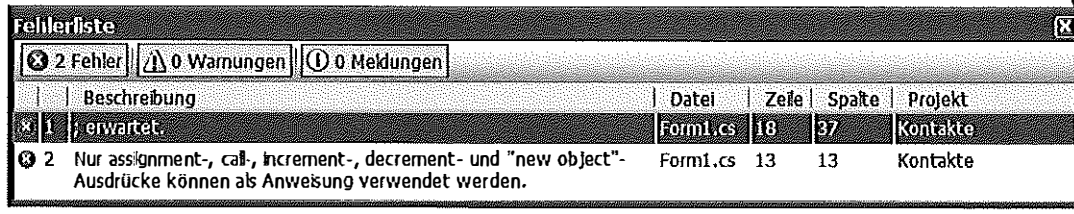
Außerdem besitzt die IDE Kurzformen, die als **Codeausschnitte** bezeichnet werden. Sie müssen nur eine Abkürzung eingeben, und die IDE fügt dann den Rest des Codes ein. Hier ist ein nützlicher Codeausschnitt: Geben Sie `mbox` ein und drücken Sie dann zweimal die Tabulatortaste. Sie füllt anschließend die Methode `MessageBox.Show` für Sie aus:

```
MessageBox.Show("Test");
```

Starten Sie Ihr Programm in der IDE mit Debuggen starten, kompiliert diese zunächst Ihr Programm. Ist das Kompilieren erfolgreich, wird Ihr Programm ausgeführt, ist es das nicht, werden Ihnen stattdessen die Fehler angezeigt.

Die Fehlerliste hilft Ihnen, Compiler-Fehler zu beheben.

Sollten Sie noch nicht herausgefunden haben, wie leicht man in C# Tippfehler machen kann, werden Sie das sicher bald! Glücklicherweise gibt Ihnen die IDE ein gutes Mittel, diese zu korrigieren. Wenn Sie Ihre Projektmappe erstellen, werden alle Fehler, die eine Kompilierung verhindern, im Fenster Fehlerliste unten in der IDE angezeigt:



Klicken Sie doppelt auf einen Fehler, springt die IDE zum Problem im Code:

```
private void pictureBox1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Adressbuch 1.0.")
}
```

Die IDE zeigt einen roten Unterstrich an der Stelle, an der Ihrer Meinung nach ein Semikolon fehlt.

Wenn Sie Dinge in der IDE ändern, ändern Sie auch Ihren Code

Sehen Sie dieses >>Tun Sie das!<<, sollten Sie die IDE öffnen und die Schritte nacharbeiten. Wir werden Ihnen genau sagen, was Sie tun sollen, und auf das hinweisen, worauf Sie achten müssen, um am meisten aus dem jeweiligen Beispiel zu lernen.

Tun Sie das!

Visuellen Code kann die IDE gut für Sie schreiben. Aber verlassen Sie sich nicht einfach auf unser Wort. Öffnen Sie Visual Studio, **erzeugen Sie ein neues Windows-Anwendung-Projekt** und sehen Sie selbst.

1. **Öffnen Sie den Designer-Code.**
Öffnen Sie die Datei `Form1.Designer.cs` in der IDE. Aber statt sie im Form-Designer zu öffnen, öffnen Sie diesmal den Code, indem Sie im Projektmappen-Explorer darauf mit rechts klicken und »View Code« wählen. Suchen Sie nach der Klassendeklaration für `Form1`:

```
partial class Form1
```

Haben Sie gesehen, dass es eine partielle Klasse ist?

2. **Öffnen Sie den Form-Designer und fügen Sie Ihrem Formular eine PictureBox hinzu.**
Gewöhnen Sie sich daran, mit mehreren Registerkarten zu arbeiten. Gehen Sie in den Projektmappen-Explorer und öffnen Sie den Formular-Designer, indem Sie auf `Form1.cs` doppelklicken. **Ziehen Sie eine neue PictureBox** auf das neue Formular.

3. **Suchen Sie den vom Designer generierten Code für das PictureBox-Steuerelement.**
Kehren Sie dann in der IDE zur Registerkarte `Form1.Designer.cs` zurück. Scrollen Sie nach unten und suchen Sie nach dieser Codezeile:

Klicken Sie auf das Pluszeichen.

```
⊞   Vom Windows Form-Designer generierter Code
```

Klicken Sie links des Codes auf das +, um den Code zu expandieren. Scrollen Sie nach unten und suchen Sie nach diesen Zeilen:

```
//
// pictureBox1
//

this.pictureBox1.Location = new System.Drawing.Point(276, 28);

this.pictureBox1.Name = "pictureBox1";

this.pictureBox1.Size = new System.Drawing.Size(100, 50);

this.pictureBox1.TabIndex = 1;

this.pictureBox1.TabStop = false;
```

Machen Sie sich keine Sorgen, wenn in Ihrem Code bei den Location- und Size-Zeilen etwas andere Werte stehen ...

Halt, halt! Was soll das heißen?

Scrollen Sie einen Augenblick nach oben. Oben im Abschnitt mit dem vom Windows Form-Designer generierten Code stehen folgende Zeilen.

```
/// <summary>
/// Erforderliche Methode für die Designerunterstützung. Der Inhalt
/// der Methode darf nicht mit dem Code-Editor geändert werden.
/// </summary>
```

Die meisten Kommentare beginnen mit nur zwei Schrägstrichen (//). Aber manchmal fügt die IDE diese Kommentare mit drei Schrägstrichen ein.

Es gibt nichts, was ein Kind so verführerisch findet wie ein großes Schild mit der Aufschrift »Nicht berühren!«. Na los, wir wissen doch, dass es Sie in den Fingern juckt … lassen Sie uns den Inhalt dieser Methode mit dem Code-Editor verändern! **Fügen Sie Ihrem Formular einen Button hinzu und machen Sie dann Folgendes:**

1. **Ändern Sie den Code, der die Eigenschaft button1.Text setzt. Was, glauben Sie, passiert jetzt mit dem Eigenschaften-Fenster der IDE?**
   Probieren Sie es aus – schauen Sie sich an, was passiert! Kehren Sie jetzt in den Form-Designer zurück und prüfen Sie die Eigenschaft Text. Hat Sie sich geändert?

2. **Bleiben Sie im Designer und nutzen Sie das Eigenschaften-Fenster, um die Name-Eigenschaft zu ändern.**
   Schauen Sie, ob Sie die IDE dazu bringen können, die Eigenschaft Name zu ändern. Sie finden sie ganz oben im Fenster Eigenschaften direkt unter »(Name)«. Was passiert mit dem Code? Was mit dem Kommentar im Code?

3. **Ändern Sie den Code, der die Eigenschaft Location auf (0,0) setzt. Ändern Sie die Eigenschaft Size, um den Button richtig groß zu machen.**
   Hat es funktioniert?

4. **Kehren Sie in den Designer zurück und ändern Sie die BackColor-Eigenschaft des Buttons.**
   Sehen Sie sich den Code von Form1.Designer.cs genau an. Wurden Zeilen hinzugefügt?

Sie müssen das Formular nicht speichern oder das Programm ausführen, um die Änderungen zu sehen. Nehmen Sie einfach die Änderungen im Code-Editor vor und klicken Sie dann auf die Designer-Registerkarte, um in den Form-Designer zu wechseln – die Änderungen sollten sofort sichtbar werden.

Es ist immer leichter, den vom Form-Designer generierten Code mit der IDE zu ändern. Aber wenn Sie das tun, führt jede Änderung in der IDE zu einer Änderung am Code Ihres Projekts.

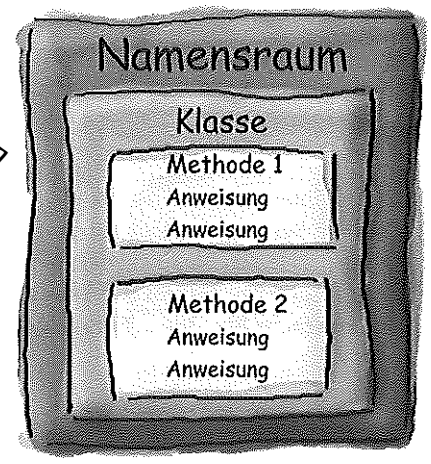
Anatomie eines Programms

Der Code aller C#-Programme ist auf genau die gleiche Weise strukturiert. Alle Programme nutzen **Namensräume, Klassen** und **Methoden,** um die Verwaltung des Codes zu vereinfachen.

Jedes Mal, wenn Sie ein neues Programm erstellen, definieren Sie einen Namensraum dafür, damit sein Code von dem der Klassen des .NET Framework getrennt ist.

Eine Klasse enthält einen Teil Ihres Programms (obwohl einige sehr kleine Programme auch nur eine Klasse enthalten).

Eine Klasse enthält eine oder mehrere Methoden. Ihre Methoden müssen immer innerhalb einer Klasse existieren. Und Methoden bestehen aus Anweisungen – wie denen, die Sie bereits gesehen haben.



Sehen wir uns Ihren Code genauer an

Öffnen Sie den Code der `Form1.cs`-Datei Ihres Kontakte-Projekts, damit wir ihn schrittweise durchgehen können.

1 **Die Codedatei beginnt damit, dass sie .NET Framework-Werkzeuge referenziert.**

Am Anfang jeder Datei finden Sie eine Gruppe von `using`-Zeilen. Sie sagen C#, welche Teile des .NET Framework verwendet werden sollen. Brauchen Sie andere Klassen, die sich in anderen Namensräumen befinden, fügen Sie auch `using`-Zeilen für diese hinzu. Da Formulare häufig viele unterschiedliche Werkzeuge aus dem .NET Framework einsetzen, fügt die IDE automatisch einen Haufen `using`-Zeilen hinzu, wenn sie ein Formular erzeugt und Ihrem Projekt hinzufügt.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
```

Diese `using`-Zeilen stehen am Anfang jeder Codedatei. Sie sagen C#, dass es all diese Klassen des .NET Framework verwenden soll. Jede sagt dem Programm, dass die Klassen in dieser spezifischen `.cs`-Datei alle Klassen in einem bestimmten Namensraum des .NET Framework verwenden werden.

2 C#-Programme sind aus Klassen aufgebaut.

Jedes C#-Programm ist aus **Klassen** aufgebaut. Eine Klasse kann beliebige Dinge tun, aber die meisten Klassen tun eine bestimmte Sache. Als Sie das neue Programm erstellt haben, hat die IDE eine Klasse namens Form1 hinzufügt, die ein Formular anzeigt.

```
namespace Contacts
{
  public partial class Form1 : Form
  {
```

Da Sie dem Programm den Namen Kontakte gaben, erzeugte die IDE einen Kontakte-Namensraum dafür, indem sie am Anfang der Codedatei eine namespace-Zeile einfügte. Alles innerhalb ihrer geschweiften Klammern ist ein Bestandteil des Namensraums Kontakte.

Diese Klasse heißt Form1. Sie enthält den gesamten Code zum Zeichnen des Formulars und der Toolbox-Steuerelemente darauf. Die IDE erzeugte es, als Sie sie anwiesen, ein neues Windows-Anwendung-Projekt zu erstellen.

Achten Sie auf Paare geschweifter Klammern. Jedes { wird irgendwann von einem } geschlossen. Paare können in andere eingebettet sein.

3 Klassen enthalten Methoden, die Aktionen ausführen.

Muss eine Klasse etwas tun, nutzt sie eine **Methode**. Eine Methode nimmt Eingaben an, führt eine Aktion aus und erzeugt manchmal eine Ausgabe. Eingaben übergeben Sie einer Methode mit **Parametern**. Je nach Eingabe können sich Methoden unterschiedlich verhalten. Einige Methoden erzeugen Ausgaben, die man als **Rückgabewert** bezeichnet. Sehen Sie vor einer Methode das Schlüsselwort `void`, bedeutet das, dass sie nichts zurücklicfert.

```
    public Form1()
    {
      InitializeComponent();
    }
```

Diese Zeile ruft eine Methode auf, die InitializeComponent() heißt und ebenfalls von der IDE für Sie erzeugt wurde.

4 Eine Anweisung führt eine einzige Aktion aus.

Als Sie Ihrem Programm die Zeile MessageBox.Show() hinzufügten, haben Sie ihm eine **Anweisung** gegeben. Jede Methode besteht aus Anweisungen. Rufi Ihr Programm eine Methode auf, führt es die erste Anweisung in der Methode aus, dann die zweite usw. Sind alle Anweisungen der Methode abgearbeitet oder stößt sie auf eine `return`-Anweisung, endet sie, und das Programm nimmt die Verarbeitung hinter der Anweisung wieder auf, von der die Methode ursprünglich aufgerufen wurde.

Das ist eine Methode namens pictureBox1_Click(), die aufgerufen wird, wenn der Benutzer auf die PictureBox kliekt.

Diese Methode hat zwei Parameter mit den Namen `sender` und `e`.

```
    private void pictureBox1_Click(object sender, EventArgs e)
    {
      MessageBox.Show("Contact List 1.0", "About");
    }
  }
}
```

Das ist eine Anweisung. Sie wissen bereits, was sie macht – sie blendet ein kleines Nachrichtenfenster ein.

Ihre Anweisung ruft die Show()-Methode auf, die Teil der Klasse MessageBox ist, die sich im Namensraum System.Windows.Forms befindet.

Ihre Anweisung übergibt der Methode Show() zwei Parameter. Der erste ist der Text, der in der MessageBox angezeigt wird, und der zweite ist der Text, der in der Titelleiste angezeigt wird.

Ihr Programm weiß, wo es anfangen muss

Jedes C#-Programm kann nur eine Einstiegspunkt-Methode haben. Diese heißt immer Main(). So weiß es, wo es anfangen muss.

Als Sie die neue Windows-Anwendung-Projektmappe erzeugten, generierte die IDE unter anderem die Datei **Program.cs**. Gehen Sie in den Projektmappen-Explorer und klicken Sie doppelt darauf. Die Datei enthält eine Klasse namens Program, und in der Klasse ist eine Methode namens Main(). Diese Methode ist der **Einstiegspunkt**. Das bedeutet, dass sie das Allerste ist, was in Ihrer Anwendung ausgeführt wird.

Hier ist etwas Code, den die IDE im letzten Kapitel automatisch für Sie erzeugt hat. Sie finden ihn in Program.cs.



Ihr Code unter der Lupe

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Windows.Forms;

namespace Kontakte
{
    static class Program
    {
        /// <summary>
        /// Der Haupteinstiegspunkt für die Anwendung.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

1 2 3 4 5 6

Der Namensraum für diesen ganzen Code ist Kontakte. Über Namensräume werden wir auf den nächsten paar Seiten noch ausführlicher reden.

Zeilen, die mit Schrägstrichen beginnen, sind Kommentare. Diese können Sie an beliebiger Stelle einfügen. Die Schrägstriche sagen C#, dass es die Zeilen ignorieren soll.

Jedes Mal, wenn Sie Ihr Programm ausführen, beginnt es hier – am Einstiegspunkt.

Diese Anweisung erzeugt das Kontakte-Formular, zeigt es an und beendet das Programm, wenn das Formular geschlossen wird.

Ich deklariere!

Die erste Zeile jeder Klasse oder Methode nennt man Deklaration.

Denken Sie daran, dass das nur der Punkt ist, an dem Sie beginnen, sich in Ihrem Code nach unten zu bewegen. Aber bevor Sie das tun, müssen Sie wissen, was Sie sich ansehen.

1 **C# und .NET haben viele eingebaute Features.**
Zeilen wie diese finden Sie am Anfang fast jeder C#-Klassendatei. `System.Windows.Forms` ist ein **Namensraum**. Die Zeile `using System.Windows.Forms` macht Ihrem Programm alles verfügbar, das sich in diesem Namensraum befindet. Dieser spezielle Namensraum enthält viele grafische Elemente wie Buttons und Formulare.

Ihre Programme werden mehr und mehr Namensräume wie diesen nutzen, während Sie in diesem Buch mehr über C# und die anderen eingebauten Features von .NET lernen.

2 **Die IDE wählt einen Namensraum für Ihren Code.**
Hier ist der Namensraum, den die IDE für Sie erstellt hat – auf Basis des Namens Ihres Projekts hat sie `Kontakte` gewählt. Der gesamte Code in Ihrem Programm existiert in diesem Namensraum.

Namensräume ermöglichen es Ihnen, den gleichen Namen in unterschiedlichen Programmen zu verwenden, solange sich diese Programme nicht auch in diesem Namensraum befinden.

3 **Ihr Code wird in einer Klasse gespeichert.**
Diese spezielle Klasse heißt `Program`. Die IDE hat sie erzeugt und ihr Code hinzugefügt, der das Programm startet und das Kontakte-Formular öffnet.

In einem Namensraum können Sie mehrere Klassen haben.

4 **Dieser Code hat eine Methode, die drei Anweisungen enthält.**
Ein Namensraum enthält Klassen, und Klassen enthalten Methoden. In jeder Methode gibt es einen Satz von Anweisungen. In diesem Programm kümmern sich die Anweisungen darum, das Kontakte-Formular zu öffnen. In Methoden werden Aktionen abgewickelt – jede Methode **macht** etwas.

5 **Jedes Programm hat eine spezielle Methode, die als Einstiegspunkt bezeichnet wird.**
Jedes C#-Programm **muss** genau eine Methode mit dem Namen `Main` haben. Obgleich Ihr Programm viele Methoden haben kann, kann nur eine die erste sein, die ausgeführt wird, und das ist die `Main`-Methode. C# prüft jede Klasse in Ihrem Code auf eine Methode, die als `static void Main()` deklariert ist. Wenn das Programm ausgeführt wird, wird die erste Anweisung in dieser Methode ausgeführt und alles, was auf diese erste Anweisung folgt.

Jedes C#-Programm muss genau eine Methode namens Main haben. Diese Methode ist der Einstiegspunkt für Ihren Code.

Führen Sie Ihren Code aus, wird ZUERST der Code in Main() ausgeführt.

Sie können den Einstiegspunkt Ihres Programms ändern

Solange Ihr Programm einen Einstiegspunkt hat, spielt es keine Rolle, in welcher Klasse sich diese Einstiegspunktmethode befindet oder was sie macht. Entfernen wir die `Main`-Methode aus `Program.cs` und erzeugen wir einen neuen Einstiegspunkt.



1. Kehren Sie zu `Program.cs` zurück und ändern Sie den Namen der `Main`-Methode in `NichtMain`. **Probieren Sie jetzt,** das Programm **zu erstellen und auszuführen**. Was passiert?

Schreiben Sie auf, was passiert ist, als Sie den Methodennamen geändert haben, und was Ihrer Meinung nach der Grund dafür ist.

2. Erstellen Sie jetzt einen neuen Einstiegspunkt. **Fügen Sie eine neue Klasse hinzu** und geben Sie ihr den Namen `AndereKlasse.cs`. Das machen Sie, indem Sie im Projektmappen-Explorer mit rechts auf den Projektnamen klicken und »Hinzufügen→Klasse...« wählen. Nennen Sie die Klassendatei `AndereKlasse.cs`. Die IDE fügt Ihrem Programm die Klasse `AndereKlasse` hinzu. Dies ist die hinzugefügte Datei:

Klicken Sie im Projektmappen-Explorer mit rechts auf das Projekt und wählen Sie >>Hinzufügen<< und >>Klasse...<<.

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;

namespace Kontakte
{
    class AndereKlasse
    {
    }
}
```

Der Datei wurden diese vier Standard-`using`-Zeilen hinzugefügt.

Auch diese Klasse befindet sich im Namensraum Kontakte, den die IDE hinzufügte, als Sie das Windows-Anwendung-Projekt erstellten.

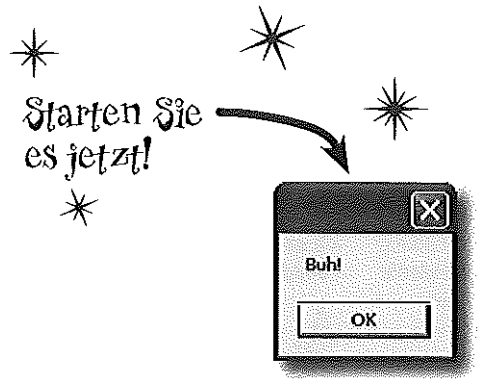
Die IDE gab der Klasse auf Basis des Dateinamens automatisch diesen Namen.

3. Fügen Sie am Anfang der Datei eine neue using-Zeile ein: **`using System.Windows.Forms;`**. Vergessen Sie nicht, die Zeile mit einem Semikolon abzuschließen!

4. Fügen Sie der Klasse **`AndereKlasse`** zwischen den geschweiften Klammern diese Methode hinzu.

```
class AndereKlasse
{
    public static void Main()
    {
        MessageBox.Show("Buh!");
    }
}
```

MessageBox ist eine Klasse im Namensraum System.Windows.Forms. Deswegen mussten Sie in Schritt 3 die using-Zeile hinzufügen. Show() ist eine Methode, die Teil der Klasse MessageBox ist.



Und was ist passiert?

Statt die Adressbuchanwendung zu öffnen, zeigt Ihr Programm jetzt dieses Meldungsfenster an. Als Sie die neue Main()-Methode erstellt haben, haben Sie Ihrem Programm einen neuen Einstiegspunkt gegeben. Jetzt führt das Programm als Erstes die Anweisung in dieser Methode aus – und das ist diese MessageBox.Show()-Anweisung. In der Methode gibt es sonst nichts. Klicken Sie auf den OK-Button, hat das Programm keine Anweisungen mehr auszuführen und endet.

5 Finden Sie heraus, wie Sie das Programm ändern müssen, damit es wieder die Adressbuchanwendung öffnet.

Hinweis: Sie müssen nur zwei Zeilen in zwei Dateien ändern.

Spitzen Sie Ihren Bleistift

Ergänzen Sie in dieser C#-Datei jeweils eine Beschreibung der Zeile, auf die ein Pfeil zeigt. Die erste haben wir für Sie erledigt.

```
using System;
using System.Linq;
using System.Text;
using System.Windows.Forms;


namespace EinNamensraum
{
      class MeineKlasse {
            public static void MachWas() {
                  MessageBox.Show("Das ist eine Nachricht!");
            }
      }
}
```

C#-Klassen nutzen diese >>using<<-Zeilen, um Methoden aus anderen Namensräumen hinzuzufügen.

Es gibt keine Dummen Fragen

F: Was sollen all diese geschweiften Klammern?

A: C# nutzt geschweifte Klammern, um Anweisungen zu **Blöcken** zu gruppieren. Geschweifte Klammern treten immer paarweise auf. Eine schließende geschweifte Klammer folgt immer auf eine öffnende. Die IDE hilft Ihnen, geschweifte Klammern zuzuordnen – klicken Sie auf eine, wird ihr Gegenstück hervorgehoben, indem es dunkler schattiert wird.

F: Ich verstehe noch nicht so recht, was ein Einstiegspunkt ist. Können Sie das noch einmal erklären?

A: Ihr Programm enthält einen ganzen Haufen an Anweisungen, aber diese werden nicht alle auf einmal ausgeführt. Das Programm beginnt mit seiner ersten Anwendung und geht dann zur nächsten über und wieder zur nächsten und so weiter. Diese Anweisungen sind üblicherweise in diverse von Klassen gegliedert. Die Frage ist also, woher Ihr Programm weiß, mit welcher Anweisung es beginnen soll, wenn Sie es starten.

Hier kommt der Einstiegspunkt ins Spiel. Der Compiler erstellt Ihren Code nur dann, wenn er **genau eine Methode mit dem Namen Main()** enthält. Diese Methode bezeichnen wir als den Einstiegspunkt. Das Programm beginnt die Ausführung mit der ersten Anweisung in Main().

F: Wie kommt es, dass Ich die Fehler im Fenster Fehlerliste erhalte, wenn Ich versuche, mein Programm auszuführen? Ich dachte, das passiert nur, wenn Ich »Bulld Projektmappe« verwende.

A: Weil die IDE, wenn Sie im Menü »Start Debugging« wählen oder den Symbolleisten-Button anklicken, als Erstes alle Dateien in der Projektmappe speichert und dann versucht, sie zu kompilieren. Und wenn Sie Ihren Code kompilieren – ganz gleich, ob das passiert, wenn Sie ihn ausführen oder die Projektmappe erstellen –, zeigt die IDE, wenn es Fehler gibt, diese in der Fehlerliste an und führt das Programm nicht aus.

Spitzen Sie Ihren Bleistift
Lösung

Ergänzen Sie in dieser C#-Datei jeweils eine Beschreibung der Zeile, auf die ein Pfeil zeigt. Die erste haben wir für Sie erledigt.

```
using System;
using System.Linq;
using System.Text;
using System.Windows.Forms;


namespace EinNamensraum
{
      class MeineKlasse {

            public static void MachWas() {

                  MessageBox.Show("Das ist eine Nachricht!");

            }

      }
}
```

C#-Klassen nutzen diese »using«-Zeilen, um Methoden aus anderen Namensräumen hinzuzufügen.

Code befindet sich immer in Klassen, deswegen braucht das Programm hier eine Klasse.

Das ist eine Methode. Jede Methode im Programm macht etwas. Methoden werden genutzt, um Anweisungen zu gruppieren.

Das ist eine Anweisung. Wird sie ausgeführt, wird ein kleines Fenster mit einer Meldung eingeblendet.

WAS IST MEINE ABSICHT?

Ordnen Sie jedem dieser von der IDE generierten Codefragmente die passende Beschreibung zu. (Es sind ein paar neue Dinge darunter – geben Sie einen Tipp ab und schauen Sie, ob Sie richtig lagen.

```
partial class Form1
{
  :
  this.BackColor = Color.DarkViolet;
  :
}
```

```
// Diese Schleife wird dreimal ausgeführt.
```

```
partial class Form1
{
  private void InitializeComponent()
  {
  :
  }
}
```

```
anzahl_boxenstoppsLabel.Name
        = "anzahl_boxenstoppsLabel";
anzahl_boxenstoppsLabel.Size
        = new System.Drawing.Size(135, 17);
anzahl_boxenstoppsLabel.Text
        = "Anzahl an Boxenstopps:";
```

```
/// <summary>
/// Bild von Rover einblenden,
/// wenn auf den Button geklickt wird.
/// </summary>
```

```
partial class Form1
{
  :
  this.MaximizeBox = false;
  :
}
```

Setzt Eigenschaften für ein Label.

Nichts – das ist ein vom Programmierer ergänzter Kommentar, der den Code anderen erklärt, die ihn lesen.

Deaktiviert das Maximieren-Symbol ( ) in der Titelleiste des Form1-Fensters.

Eine besondere Art von Kommentar, den die IDE nutzt, um zu erklären, was ein Codeblock macht.

Ändert die Hintergrundfarbe des Form1-Fensters.

Ein Codeblock, der ausgeführt wird, wenn ein Programm ein Form1-Fenster öffnet.



Ordnen Sie jedem dieser von der IDE generierten Codefragmente die passende Beschreibung zu. (Es sind ein paar neue Dinge darunter – geben Sie einen Tipp ab und schauen Sie, ob Sie richtig lagen.)

```
partial class Form1
{
  ⋮
  this.BackColor = Color.DarkViolet;
  ⋮
}
```

```
// Diese Schleife wird dreimal ausgeführt.
```

```
partial class Form1
{
  private void InitializeComponent()
  {
  ⋮
  }
}
```

```
anzahl_boxenstoppsLabel.Name
        = "anzahl_boxenstoppsLabel";
anzahl_boxenstoppsLabel.Size
        = new System.Drawing.Size(135, 17);
anzahl_boxenstoppsLabel.Text
        = "Anzahl an Boxenstopps:";
```

```
/// <summary>
/// Bild von Rover einblenden,
/// wenn auf den Button geklickt wird.
/// </summary>
```

```
partial class Form1
{
  ⋮
  this.MaximizeBox = false;
  ⋮
}
```

Setzt Eigenschaften für ein Label.

Nichts – das ist ein vom Programmierer ergänzter Kommentar, der den Code anderen erklärt, die ihn lesen.

Deaktiviert das Maximieren-Symbol ( ) in der Titelleiste des Form1-Fensters.

Eine besondere Art von Kommentar, den die IDE nutzt, um zu erklären, was ein Codeblock macht.

Ändert die Hintergrundfarbe des Form1-Fensters.

Ein Codeblock, der ausgeführt wird, wenn ein Programm ein Form1-Fenster öffnet.

Zwei Klassen können im gleichen Namensraum sein

Schauen Sie sich diese beiden Klassendateien aus einem Programm namens Tiere2 an. Sie enthalten drei Klassen: eine Klasse Hund, eine Klasse Katze und eine Klasse Fisch. Da sie alle im gleichen Namensraum Tiere2 sind, können Anweisungen in der Methode Hund.Bellen() die Methoden Katze.Miauen() und Fisch.Schwimmen() aufrufen. Es spielt keine Rolle, wie die verschiedenen Namensräume und Klassen in Dateien aufgeteilt sind. Wenn sie ausgeführt werden, agieren sie immer noch auf die gleiche Weise.

EinigeKlassen.cs

```
namespace Tiere2 {

  public class Hund {

    public void Bellen() {
      // Anweisungen
    }

  }

  public partial class Katze
{

    public void Miauen() {
      // Anweisungen
    }

  }
}
```

Wenn eine Klasse >>public<< ist, bedeutet das, dass alle anderen Klassen im Programm auf sie zugreifen können.

MehrKlassen.cs

```
namespace Tiere2 {

  public class Fisch {

    public void Schwimmen() {
      // Anweisungen
    }

  }

  public partial class Katze {

    public void Schnurren() {
      // Anweisungen
    }

  }
}
```

Da diese beiden Klassen im gleichen Namensraum sind, können sie sich gegenseitig >>sehen<< – obwohl sie in unterschiedlichen Dateien sind. Eine Klasse kann auch mehrere Dateien umspannen, aber dann müssen Sie das spezielle Schlüsselwort >>partial<< verwenden, wenn Sie sie deklarieren.

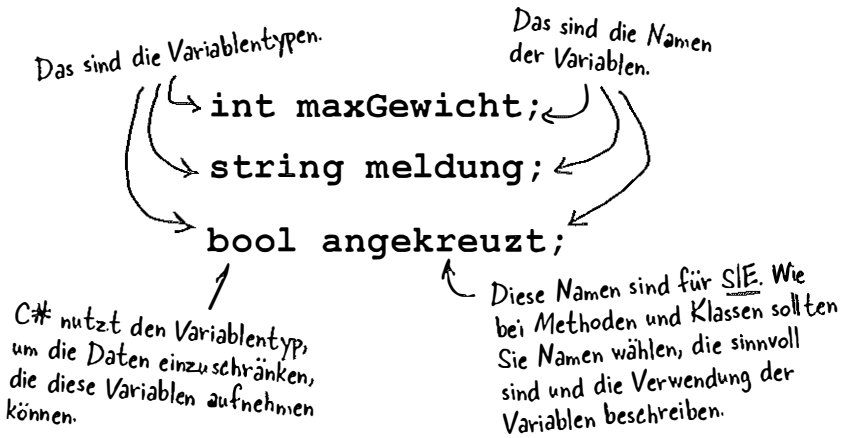
Sie können eine Klasse nur auf mehrere Dateien aufteilen, wenn Sie das Schlüsselwort `partial` verwenden. Sie werden das wahrscheinlich bei keinem Programm machen, das Sie in diesem Buch schreiben, aber die IDE nutzt es, um Ihr Formular in zwei Dateien aufzuteilen, Form1.cs und Form1.Designer.cs.

Ihre Programme nutzen Variablen, um mit Daten zu arbeiten

Wenn man es ganz simpel betrachtet, ist jedes Programm eigentlich nur ein Datenfresser. Manchmal haben diese Daten die Form eines Dokuments, manchmal die eines Bilds in einem Computerspiel oder einer Chat-Nachricht. Aber all das sind nur Daten. Und das ist der Punkt, an dem **Variablen** ins Spiel kommen. Variablen werden von Ihrem Programm genutzt, um diese Daten zu speichern.

Variablen deklarieren

Wenn Sie eine Variable **deklarieren**, teilen Sie Ihrem Programm den *Typ* und den *Namen* der Variablen mit. Kennt C# einmal den Typ Ihrer Variablen, verhindert es die Kompilierung Ihres Programms, wenn Sie einen Fehler gemacht haben und etwas zu tun versuchen, das sinnlos ist, beispielsweise »Fido« von 48353 abzuziehen.



Variablen variieren

Während Ihr Programm ausgeführt wird, entspricht eine Variable zu verschiedenen Zeiten unterschiedlichen Werten. Mit anderen Worten: Eine Variable ist ***variabel***. (Deswegen ist »Variable« ja auch ein so guter Name.) Das ist wirklich wichtig, weil dieser Grundgedanke der Kern jedes Programms ist, das Sie geschrieben haben und jemals schreiben werden. Nehmen wir an, Ihr Programm setzt die Variable `meinHöhe` auf 63:

```
int meinHöhe = 63;
```

Dann ersetzt C# `meinHöhe` jedes Mal, wenn es im Code erscheint, durch seinen Wert, 63. Ändern Sie den Wert später mal in 12

```
meinHöhe = 12;
```

ersetzt C# `meinHöhe` mit 12 – die Variable heißt aber immer noch `meinHöhe`.

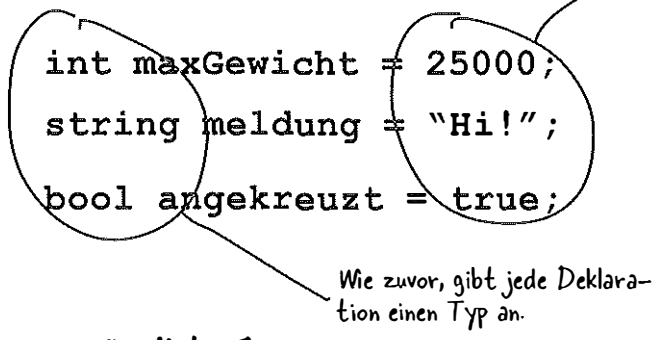
Immer wenn Ihr Programm Zahlen, Text, True/False-Werte oder irgendeine andere Art von Daten verwendet, nutzen Sie Variablen, um diese Daten festzuhalten.

Sie müssen Variablen Werte zuweisen, bevor Sie sie verwenden

Versuchen Sie, diese Anweisungen in ein C#-Programm einzufügen:

```
int z;
MessageBox.Show("Die Antwort ist " + z);
```

Legen Sie los und probieren Sie es aus. Sie erhalten eine Fehlermeldung, und die IDE weigert sich, Ihren Code zu kompilieren. Das liegt daran, dass die IDE jede Variable daraufhin prüft, ob Sie ihr einen Wert zugewiesen haben, bevor Sie sie verwenden. Dass Sie Ihren Variablen Werte zugewiesen haben, können Sie am leichtesten sicherstellen, indem Sie die Anweisung, die eine Variable deklariert, mit einer Anweisung kombinieren, die ihr einen Wert zuweist:



Ein paar nützliche Typen

Jede Variable hat einen Typ, der C# sagt, welche Arten von Daten sie speichern kann. In Kapitel 4 werden wir uns die verschiedenen Typen von C# noch genauer ansehen. Bis dahin beschränken wir uns auf die drei wichtigsten Typen. `int` speichert Integer (ganze Zahlen), `string` Text und `bool` Boolesche Wahr/Falsch-Werte.

va-ri-a-bel, Adjektiv.
veränderlich oder veränderbar
*Die **variable** Umdrehungszahl der Bohrmaschine ermöglicht Klaus, die Geschwindigkeit des Bohrers an den jeweiligen Werkstoff anzupassen.*

Wenn Sie Code schreiben, der eine Variable nutzt, der kein Wert zugewiesen wurde, lässt sich Ihr Code nicht kompilieren. Diesen Fehler kann man leicht vermeiden, indem man die Variablendeklaration und die Wertzuweisungen zu einer Anweisung kombiniert.

Haben Sie Ihrer Variablen einmal einen Wert zugewiesen, kann sich dieser Wert ändern. Es gibt also keine Nachteile, wenn Sie einer Variablen bei der Deklaration einen Anfangswert zuweisen.

C# nutzt vertraute mathematische Symbole

Was Sie machen können, wenn Sie Daten in einer Variablen gespeichert haben? Na, wenn es eine Zahl ist, wollen Sie mit ihr sicher Additionen, Subtraktionen, Multiplikationen oder Divisionen durchführen. Das ist der Punkt, an dem **Operatoren** ins Spiel kommen. Die grundlegenden Operatoren kennen Sie bereits. Reden wir jetzt über ein paar weitere. Hier ist ein Codeblock, der Operatoren nutzt, um einfache mathematische Operationen durchzuführen.

Wir deklarieren eine neue int-Variable namens zahl und setzen sie auf 15. Dann fügen wir ihr 10 hinzu. Nach der zweiten Anweisung ist zahl gleich 25.

Die dritte Anweisung ändert den Wert von zahl, indem sie ihn auf das Ergebnis von 36 mal 15, d.h. 540, setzt. Dann wird er auf das Ergebnis von 12. – (42. / 7) gesetzt, und das ist 6.

Dieser Operator ist etwas anders. += heißt: Nimm den Wert von zahl und füge ihm 10 hinzu. Da zahl aktuell gleich 6 ist, setzt das Hinzufügen von 10 ihren Wert auf 16.

Der *=-Operator ist wie +=, multipliziert allerdings den aktuellen Wert von zahl mit 3, sodass der Wert nachher 48 ist.

71 geteilt durch 3 ist 23.666666. Da zahl ein int ist, können in dieser Variablen nur ganze Zahlen gespeichert werden. Der Wert wird deswegen auf 23 gerundet.

int werden Sie häufig zum Zählen verwenden. Wenn Sie das tun, sind die Operatoren ++ und -- praktisch. ++ inkrementiert anzahl, indem dem Wert der Variablen eins hinzugefügt wird, und -- dekrementiert anzahl, indem vom Wert eins abgezogen wird. Am Ende ist der Wert also wieder null.

Diese MessageBox enthält den Text "Hallo noch mal Hallo"

Nutzen Sie den Operator + auf einem String, fügt er die beiden Strings einfach zusammen. Zahlen wandelt er für Sie automatisch in Strings um.

Das "" ist der leere String. Er enthält keine Zeichen. (Er ist so etwas wie eine 0, der man Strings hinzufügen kann.)

Ein bool speichert true oder false. Der Operator ! heißt NICHT. Er macht true zu false und umgekehrt.

```
int zahl = 15;
zahl = zahl + 10;
zahl = 36 * 15;
zahl = 12 - (42 / 7);
zahl += 10;
zahl *= 3;
zahl = 71 / 3;

int anzahl = 0;
anzahl ++;
anzahl --;

string ergebnis = "Hallo";
ergebnis += " noch mal " + ergebnis;
MessageBox.Show(ergebnis);
ergebnis = "Der Wert ist: " + anzahl;
ergebnis = "";

bool jaNein = false;
bool nocheinBool = true;
jaNein = !nocheinBool;
```



Versuchen Sie nicht, sich diese Operatoren jetzt einzuprägen.

Sie werden sie schon lernen, weil Sie ständig auf sie stoßen.

Schleifen führen eine Aktion immer wieder aus

Eine Sache ist an den meisten größeren Programmen schon seltsam: Sie enthalten fast immer Aktionen, die immer wieder ausgeführt werden. Dazu sind **Schleifen** da – diese sagen Ihrem Programm, dass es eine Gruppe bestimmter Anweisungen so lange wiederholen soll, bis eine bestimmte Bedingung `true` (oder `false`!) ist.

Das ist einer der Gründe dafür, dass Boolesche Werte so wichtig sind. Eine Schleife nutzt einen Test, um herauszufinden, ob sie weiterlaufen soll.

```
while (x > 5)
{
    x = x - 3;
}
```

Bei einer while-Schleife werden alle Anweisungen innerhalb der geschweiften Klammern so lange ausgeführt, wie die Bedingung in der Klammer wahr ist.

Jede for-Schleife hat drei Anweisungen. Die erste richtet die Schleife ein. Die Schleife wird fortgesetzt, solange die zweite Anweisung wahr ist. Und die dritte Anweisung wird nach jedem Schleifendurchlauf ausgeführt.

```
for (int i = 0; i < 8; i = i + 2)
{
    MessageBox.Show("Mich sehen Sie viermal");
}
```

Nutzen Sie eine Codeschablone, um einfache for-Schleifen zu schreiben

Gleich werden Sie for-Schleifen schreiben, und die IDE kann Ihre Programmierarbeit etwas beschleunigen. Geben Sie das Schlüsselwort `for` ein und drücken dann zweimal die Tabulatortaste, fügt die IDE automatisch Code für Sie ein. Ändern Sie den Namen der Variablen, aktualisiert sie automatisch den Rest der Schablone. Drücken Sie die Tabulatortaste erneut, spring der Cursor zu »length«.

Drücken Sie die Tabulatortaste, damit der Cursor zu »length« springt. Die Anzahl der Schleifendurchläufe wird durch den Wert bestimmt, den Sie für length angeben. Sie können length in eine Zahl oder eine Variable ändern.

```
for (int i = 0; i < length; i++)
{
}
```

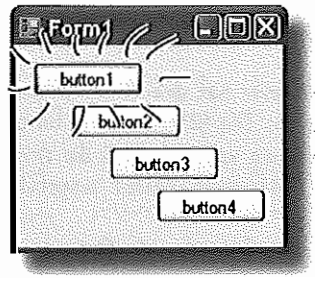
Ändern Sie den Namen der Variablen, ändert die Schablone automatisch die anderen beiden Vorkommen.

Zeit, mit dem Programmieren zu beginnen

Die eigentliche Arbeit in jedem Programm besteht aus den Anweisungen darin. Aber Anweisungen leben nicht im luftleeren Raum. Richten wir also die Bühne ein, auf der wir uns austoben und etwas Code schreiben können. **Erzeugen Sie ein neues Windows-Anwendung-Projekt.**

Das sagt der IDE, dass Sie ein neues Projekt mit einem leeren Formular und einem Einstiegspunkt erzeugen soll. Am besten geben Sie ihm den Namen »Kapitel 2 Programm 1« – Sie werden in diesem Buch noch eine Menge Programme schreiben.

Bauen Sie dieses Formular auf



Syntax 1x1

★ Jede Anweisung muss mit einem Semikolon enden.

```
x = x + 1;
```

★ Ein Ein-Zeilen-Kommentar beginnt mit zwei Schrägstrichen.

```
// Eine Zeile ignorieren.
```

★ Leerräume spielen meist keine Rolle.

```
x       =       3   ;
```

ist das Gleiche wie

```
x = 3;
```

★ Variablen werden mit einem **Typ** und einem **Namen** deklariert (weitere Typen lernen Sie in Kapitel 4 kennen).

```
int gewicht;
// gewicht ist ein
// Integer.
```

★ Klassen und Methoden werden in einem Paar geschweifter Klammern definiert.

```
public void los() {
   // Toller Code.
}
```

Fügen Sie Anweisungen zur Anzeige einer Meldung hinzu

Beginnen Sie damit, dass Sie doppelt auf den ersten Button klicken. Fügen Sie der Methode `button1_Click()` dann diese sechs Anweisungen hinzu. Schauen Sie sich den Code und die Ausgabe, die er erzeugt, genau an.

x ist eine Variable. Das »int« sagt C#, dass sie ein Integer ist, und der Rest der Anweisung setzt ihren Wert auf 3.

```
private void button1_Click(object sender, EventArgs e)
{
    // Das ist ein Kommentar.
    String name = "Quentin";
    int x = 3;
    x = x * 17;
    double d = Math.PI / 2;
    MessageBox.Show("Der Name ist " + name
        + "\nx ist " + x
        + "\nd ist " + d);
}
```

Es gibt eine eingebaute Klasse namens Math, die ein Member namens PI besitzt. Math befindet sich im Namensraum System. Die Datei, in der dieser Code steht, muss am Anfang also die Zeile `using System;` haben.

Das \n ist eine Escape-Sequenz, mit der in die MessageBox ein Zeilenumbruch eingefügt wird.



if/else-Anweisungen fällen Entscheidungen

Nutzen Sie **if/else-Anweisungen**, um Ihrem Programm mitzuteilen, dass es bestimmte Dinge nur tun soll, wenn die **Bedingungen**, die Sie aufgestellt haben, (nicht) wahr sind. Eine Menge if/else-Anweisungen prüfen, ob zwei Dinge gleich sind. Dann verwenden Sie den ==-Operator. Das ist ein anderer Operator als das einfache Gleichheitszeichen (der =-Operator), das sie benutzen, um einen Wert zu setzen.

Jede if-Anweisung beginnt mit einer Testbedingung.

```
if (einWert == 24)
{
    MessageBox.Show("Der Wert war 24.");
}
```

Die Anweisung in den geschweiften Klammern wird nur ausgeführt, wenn der Test wahr ist.

Nutzen Sie immer zwei Gleichheitszeichen, um zu prüfen, ob zwei Dinge einander gleich sind.

```
if (einWert == 24)
{
    // In geschweiften Klammern können
    // beliebig viele Anweisungen stehen.
    MessageBox.Show("Der Wert war 24.");
} else {
    MessageBox.Show("Der Wert war nicht 24.");
}
```

if/else-Anweisungen sind ziemlich einfach. Wenn die Testbedingung wahr ist, werden die Anweisungen zwischen den ersten geschweiften Klammern ausgeführt, andernfalls die in den zweiten geschweiften Klammern.



Verwechseln Sie die Operatoren nicht mit den Gleichheitszeichen!

Ein Gleichheitszeichen (=) wird verwendet, um den Wert einer Variablen zu setzen. Zwei Gleichheitszeichen (==) werden verwendet, um zwei Werte zu vergleichen. Sie können sich nicht vorstellen, wie viele Programmfehler – selbst bei Programmen, die erfahrene Programmierer geschrieben haben! – durch eine falsche Verwendung von = statt == verursacht werden. Sollte sich die IDE mit folgender Meldung beschweren: »Eine implizite Konvertierung vom Typ 'int' in 'bool' ist nicht möglich.«, ist das wahrscheinlich das Problem.

Bedingungen aufstellen und prüfen

Nutzen Sie **if/else-Anweisungen**, um Ihrem Programm zu sagen, dass es bestimmte Dinge nur tun soll, wenn die von Ihnen aufgestellten **Bedingungen** (nicht) wahr sind.

Vergleichen Sie zwei Werte mit einem Vergleichsoperator, bezeichnet man das als Testbedingung.

Prüfen Sie Bedingungen mit logischen Operatoren

Wir haben gerade den ==-Operator betrachtet, den man einsetzt, um zu prüfen, ob zwei Werte gleich sind. Es gibt noch ein paar weitere Operatoren. Versuchen Sie nicht, sich diese jetzt sofort einzuprägen – in den nächsten paar Kapiteln werden Sie sie noch gründlich kennenlernen:

- Der !=-Operator funktioniert ähnlich wie ==, ergibt aber wahr, wenn die verglichenen Dinge **nicht gleich** sind.
- Mit den Operatoren > und < können Zahlen verglichen werden, um zu prüfen, ob die eine größer oder kleiner als die andere ist.
- ==, !=, > und < werden als **Vergleichsoperatoren** bezeichnet. Nutzen Sie diese, um zwei Variablen oder Werte zu vergleichen, geben Sie eine **Testbedingung** an.
- Über den `&&`-Operator (der für UND steht) und den `||`-Operator (für ODER) können Sie Testbedingungen zu einem langen Test verknüpfen. Um zu prüfen, ob `i` gleich 3 oder `j` kleiner als 5 ist, verwenden Sie also `(i == 3) || (j < 5)`.

Eine Variable setzen und ihren Wert prüfen

Hier ist der Code für den zweiten Button. Es ist eine `if/else`-Anweisung, die die int-**Variable** x darauf prüft, ob sie gleich 10 ist.

Denken Sie daran, Ihr Programm anzuhalten, bevor Sie das tun – die IDE lässt Sie den Code nicht bearbeiten, während das Programm läuft. Sie können das Programm beenden, indem Sie den Stopp-Button in der Symbolleiste anklicken oder im Menü Debuggen den Eintrag »Debuggen beenden« wählen.

```
private void button2_Click(object sender, EventArgs e)
{
    int x = 5;
    if (x == 10)
    {
        MessageBox.Show("x muss 10 sein");
    }
    else
    {
        MessageBox.Show("x ist nicht 10");
    }
}
```

Erst richten wir eine Variable namens x ein und setzen sie auf 5. Dann prüfen wir, ob sie gleich 10 ist.



Hier ist die Ausgabe. Schauen Sie, ob Sie eine Codezeile so ändern können, dass hier stattdessen »x muss 10 sein« steht.

Noch eine Testbedingung

Der dritte Button erzeugt diese Ausgabe. Ändern Sie jetzt zwei Codezeilen, damit beide MessageBox-Fenster angezeigt werden.



Diese Zeile prüft, ob einWert gleich 3 und name gleich »Tim« ist.

```
private void button3_Click(object sender, EventArgs e)
{
    int einWert = 4;
    String name = "Richard";
    if ((einWert = 3) && (name.Equals("Tim")))
    {
        MessageBox.Show("x ist 3 und der Name ist Tim.");
    }
    MessageBox.Show("Diese Anweisung läuft immer.");
}
```

Fügen Sie Ihrem Programm Schleifen hinzu

Hier ist der Code für den letzten Button. Er enthält zwei Schleifen. Die erste ist eine **while**-Schleife, die die Anweisungen innerhalb der geschweiften Klammern so lange wiederholt, wie die Bedingung wahr ist. Die zweite ist eine **for**-Schleife. Werfen Sie einen Blick darauf und sehen Sie sich an, wie sie funktioniert.

```
private void button4_Click(object sender, EventArgs e)
{
    int anzahl = 0;

    while (anzahl < 10)
    {
        anzahl = anzahl + 1;
    }

    for (int i = 0; i < 5; i++)
    {
        anzahl = anzahl - 1;
    }

    MessageBox.Show("Die Antwort ist " + anzahl);
}
```

Diese Schleife wird wiederholt, solange die Variable anzahl kleiner als 10 ist.

Der zweite Teil der for-Anweisung ist der Test. Er sagt: »Die Schleife soll laufen, solange i kleiner als fünf ist.«

Das richtet die Schleife ein. Es setzt einfach die int-Variable, die in der Schleife verwendet wird, auf einen Wert.

Das ist der Punkt, an dem die Schleife tatsächlich etwas macht. Hier fügt sie i eins hinzu. Jedes Mal, wenn die Schleife durchlaufen wird, wird i 1 hinzuaddiert.

Lesen Sie den Code genau durch, bevor Sie auf den Button klicken, und versuchen Sie herauszufinden, was angezeigt wird. Prüfen Sie dann, ob Sie richtig lagen.

Spitzen Sie Ihren Bleistift

Verschaffen wir uns etwas mehr Praxis mit Bedingungen und Schleifen. Werfen Sie einen Blick auf den Code unten. Kreisen Sie die Testbedingungen ein, und füllen Sie die Lücken in den Kommentaren so aus, dass der ausgeführte Code genau beschrieben wird.

```
int ergebnis = 0; // Diese Variable enthält das Endergebnis.
int x = 6; // Deklariert die Variable x und setzt sie auf 6.
while (x > 3) {
  // Diese Anweisungen so lange ausführen, wie .....
  ergebnis = ergebnis + x; // Addiere x .....
  x = x - 1; // Subtrahiere .....
}
for (int z = 1; z < 3; z = z + 1) {
  // Beginne eine Schleife, indem .....
  // Durchlaufe die Schleife, solange .....
  // ..... nach jedem Schleifendurchlauf.
  ergebnis = ergebnis + z; // .....
}
// Die nächste Anweisung zeigt eine MessageBox mit der Meldung:
// .....
MessageBox.Show("Das Ergebnis ist " + ergebnis);
```

Die erste Aufgabe haben wir für Sie erledigt.

Mehr zu Testbedingungen

Sie können einfache Testbedingungen durchführen, indem Sie den Wert einer Variablen mit Vergleichsoperatoren prüfen. So können Sie zwei Zahlen, x und y, vergleichen:

```
x < y (kleiner als)
x > y (größer als)
x == y (gleich - ja, mit zwei Gleichheitszeichen)
```

Diese werden Sie am häufigsten verwenden.



Moment! Gibt's dabei nicht einen Haken? Was passiert, wenn ich eine Schleife mit einem Test schreibe, der nie falsch wird?

Dann läuft Ihre Schleife ewig!

Jedes Mal, wenn Ihr Programm eine Testbedingung durchführt, ist das Ergebnis entweder **`true`** oder **`false`**. Ist es **`true`**, durchläuft Ihr Programm die Schleife ein weiteres Mal. Jede Schleife sollte Code einschließen, der bewirkt, dass die Testbedingung **`false`** liefert, wenn er oft genug ausgeführt wurde. Tut sie das nicht, läuft die Schleife immer weiter, bis Sie das Programm abbrechen oder den Computer ausschalten!

Manchmal nennt man das eine Endlosschleife.

Spitzen Sie Ihren Bleistift

Hier sind ein paar Schleifen. Schreiben Sie auf, ob die einzelnen Schleifen endlos laufen oder ob sie irgendwann beendet werden. Falls eine beendet wird, schreiben Sie auf, wie oft die Schleife durchlaufen wird.

Schleife 1

```
int anzahl = 5;
while (anzahl > 0) {
   anzahl = anzahl * 3;
   anzahl = anzahl * -1;
}
```

Schleife 2

```
int i = 0;
while (i == 0) {
   anzahl = anzahl * 3;
   anzahl = anzahl * -1;
}
```

Schleife 3

```
int j = 2;
for (int i = 1; i < 100;
     i = i * 2)
{
    j = j - i;
    while (j < 25)
    {
        j = j + 5;
    }
}
```

Schleife 4

```
while (true) { int i = 1;}
```

Schleife 5

```
int p = 2;
for (int q = 2; q < 32;
     q = q * 2)
{
   while (p < q)
   {
      p = p * 2;
   }
   q = p - q;
}
```



KOPF-NUSS

Fällt Ihnen ein Grund ein, eine Schleife zu schreiben, die endlos läuft?

Spitzen Sie Ihren Bleistift
Lösung

Verschaffen wir uns etwas mehr Praxis mit Bedingungen und Schleifen. Werfen Sie einen Blick auf den Code unten. Kreisen Sie die Testbedingungen ein und füllen Sie die Lücken in den Kommentaren so aus, dass der ausgeführte Code genau beschrieben wird.

```
int ergebnis = 0; // Diese Variable enthält das Endergebnis.
int x = 6; // Deklariert die Variable x und setzt sie auf 6.
while (x > 3) {
  // Diese Anweisungen so lange ausführen, wie x größer als 3 ist.
  ergebnis = ergebnis + x; // Addiere x zur Variablen ergebnis hinzu.
  x = x - 1; // Subtrahiere 1 vom Wert von x.
}
for (int z = 1; z < 3; z = z + 1) {
  // Beginne eine Schleife, indem die Variable z deklariert und auf 1 gesetzt wird.
  // Durchlaufe die Schleife, solange z kleiner als 3 ist.
  // Addiere zu z 1 hinzu nach jedem Schleifendurchlauf.
  ergebnis = ergebnis + z; // Addiere den Wert von z zu ergebnis hinzu.
}
// Die nächste Anweisung zeigt eine MessageBox mit der Meldung:
// Das Ergebnis ist 18
MessageBox.Show("Das Ergebnis ist " + ergebnis);
```

Diese Schleife läuft zweimal – erst mit z gleich 1 und dann mit z gleich 2. Ist z gleich 3, wird die Schleife kein weiteres Mal durchlaufen.

Spitzen Sie Ihren Bleistift
Lösung

Hier sind ein paar Schleifen. Schreiben Sie auf, ob die einzelnen Schleifen endlos laufen oder ob sie irgendwann beendet werden. Falls eine beendet wird, schreiben Sie auf, wie oft die Schleife durchlaufen wird.

Schleife 1
Wird einmal ausgeführt.

Schleife 2
Läuft endlos.

Schleife 3
Wird siebenmal ausgeführt.

Schleife 4
Noch eine Endlosschleife.

Schleife 5
Wird achtmal ausgeführt.

Es gibt keine Dummen Fragen

F: Steckt der gesamte Code wirklich immer in einer Klasse?

A: Ja. Immer wenn ein C#-Programm etwas macht, liegt das daran, dass Anweisungen ausgeführt werden. Diese Anweisungen sind Bestandteile von Klassen, und diese Klassen sind Bestandteile von Namensräumen. Selbst wenn es scheint, als wäre etwas keine Anweisung in einer Klasse – beispielsweise wenn Sie mit dem Designer die Eigenschaft eines Objekts in einem Formular setzen –, werden Sie bei einem genauen Blick in Ihren Code feststellen, dass die IDE irgendwo in einer Klasse Anweisungen hinzugefügt oder geändert hat.

F: Gibt es Namensräume, die Ich nicht verwenden darf? Gibt es welche, die Ich verwenden *muss*?

A: Ja, es gibt ein paar Namensräume, die Sie nicht verwenden dürfen. Haben Sie bemerkt, dass alle `using`-Zeilen am Anfang von C#-Kiassendateien immer mit `System` beginnen? Das liegt daran, dass es einen Namensraum System gibt, der vom .NET Framework verwendet wird. In ihm finden Sie all die wichtigen Werkzeuge, die Ihren Programmen Macht verleihen – wie beispielsweise `System.Data`, über den Sie mit Datenbanken und Tabellen arbeiten können, oder `System.Math`, der mathematische Funktionen enthält. Aber meistens können Sie beliebige Namen für Namensräume wählen (solange diese nur Buchstaben, Zahlen und Unterstriche enthalten). Wenn Sie ein neues Programm erstellen, wählt die IDE auf Basis des Projektnamens automatisch einen Namensraum für Sie.

F: Das mit den partiellen Klassen verstehe ich noch nicht.

A: Mit partiellen Klassen können Sie den Code für eine Klasse auf mehrere Dateien aufteilen. Die IDE macht das, wenn sie ein Formular erzeugt– in einer Datei hält sie den Code, den Sie bearbeiten (wie Form1.cs), und in einer anderen den Code, den sie automatisch verändert (Form1.Designer.cs). Mit einem Namensraum müssen Sie das allerdings nicht machen. Ein Namensraum kann zwei, drei oder auch zehn und mehr Dateien umspannen. Geben Sie einfach am Anfang der Datei eine Namensraumdeklaration an, und alles in den geschweiften Klammern hinter der Deklaration befindet sich im gleichen Namensraum. Eine Sache noch: In einer Datei können sich mehrere Klassen befinden und ebenso mehrere Namensräume. In den kommenden Kapiteln werden Sie noch viel mehr über Klassen lernen.

F: Nehmen wir an, Ich ziehe etwas auf ein Formular und die IDE generiert automatisch einen Haufen Code. Was passiert, wenn ich dann »Rückgängig« anklicke?

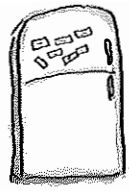
A: Die Frage beantworten Sie am besten selbst, indem Sie es ausprobieren! Tun Sie etwas in der IDE, das diese Code für Sie generieren lässt. Ziehen Sie einen Button auf ein Formular, ändern Sie Eigenschaften. Versuchen Sie dann, es rückgängig zu machen. Was passiert? Sie sehen, dass die IDE clever genug ist, einfache Dinge selbstständig rückgängig zu machen. Bei komplexeren Dingen, beispielsweise wenn Sie Ihrem Projekt eine neue SQL-Datenbank hinzugefügt haben, erhalten Sie eine Warnung. Sie weiß dann immer noch, wie sie die Aktion rückgängig machen muss, aber nicht, wie sie sie wiederherstellen kann.

F: Wie vorsichtig muss Ich denn mit dem Code umgehen, den die IDE automatisch generiert hat?

A: in der Regel sollten Sie sehr vorsichtig sein. Es ist wirklich nützlich, wenn man weiß, was die IDE mit dem eigenen Code macht, und gelegentlich muss man wissen, was darin steckt, um ernsthafte Probleme zu beheben. Aber in den meisten Fällen können Sie alles, was Sie machen müssen, über die IDE tun.

Punkt für Punkt

- Mit Anweisungen weisen Sie das Programm an, Aktionen durchzuführen. Anweisungen stehen in Klassen und Klassen in Namensräumen.
- Anweisungen enden immer mit einem Semikolon (`;`).
- Die grafischen Werkzeuge in der Visual Studio IDE bewirken Codeänderungen in Ihrem Programm.
- Codeblöcke werden von geschweiften Klammern `{ }` eingeschlossen. Klassen, Schleifen und viele andere Anweisungen nutzen Blöcke.
- Eine Testbedingung ist entweder `true` oder `false`. Sie nutzen sie als Endbedingung einer Schleife oder in if/else-Anweisungen.
- Mit Variablen speichern Sie in Programmen Daten. Mit `=` weisen Sie Variablen einen Wert zu, mit `==` prüfen Sie, ob die Werte zweier Variablen gleich sind.
- Eine `while`-Schleife führt alles in ihrem Block aus, solange die *Testbedingung* `true` ist.
- Ist die Testbedingung `false`, läuft die Schleife nicht. Die Ausführung geht mit dem Code nach dem Schleifenblock weiter.



Code-Magneten

Die Teile eines C#-Programms sind vollkommen durcheinandergeraten. Können Sie die Codeschnipsel so anordnen, dass sie ein funktionierendes C#-Programm ergeben, das die Ausgabe in dem Dialogfenster unten hervorbringt? Einige der geschweiften Klammern sind zu Boden gefallen. Sie waren einfach zu klein, um sie aufzuheben. Fügen Sie also so viele davon hinzu, wie Sie benötigen!

Das "" ist der leere String – dies bedeutet, dass das Ergebnis noch keine Zeichen enthält.

```
string ergebnis = "";
```

Dieser Magnet ist an seinem alten Platz geblieben.

```
if (x == 1) {
ergebnis = ergebnis + "d";
x = x - 1;
}
```

```
if (x == 2) {
ergebnis = ergebnis + "b c";
}
```

```
if (x > 2) {
ergebnis = ergebnis + "a";
}
```

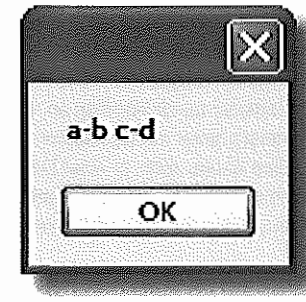
```
int x = 3;
```

```
x = x - 1;
ergebnis = ergebnis + "-";
```

```
while (x > 0) {
```

```
MessageBox.Show(ergebnis);
```

Ausgabe:



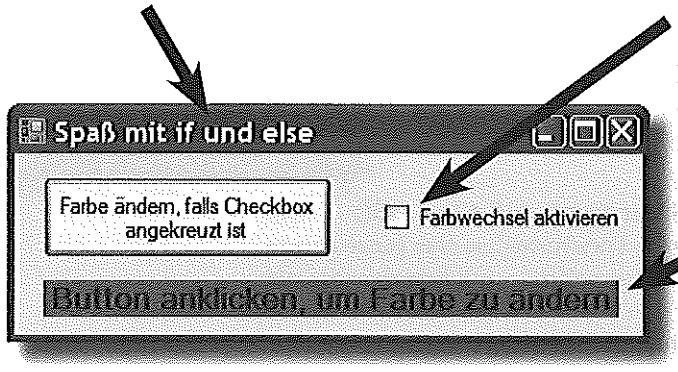
→ Antworten auf Seite 82.

Wir geben Ihnen in diesem Buch eine Menge Übungen wie diese auf. Die Lösung werden wir Ihnen in ein paar Seiten geben. Sollten Sie hängen bleiben, dürfen Sie keine Angst haben, bei der Antwort zu spinksen. Mit Mogeln hat das nichts zu tun!



Es ist an der Zeit, dass Sie die Arbeit mit if/else-Anweisungen etwas trainieren. Können Sie dieses Programm erstellen?

Hier ist das Formular.

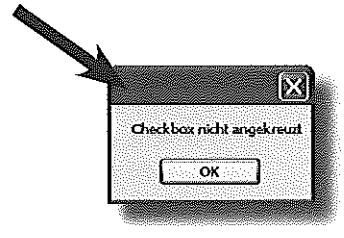


Fügen Sie diese Checkbox hinzu.
Ziehen Sie sie aus der Toolbox auf Ihr Formular. Nutzen Sie die Eigenschaft **`Text`**, um den danebenstehenden Text zu ändern. (Die Eigenschaft **`Text`** nutzen Sie auch, um den Text auf dem Button und in dem Label zu ändern.

Das ist ein Label.
Sie können die Eigenschaften nutzen, um die Schriftgröße und das Schriftgewicht zu ändern. Setzen Sie die Eigenschaft **`BackColor`** auf Rot – wählen Sie »**Red**« in der Auswahl der Webfarben.

Blenden Sie diese Meldung ein, falls der Benutzer auf den Button klickt, wenn die Checkbox NICHT ausgewählt ist.
Trägt Ihre Checkbox den Namen **`checkBox1`** (Sie können die Eigenschaft **`Name`** ändern, wenn Sie möchten), können Sie mit folgender Testbedingung prüfen, ob sie ausgewählt ist:

```
checkBox1.Checked == true
```



Ändern Sie die Hintergrundfarbe des Labels, falls der Benutzer auf den Button klickt, wenn die Checkbox ausgewählt IST.
Ändern Sie die Hintergrundfarbe des Labels in Blau, falls diese rot ist, wenn der Benutzer auf den Button klickt. Ist sie blau, ändern Sie sie wieder in Rot. Hier ist eine Anweisung, die die Hintergrundfarbe eines Labels namens **`label1`** setzt:

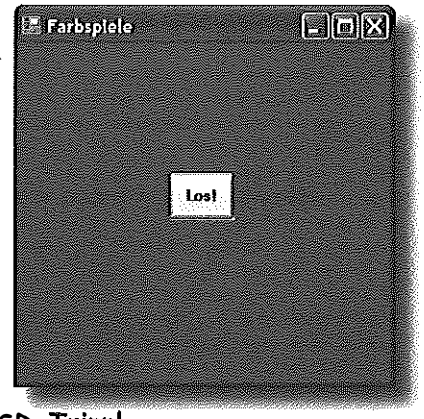
```
label1.BackColor = Color.Red;
```

(TIPP: Die Testbedingung, mit der Sie prüfen, ob Rot die Hintergrundfarbe des Labels ist, hat große Ähnlichkeit mit dieser Anweisung – es gibt allerdings einen wichtigen Unterschied!)



Bauen wir etwas schön Buntes auf!

1 **Hier ist das zu erstellende Formular.**



Tipp: Wenn Sie in einer for-Schleife eine Variable deklarieren – for (int c = 0; ...) –, dann ist diese Variable nur innerhalb der geschweiften Klammern dieser for-Schleife gültig. Haben Sie zwei for-Schleifen, die beide die gleiche Variable verwenden, müssen Sie diese entweder in beiden Schleifen deklarieren oder eine Deklaration außerhalb der Schleifen angeben.

2 **Schicken Sie den Formularhintergrund auf einen LSD-Tripp!**
Sorgen Sie dafür, dass der Hintergrund des Formulars eine ganze Palette Farben durchläuft, wenn auf den Button geklickt wird! Erzeugen Sie eine Schleife mit einer Variablen **c**, die von 0 bis 253 läuft. Hier ist der Codeblock, der in die geschweiften Klammern kommt:

```
this.BackColor = Color.FromArgb(c, 255 - c, c);

Application.DoEvents();
```

Diese Zeile sagt dem Programm, dass es das Betriebssystem neben Ihrem Programm andere Dinge machen lassen soll. Ohne sie frisst Ihr Programm alle CPU-Zyklen – und achtet noch nicht einmal auf Ereignisse, die eintreten (beispielsweise dass der Benutzer auf das X-Kästchen klickt, um die Anwendung zu schließen).

Beeindruckende Farben!

.NET bietet einen Haufen an vordefinierten Farben wie Blue und Red, ermöglicht Ihnen über die Methode `Color.FromArgb()` aber auch, Ihre eigenen Farben zu definieren, indem Sie drei Zahlen angeben: einen Rot-, einen Grün- und einen Blauwert.

3 **Einen Gang rausnehmen.**
Bremsen Sie die Farbkaskade ab, indem Sie hinter der `Application.DoEvents()`-Zeile diese Zeile einfügen.

```
System.Threading.Thread.Sleep(3);
```

Diese Anweisung fügt in die Schleife eine Verzögerung von 3 Millisekunden ein. Sie ist Teil der .NET-Bibliothek und befindet sich im Namensraum `System.Threading`.

Im Verlauf dieses Buchs werden Sie eine Menge Anwendungen erstellen. Sie müssen jeder davon einen eigenen Namen geben. Wir empfehlen Ihnen, diese auf Basis der Kapitelnummer und des Texts in der Titelleiste des Formulars »2. Farbspiele« zu nennen.

4 **Machen Sie es glatter.**
Sorgen Sie dafür, dass die Farben wieder zu der Farbe zurücklaufen, mit der angefangen wurde. Fügen Sie eine weitere Schleife ein, bei der `c` von 254 zurück zu 0 läuft. Setzen Sie den gleichen Codeblock zwischen die geschweiften Klammern.

3 **Machen Sie die Sache endlos.**
Umgeben Sie die beiden Schleifen mit einer weiteren Schleife, die unablässig läuft und nicht endet, damit der Hintergrund durch einen Klick auf den Button beginnt, die Farbe zu ändern, und damit dann nicht mehr aufhört. (Tipp: Die Schleife `while (true)` läuft endlos!)

Befindet sich eine Schleife in einer anderen, nennt man diese eine »eingebettete« Schleife.

Oh! Das Programm endet nicht mehr!

Führen Sie das Programm in der IDE aus. Lassen Sie es Schleifen drehen. Schließen Sie jetzt das Fenster. Warten Sie eine Minute – die IDE kehrt nicht in den Bearbeitungsmodus zurück! Sie tut, als würde das Programm noch immer laufen. Endgültig stoppen müssen Sie das Programm, indem Sie in der IDE auf den quadratischen Stopp-Button der IDE klicken (oder im Debug-Menü »Debuggen beenden« wählen).

3 **Sorgen Sie dafür, dass das Programm endet.**
Sorgen Sie dafür, dass die Schleife, die Sie in Schritt 5 hinzugefügt haben, beendet wird, wenn das Programm geschlossen wird. Ändern Sie die äußere Schleife folgendermaßen:

```
while (Visible)
```

Führen Sie jetzt das Programm aus und klicken Sie auf das X-Kästchen in der Ecke. Das Fenster wird geschlossen, und das Programm endet! Aber es gibt eine Verzögerung von ein paar Sekunden, bevor die IDE wieder in den Bearbeitungsmodus wechselt.

Wenn man in einer if-Anweisung oder einer Schleife einen Booleschen Wert wie Visible prüft, ist man manchmal geneigt, auf `(Visible == true)` zu testen. Das »== true« können Sie sich sparen – es reicht, den Booleschen Wert einzuschließen.

Bei der Arbeit mit einem Formular oder einem Steuerelement ist Visible so lange true, wie das Formular oder Steuerelement angezeigt wird. Setzen Sie es auf false, bewirkt dies, dass das Formular oder Steuerelement verschwindet.

Tipp: Der &&-Operator heißt »UND«. Mit ihm knüpfen Sie mehrere Testbedingungen zu einem großen Test zusammen, der nur wahr ist, wenn der erste Test UND der zweite UND der dritte usw. wahr sind. Er könnte bei der Lösung dieses Problems nützlich sein.

Haben Sie eine Idee, was die Verzögerung bewirken könnte? Können Sie es reparieren, damit das Programm sofort endet, wenn Sie das Fenster schließen?



LÖSUNGEN ZU DEN ÜBUNGEN

Es ist an der Zeit, dass Sie die Arbeit mit if/else-Anweisungen etwas trainieren. Können Sie dieses Programm erstellen?

Hier ist die vollständige Form1.cs-Datei für die Übung »Spaß mit if und else«. Wenn wir sehr viel Code in einer Datei zeigen, ziehen wir einen grauen Kasten um den Teil, den Sie hinzufügen sollen.

Hier ist der Code für das Formular. Wir haben unserer Projektmappe den Namen »Kapitel 2 Spaß mit if und else« gegeben. Deswegen hat die IDE den Namensraum Kapitel_2_Spaß_mit_if_und_else gewählt. Wenn Sie Ihrer Projektmappe einen anderen Namen gegeben haben, finden Sie hier einen anderen Namensraum.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace Kapitel_2_Spaß_mit_if_und_else
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
```

Als Sie doppelt auf den Button geklickt haben, hat die IDE dem Formular die Methode button1_Click() hinzugefügt. Diese Methode wird jedes Mal ausgeführt, wenn auf den Button geklickt wird.

```
        private void button1_Click(object sender, EventArgs e)
        {
            if (checkBox1.Checked == true)
            {
                if (label1.BackColor == Color.Red)
                {
                    label1.BackColor = Color.Blue;
                }
                else
                {
                    label1.BackColor = Color.Red;
                }
            }
            else
            {
                MessageBox.Show("Checkbox nicht angekreuzt");
            }
        }
    }
}
```

Die äußere if-Anweisung prüft, ob die Checkbox angekreuzt ist!

Die innere if-Anweisung prüft die Farbe des Labels. Ist das Label aktuell rot, wird eine Anweisung ausgeführt, die es blau macht.

Diese Anweisung wird ausgeführt, wenn der Hintergrund des Labels nicht rot ist, um ihn wieder auf Rot zurückzusetzen.

Diese MessageBox springt auf, wenn die Checkbox nicht angekreuzt ist.

Den Code für alle Übungslösungen zu diesem Buch können Sie von http://examples.oreilly.de/german_examples/hfcsharpger/ herunterladen.



Lösungen zu den Übungen

Bauen wir etwas schön Buntes auf!

Manchmal zeigen wir Ihnen nicht den gesamten Code in der Projektmappe, sondern nur die Teile, die geändert wurden. Die gesamte Logik des Projekts Farbspiele steckt in der Methode button1_Click(), die die IDE eingefügt hat, als Sie doppelt auf den Button geklickt haben.

Als die IDE diese Methode hinzufügte, schob sie vor der geschweiften Klammer einen zusätzlichen Zeilenumbruch ein. Manchmal ziehen wir die Klammer in die vorangehende Zeile, um Platz zu sparen – C# ist der zusätzliche Leerraum gleichgültig, deswegen ist das kein Problem.

```
private void button1_Click(object sender, EventArgs e)
{
    while (Visible) {
        for (int c = 0; c < 254 && Visible; c++)
        {
            this.BackColor = Color.FromArgb(c, 255 - c, c);
            Application.DoEvents();
            System.Threading.Thread.Sleep(3);
        }
        for (int c = 254; c >= 0 && Visible; c--) {
            this.BackColor = Color.FromArgb(c, 255 - c, c);
            Application.DoEvents();
            System.Threading.Thread.Sleep(3);
        }
    }
}
```

Die äußere Schleife läuft, solange das Formular sichtbar ist. Wird es geschlossen, wird Visible false, und das bewirkt, dass die while-Schleife beendet wird.

Die erste Schleife lässt die Farben in die eine Richtung laufen, und die zweite Schleife kehrt das um, damit es harmonischer aussieht.

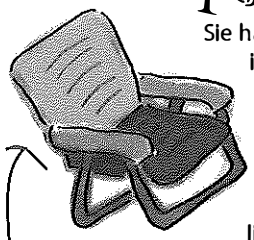
Die zusätzliche Verzögerung haben wir behoben, indem wir mit dem &&-Operator in beiden for-Schleifen Visible prüfen. Dann endet die Schleife, sobald Visible falsch wird.

Haben Sie eine Idee, was die Verzögerung bewirken könnte? Können Sie es reparieren, damit das Programm sofort endet, wenn Sie das Fenster schließen?

Die Verzögerung tritt ein, weil die for-Schleife erst enden muss, bevor die while-Schleife prüfen kann, ob Visible immer noch wahr ist. Das können Sie beheben, indem Sie der Testbedingung der beiden Schleifen `&& Visible` hinzufügen.

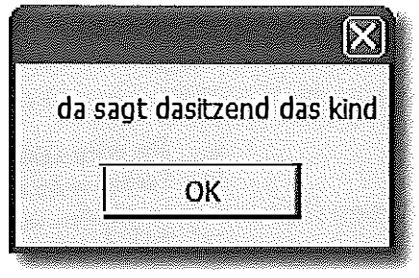
Sah Ihr Code etwas anders aus? Meist gibt es mehrere Wege, Programmierprobleme zu lösen – Sie hätten beispielsweise while- statt for-Schleifen verwenden können. Wenn Ihr Programm funktioniert, ist Ihre Lösung richtig.

Pool-Puzzle



Sie haben die ***Aufgabe***, die leeren Zeilen im Code mit den Codeschnipseln aus dem Pool zu füllen. Jedes Schnipsel **darf** nur einmal verwendet werden, und Sie werden nicht alle Schnipsel benötigen. Das ***Ziel*** ist es, eine Klasse zu erstellen, die sich kompilieren lässt und die gezeigte Ausgabe erzeugt. Lassen Sie sich nicht täuschen – das ist schwerer, als es scheint.

Ausgabe:



Auf diese >>Pool-Puzzle<<-Übungen werden Sie im gesamten Buch stoßen. Sie sollen Ihren kleinen grauen Zellen eine extra-hartes Training verschaffen. Wenn Sie verzwickte Logikrätsel mögen, wird Ihnen das hier gefallen. Wenn nicht, probieren Sie es trotzdem – und haben Sie keine Angst, einen Blick auf die Lösung zu werfen, um zu sehen, was zu tun ist. Und sollten Sie bei einem Pool-Puzzle nicht weiterwissen, übergehen Sie es einfach!

```
using System;
using System.Windows.Forms;
namespace Kapitel_2 {
  class Kapitel2PoolPuzzle {
    public static void Main() {
      int x = 0;
      String Vers = "";

      while ( _____ ) {

        _____
        if ( x < 1 ) {
          _____
        }
        _____

        if ( _____ ) {

          _____

          _____
        }
        if ( x == 1 ) {

          _____
        }
        if ( _____ ) {

          _____
        }

        _____
      }
      _____
    }
  }
}
```

Hinweis: Jeder Schnipsel darf nur einmal verwendet werden.

```
x > 0
x < 1
x > 1
x > 3
x < 4

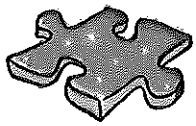
x = x + 1;
x = x + 2;
x = x - 2;
x = x - 1;

Vers = Vers + " ";
Vers = Vers + "da";
Vers = Vers + "s";
Vers = Vers + "das";

MessageBox.Show(Vers);

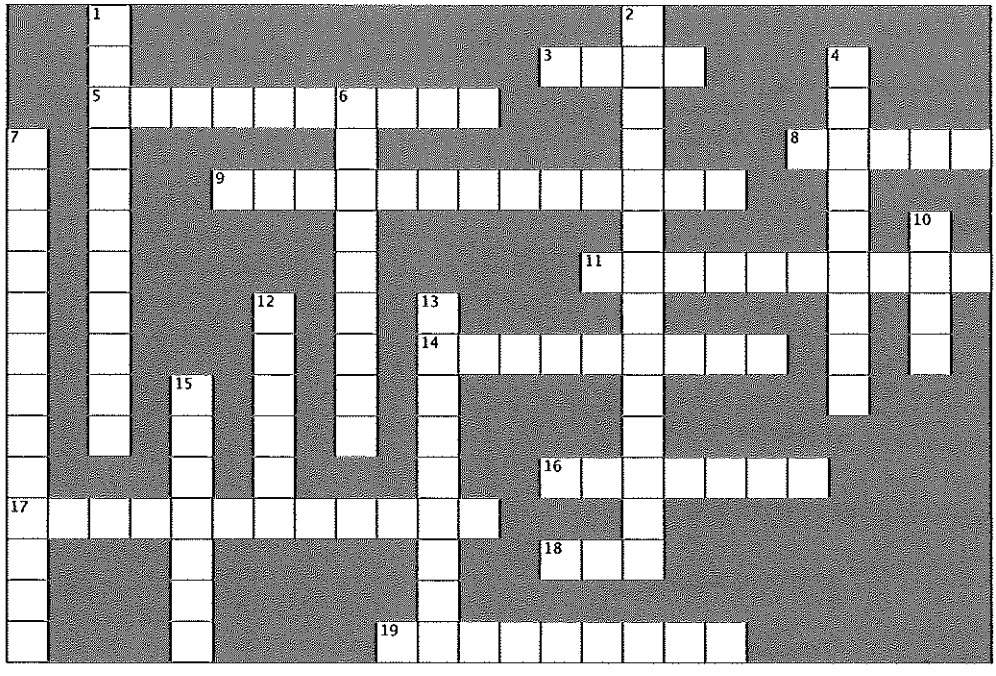
Vers = Vers + "itzend ";
Vers = Vers + "agt ";
Vers = Vers + " kind";
Vers = Vers + "dasitzend";
Vers = Vers + "sagt";
```


Antworten auf Seite 83.



C#-Kreuzworträtsel

Wie Ihnen ein Kreuzworträtsel hilft, C# zu lernen? Alle Wörter beziehen sich auf C# und stammen aus diesem Kapitel. Die Hinweise enthalten mentale Haken und Ösen, die Ihnen helfen, alternative Wege zu C# direkt in Ihr Hirn zu brennen.

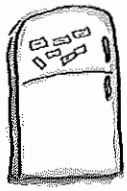


Waagerecht

3. Der Name des Einstiegspunkt in allen C#-Programmen.
5. System.Windows.Forms ist ein Beispiel dafür.
8. Enthält eine Liste mit Dingen.
9. button1.Text und checkBox3.Name sind Beispiele dafür.
11. _____.Show() rufen Sie auf, um ein einfaches Windows-Dialogfenster zu öffnen.
14. Ein kleiner Teil eines Programms, der etwas macht.
16. Ein Typ von Anwendung, bei dem die Interaktion über die Kommandozeile abgewickelt wird.
17. Ein Codeblock wird von solchen Klammern eingeschlossen.
18. Ein Typ von Variable, der nur ganze Zahlen speichern kann.
19. Damit endet jede Anweisung.

Senkrecht

1. Eine spezielle Methode, die ausgeführt wird, wenn ein Formular erzeugt wird.
2. Eine besondere Methode, die ausgeführt wird, wenn Ihr Programm gestartet wird.
4. Darüber übergeben Sie einer Methode Informationen.
6. Die Ausgabe einer Methode ist ihr _____wert.
7. Der Ausdruck, der einer Schleife sagt, wann sie enden soll.
10. Der Typ einer Variablen, die entweder true oder false ist.
12. Enthält Methoden.
13. Diese Art von Klasse umspannt mehrere Dateien.
15. Darin stehen Ihre Anweisungen.



Code-Magneten, Lösung zu Seite 74

Die Teile eines C#-Programms sind vollkommen durcheinandergeraten. Können Sie die Codeschnipsel so anordnen, dass sie ein funktionierendes C#-Programm ergeben, das die Ausgabe in dem Dialogfenster unten hervorbringt? Einige der geschweiften Klammern sind zu Boden gefallen. Sie waren einfach zu klein, um sie aufzuheben. Fügen Sie also so viele davon hinzu, wie Sie benötigen!

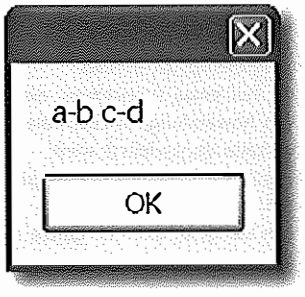
```
string ergebnis = "";
int x = 3;
while (x > 0) {
{
    if (x > 2) {
    ergebnis = ergebnis + "a";
    }
    x = x - 1;
    ergebnis = ergebnis + "-";
    if (x == 2) {
    ergebnis = ergebnis + "b c";
    }
        if (x == 1) {
        ergebnis = ergebnis + "d";
        x = x - 1;
        }
}
MessageBox.Show(ergebnis);
```

Dieser Magnet ist an seinem alten Platz geblieben.

Beim ersten Schleifendurchlauf ist x gleich 3. Diese Testbedingung liefert also wahr.

Beim ersten Schleifendurchlauf macht diese Anweisung x gleich 2 und beim zweiten gleich 1.

Ausgabe:



Pool-Puzzle, Lösung



Sie hatten die ***Aufgabe***, die Lücken im Code mit den Codeschnipseln aus dem Pool zu füllen. Ihr ***Ziel*** war es, eine Klasse zu erstellen, die sich kompilieren und ausführen lässt.

```
using System;
using System.Windows.Forms;
namespace Kapitel_2 {
  class Kapitel2PoolPuzzle {
    public static void Main() {
      int x = 0;
      String Vers = "";

      while ( x < 4 ) {

        Vers = Vers + "da";
        if ( x < 1 ) {
          Vers = Vers + " ";
        }
        Vers = Vers + "s";

        if ( x > 1 ) {

          Vers = Vers + " kind";

          x = x + 2;
        }
        if ( x == 1 ) {

          Vers = Vers + "itzend ";
        }
        if ( x < 1 ) {

          Vers = Vers + "agt ";
        }

        x = x + 1;
      }
      MessageBox.Show(Vers);
    }
  }
}
```



C#-Kreuzworträtsel, Lösung

	[1]K											[2]E								
	O										[3]M	A	I	N			[4]P			
	[5]N	A	M	E	N	S	[6]R	A	U	M				N			A			
[7]T	S					U								S		[8]A	R	R	A	Y
E	T			[9]E	I	G	E	N	S	C	H	A	F	T	E	N		A			
S	R					C								I			M		[10]B	
T	U					K							[11]M	E	S	S	A	G	E	B	O	X
B	K			[12]K		G		[13]P						G				T		O	
E	T			L		A		[14]A	N	W	E	I	S	U	N	G		E		L	
D	O		[15]M	A		B		R					P					R			
I	R		E	S		E		T					U								
N			T	S				I				[16]K	O	N	S	O	L	E			
[17]G	E	S	C	H	W	E	I	F	T	E	N			K							
U			O					L				[18]I	N	T							
N			D					L													
G			E				[19]S	E	M	I	K	O	L	O	N						

3 Objekte: Orientieren Sie sich!

Aufgabenspezifischer Code



Jedes Programm, das Sie schreiben, löst ein Problem.

Bevor Sie ein Programm erstellen, überlegen Sie sich am besten, welches *Problem* Ihr Programm lösen soll. Deswegen sind ***Objekte*** so ungemein hilfreich. Mit diesen können Sie Ihren Code auf Basis des Problems strukturieren, das er lösen soll. Und dann können Sie Ihre Zeit damit verbringen, *über das Problem nachzudenken*, an dem Sie arbeiten müssen, anstatt gleich beim Schreiben des Codes im Sumpf der Implementierungsanforderungen zu versinken. Wenn Sie Objekte richtig einsetzen, erhalten Sie Code, der *intuitiv* zu schreiben sowie leicht zu lesen und zu ändern ist.

Wie Mark über sein Problem nachdenkt

Mark ist Programmierer. Gleich muss er sich auf den Weg zu einem Vorstellungsgespräch machen. Er kann's kaum erwarten, seine C#-Fertigkeiten vorzuführen. Aber erst einmal muss er den Weg finden – und langsam wird die Zeit knapp.

1 **Mark überlegt, welchen Weg er zum Vorstellungsgespräch nehmen soll.**



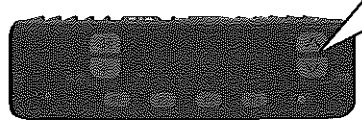
Ich nehme die Talbrücke, fahre den Mauspfad hoch und passiere dann Neustadt.

Mark legt sein Ziel fest und überlegt sich dann den Weg.

2 **Zum Glück hatte er das Radio an. Es gibt einen fetten Stau, der ihn zu spät kommen ließe!**

Mark erhält neue Informationen zu einer Straße, die er umgehen muss.

Hier ist Frank Kreischer mit den aktuellsten Verkehrsmeldungen: Mauspfad, Altstadt, Richtung Neustadt, 2 Kilometer Stau nach Verkehrsunfall.



3 **Mark überlegt sich einen neuen Weg, damit er pünktlich zu seinem Vorstellungsgespräch kommt.**

Jetzt muss er sich einen neuen Weg zum Vorstellungsgespräch überlegen.

Kein Problem. Wenn ich stattdessen die B3 nehme, bin ich immer noch pünktlich.



Wie Marks Navi über das Problem nachdenkt

Mark hat sich ein eigenes GPS-Navigationssystem gebastelt, das er einsetzt, um in der Stadt den Weg zu finden.

Hier ist das Diagramm einer Klasse in Marks Programm. Oben steht der Name und darunter stehen die Methoden.

Navigator

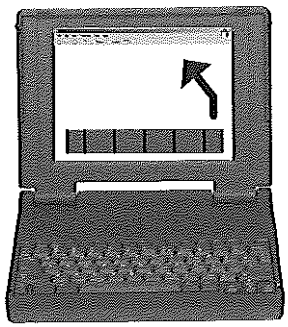
ZielFestlegen()
StraßeAusschließen()
StraßeEinschließen()
RouteBerechnen()
FahrzeitBerechnen()
EntfernungBerechnen()

```
ZielFestlegen("Marktplatz");
string route;
route = RouteBerechnen();
```

Hier ist die Ausgabe der Methode RouteBerechnen() – es ist ein String mit den Richtungshinweisen, denen Mark folgen soll.

Das Navi nimmt ein Ziel an und gibt eine Route aus.

```
»Nimm die Talbrücke zum Mauspfad und folge ihm durch Neustadt.«
```



Das Navi erhält Informationen zu einem Weg, den es vermeiden soll.

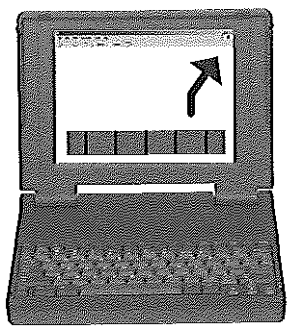
```
StraßeAusschließen("Mauspfad");
```

Jetzt berechnet es eine neue Route zum Ziel.

```
string route;
route = RouteBerechnen();
```

```
»Nimm die B3 über die Parkbrücke zur Berliner Straße.«
```

RouteBerechnen() liefert eine neue Route, die die Straße umgeht, die Mark vermeiden will.



Marks Navigationssystem löst das Navigationsproblem auf gleiche Weise wie er.

Die Methoden von Marks Navigator-Klasse

Marks Navigator-Klasse besitzt Methoden, in denen die Aktionen abgewickelt werden. Aber im Unterschied zu den `button_Click()`-Methoden in den von Ihnen erstellten Formularen befassen diese sich alle mit einem einzigen Problem: der Suche nach einem Weg durch eine Stadt. Das ist der Grund dafür, dass Mark sie alle in eine Klasse gesteckt und dieser Klasse den Namen `Navigator` gegeben hat.

Mark hat seine Navigator-Klasse so entworfen, dass es leicht ist, Routen zu generieren und zu verändern. Um eine Wegbeschreibung zu erhalten, ruft Marks Programm die Methode `ZielFestlegen()` auf, um das Ziel anzugeben, und nutzt dann die Methode `RouteBerechnen()`, um die Wegbeschreibung in einen String zu packen. Wenn er die Route ändern muss, ruft sein Programm die Methode `StraßeAusschließen()` auf, um die Route so zu ändern, dass bestimmte Straßen umgangen werden. Anschließend wird die Methode `RouteBerechnen()` eingesetzt, um die neue Wegbeschreibung abzurufen.

Mark wählte Methodennamen, die für jeden sinnvoll sind, der sich überlegt, wie man durch eine Stadt navigiert.

```
public class Navigator() {
    public void ZielFestlegen(string zielname) { ... };
    public void StraßeAusschließen(string straßenname) { ... };
    public string RouteBerechnen() { ... };
}
```

Das ist der Rückgabetyp der Methode. Er bedeutet, dass die Anweisung, die die Methode RouteBerechnen() aufruft, eingesetzt werden kann, um eine String-Variable zu setzen, die eine Wegbeschreibung enthält. Ist der Rückgabetyp void, heißt das, dass die Methode nichts zurückliefert.

```
string route;
route = RouteBerechnen();
```

Manche Methoden haben Rückgabewerte

Jede Methode besteht aus Anweisungen, die Dinge tun. Einige Methoden führen ihre Anweisungen aus und enden dann. Andere haben einen **Rückgabewert**, d.h. einen Wert, der in der Methode berechnet oder generiert wird und an die Methode zurückgeliefert wird, die die andere Methode aufrief. Der Typ des Rückgabewerts (wie `string` oder `int`) wird als **Rückgabetyp** bezeichnet.

Die **`return`**-Anweisung sagt der Methode, dass sie sofort enden soll. Hat Ihre Methode keinen Rückgabewert – d.h., deklariert sie den Rückgabetyp **`void`** –, endet die `return`-Anweisung einfach mit einem Semikolon. Ihre Methode muss dann nicht immer eine `return`-Anweisung enthalten. Aber wenn eine Methode einen Rückgabetyp hat, muss sie die `return`-Anweisung verwenden:

Hier ist ein Beispiel für eine Methode mit einem Rückgabetyp – sie liefert einen `int`. Die Methode nutzt die beiden Parameter, um das Ergebnis zu berechnen, und liefert dieses dann mit der return-Anweisung an die aufrufende Anweisung zurück.

```
public int ZahlenMultiplizieren(int ersteZahl, int zweiteZahl) {
        int ergebnis = ersteZahl * zweiteZahl;
        return ergebnis;
}
```

Hier ist eine Anweisung, die eine Methode zur Multiplikation zweier Zahlen aufruft:

```
int einErgebnis = ZahlenMultiplizieren(3, 5);
```

Methoden können Werte wie 3 und 5 annehmen. Sie können Ihnen aber auch Variablen übergeben.

Nutzen Sie das Gelernte, um eine einfache Anwendung zu erstellen

Verknüpfen wir ein Formular mit einer Klasse und lassen wir seinen Button eine Methode in dieser Klasse aufrufen:

Tun Sie das!

1. Erzeugen Sie in der IDE eine neue Windows-Forms-Anwendung. Fügen Sie ihr eine Klassendatei namens `Redner.cs` hinzu, indem Sie im Projektmappen-Explorer rechts auf das Projekt klicken und im Menü Hinzufügen »Klasse...« wählen. Nennen Sie die Datei »Redner.cs«, gibt die IDE der Klasse darin automatisch den Namen Redner. Dann öffnet sie die neue Datei in einer neue Registerkarte.

2. Fügen Sie am Anfang der Datei `using System.Windows.Forms;` und am Ende diesen Code ein:

```
class Redner {
    public static int BlahBlahBlah(string aussage, int wieOft) {
        string ergebnisString = "";
        for (int zähler = 1; zähler <= wieOft; zähler++) {
            ergebnisString = ergebnisString + aussage + "\n";
        }
        MessageBox.Show(ergebnisString);
        return ergebnisString.Length;
    }
}
```

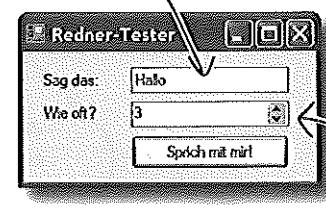
Diese Anweisung deklariert die Variable ergebnisString und setzt sie auf den leeren String.

Der Rückgabewert der Methode BlahBlahBlah() ist ein int, der die Gesamtlänge der angezeigten Nachricht enthält. >>.Length<< können Sie an jeden String anhängen, um seine Länge zu bestimmen.

Die neue Klasse hat eine Methode namens `BlahBlahBlah()`, die zwei Parameter erwartet. Der erste Parameter ist ein String, der die Aussage enthält, und der zweite ist eine Zahl, die angibt, wie oft die Aussage gemacht werden soll. Wird die Methode aufgerufen, springt eine MessageBox mit der entsprechenden Anzahl an Wiederholungen der Aussage auf. Der Rückgabewert ist die Länge des Strings. Den String für den Parameter aussage und die Zahl für den Parameter wieOft erhält die Methode aus einem Formular, in das der Benutzer über ein **TextBox**- und ein **NumericUpDown**-Steuerelement einen Text bzw. eine Zahl eingeben kann.

Setzen Sie den Standardtext der TextBox über die Eigenschaft Text auf >>Hallo<<.

3. Fügen Sie Ihrem Projekt dieses Formular hinzu.



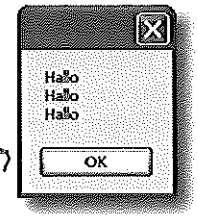
Doppelklicken Sie auf den Button und geben Sie für ihn diesen Code an:

```
int len = Redner.BlahBlahBlah(textBox1.Text, (int) numericUpDown1.Value);
MessageBox.Show("Die Länge der Nachricht ist " + len);
```

Das ist ein NumericUpDown-Steuerelement. Setzen Sie die Eigenschaft Minimum auf 1, die Eigenschaft Maximum auf 10 und die Eigenschaft Value auf 3.

4. Führen Sie das Programm dann aus! Klicken Sie auf den Button und beobachten Sie die beiden Dialogfenster. Die Klasse öffnet das erste, das Formular das zweite.

Dieses Fenster öffnet die Methode BlahBlahBlah() auf Basis der übergebenen Parameter.

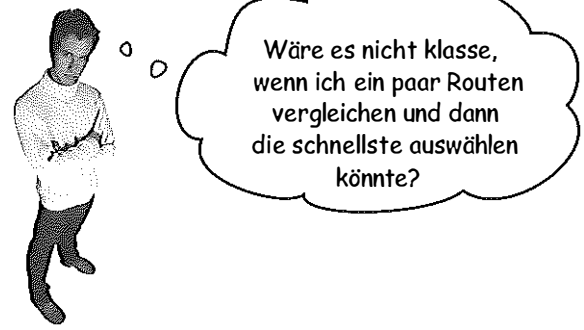


Liefert die Methode einen Wert, zeigt das Formular dieses Fenster an.



Mark hat eine Idee

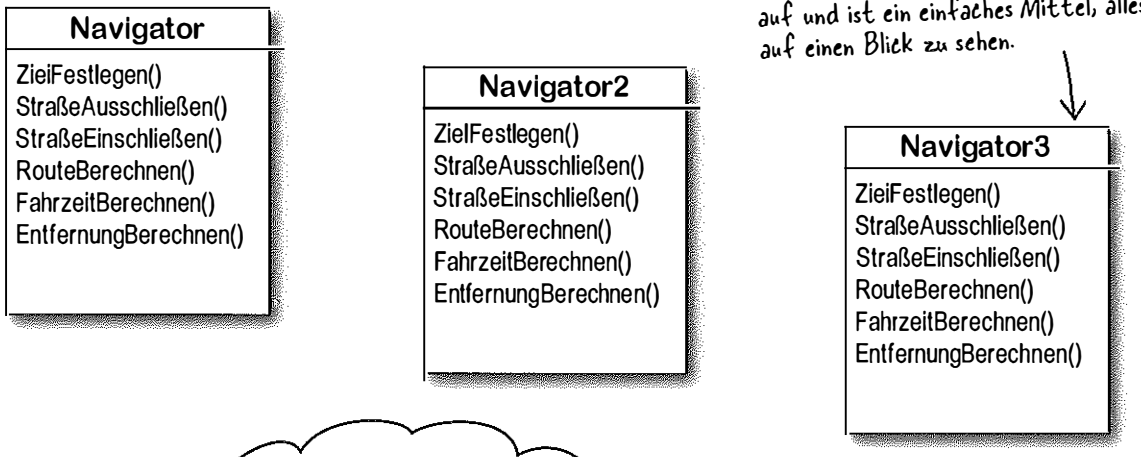
Das Vorstellungsgespräch lief ganz nach Wunsch! Aber der Verkehrsstau heute morgen brachte Mark dazu, darüber nachzudenken, wie er sein Navi verbessern könnte.



Er könnte drei verschiedene Navigator-Klassen erstellen ...

Mark könnte den Code der Klasse Navigator kopieren und in zwei weitere Klassen einfügen. Dann könnte sein Programm drei Routen gleichzeitig speichern.

Dieser Kasten ist ein Klassendiagramm. Es führt alle Methoden in einer Klasse auf und ist ein einfaches Mittel, alles auf einen Blick zu sehen.

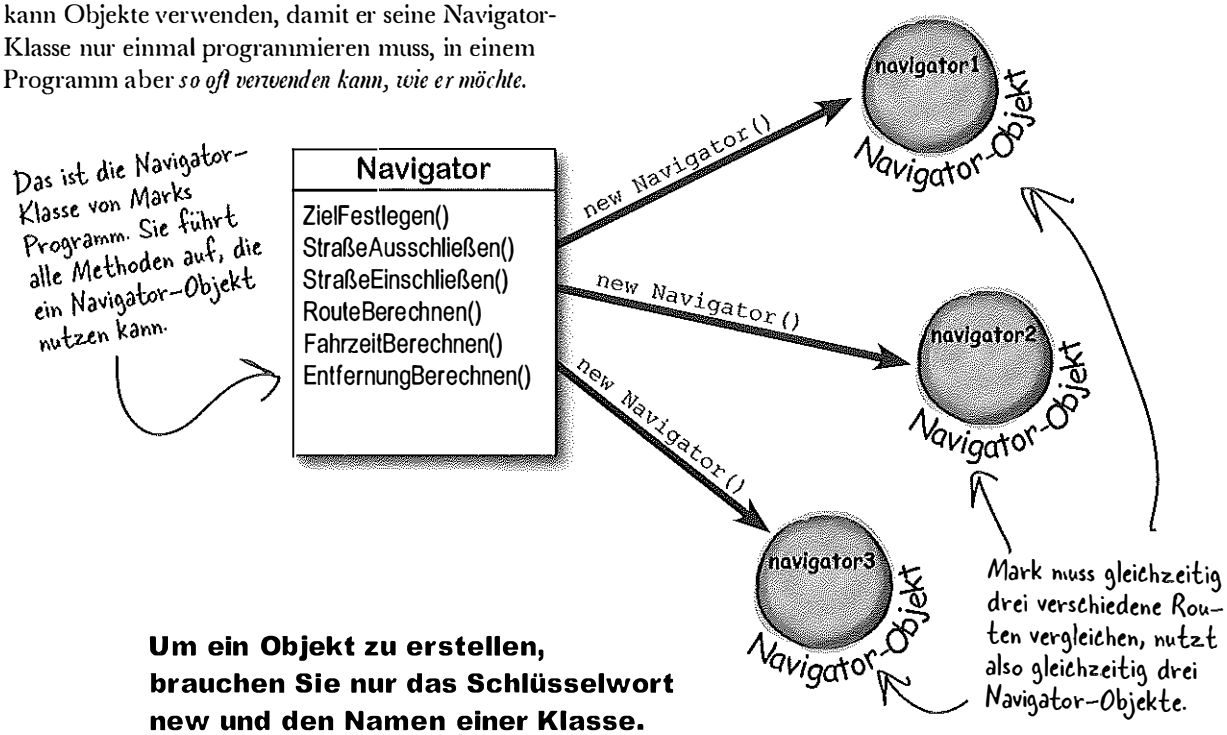




Stimmt! Drei Kopien des gleichen Codes zu pflegen ist wirklich umständlich. Bei vielen Problemen, die Sie lösen müssen, brauchen Sie eine Möglichkeit, ein ***Ding*** mehrfach darzustellen. Hier sind das mehrere Routen. Aber es könnten auch mehrere Turbinen, Hunde, Musikdateien oder sonst irgendwas sein. Alle diese Programme haben eine Sache gemeinsam: Sie brauchen ein Mittel, um eine Art von Ding immer auf die gleiche Weise zu behandeln, egal mit wie vielen dieser Dinger sie es zu tun haben.

Mark kann sein Problem mit Objekten lösen

Objekte sind das C#-Werkzeug, mit dessen Hilfe Sie mit einem Haufen gleicher Dinge arbeiten. Mark kann Objekte verwenden, damit er seine Navigator-Klasse nur einmal programmieren muss, in einem Programm aber *so oft verwenden kann, wie er möchte.*



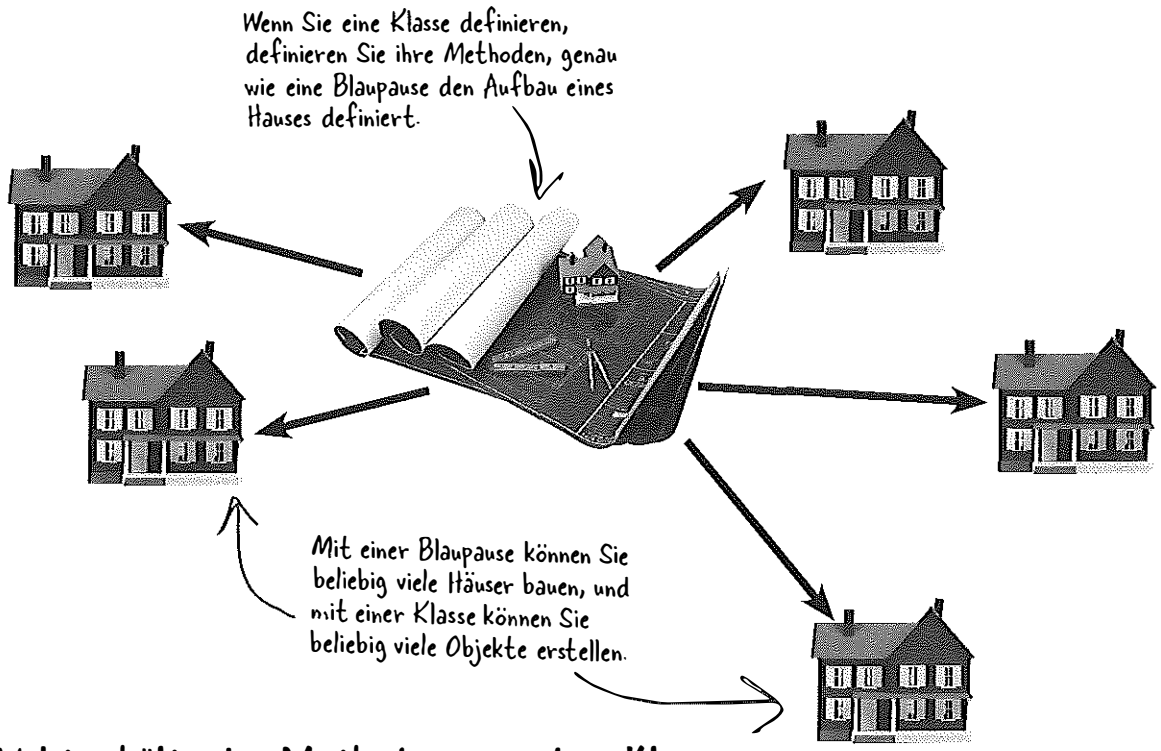
Um ein Objekt zu erstellen, brauchen Sie nur das Schlüsselwort new und den Namen einer Klasse.

```
Navigator navigator1 = new Navigator();
navigator1.ZielFestlegen("Marktplatz");
string route;
route = navigator1.RouteBerechnen();
```

Jetzt können Sie das Objekt verwenden! Wenn Sie ein Objekt einer Klasse erstellen, erhält es alle Methoden dieser Klasse.

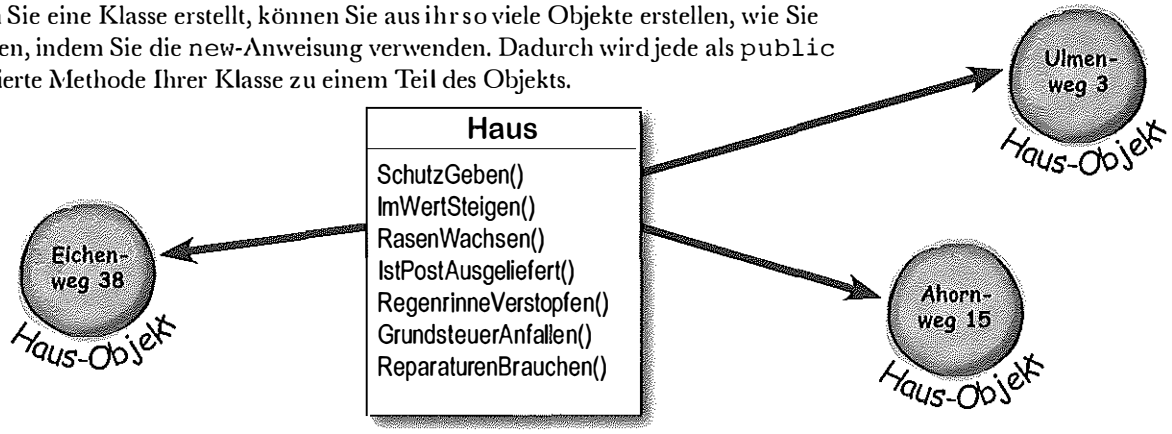
Sie nutzen eine Klasse, um ein Objekt zu erstellen

Eine Klasse ist wie eine Blaupause für ein Objekt. Wenn Sie in einem Vorort fünf identischer Häuser bauen lassen wollen, würden Sie Ihren Architekten natürlich nicht beauftragen, fünf identische Sätze von Bauplänen zu zeichnen, Sie würden einen einzigen Bauplan verwenden, um fünf Häuser zu bauen.



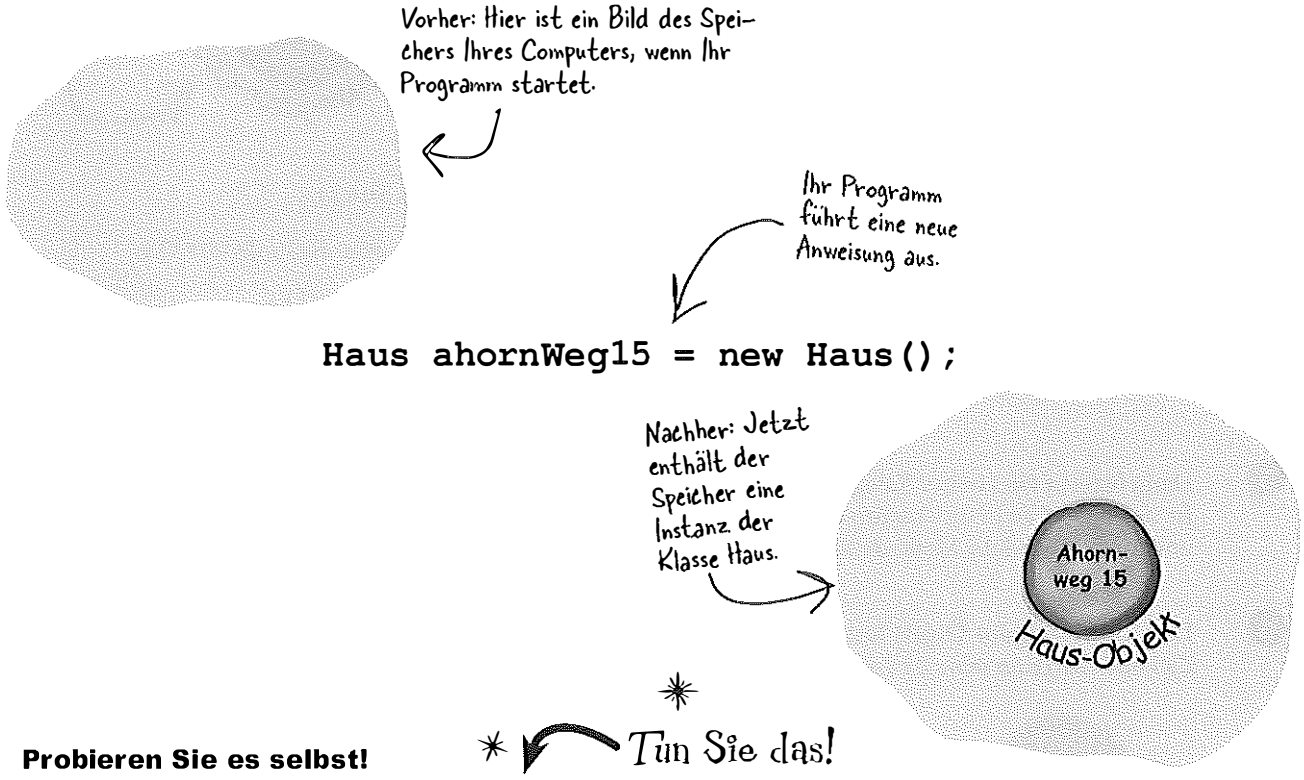
Ein Objekt erhält seine Methoden von seiner Klasse

Haben Sie eine Klasse erstellt, können Sie aus ihr so viele Objekte erstellen, wie Sie möchten, indem Sie die `new`-Anweisung verwenden. Dadurch wird jede als `public` deklarierte Methode Ihrer Klasse zu einem Teil des Objekts.



Wenn Sie ein neues Objekt einer Klasse erstellen, bezeichnet man dieses als Instanz der Klasse

Soll ich Ihnen was sagen … den Kram wissen Sie schon! Alles in der Toolbox ist eine Klasse: Es gibt eine Klasse `Button`, eine Klasse `TextBox`, eine Klasse `Label` usw. Wenn Sie einen Button aus der Toolbox ziehen, erzeugt die IDE automatisch eine Instanz der Klasse `Button` und nennt sie `button1`. Ziehen Sie einen weiteren Button aus der Toolbox, erzeugt sie eine weitere Instanz mit dem Namen `button2`. Jede Instanz von `Button` hat ihre eigenen Eigenschaften und Methoden. Aber jeder Button verhält sich auf genau die gleiche Weise, weil alle Instanzen der gleichen Klasse sind.



Tun Sie das!

Probieren Sie es selbst!

Öffnen Sie ein beliebiges Projekt, das einen Button namens `button1` verwendet, und nutzen Sie die IDE, um im gesamten Projekt nach dem Text »**`button1 = new`**« zu suchen. Sie werden dann den Code finden, den die IDE dem Formularentwurf hinzugefügt hat, um die Instanz der Klasse `Button` zu erzeugen.

In-s-tanz, Nomen.
Ein Beispiel oder Vorkommen von etwas. *Die Suchen/Ersetzen-Funktion der IDE findet jede* ***Instanz*** *eines Worts und ändert sie.*

Eine bessere Lösung ... dank Objekten!

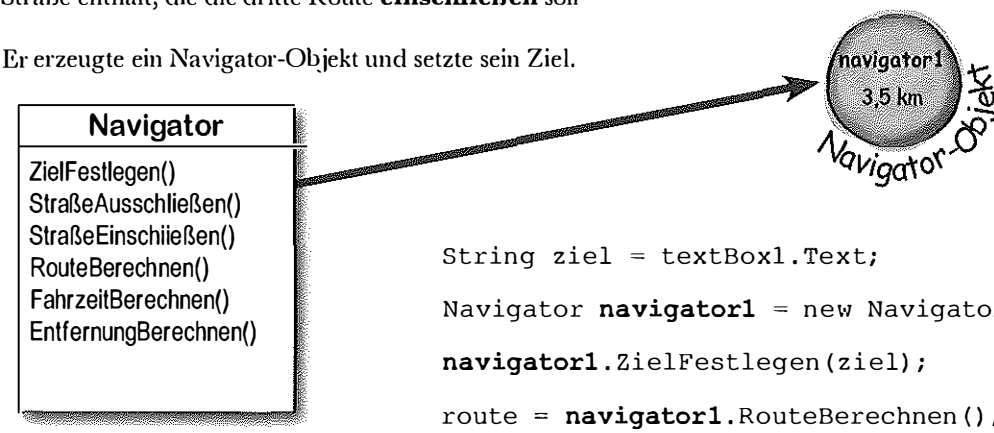
Mark entwickelte ein neues Programm zum Vergleich von Routen, das Objekte nutzt, um die kürzeste von drei verschiedenen Routen zum selben Ziel zu finden. Folgendermaßen ist das Programm aufgebaut.

GUI steht für Graphical User Interface bzw. grafische Benutzerschnittstelle. Das ist das, was Sie aufbauen, wenn Sie im Windows Forms-Designer ein Formular entwerfen.

1. Mark richtete ein GUI mit einem Textfeld, `textBox1`, ein, das das **Ziel** für die drei Routen enthält. Dann fügte er eine `textBox2` hinzu, die eine Straße enthält, die eine der Routen **vermeiden** soll, sowie eine `textBox3`, die eine weitere Straße enthält, die die dritte Route **einschließen** soll

2. Er erzeugte ein Navigator-Objekt und setzte sein Ziel.

Das Objekt navigator1 ist eine Instanz der Klasse Navigator.




```
String ziel = textBox1.Text;
Navigator navigator1 = new Navigator();
navigator1.ZielFestlegen(ziel);
route = navigator1.RouteBerechnen();
```

3. Dann fügte er ein zweites Navigator-Objekt namens `navigator2` hinzu. Er rief seine `ZielFestlegen()`-Methode auf, um das Ziel festzulegen, und anschließend die Methode `StraßeAusschließen()`.

Die Methoden ZielFestlegen(), StraßeAusschließen() und StraßeEinschließen() erwarten alle einen String-Parameter.

4. Das dritte Navigator-Objekt heißt `navigator3`. Mark setzte sein Ziel und rief dann seine `StraßeEinschließen()`-Methode auf.



5. Jetzt kann Mark die `EntfernungBerechnen()`-Methode der einzelnen Objekte aufrufen, um zu ermitteln, welche Route die kürzeste ist. Und er musste den Code nur einmal schreiben, nicht dreimal.

Jedes Mal, wenn Sie ein neues Objekt einer Klasse erstellen, erstellen Sie eine Instanz der Klasse.

Spitzen Sie Ihren Bleistift

Führen Sie die gleichen Schritte durch wie Mark auf der gegenüberliegenden Seite, um die Navigator-Objekte zu erstellen und ihre Methoden aufzurufen.

```
String ziel = textBox1.Text;
String StraßeAusschließen = textBox2.Text;
String StraßeEinschließen = textBox3.Text;

Navigator navigator1 = new Navigator();
navigator1.ZielFestlegen(ziel);
int distanz1 = navigator1.EntfernungBerechnen();
```

Wir haben Ihnen einen Ansatzpun t gegeben. Hier ist der Code, den Mark schrieb, um die Ziel- und Straßennamen aus den Textfeldern abzurufen.

Und hier ist der Code, der das Navigator-Objekt erstellt, sein Ziel setzt und die Entfernung abruft.

1. Erzeugen Sie das **navigator2**-Objekt, setzen Sie sein Ziel, rufen Sie seine **StraßeAusschließen()**-Methode auf und nutzen Sie die Methode **EntfernungBerechnen()**, um eine int-Variable namens **distanz2** zu setzen.

```
Navigator navigator2 = .....

navigator2.....

navigator2.....

int distanz2 = .....
```

2. Erzeugen Sie das **navigator3**-Objekt, setzen Sie sein Ziel, rufen Sie seine **StraßeEinschließen()**-Methode auf und nutzen Sie die Methode **EntfernungBerechnen()**, um eine int-Variable namens **distanz3** zu setzen.

```
.....

.....

.....

.....
```

Die eingebaute C#-Methode Math.Min() vergleicht zwei Zahlen und liefert die kleinere zurück. Mit ihr hat Mark die kürzeste Distanz zum Ziel berechnet.

```
int kleinsteDistanz = Math.Min(distanz1, Math.Min(distanz2, distanz3));
```

Spitzen Sie Ihren Bleistift
Lösung

Führen Sie die gleichen Schritte durch wie Mark auf der gegenüberliegenden Seite, um die Navigator-Objekte zu erstellen und ihre Methoden aufzurufen.

```
String ziel = textBox1.Text;
String StraßeAusschließen = textBox2.Text;
String StraßeEinschließen = textBox3.Text;

Navigator navigator1 = new Navigator();
navigator1.ZielFestlegen(ziel);
int distanz1 = navigator1.EntfernungBerechnen();
```

Wir haben Ihnen einen Ansatzpunkt gegeben. Hier ist der Code zum Abrufen des Ziels und der Straßennamen sowie der Code zur Erstellung des ersten Navigator-Objekts, zum Setzen seiner Route und zum Abrufen der Entfernung.

1. Erzeugen Sie das **navigator2**-Objekt, setzen Sie sein Ziel, rufen Sie seine **StraßeAusschließen()**-Methode auf und nutzen Sie die Methode **EntfernungBerechnen()**, um eine int-Variable namens **distanz2** zu setzen.

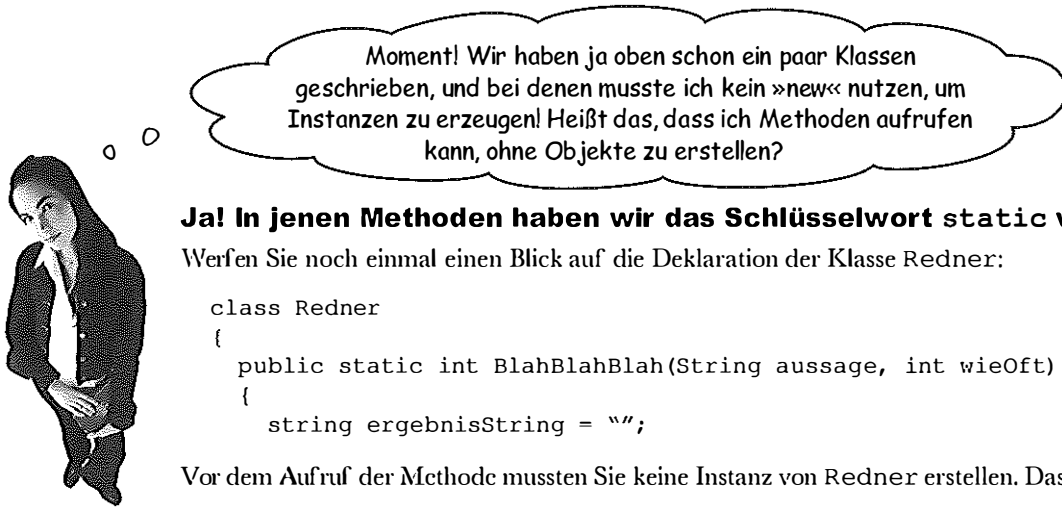
```
Navigator navigator2 = new Navigator()
navigator2.ZielFestlegen(ziel);
navigator2.StraßeAusschließen(StraßeAusschließen);
int distanz2 = navigator2.EntfernungBerechnen();
```

2. Erzeugen Sie das **navigator3**-Objekt, setzen Sie sein Ziel, rufen Sie seine **StraßeEinschließen()**-Methode auf und nutzen Sie die Methode **EntfernungBerechnen()**, um eine int-Variable namens **distanz3** zu setzen.

```
Navigator navigator3 = new Navigator()
navigator3.ZielFestlegen(ziel);
navigator3.StraßeEinschließen(StraßeEinschließen);
int distanz3 = navigator3.EntfernungBerechnen();
```

Die eingebaute C#-Methode Math.Min() vergleicht zwei Zahlen und liefert die kleinere zurück. Mit ihr hat Mark die kürzeste Distanz zum Ziel berechnet.

```
int kleinsteDistanz = Math.Min(distanz1, Math.Min(distanz2, distanz3));
```



Ja! In jenen Methoden haben wir das Schlüsselwort `static` verwendet.

Werfen Sie noch einmal einen Blick auf die Deklaration der Klasse `Redner`:

```
class Redner
{
  public static int BlahBlahBlah(String aussage, int wieOft)
  {
    string ergebnisString = "";
```

Vor dem Aufruf der Methode mussten Sie keine Instanz von `Redner` erstellen. Das ging einfach so:

```
Redner.BlahBlahBlah("Hallo Hallo Hallo", 5);
```

Auf diese Weise ruft man `static`-Methoden auf, und das haben Sie die ganze Zeit gemacht. Entfernen Sie aus der Deklaration der Methode `BlahBlahBlah()` das Schlüsselwort `static`, müssen Sie eine Instanz von `Redner` erstellen, um die Methode aufzurufen. Davon abgesehen, sind statische Methoden wie Objektmethoden. Sie können ihnen Parameter übergeben, die Methoden können Werte zurückliefern und leben in Klassen.

Es gibt noch eine weitere Sache, die Sie mit dem Schlüsselwort `static` tun können. Sie können eine **vollständige Klasse** als `static` markieren. Dann **müssen** auch alle ihre Methoden `static` sein. Versuchen Sie, einer statischen Klasse eine nicht statische Methode hinzuzufügen, lässt sie sich nicht kompilieren.

Es gibt keine Dummen Fragen

F: Wenn ich mir etwas »Statisches« vorstelle, denke ich immer an Dinge, die sich nicht ändern. Bedeutet das, dass sich nicht statische Methoden ändern können, statische jedoch nicht? Verhalten sie sich unterschiedlich?

A: Nein, statische und nicht statische Methoden verhalten sich vollkommen gleich. Der einzige Unterschied ist, dass statische Methoden keine Instanz verlangen, nicht statische hingegen schon. Eine Menge Menschen haben Probleme, sich das zu merken, weil das Wort »static« nicht so intuitiv ist.

F: Ich kann meine Klasse also erst verwenden, wenn ich eine Instanz erstelle?

A: Sie können ihre statischen Methoden verwenden. Aber wenn Sie Methoden haben, die nicht statisch sind, brauchen Sie eine Instanz, bevor Sie sie verwenden können.

F: Was will ich denn dann mit einer Methode, die eine Instanz braucht? Warum mache ich nicht einfach alle meine Methoden statisch?

A: Na, wenn Sie Objekte haben, die Daten festhalten -- wie Marks Instanzen der Klasse Navigator, die jeweils eine andere Route festhalten –, können Sie die jeweiligen Instanzmethoden nutzen, um mit diesen Daten zu arbeiten. Als Mark auf der navigator2-Instanz die Methode StraßeAusschließen() aufrief, war nur die Route in dieser speziellen Instanz betroffen. Die Objekte navigator1 und navigator3 wurden nicht beeinträchtigt. Auf diese Weise konnte er gleichzeitig mit drei verschiedenen Routen arbeiten – und sein Programm konnte sie alle festhalten.

F: Wie hält eine Instanz denn dann die Daten fest?

A: Blättern Sie um, wenn Sie das herausfinden wollen!

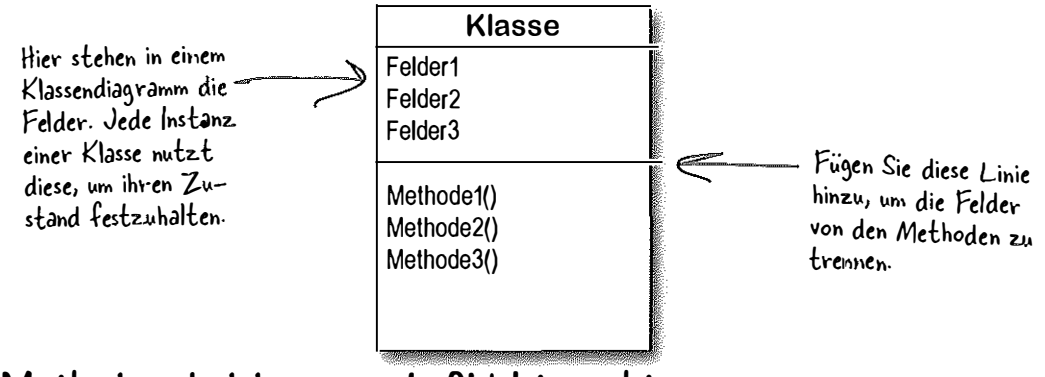
Eine Instanz nutzt Felder, um Dinge festzuhalten

Den Text auf einem Button ändern Sie, indem Sie in der IDE seine `Text`-Eigenschaft setzen. Wenn Sie das tun, fügt die IDE dem Designer Code wie den folgenden hinzu:

```
button1.Text = "Text für den Button";
```

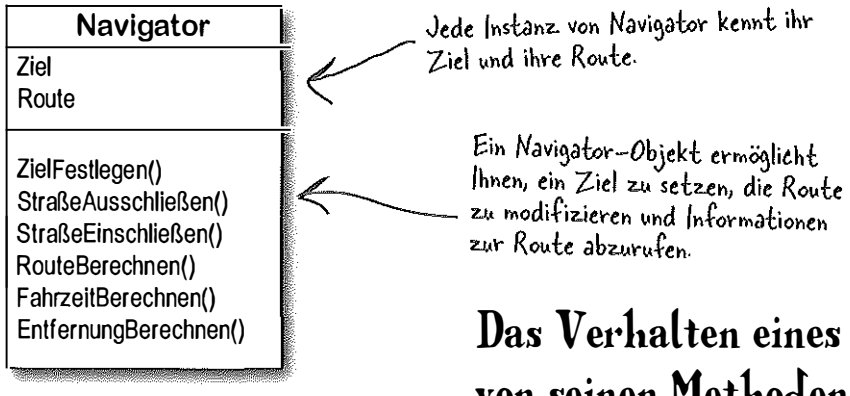
Technisch gesehen, wird dabei eine Eigenschaft gesetzt. Eine Eigenschaft ist einem Feld sehr ähnlich – aber dazu werden wir etwas später noch kommen.

Sie wissen jetzt, dass `button1` eine Instanz der Klasse `Button` ist. Dieser Code modifiziert ein **Feld** für die Instanz `button1`. Sie können einem Klassendiagramm Felder hinzufügen – ziehen Sie einfach in seiner Mitte eine horizontale Linie. Felder kommen über diese Linie, Methoden darunter.



Methoden sind das, was ein Objekt macht. Felder sind das, was das Objekt weiß.

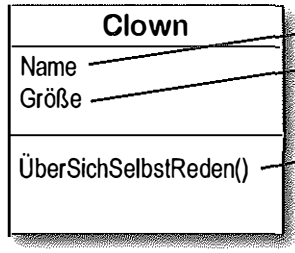
Als Mark seine drei Instanzen der Klasse `Navigator` schrieb, erstellte sein Programm drei Objekte. Jedes dieser Objekte wurde genutzt, um eine andere Route festzuhalten. Als das Programm die Instanz `navigator2` erstellte und seine `ZielFestlegen()`-Methode aufrief, setzte es das Ziel für diese eine Instanz. Das hatte jedoch keine Auswirkungen auf die `navigator1`- oder die `navigator3`-Instanz.



Das Verhalten eines Objekts wird von seinen Methoden definiert, seine Felder halten seinen Zustand fest.

Erstellen wir mal ein paar Instanzen!

Es ist leicht, Ihrer Klasse Felder hinzuzufügen. Deklarieren Sie einfach Variablen außerhalb ihrer Methoden. Jetzt erhält jede Instanz eine eigene Kopie dieser Variablen.



Denken Sie daran: Das »void« vor der Methode bedeutet, dass sie keinen Wert zurückliefert.

```
public class Clown {
  public String Name;
  public int Größe;

  public void ÜberSichSelbstReden() {
    MessageBox.Show("Ich heiße "
      + Name + " und bin "
      + Größe + " cm groß.");
  }
}
```

Wenn Sie Instanzen Ihrer Klasse erstellen wollen, dürfen Sie in der Klassen- und Methodendefinition das Schlüsselwort static nicht verwenden.

Bedenken Sie, dass der *=-Operator C# sagt, dass es den Wert links des Operators mit dem rechts des Operators multiplizieren soll.

Spitzen Sie Ihren Bleistift

Notieren Sie den Inhalt der Aussage, nachdem die Anweisung zu ihrer Linken ausgeführt wurde.

```
Clown einClown = new Clown();
einClown.Name = "Piff";
einClown.Größe = 30;
```

einClown.ÜberSichSelbstReden(); »Ich heiße _____ und bin _____ cm groß.«

```
Clown andererClown = new Clown();
andererClown.Name = "Paff";
andererClown.Größe = 40;
```

andererClown.ÜberSichSelbstReden(); »Ich heiße _____ und bin _____ cm groß.«

```
Clown clown3 = new Clown();
clown3.Name = andererClown.Name;
clown3.Größe = einClown.Größe - 8;
```

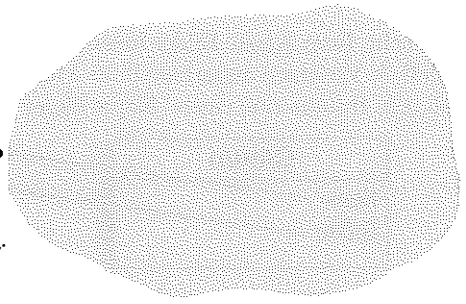
clown3.ÜberSichSelbstReden(); »Ich heiße _____ und bin _____ cm groß.«

```
andererClown.Größe *= 2;
```

andererClown.ÜberSichSelbstReden(); »Ich heiße _____ und bin _____ cm groß.«

Objekte auf dem Speicher

Wenn Ihr Programm ein Objekt erstellt, besteht dieses in einem Teil des Speichers des Computers, der als **Heap** (oder Halde) bezeichnet wird. Wenn Ihr Code mit der `new`-Anweisung ein Objekt erstellt, reserviert C# sofort Platz auf dem Heap, damit es die Daten für das Objekt speichern kann.



Hier ist ein Bild des Heaps, bevor das Projekt gestartet wurde. Beachten Sie, dass er leer ist.

Sehen wir uns genauer an, was hier passiert ist

Spitzen Sie Ihren Bleistift — Lösung

Notieren Sie den Inhalt der Aussage, nachdem die Anweisung zu ihrer Linken ausgeführt wurde.

Jede dieser new-Anweisungen erstellt eine Instanz der Klasse Clown, indem sie für dieses Objekt einen Speicherhappen auf dem Heap reserviert und mit den Daten des Objekts füllt.

```
Clown einClown = new Clown();
einClown.Name = "Piff";
einClown.Größe = 30;
einClown.ÜberSichSelbstReden();
```

»Ich heiße Piff und bin 30 cm groß.«

```
Clown andererClown = new Clown();
andererClown.Name = "Paff";
andererClown.Größe = 40;
andererClown.ÜberSichSelbstReden();
```

»Ich heiße Paff und bin 40 cm groß.«

```
Clown clown3 = new Clown();
clown3.Name = andererClown.Name;
clown3.Größe = einClown.Größe - 8;
clown3.ÜberSichSelbstReden();
```

»Ich heiße Paff und bin 22 cm groß.«

```
andererClown.Größe *= 2;
andererClown.ÜberSichSelbstReden();
```

»Ich heiße Paff und bin 80 cm groß.«

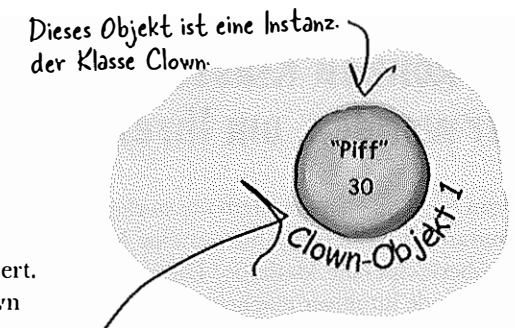
Erstellt Ihr Programm ein neues Objekt, kommt dieses auf den Heap.

Was Ihr Programm im Kopf hat

So erstellt Ihr Programm eine neue Instanz der Klasse `Clown`:

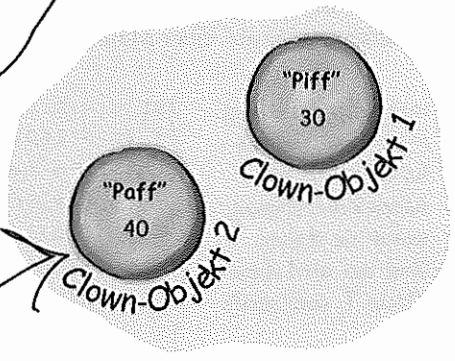
```
Clown meineInstanz = new Clown();
```

Eigentlich werden hier zwei Anweisungen zu einer kombiniert. Die erste Anweisung deklariert eine Variable des Typs Clown (`Clown meineInstanz;`). Die zweite erzeugt ein neues Objekt und weist es der Variablen zu, die gerade erstellt wurde (`meineInstanz= new Clown();`). So sieht der Heap nach den einzelnen Anweisungen aus:



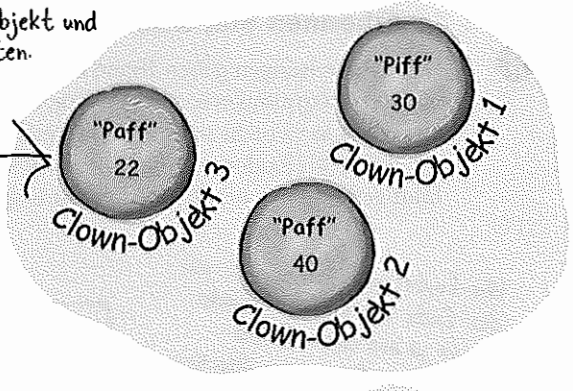
1
```
Clown einClown = new Clown();
einClown.Name = "Piff";
einClown.Größe = 30;
einClown.ÜberSichSelbstReden();
```

Das erste Objekt wird erstellt, und seine Felder werden gesetzt.



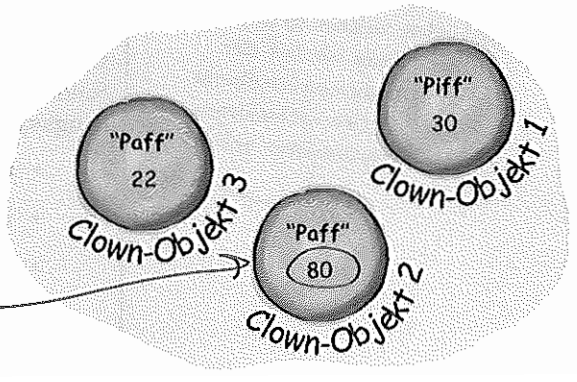
2
```
Clown andererClown = new Clown();
andererClown.Name = "Paff";
andererClown.Größe = 40;
andererClown.ÜberSichSelbstReden();
```

Diese Anweisungen erstellen das zweite Objekt und füllen es mit Daten.



3
```
Clown clown3 = new Clown();
clown3.Name = andererClown.Name;
clown3.Größe = einClown.Größe - 8;
clown3.ÜberSichSelbstReden();
```

Das dritte Clown-Objekt wird erstellt und gefüllt.



4
```
andererClown.Größe *= 2;
andererClown.ÜberSichSelbstReden();
```

Es gibt keinen `new`-Befehl. Das bedeutet, dass diese Anweisungen kein neues Objekt erzeugen. Sie verändern bloß eins, das bereits im Speicher ist

Sie können Klassen- und Methodennamen verwenden, um Ihren Code intuitiv zu machen

Wenn Sie Code in eine Methode stecken, treffen Sie eine Entscheidung über die Struktur Ihres Programms. Verwenden Sie nur eine Methode? Teilen Sie den Code auf mehrere auf? Brauchen Sie überhaupt eine Methode? Die Entscheidungen, die Sie zu Ihren Methoden treffen, können Ihren Code intuitiver machen – oder, wenn Sie nicht aufpassen, viel unübersichtlicher.

1 Hier ist ein netter, kompakter Happen Code. Er stammt aus einem Steuerungsprogramm, das auf einer Maschine läuft, die Schokoriegel herstellt:

```
int t = m.tmpPrf();
if (t > 160) {
  T tb = new T();
  tb.drklSchl(2);
  iks.Füllen();
  iks.Entlüften();
  m.luftsysPrf();
}
```

Die Methode tmpPrf() liefert einen int ... aber was macht sie?

>>obj<<, >>iks<< und >>m<< sind schreckliche Namen! Wir haben keine Ahnung, wofür sie stehen. Und was wird diese Klasse T wohl sein?

Die Methode drklSchl() hat einen Parameter, aber was das ist, wissen wir nicht.

Werfen Sie einen genaueren Blick auf den Code. Erkennen Sie, was er macht?

2 Diese Anweisungen geben Ihnen keinerlei Hinweise darauf, warum der Code macht, was er macht. Hier war der Programmierer mit den Ergebnissen zufrieden, weil er alles in eine Methode stecken konnte. Aber eigentlich ist es nicht sonderlich hilfreich, wenn man seinen Code so kompakt wie möglich macht! Zerlegen wir ihn in Methoden, damit er besser lesbar wird, und kümmern wir uns darum, dass die Klassen aussagekräftigere Namen erhalten. Aber wir beginnen damit herauszufinden, was der Code überhaupt machen soll.

Wie findet man nun heraus, was der Code tun soll? Der Code wurde ja aus einem Grund geschrieben. Sie müssen also herausfinden, welcher Grund das ist! Hier können wir auf der Seite des Bedienungshandbuchs die Spezifikation nachschlagen, der der Programmierer gefolgt ist.

Süß & Sauer Schokoriegel-Maschine B2B
Bedienungshandbuch

Die Schokoladentemperatur muss alle drei Minuten von einem automatisierten System geprüft werden. Übersteigt die Temperatur **160 °C**, ist die Schokolade zu heiß. Das System muss dann die **Schoko-Isolationskühlungssystem-Entlüftungsprozedur (S-IKS-EP)** einleiten:

- Drosselklappe an Turbine 2 schließen.
- Das Isolationskühlungssystem mit Wasser füllen.
- Das Wasser entlüften.
- Sicherstellen, dass im System keinerlei Luftspuren verbleiben.

3. Diese Handbuchseite hat es deutlich leichter gemacht, den Code zu verstehen. Außerdem hat sie uns ein paar gute Hinweise darauf gegeben, wie wir den Code verständlicher machen können. Wir wissen jetzt, warum die Testbedingung die Variable t mit 160 vergleicht – das Handbuch sagt, dass eine Temperatur über 160 °C bedeutet, dass die Schokolade zu heiß ist. Und es stellt sich heraus, dass »m« eine Klasse ist, die die Schokoriegel-Maschine steuert. Sie hat statische Methoden zur Prüfung der Schokoladentemperatur und des Lüftungssystems. Packen wir die Temperaturprüfung also in eine Methode und wählen wir für die Klasse und die Methoden Namen, die ihren Zweck klarer machen.

```
public boolean IstSchokoZuHeiß() {
    int temp = Maschine.SchokoTemperaturPrüfen();
    if (temp > 160) {
      return true;
    } else {
      return false;
    }
}
```

Der Rückgabetyp der Methode IstSchokoZuHeiß().

Der Code ist viel verständlicher, wenn wir der Klasse den Namen »Maschine« und der Methode den Namen »SchokoTemperaturPrüfen« geben.

Der Rückgabetyp der Methode ist boolean. Das bedeutet, sie liefert entweder true oder false.

4. Was soll man laut Spezifikation tun, wenn die Schokolade zu heiß ist? Sie sagt uns, dass wir die Schoko-Isolationskühlungssystem-Entlüftungsprozedur (oder S-IKS-EP) einleiten sollen. Schreiben wir also eine weitere Methode und wählen wir einen klareren Namen für die Klasse »T« (die offensichtlich die Turbine steuert) und die Klasse »iks« (die das Isolationskühlungssystem steuert und zwei statische Methoden zum Füllen und Entlüften des Systems hat):

```
public void IKSEntlüftungsProzedurAusführen() {
    Turbine turbinensteuerung = new Turbine();
    turbinensteuerung.DrosselklappeSchließen(2);
    IsolationsKühlungsSystem.Füllen();
    IsolationsKühlungsSystem.Entlüften();
    Maschine.LuftSystemPrüfen();
}
```

Der Rückgabetyp void bedeutet, dass die Methode keinen Wert zurückliefert.

5. Jetzt ist der Code viel intuitiver! Selbst wenn Sie nicht wissen, dass die S-IKS-Entlüftungsprozedur eingeleitet werden muss, wenn die Schokolade zu heiß ist, **ist es jetzt viel offensichtlicher, was der Code macht.**

```
if (IstSchokoZuHeiß() == true) {
    IKSEntlüftungsProzedurAusführen();
}
```

Indem Sie auf das Problem blicken, das Ihr Code lösen soll, können Sie ihn besser lesbar und schreibbar machen. Wählen Sie Methodennamen, die für jemanden sinnvoll sind, der das Problem versteht, ist Ihr Code viel leichter zu entziffern ... und zu entwickeln!

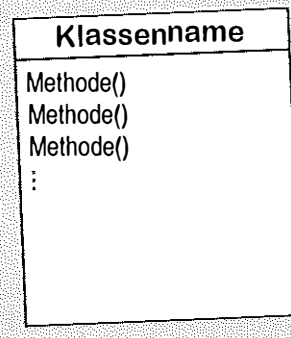
Geben Sie Ihren Klassen eine natürliche Struktur

Nehmen Sie sich einen Augenblick Zeit und überlegen Sie, warum Sie Ihre Methoden intuitiver machen möchten: **weil jedes Programm ein Problem löst oder einen Zweck hat.** Es muss kein Geschäftsproblem sein – manchmal soll ein Programm (wie unsere Farbspielerei) einfach nur cool sein oder Spaß machen! Aber ganz egal, was Ihr Programm macht, je mehr Sie Ihren Code dem zu lösenden Problem angleichen, umso leichter lässt sich das Programm schreiben (und lesen, reparieren, warten ...).

Planen Sie Ihre Klassen mit Klassendiagrammen

Ein Klassendiagramm ist ein einfaches Mittel, um Ihre Klassen auf Papier zu bringen. Es ist ein wirklich wertvolles Werkzeug zum Entwerfen Ihres Codes, BEVOR Sie damit beginnen, ihn zu schreiben.

Schreiben Sie oben in das Diagramm den Namen der Klasse. Schreiben Sie die Methoden in den Kasten darunter. Jetzt können Sie alle Teile auf einen Blick sehen!



Erstellen wir ein Klassendiagramm

Werfen wir einen weiteren Blick auf die `if`-Anweisung in Schritt 5 auf der letzten Seite. Sie wissen bereits, dass Anweisungen immer innerhalb von Methoden stehen, die immer in Klassen sind, stimmt's? Hier befand sich diese `if`-Anweisung in einer Methode namens `WartungsTestDurchführen()`, die Teil der Klasse `SchokoriegelSteuerung` ist. Werfen Sie jetzt einen Blick auf den Code und das Klassendiagramm. Sehen Sie, wie diese sich aufeinander beziehen?

```
public class SchokoriegelSteuerung {

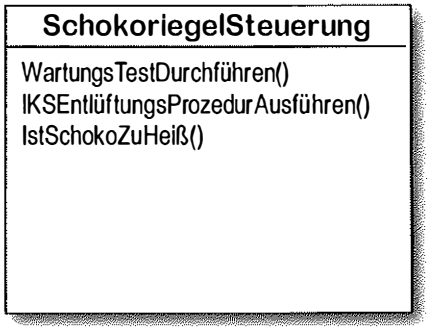
  public void WartungsTestDurchführen() {
    ...
    if (IstSchokoZuHeiß() == true) {
       IKSEntlüftungsProzedurAusführen();
    }
    ...
  }

  public void IKSEntlüftungsProzedurAusführen() ...

  public boolean IstSchokoZuHeiß() ...

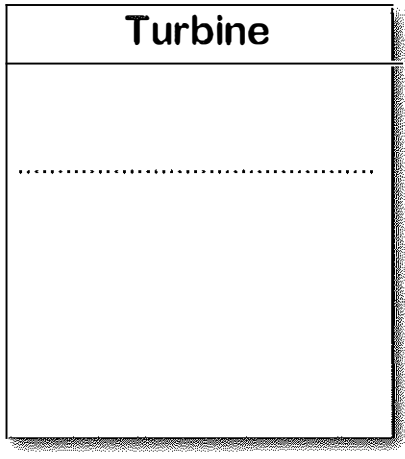
}
```



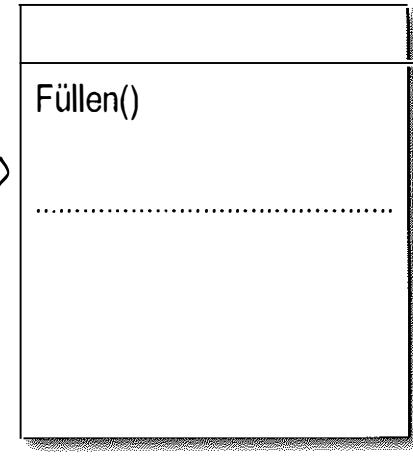


Spitzen Sie Ihren Bleistift

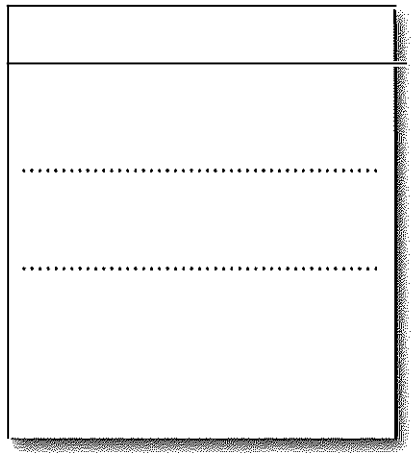
Der Code für das Schokoriegelsteuerungssystem, das wir auf der vorangehenden Seite aufgebaut haben, ruft drei weitere Klassen auf. Blätten Sie zurück, sehen Sie sich den Code an und füllen Sie die entsprechenden Klassendiagramme aus.



Hier haben wir für Sie den Klassennamen eingegeben. Welche Methode kommt hier hin?



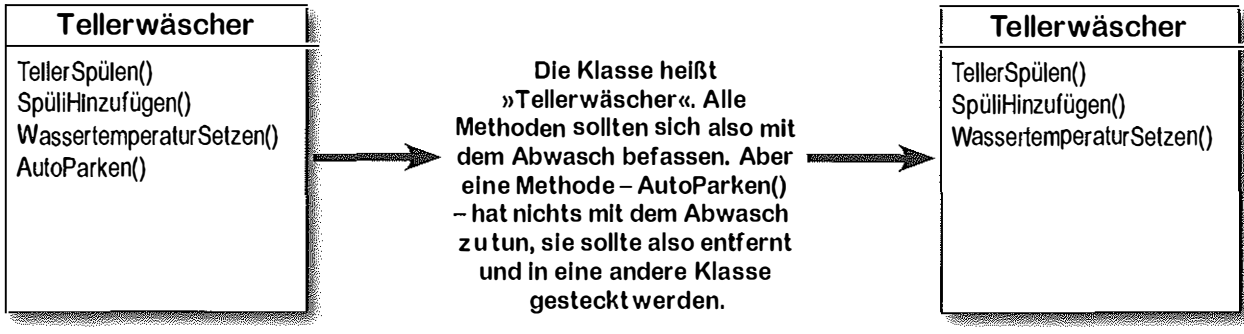
Eine der Klassen hatte eine Methode namens Füllen(). Ergänzen Sie den Klassennamen und die andere Methode.



Im Code auf der vorangegangenen Seite gab es noch eine weitere Klasse. Tragen Sie ihren Namen und die Methoden ein.

Klassendiagramme helfen Ihnen, Ihre Klassen auf sinnvolle Weise zu organisieren

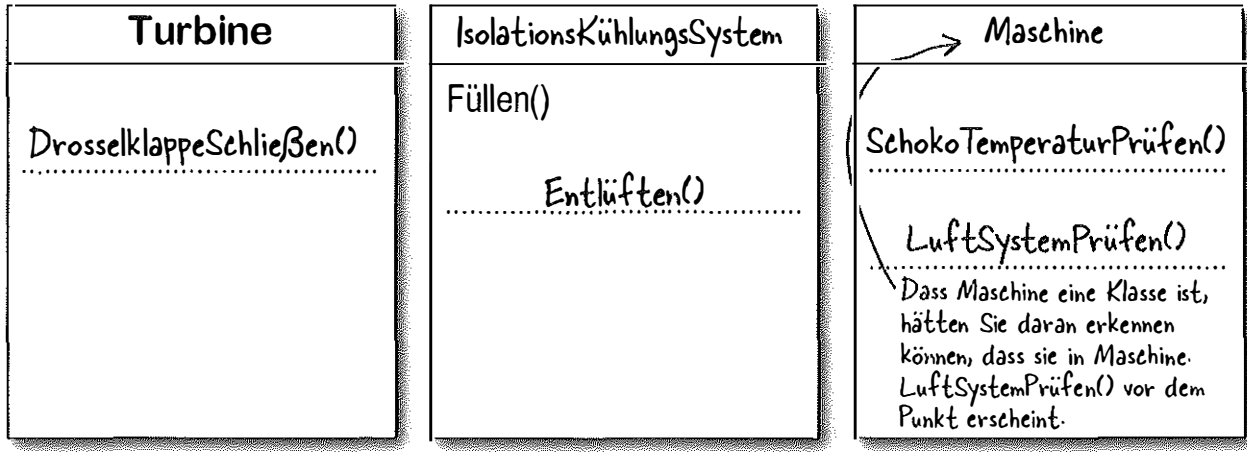
Das Formulieren von Klassendiagrammen macht es erheblich einfacher, mögliche Probleme in Ihren Klassen zu entdecken, **bevor** Sie Code schreiben. Über Ihre Klassen von einem übergeordneten Standpunkt aus nachzudenken, bevor Sie an die Details gehen, kann Ihnen helfen, eine Klassenstruktur zu entwickeln, die sicherstellt, dass Ihr Code die Probleme löst, mit denen er sich befasst. So können Sie einen Schritt zurücktreten und sicherstellen, dass Sie keine unnötigen oder schlecht strukturierten Klassen oder Methoden schreiben und dass die Klassen, die Sie schreiben, intuitiv und leicht verwendbar sind.



Spitzen Sie Ihren Bleistift

Lösung

Der Code für das Schokoriegelsteuerungssystem, das wir auf der vorangehenden Seite aufgebaut haben, ruft drei weitere Klassen auf. Blätten Sie zurück, sehen Sie sich den Code an und füllen Sie die entsprechenden Klassendiagramme aus.



Spitzen Sie Ihren Bleistift

Jede dieser Klassen weist einen ernsthaften Entwurfsfehler auf. Schreiben Sie auf, was Ihrer Meinung nach mit den Klassen nicht in Ordnung ist und wie Sie es reparieren würden.



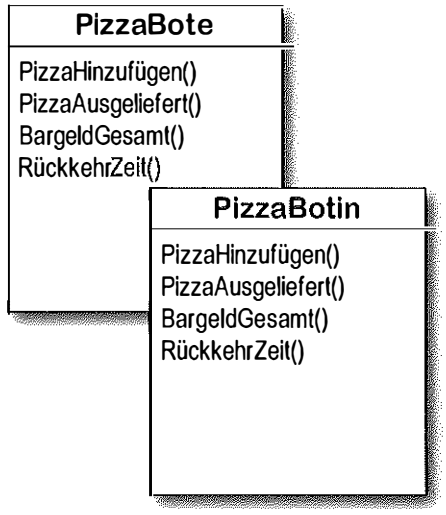
Diese Klasse ist Teil des Schokoriegel-Systems von oben.

.....

.....

.....

.....



Diese beiden Klassen sind Teil eines Systems, das ein Pizzadienst einsetzt, um die Auslieferung von Bestellungen nachzuhalten.

.....

.....

.....

.....



Die Klasse `Kassenbuch` ist Teil eines Programms, das von dem automatisierten Kassensystem eines Kaufhauses verwendet wird.

.....

.....

.....

.....

Spitzen Sie Ihren Bleistift
Lösung

So haben wir diese Klassen korrigiert. Das ist aber nur jeweils ein möglicher Weg, diese Probleme zu reparieren – es gibt viele andere, um diese Klassen auf Basis ihrer Verwendung zu entwerfen.

Diese Klasse ist Teil des Schokoriegel-System von oben.

Der Klassenname sagt nicht, was die Klasse macht. Ein Programmierer, der eine Codezeile sieht, die Klasse2.3.Los() aufruft, hat absolut keinen Schimmer, was diese Zeile macht. Wir sollten auch der Methode einen anschaulicheren Namen geben – wir haben Schokoriegelherstellen() gewählt, er könnte aber auch anders lauten.

SchokoriegelMaschine

SchokoriegelGewicht()
VerpackungDrucken()
BerichtErzeugen()
SchokoriegelHerstellen()

Diese beiden Klassen sind Teil eines Systems, das ein Pizzadienst einsetzt, um die Auslieferung von Bestellungen nachzuhalten.

Die Klassen PizzaBote und PizzaBotin scheinen das Gleiche zu tun – sie halten den Boten fest, der die Pizza ausliefert. Ein besserer Entwurf würde sie durch eine einzige Klasse ersetzen, die ein Feld für das Geschlecht enthält.

PizzaBote

Geschlecht

PizzaHinzufügen()
PizzaAusgeliefert()
BargeldGesamt()
RückkehrZeit()

Wir haben das Feld Geschlecht hinzugefügt, weil wir davon ausgehen, dass es einen Grund gibt, weibliche und männliche Boten getrennt nachzuhalten, und es deswegen zwei Klassen für sie gab.

Die Klasse `Kassenbuch` ist Teil eines Programms, das von dem automatisierten Kassensystem eines Kaufhauses verwendet wird.

Alle Methoden in der Klasse tun Dinge, die mit der Kasse zu tun haben – einen Verkauf ausführen, eine Liste der Transaktionen abrufen, Bargeld hinzufügen ... nur eine nicht: betanken. Diese Methode sollte herausgezogen und in eine andere Klasse gesteckt werden.

Kassenbuch

VerkaufDurchführen()
KeinVerkauf()
BargeldWiederauffüllen()
GeldbestandInKasse()
TransaktionslisteAbrufen()
BargeldHinzufügen()
BargeldHerausnehmen()

```
public partial class Form1 : Form
{
  private void button1_Click(object sender, EventArgs e)
  {
    String ergebnis = "";
    Echo e1 = new Echo();

    ____________________

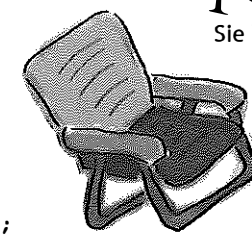
    int x = 0;
    while ( ____________ ) {
      ergebnis = ergebnis + e1.hallo() + "\n";

      ____________________

      if ( ____________ ) {
        e2.zähler = e2.zähler + 1;
      }
      if ( ____________ ) {
        e2.zähler = e2.zähler + e1.zähler;
      }
      x = x + 1;
    }
    MessageBox.Show(ergebnis + "Zähler: " + e2.zähler);
  }

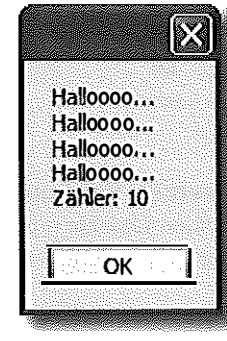
  public class ____________ {
    public int _________ = 0;
    public string ___________ {
      return "Halloooo...";
    }
  }
}
```

Pool-Puzzle



Sie haben die ***Aufgabe***, die leeren Zeilen im Code mit den Codeschnipseln aus dem Pool zu füllen. Einzelne Schnipsel **können** mehrfach verwendet werden, und Sie werden nicht alle Schnipsel benötigen. Das ***Ziel*** ist, eine Klasse zu erstellen, die sich kompilieren lässt und die gezeigte Ausgabe erzeugt. Lassen Sie sich nicht täuschen – das ist schwerer, als es scheint.

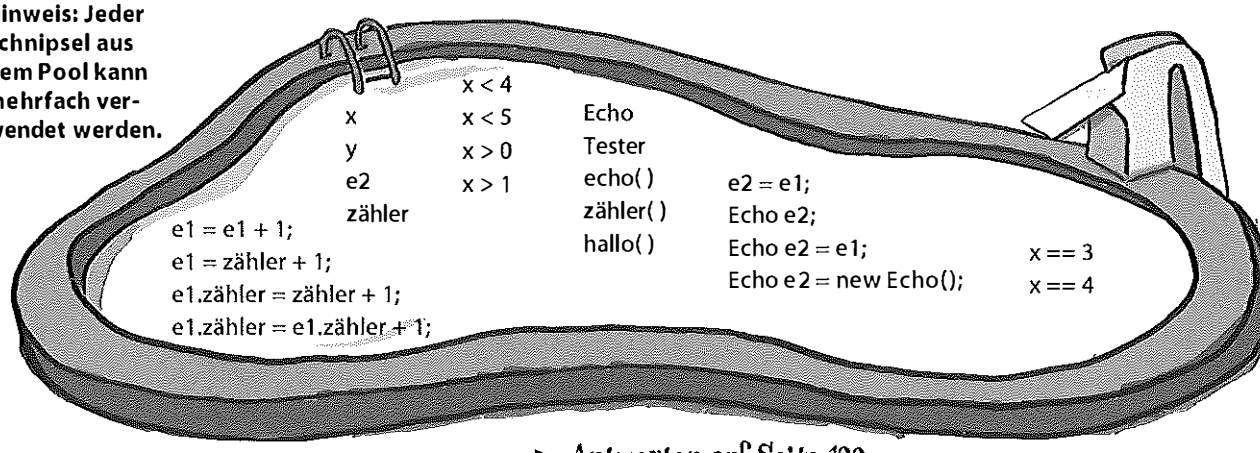
Ausgabe



Bonusfrage!

Wie würden Sie das Puzzle vervollständigen, wenn die letzte Zeile der Ausgabe **24** statt **10** wäre? Sie müssen dazu nur eine Anweisung ändern.

Hinweis: Jeder Schnipsel aus dem Pool kann mehrfach verwendet werden.



Antworten auf Seite 120.

Eine Klasse für die Arbeit mit ein paar Typen erstellen

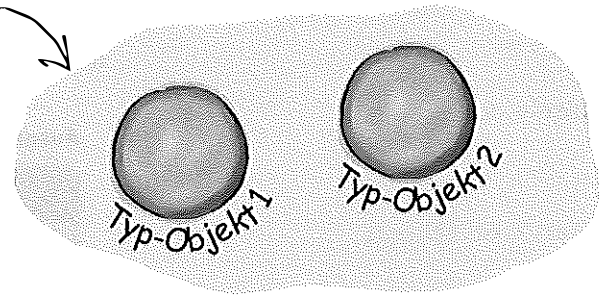
Tim und Tom leihen einander ständig Geld. Bauen wir also eine Klasse auf, um das nachzuhalten.



1 **Wir erstellen die Klasse Typ und fügen einem Formular zwei Instanzen hinzu.**

Das Formular hat zwei Felder, eins namens `tim` (um das erste Objekt nachzuhalten) und eins namens `tom` (um das zweite Objekt nachzuhalten).

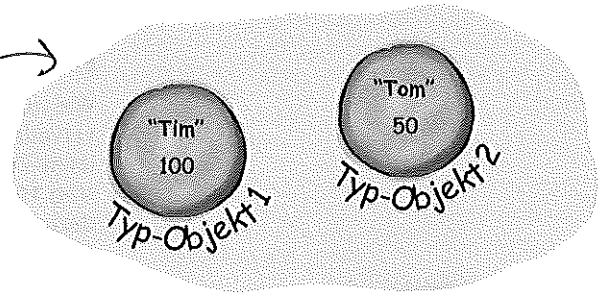
Die new-Anweisungen, die die beiden Instanzen erstellen, stehen in Code, der ausgeführt wird, sobald das Formular erstellt wird. So sieht der Heap aus, nachdem das Formular geladen wurde.



2 **Wir setzen Geld und Name für beide Typ-Objekte.**

Die beiden Objekte repräsentieren verschiedene Typen, jedes hat also einen eigenen Namen und auch einen anderen Betrag Bargeld in der Tasche.

Jeder Typ hat ein Name-Feld, das seinen Namen festhält, und ein Geld-Feld, das die Knete in seiner Börse festhält.



3 **Wir tauschen Geld zwischen den Jungs aus.**

Wie nutzen die `GeldVerleihen()`-Methode des einen Typs, um dem anderen Geld zu geben, und wir nutzen die `GeldEmpfangen()`-Methode, um Geld von ihm zurückzunehmen.

Wenn Sie eine Instanz von Typ nehmen und seine GeldEmpfangen()-Methode aufrufen, übergeben Sie den Geldbetrag als Parameter. Ein Aufruf von tim.GeldEmpfangen(25) nimmt also 25 Euro von Tim.

Das Formular ruft die GeldEmpfangen()-Methode des Objekts auf.

```
tim.GeldEmpfangen(25);
```

Die Methode liefert die Anzahl an Euros zurück, die angenommen wurden.



Ein Projekt für die Jungs erstellen



Erzeugen Sie ein neues Windows-Forms-Anwendung-Projekt (da wir ein Formular verwenden werden). Fügen Sie diesem dann über den Projektmappen-Explorer eine neue Klasse namens Typ hinzu. Achten Sie darauf, dass Sie am Anfang der Klassendatei für Typ »`using System.Windows.Forms;`« einfügen. Geben Sie dann die Klasse Typ ein. Hier ist der Code dafür:

Die Klasse Typ hat zwei Felder. Das Feld Name ist ein String und enthält den Namen des Typs (»Tim«). Und das Feld Geld ist ein int und hält fest, wie viel er in der Tasche hat.

Die Methode GeldVerleihen() hat einen Parameter namens betrag, den Sie nutzen, um dem Typ zu sagen, wie viel Geld er Ihnen geben soll.

```
public class Typ {
    public string Name;
    public int Geld;

    public int GeldVerleihen(int betrag) {
        if (betrag <= Geld && betrag > 0) {
            Geld -= betrag;
            return betrag;
        } else {
            MessageBox.Show(
             "Ich habe nicht genug, um dir " + betrag,
             " zu geben, sagt " + Name );
            return 0;
        }
    }
```

Der Typ vergewissert sich, dass er um einen positiven Geldbetrag gebeten wird, da er seinem Geld sonst etwas hinzufügt, anstatt etwas davon wegzunehmen.

Mit einer if-Anweisung prüft er, ob er genug Geld hat – ist das der Fall, nimmt er es aus der Tasche und liefert es als Rückgabewert zurück.

Hat der Typ nicht genug Geld, sagt er das über ein Dialogfenster und lässt GeldVerleihen() 0 zurückliefern.

Die Methode GeldEmpfangen() funktioniert genau wie die Methode GeldVerleihen(). Ihr wird als Parameter ein Betrag übergeben, sie stellt sicher, dass dieser Betrag größer null ist, und fügt ihn dann dem eigenen Geld hinzu.

```
    public int GeldEmpfangen(int betrag) {
        if (betrag > 0) {
            Geld += betrag;
            return betrag;
        } else {
            MessageBox.Show(betrag + " ist kein Betrag, den ich",
                  " akzeptiere, sagt " + Name);
            return 0;
        }
    }
```

War betrag positiv, liefert die Methode GeldEmpfangen() den hinzugefügten Betrag zurück. War betrag null oder kleiner, wird ein Dialogfenster eingeblendet und dann 0 zurückgeliefert.

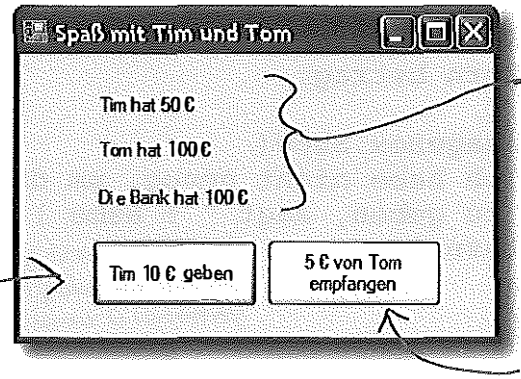
Geben Sie acht auf Ihre geschweiften Klammern. Es passiert schnell, dass ihre Anzahl nicht stimmt – prüfen Sie, ob es zu jeder öffnenden Klammer eine entsprechende schließende gibt. Sind die Klammern im Gleichgewicht, rückt die IDE sie automatisch für Sie ein, wenn Sie die letzte schließende Klammer eingeben.

Erstellen Sie ein Formular zur Interaktion mit den Jungs

Die Klasse Typ ist gut, aber sie ist erst der Anfang. Bauen Sie jetzt ein Formular auf, das zwei Instanzen der Klasse Typ nutzt. Es hat Labels, die Ihnen ihre Namen und den Geldbetrag anzeigen, den sie in der Tasche haben, sowie Buttons, mit denen von ihnen Geld angenommen bzw. ihnen Geld gegeben werden kann.



1 **Fügen Sie Ihrem Formular zwei Buttons und drei Labels hinzu.**
Die beiden oberen Labels zeigen, wie viel Geld die einzelnen Typen haben. Außerdem haben wir dem Formular eine Variable `bank` hinzugefügt – das dritte Label zeigt, wie viel Geld darin ist. Sie werden einigen der Labels, die Sie auf das Formular ziehen, Namen geben müssen. Das machen Sie, indem Sie **auf das Label klicken**, dessen Namen Sie ändern möchten, und im Eigenschaftenfenster **die Zeile »(Name)« ändern**. Das verbessert die Lesbarkeit Ihres Codes, weil Sie nun mit »timsGeld« und »tomsGeld« statt mit »label1« und »label2« arbeiten können.



Geben Sie dem obersten Label den Namen timsGeld, dem Label darunter den Namen tomsGeld und dem untersten Label den Namen bankGeld. Die Text-Eigenschaften können Sie belassen. Diese werden wir über eine Methode setzen.

Dieser Button ruft die GeldEmpfangen()-Methode des Tim-Objekts auf und zieht das Geld, das Tim übergeben wird, von der Variablen bank ab.

Dieser Button ruft die GeldVerleihen()-Methode des Tom-Objekts auf, übergibt ihr als betrag 5 und addiert der Variablen bank das Geld hinzu, das von Tom empfangen wird.

2 **Fügen Sie Ihrem Formular die Variablen hinzu.**
Ihr Formular muss die Jungs festhalten, Sie benötigen also eine Variable für jeden von beiden. Nennen Sie diese `tim` und `tom`. Fügen Sie dem Formular dann eine Variable namens `bank` hinzu, die festhält, wie viel Geld das Formular von Tom empfangen hat bzw. Tim geben musste.

```
namespace Ihr_Projektname {
    public partial class Form1 : Form {
        Typ tim;
        Typ tom;
        int bank = 100;

        public Form1() {
            InitializeComponent();
        }
```

Da wir Tim und Tom mit Typ-Objekten festhalten, deklarieren wir die entsprechenden Variablen mit Typ.

Der Geldbetrag in der Bank steigt und sinkt, je nachdem, wie viel Geld das Formular den Typ-Objekten gegeben bzw. von ihnen empfangen hat.

3 Fügen Sie dem Formular eine Methode zur Aktualisierung der Labels hinzu.

Die Labels im Formular zeigen, wie viel Geld die Jungs haben und wie viel in der Variablen `bank` ist. Fügen Sie also die Methode `FormularAktualisieren()` hinzu, um sie aktuell zu halten – **achten Sie darauf, dass der Rückgabetyp `void` ist**, damit C# weiß, dass die Methode keinen Wert zurückliefert. Geben Sie diese Methode direkt unterhalb der Stelle in das Formular ein, an der Sie die Variable `bank` eingefügt haben:

Beachten Sie, wie die Labels mit den Name- und Geld-Feldern der Typ-Objekte aktualisiert werden.

```
public void FormularAktualisieren() {
    timsGeld.Text = tim.Name + " hat " + tim.Geld + " €";
    tomsGeld.Text = tom.Name + " hat " + tom.Geld + " €";
    bankGeld.Text = "Die Bank hat " + bank + " €";
}
```

Die neue Methode muss von beiden Buttons aufgerufen werden, damit die Labels aktuell bleiben.

4 Fügen Sie den Buttons den Code für die Interaktion mit den Objekten hinzu.

Achten Sie darauf, dass der linke Button `button1` heißt und der rechte `button2`. Klicken Sie doppelt auf beide Buttons – die IDE fügt dem Formular dann die Methoden `button1_Click()` und `button2_Click()` hinzu. Fügen Sie beiden diesen Code hinzu:

```
private void button1_Click(object sender, EventArgs e) {
    if (bank >= 10) {
        bank -= tim.GeldEmpfangen(10);
        FormularAktualisieren();
    } else {
        MessageBox.Show("Die Bank ist pleite.");
    }
}

private void button2_Click(object sender, EventArgs e) {
    bank += tom.GeldVerleihen(5);
    FormularAktualisieren();
}
```

Klickt der Benutzer auf den Button »Tim 10 € geben«, wird die GeldEmpfangen()-Methode von Tims Objekt aufgerufen – aber nur, wenn die Bank genug Geld hat.

Die Bank braucht mindestens 10 €, damit sie Tim etwas geben kann. Ist nicht genug da, zeigt sie dieses Dialogfenster an.

Der Button »5 € von Tom empfangen« muss nicht prüfen, wie viel Geld die Bank hat, da sie nur hinzufügt, was Tom ihr gibt.

Hat Tom kein Geld mehr, liefert GeldVerleihen() null zurück.



3 Geben Sie Tim zu Beginn 50 € und Tom 100 €.

Wie die Geld- und Name-Felder für Tim und Tom am Anfang gesetzt werden, sollen Sie selbst herausfinden. Stecken Sie den entsprechenden Code direkt unter `InitializeComponent()` ins Formular. Das ist ein Teil einer besonderen Methode, die einmal ausgeführt wird, wenn das Formular initialisiert wird. Klicken Sie im Anschluss daran mehrfach auf beide Buttons –, prüfen Sie, ob der eine Button 10 € aus der Bank nimmt und Tim gibt und der andere 5 € von Tom nimmt und der Bank gibt.

```
public Form() {
    InitializeComponent();
    // Initialisieren Sie hier tim und tom!
}
```

Fügen Sie hier die Codezeilen ein, die zwei Objekte erzeugen und ihre Name- und Geld-Felder setzen.



Lösungen zu den Übungen

Wie die Geld- und Name-Felder für Tim und Tom am Anfang gesetzt werden, sollen Sie selbst herausfinden. Stecken Sie den entsprechenden Code direkt unter `InitializeComponent()` ins Formular.

```
public Form1() {
    InitializeComponent();

    tom = new Typ();
    tom.Name = "Tom";
    tom.Geld = 100;

    tim = new Typ();
    tim.Name = "Tim";
    tim.Geld = 50;

    FormularAktualisieren();
}
```

Hier richten wir die erste Instanz von Typ ein. Die erste Zeile erzeugt ein Objekt, und die nächsten beiden setzen seine Eigenschaften.

Dann machen wir das Gleiche für die zweite Instanz der Klasse Typ.

Achten Sie darauf, dass Sie FormularAktualisieren() aufrufen, damit die Labels schon richtig aussehen, wenn das Formular das erste Mal aufgerufen wird.

Es gibt keine Dummen Fragen

Speichern Sie das Projekt, wir kehren in ein paar Seiten zu ihm zurück.

F: Warum beginnt die Lösung nicht mit »`Typ tom = new Typ()`«? Warum haben Sie das erste »`Typ`« weggelassen?

A: Weil Sie das Feld `tom` bereits am Anfang des Formulars deklariert haben. Erinnern Sie sich, dass wir gesagt hatten, die Anweisung »`int i = 5;`« entspräche den zwei Anweisungen »`int i`« und »`i = 5;`«?Das hier ist das Gleiche. Sie könnten versuchen, das Feld `tom` in einer einzigen Zeile wie der folgenden zu deklarieren: »`Typ tom = new Typ();`«. Aber der erste Teil dieser Anweisung (»`Typ tom;`«) steht schon am Anfang des Formulars. Sie brauchen also nur die zweite Hälfte der Zeile, den Teil, der das Feld `tom` auf die neu erzeugte Instanz von `Typ` setzt.

F: Aber warum kann ich nicht einfach die »`Typ tom;`«-Zeile am Anfang des Formulars löschen?

A: Dann ist die Variable `tom` nur in der Methode »`public Form1()`« definiert. Wenn Sie eine Variable in einer Methode deklarieren, ist sie nur in dieser Methode gültig – in anderen Methoden können Sie nicht auf sie zugreifen. Deklarieren Sie sie hingegen außerhalb der Methode, aber in einem Formular oder einer Klasse, können Sie auf sie aus ***jeder Methode*** in diesem Formular oder dieser Klasse zugreifen.

F: Was passiert, wenn ich das erste »`Typ`« nicht weglasse?

A: Dann kriegen Sie Probleme – Ihr Formular funktioniert nicht, weil die Variable `tom` nie gesetzt wird. Denken Sie eine Minute darüber nach, werden Sie erkennen, warum das auf diese Weise funktioniert. Steht am Anfang Ihres Formulars diese Zeile:

```
public partial class Form1 : Form {
    Typ tom;
```

und später in Ihrem Code in einer Methode Folgendes:

```
Typ tom = new Typ();
```

haben Sie ***zwei*** Variablen deklariert. Das ist etwas verwirrend, weil beide den gleichen Namen haben. Aber eine von ihnen ist im gesamten Formular gültig und die andere – die neu hinzugefügte – nur innerhalb dieser Methode. Die nächste Zeile (`tom.Name = "Tom";`) aktualisiert nur diese ***lokale*** Variable, ohne die im Formular anzurühren. Versuchen Sie dann, Ihren Code auszuführen, erhalten Sie daher eine gemeine Fehlermeldung (»NullReferenceException wurde nicht behandelt«), die einfach sagt, dass Sie versucht haben, ein Objekt zu verwenden, bevor Sie es mit `new` erstellt haben.

Die Objektinitialisierung kann noch einfacher sein

Fast jedes Objekt, das Sie erstellen, muss auf irgendeine Weise initialisiert werden. Das Typ-Objekt ist da keine Ausnahme – es ist vollkommen nutzlos, bis seine Name- und Geld-Felder gesetzt sind. Die Initialisierung von Feldern ist so häufig erforderlich, dass C# eine Kurzform dafür bietet, die als **Objektinitialisierer** bezeichnet wird. Und die IntelliSense-Funktion der IDE hilft Ihnen, diese zu nutzen.



Objektinitialisierer funktionieren nur in C# 3.0.

Mit Visual Studio 2005 funktioniert das nicht. Sie sollten unbedingt überlegen, die Visual Studio 2008 Express Edition herunterzuladen – sie ist kostenlos und kann neben VS2005 installiert werden.

1. Hier ist der Code, den Sie zur Initialisierung von Tims Typ-Objekt geschrieben haben.

```
tim = new Typ();
tim.Name = "Tim";
tim.Geld = 50;
```

2. Löschen Sie die beiden letzten Zeilen und das Semikolon nach »Typ()« und geben Sie eine öffnende geschweifte Klammer ein.

```
tim = new Typ() {
```

3. Drücken Sie die Leertaste. Die IDE blendet dann ein IntelliSense-Fenster ein, in dem Sie alle Felder sehen, die Sie initialisieren können.

```
tim = new Typ() {
```



4. Drücken Sie die Tabulatortaste, um das Feld Geld hinzuzufügen, und setzen Sie es dann auf 50.

```
tim = new Typ() { Geld = 50
```

5. Geben Sie ein Komma ein. Dann wird sofort das andere Feld eingeblendet.

```
tim = new Typ() { Geld = 50,
```



6. Schließen Sie den Objektinitialisierer. Damit haben Sie sich zwei Zeilen Code gespart!

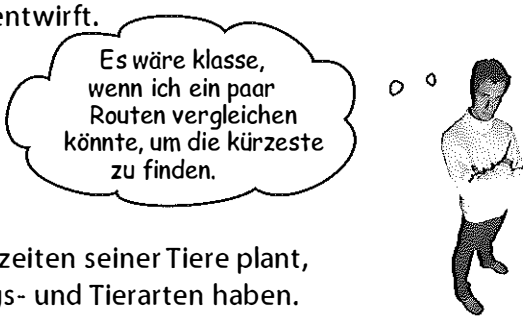
```
tim = new Typ() { Geld = 50, Name = "Tim" };
```

Diese neue Deklaration macht das Gleiche wie die drei Codezeilen, die Sie ursprünglich geschrieben hatten. Sie ist nur kürzer und besser lesbar.

Objektinitialisierer sparen Ihnen Zeit und machen Ihren Code kompakter und besser lesbar ... und die IDE hilft Ihnen dabei, sie zu schreiben.

Ein paar Ideen zum Entwurf intuitiver Klassen

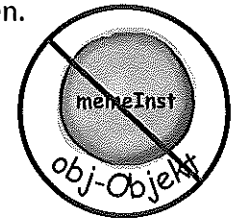
★ **Sie erstellen Ihr Programm, um ein Problem zu lösen.**
Nehmen Sie sich die Zeit, über das Problem nachzudenken. Lässt es sich leicht in Einheiten zerlegen? Wie würden Sie es einem anderen erklären? Das sind Dinge, über die man nachdenken sollte, wenn man Klassen entwirft.



Wer oder was wird Ihr Programm nutzen?
Ein Programm, mit dem ein Tierpfleger die Fütterungszeiten seiner Tiere plant, könnte separate Klassen für unterschiedliche Nahrungs- und Tierarten haben.

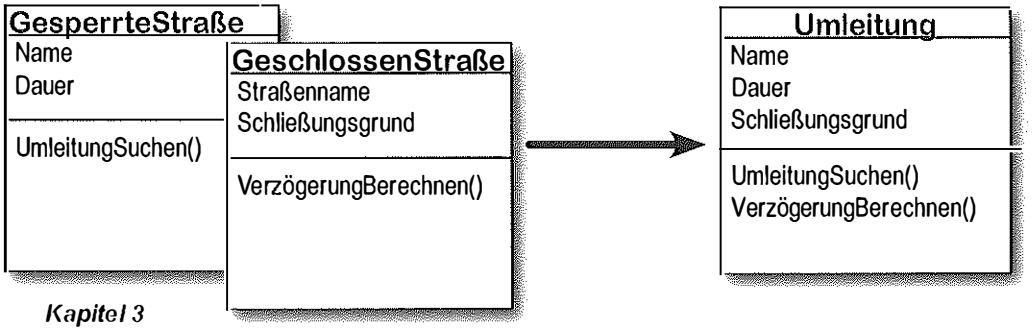


★ **Nutzen Sie aussagekräftige Namen für Klassen und Methoden.**
Was Ihre Klassen und Methoden tun, sollte jemand, der sie betrachtet, allein anhand ihrer Namen herausfinden können.





★ **Achten Sie auf Ähnlichkeiten zwischen Klassen.**
Manchmal können zwei Klassen zu einer kombiniert werden, wenn sie wirklich ähnlich sind. Das Schokoriegel-System kann drei oder vier Turbinen haben, aber es gibt nur eine Methode zum Schließen der Drosselklappe, die die Nummer der Turbine als Parameter erwartet.





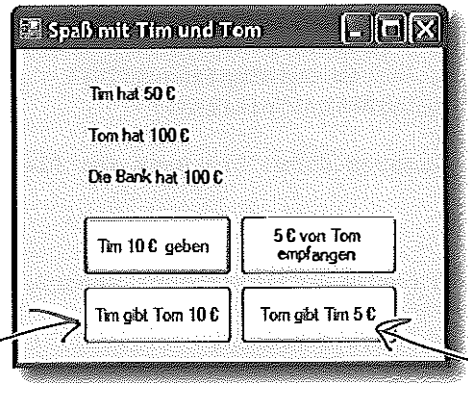
Fügen Sie dem Programm »Spaß mit Tim und Tom« Buttons hinzu, damit die Jungs sich gegenseitig Geld geben können.

1 **Nutzen Sie einen Objektinitialisierer, um Toms Typ-Instanz zu initialisieren.**
Für Tim haben Sie das schon gemacht. Sorgen Sie jetzt dafür, dass auch Toms Instanz einen Objektinitialisierer verwendet.

Sollten Sie bereits auf den Button geklickt haben, löschen Sie ihn einfach, fügen ihn dem Formular erneut hinzu und nennen ihn dann um. Löschen Sie danach die alte button3_Click() -Methode, die die IDE zuvor hinzugefügt hat, und verwenden Sie die neue Methode, die sie jetzt hinzufügt.

2 **Fügen Sie dem Formular zwei Buttons hinzu.**
Der erste Button sagt Tim, Tom 10 € zu geben, und der zweite sagt Tom, Tim 5 € zu geben. Gehen Sie, **bevor Sie auf den Button doppelklicken,** zum Eigenschaftenfenster und ändern Sie dort über die »(Name)«-Zeile die Namen beider Buttons – sie steht **ganz oben** in der Liste der Eigenschaften. Geben Sie dem ersten Button den Namen **`timGibtTom`** und dem zweiten den Namen **`tomGibtTim`**.

Dieser Button weist Tim an, Tom 10 € zu geben, deswegen sollten Sie ihm über die »(Name)«-Zeile im Eigenschaftenfenster den Namen timGibtTom geben.



Dieser Button weist Tom an, Tim 5 € zu geben. Geben Sie ihm den Namen tomGibtTim.

3 **Machen Sie die beiden Buttons funktionsfähig.**
Klicken Sie im Designer doppelt auf den Button `timGibtTom`. Die IDE fügt dem Formular dann die Methode `timGibtTom_Click()` hinzu, die ausgeführt wird, wenn auf den Button geklickt wird. Geben Sie in dieser Methode Code an, der Tim 10 € an Tom schicken lässt. Klicken Sie dann doppelt auf den anderen Button und schreiben Sie in der von der IDE erzeugten Methode `tomGibtTim_Click()` Code, der Tom 5 € Tim geben lässt. Achten Sie darauf, dass das Formular aktualisiert wird, nachdem das Geld die Hände gewechselt hat.



LÖSUNGEN ZU DEN ÜBUNGEN

Fügen Sie dem Programm »Spaß mit Tim und Tom« Buttons hinzu, damit die Jungs sich gegenseitig Geld geben können.

```
public partial class Form1 : Form {
    Typ tim;
    Typ tom;
    int bank = 100;

    public Form1() {
        InitializeComponent();
        tom = new Typ() { Geld = 100, Name = "Tom" };
        tim = new Typ() { Geld = 50, Name = "Tim" };
        FormularAktualisieren();
    }

    public void FormularAktualisieren() {
        timsGeld.Text = tim.Name + " hat " + tim.Geld + " €";
        tomsGeld.Text = tom.Name + " hat " + tom.Geld + " €";
        bankGeld.Text = "Die Bank hat " + bank + " €";
    }

    private void button1_Click(object sender, EventArgs e) {
        if (bank >= 10) {
            bank -= tim.GeldEmpfangen(10);
            FormularAktualisieren();
        } else {
            MessageBox.Show("Die Bank ist Pleite.");
        }
    }

    private void button2_Click(object sender, EventArgs e) {
        bank += tom.GeldVerleihen(5);
        FormularAktualisieren();
    }

    private void timGibtTom_Click(object sender, EventArgs e) {
        tom.GeldEmpfangen(tim.GeldVerleihen(10));
        FormularAktualisieren();
    }

    private void tomGibtTim_Click(object sender, EventArgs e) {
        tim.GeldEmpfangen(tom.GeldVerleihen(5));
        FormularAktualisieren();
    }

}
```

Hier sind die Objektinitialisierer für die beiden Instanzen der Klasse Typ. Tom wird mit 100 € und seinem Namen initialisiert.

Damit Tim Tom Geld gibt, rufen wir Tims GeldVerleihen()-Methode auf und senden ihr Ergebnis in Toms GeldEmpfangen()-Methode.

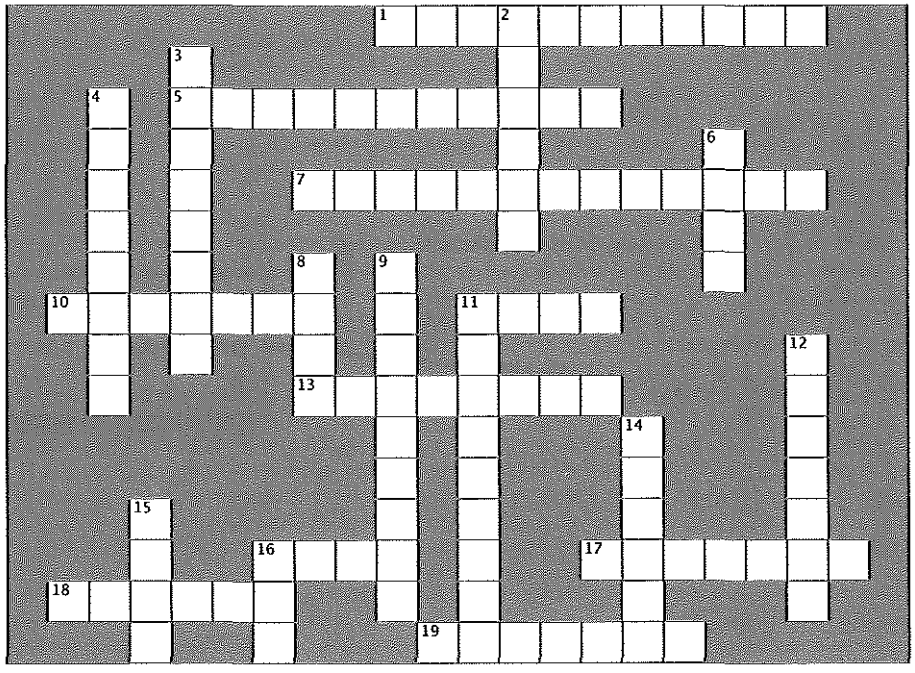
Sehen Sie sich genau an, wie die Methoden von Typ aufgerufen werden. Die von GeldVerleihen() gelieferten Ergebnisse werden als Parameter direkt in GeldEmpfangen() gepumpt.

Der Trick ist hier, dass man durchdenkt, wer Geld gibt und wer es empfängt.



Objekt-Kreuzworträtsel

Es wird Zeit, dass wir Ihrer linken Gehirnhälfte eine Pause geben und die rechte zum Arbeiten bringen: Alle Lösungswörter beziehen sich auf Objekte und stammen aus diesem Kapitel.



Waagerecht

1. Diese optionale Methode wird jedes Mal ausgeführt, wenn ein Objekt erzeugt wird.
5. Eine besondere Art von Feld, das von den Steuerelementen des Formulars verwendet wird.
7. Über dieses Formularsteuerelement kann der Benutzer eine Zahl in einem von Ihnen festgelegten Bereich auswählen.
10. Instanzen von Klassen, angezeigte Formulare und aus der Toolbox gezogene Steuerelemente sind alles _____.
11. Ist der Rückgabetyp einer Methode _____, liefert sie gar nichts.
13. Bevor man mit dem Programmieren anfängt, sollte man auf Papier ein Klassen_____ entwerfen.
16. Ein guter Methoden_____ macht deutlich, was eine Methode macht.
17. Die Felder eines Objekts definieren seinen _____.
18. Das ist die Ausgangsbasis, auf der Sie ein Objekt erstellen.
19. Ein Objekt ist eine _____ einer Klasse.

Senkrecht

2. Dieses Schlüsselwort sollten Sie in Ihrer Klassendeklaration vermeiden, wenn Sie Instanzen der Klasse erstellen können möchten.
3. Sie definieren, was eine Klasse macht.
4. Mit diesem Formularsteuerelement können Sie andere Steuerelemente gruppieren.
6. Der Typ des Objekts, das an die Application.Run()-Methode des Einstiegspunkts einer Windows-Anwendung übergeben wird.
8. Das nutzt ein Objekt, um nachzuhalten, was es weiß.
9. Das, was Sie verwenden, um Methoden Informationen zu übermitteln.
11. Die Methoden eines Objekts definieren sein _____.
12. Was Sie tun, wenn Sie (int) einsetzen, um einen Wert mit Nachkommastellen in eine Ganzzahl umzuwandeln.
14. Diese Anweisung sagt einer Methode, dass sie die Ausführung sofort beenden soll, und gibt einen Wert an, der an die Anweisung zurückgeliefert werden soll, die diese Methode aufrief.
15. Wo die Objekte leben.
16. Diese Anweisung nutzen Sie, um ein Objekt zu erstellen.

Pool-Puzzle, Lösung von Seite 109



Sie hatten die ***Aufgabe***, die Lücken im Code mit den Codeschnipseln aus dem Pool zu füllen. Ihr ***Ziel*** war es, eine Klasse zu erstellen, die sich kompilieren und ausführen lässt.

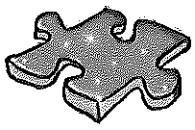
```
public partial class Form1 : Form
{
  private void button1_Click(object sender, EventArgs e)
  {
    String ergebnis = "";
    Echo e1 = new Echo();
    Echo e2 = new Echo();
    int x = 0;
    while ( x < 4 ) {
      ergebnis = ergebnis + e1.hallo() + "\n";
      e1.zähler = e1.zähler + 1;
      if ( x == 3 ) {
        e2.zähler = e2.zähler + 1;
      }
      if ( x > 0 ) {
        e2.zähler = e2.zähler + e1.zähler;
      }
      x = x + 1;
    }
    MessageBox.Show(ergebnis + "Zähler: " + e2.zähler);
  }

  public class Echo {
    public int zähler = 0;
    public string hallo() {
      return "Halloooo...";
    }
  }
}
```

Das ist die richtige Antwort.

Die Bonusantwort ist:

```
Echo e2 = e1;
```



Objekt-Kreuzworträtsel, Lösung



4 Typen und Referenzen

Es ist 10:00. Wissen Sie, wo Ihre Daten sind?



Datentyp, Datenbank, Lieutenant Commander Data ... all das ist wichtiges Zeug. Ohne Daten sind Ihre Programme nutzlos. Sie brauchen **Informationen** von Ihren Anwendern und nutzen diese, um andere Informationen nachzuschlagen oder neue Informationen zu produzieren, die Sie dann wieder an Ihre User zurückliefern. Eigentlich schließt fast alles, was Sie beim Programmieren tun, auf die eine oder andere Weise **Arbeit mit Daten** ein. In diesem Kapitel werden Sie alles über die **Datentypen** von C# lernen, erfahren, wie Sie in Ihrem Programm mit Daten arbeiten, und Sie werden sogar auf einige schmutzige Geheimnisse über **Objekte** stoßen (*psst ... auch Objekte sind Daten*).

Der Typ einer Variablen bestimmt, welche Art Daten sie speichern kann

In C# sind 15 **Typen** eingebaut, die jeweils Daten unterschiedlicher Art speichern. Einige der am weitesten verbreiteten haben Sie bereits gesehen, und Sie wissen, wie man diese verwendet. Aber es gibt noch ein paar, die Ihnen bisher nicht begegnet sind, die aber ebenfalls ziemlich praktisch sein können.

Typen, die Sie permanent verwenden

Es sollte Sie nicht überraschen, dass `int`, `string`, `bool` und `float` die gebräuchlichsten Typen sind.

Eine ganze Zahl hat keinen Nachkommateil.

- `int` kann **ganze** Zahlen von –2.147.483.648 bis 2.147.483.647 speichern.
- `string` kann Text beliebiger Länge (einschließlich des leeren Strings `""`) speichern.
- `bool` ist ein Boolescher Wert – er ist entweder `true` oder `false`.
- `float` kann eine **dezimale** Zahl von $\pm 1{,}5 \times 10^{-45}$ bis $\pm 3{,}4 \times 10^{38}$ mit bis zu 7 signifikanten Stellen speichern. Dieser Bereich scheint seltsam und kompliziert, ist eigentlich aber ziemlich einfach. »Signifikante Stellen« bezieht sich auf die *Genauigkeit* der Zahl: 35.048.410.000.000, 1.743.059, 14,43857 und 0,00004374155 haben jeweils 7 signifikante Stellen. 10^{38} bedeutet, dass Sie Zahlen bis zu einer Größe von 10^{38} (oder 1 gefolgt von 38 Nullen) speichern können – solange sie 7 oder weniger signifikante Stellen haben. Auf der anderen Seite des Wertebereichs bedeutet 10^{-45}, dass Sie eine Zahl so klein wie 10^{-45} (oder einen Dezimaltrenner gefolgt von 45 Nullen gefolgt von 1) speichern können … aber nur, wie Sie sicher schon erraten haben, solange sie 7 oder weniger signifikante Stellen hat.

Das »float« sagt, dass es sich um eine Gleitkommazahl, keine Festkommazahl handelt, die eine festgelegte Anzahl von Nachkommastellen hat.

Weitere Typen für ganze Zahlen

Es gab einmal eine Zeit, in der Computerspeicher verdammt teuer und Computerprozessoren verdammt langsam waren. Und, glauben Sie es oder nicht, wenn man den falschen Datentyp verwendete, konnte das ein Programm ernstlich ausbremsen. Glücklicherweise haben die Zeiten sich geändert, und wenn man eine ganze Zahl speichern muss, kann man meist sorglos einen `int` verwenden. Manchmal aber brauchen Sie etwas Größeres … und gelegentlich auch mal etwas Kleineres. Deswegen gibt Ihnen C# weitere Optionen:

Diese Typen nutzen Sie oft, wenn Sie ein Problem lösen, bei dem das »Überschlagen«, das Ihnen in ein paar Minuten begegnen wird, wirklich nützlich ist.

- `byte` speichert **ganze** Zahlen zwischen 0 und 255.
- `sbyte` speichert **ganze** Zahlen von –128 bis 127.
- `short` speichert **ganze** Zahlen von –32.768 bis 32.767.
- `ushort` speichert **ganze** Zahlen von 0 bis 65.535.
- `uint` speichert **ganze** Zahlen von 0 bis 4.294.967.295.
- `long` kann jede ganze Zahl zwischen minus und plus 9 Milliarden Milliarden speichern.
- `ulong` kann jede ganze Zahl zwischen 0 und 18 Milliarden Milliarden speichern.

Das »u« steht für »unsigned« (vorzeichenlos) und heißt, dass die Werte nicht negativ sein können (deswegen gibt es kein Minuszeichen).

Typen zum Speichern *echt* riesiger und *echt* winziger Zahlen

Manchmal sind 7 signifikante Stellen einfach nicht genau genug. Und manchmal, glauben Sie es oder nicht, ist 10^{38} einfach nicht groß oder nicht 10^{-45} klein genug. Viele Programme, die für Finanzangelegenheiten oder wissenschaftliche Belange geschrieben werden, stoßen ständig auf diese Probleme, deswegen gibt Ihnen C# zwei weitere Typen:

Der Typ `double` ist eigentlich genauso verbreitet wie `float`. Viele nutzen ihn ständig und `float` nur selten.

- `double` speichert beliebige Zahlen von $\pm 5.0 \times 10^{-324}$ bis $\pm 1.7 \times 10^{308}$ mit 15 bis 16 signifikanten Stellen.
- `decimal` speichert beliebige Zahlen von $\pm 1.0 \times 10^{-28}$ bis $\pm 7.9 \times 10^{28}$ mit 28 bis 29 signifikanten Stellen.

Als Sie die Value-Eigenschaft des NumericUp-Down-Steuerelements verwendeten, haben Sie einen decimal eingesetzt.

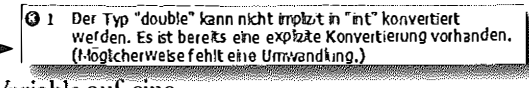
Auch Literale haben Typen

Literal bezeichnet einfach eine Zahl, die Sie in Ihren Code eingeben.

Geben Sie in ein C#-Programm direkt eine Zahl ein, nutzen Sie ein **Literal** ... und jedem Literal wird automatisch ein Typ zugewiesen. Das können Sie selbst sehen – geben Sie einfach folgende Zeile ein, um einer **`int`**-Variablen den Wert `14.7` zuzuweisen:

```
int meinInt = 14.7;
```

Versuchen Sie, das Programm zu erstellen, erhalten Sie dies:



Den gleichen Fehler erhalten Sie, wenn Sie versuchen, eine `int`-Variable auf eine `double`-Variable zu setzen. Die IDE sagt Ihnen damit, dass das Literal `14.7` einen Typ hat – und der ist `double`. Sie können den Typ in `float` ändern, indem Sie ans Ende ein `F` anhängen (`14.7F`). Und `14.7M` ist ein Dezimal.

Versuchen Sie, einer float- oder einer decimal-Variablen einfach so ein Literal zuzuweisen, liefert Ihnen die IDE eine hilfreiche Meldung, die Sie daran erinnert, das richtige Suffix hinzuzufügen. Cool!

Ein paar weitere nützliche eingebaute Typen

Gelegentlich muss man einzelne Zeichen wie `Q`, `7` oder `$` speichern. Dazu nutzt man den Typ `char`. Literale Werte für `char` stehen immer in einfachen Anführungszeichen (`'x'`, `'3'`). In die Anführungszeichen können Sie auch **Escape-Sequenzen** einschließen (`'\n'` ist ein Zeilenumbruch, `'\t'` ein Tabulator). Sie schreiben Escape-Sequenzen in Ihrem C#-Code mit mehreren Zeichen, aber Ihr Programm speichert jede Escape-Sequenz im Speicher als ein einziges Zeichen.

In Kapitel 9 lernen Sie mehr zum Verhältnis von `char` und `byte`.

Und dann gibt es noch einen wichtigen Typ: **`object`**. Sie haben bereits ein paar Typen von Objekten gesehen. `object` ist der allgemeinste Typ. Später werden Sie lernen, das Typen in Hierarchien stehen – `object` bildet die Wurzel dieser Hierarchie. Das ist nützlich, denn es bedeuet, **dass Sie einer `object`-Variablen jeden Wert, jede Variable und jedes Objekt zuweisen können.**



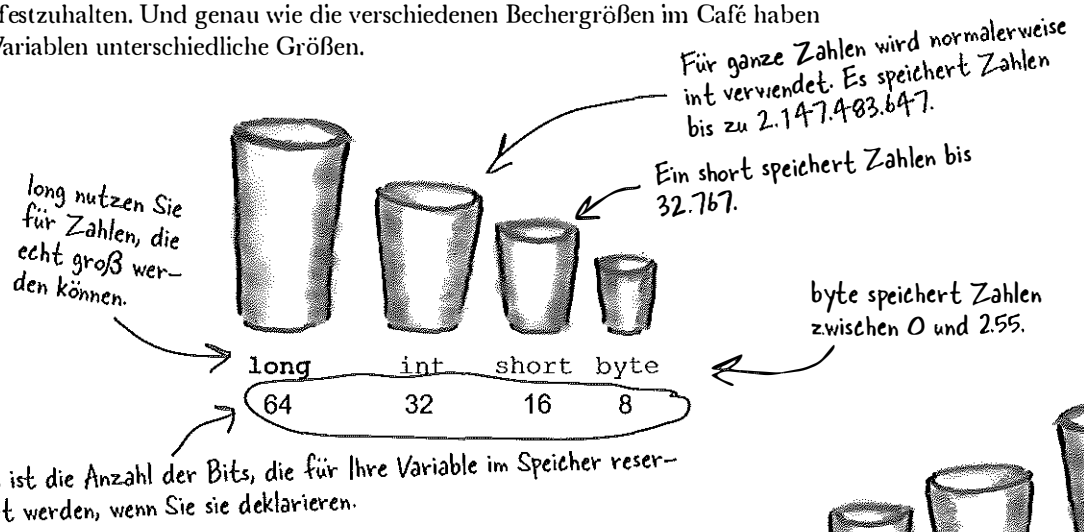
KOPF-NUSS

Mit dem Windows-Rechner können Sie zwischen dezimalen Zahlen (zur Basis 10) und binären Zahlen (Zahlen zur Basis 2, die nur mit 0 und 1 geschrieben werden) wechseln – gehen Sie in den Wissenschaftlich-Modus, geben Sie eine Zahl ein und klicken Sie auf **Bin**, um die Zahl in Binär zu konvertieren. Klicken Sie dann auf **Dec**, um die Konvertierung rückgängig zu machen. Geben Sie jetzt **einige der Ober- und Untergrenzen für die ganzzahligen Typen** (wie –32.768 und 255) ein und wandeln Sie sie in Binär um. Haben Sie jetzt eine Idee, ***warum*** C# gerade diese Grenzen wählt?

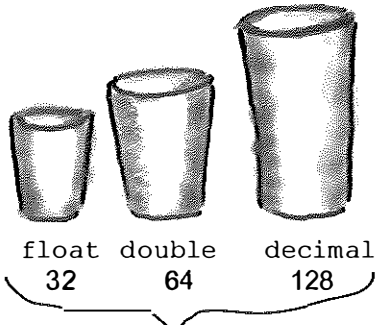
Eine Variable ist wie ein Daten-To-Go-Becher

Alle Datentypen brauchen Speicherplatz. (Erinnern Sie sich an den Heap aus dem letzten Kapitel?) Ihre Aufgabe schließt also ein, dass Sie sich darüber Gedanken machen, wie *viel* Speicherplatz Sie benötigen, wenn Sie in einem Programm einen String oder eine Zahl verwenden. Das ist einer der Gründe dafür, dass Sie Variablen verwenden. Über sie reservieren Sie so viel Platz im Speicher, wie Sie für Ihre Daten benötigen.

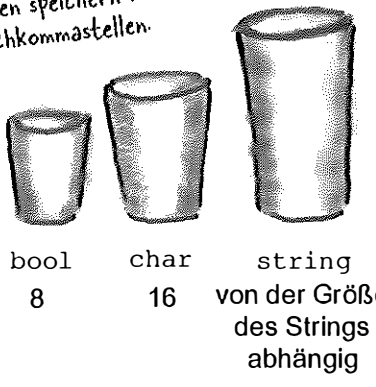
Stellen Sie sich eine Variable wie einen Becher vor, in dem Sie Ihre Daten lagern. C# nutzt eine Menge unterschiedlicher Arten von Bechern, um unterschiedliche Arten von Daten festzuhalten. Und genau wie die verschiedenen Bechergrößen im Café haben auch Variablen unterschiedliche Größen.



Zahlen mit Nachkommastellen werden anders gespeichert als ganze Zahlen. Die meisten Ihrer Zahlen mit Nachkommastellen können Sie mit `float` speichern, dem kleinsten Datentyp zur Speicherung von Dezimalzahlen. Wenn Sie eine höhere Genauigkeit brauchen, verwenden Sie `double`, und für das Schreiben einer wissenschaftlichen Anwendung, bei der die Zahlen sehr genau sein müssen, bietet der Typ `decimal` die größte Genauigkeit.



Es geht aber nicht immer um Zahlen. (Sie würden ja auch nicht erwarten, einen heißen Kaffee in einem Plastikbecher zu erhalten und einen kalten in einem Papierbecher.) Der C#-Compiler kann auch mit Zeichen und anderen nicht numerischen Typen umgehen. Der Typ `char` speichert ein Zeichen, und `string` wird für viele aneinandergehängte Zeichen verwendet. Für eine `string`-Variable muss keine Größe vorgegeben werden. Sie wird so erweitert, dass Sie soviele Daten aufnehmen kann, wie Sie darin speichern müssen. Der Typ `bool` wird zur Speicherung von Wahr/Falsch-Werten verwendet, wie Sie sie in Ihren `if`-Anweisungen verwendet haben.



10 Kilo Daten in einem 5-Kilo-Sack

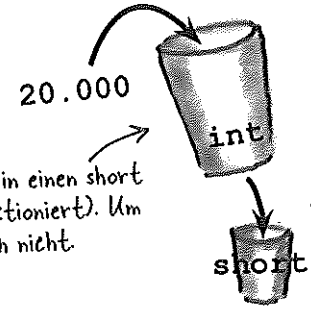


Deklarieren Sie eine Variable eines bestimmten Typs, wird sie vom Compiler auch so betrachtet. Selbst wenn ein Wert nicht einmal in die Nähe der Obergrenze des Typs kommt, den Sie deklariert haben, blickt der Compiler nur auf den Becher, in dem er sich befindet, nicht auf den Wert darin. Folgendes funktioniert also nicht:

```
int meilenUnterDemMeer = 20000;

short wenigerMeilen = meilenUnterDemMeer;
```

20.000 würde in einen `short` passen. Kein Problem. Aber da **`meilenUnterDemMeer`** als `int` deklariert ist, berücksichtigt der Compiler nur die maximale Größe eines `int` und sieht, dass diese zu groß für einen `short`-Container ist. Diese Übersetzung führt der Compiler nicht im Vorübergehen für Sie aus. Sie müssen sicherstellen, dass Sie den richtigen Typ für die Daten verwenden, mit denen Sie arbeiten.



Der Compiler sieht nur, dass ein int in einen short gesteckt werden soll (was nicht funktioniert). Um den Wert im Becher kümmert er sich nicht.

Das ist vernünftig. Was wäre, wenn Sie später einen größeren Wert in den int-Becher steckten, der nicht mehr in den short-Becher passt? Der Compiler versucht nur, Sie zu schützen.

Spitzen Sie Ihren Bleistift

Drei dieser Anweisungen lassen sich nicht kompilieren, weil sie versuchen, zu große Daten oder Daten des falschen Typs in eine Variable zu stecken. Kreisen Sie sie ein.

```
int stunden = 24;

short y = 78000;

bool fertig = ja;

short RPM = 33;

int kontostand = 345667 - 567;

string drohung = "Deine Mutter";

byte tage = 365;

long radius = 3;

char initial = 'S';

string monate = "12";
```

Auch eine Zahl der richtigen Größe können Sie nicht einfach irgendeiner Variablen zuweisen

Schauen wir, was passiert, wenn Sie versuchen, einer `int`-Variablen einen `decimal`-Wert zuzuweisen.

Tun Sie das!

1. Erzeugen Sie ein neues Projekt und fügen Sie ihm einen Button hinzu. Fügen Sie dann der **`Click()`**-Methode des Buttons folgende Zeilen hinzu:

```
decimal meinDezimalWert = 10;
int meinIntWert = meinDezimalWert;

MessageBox.Show("meinIntWert ist " + meinIntWert);
```

2. Versuchen Sie, das Programm zu erstellen. Oh, Sie erhalten diesen Fehler:



Sehen Sie sich an, wie die IDE herausfand, dass Ihnen wahrscheinlich eine Umwandlung fehlt.

3. Lassen Sie den Fehler verschwinden, indem Sie einen **Cast** (auch Umwandlung genannt) einfügen, der den `decimal` zu einem `int` macht. Nachdem Sie die zweite Zeile in der nachfolgenden Weise geändert haben, lässt sich des Programm kompilieren und ausführen:

```
int einIntValue = (int) myDecimalValue;
```

Dies ist die Umwandlung von einem `decimal`-Wert in einen `int`.

Was ist passiert?

Der Compiler lässt Sie einer Variablen keine Werte des falschen Typs zuweisen – selbst wenn die Variable den Wert aufnehmen könnte–, weil das der Grund für eine Unzahl von Fehlern ist. Wenn Sie umwandeln, ist das, als würden Sie dem Compiler ein Versprechen geben: »Ich weiß, die Typen sind unterschiedlich, aber in diesem Fall wird trotzdem nichts schiefgehen!« C# versucht dann, die Daten in die neue Variable zu packen.

Blättern Sie kurz zum Anfang des letzten Kapitels zurück und schauen Sie sich an, wie diese Umwandlungstechnik eingesetzt wurde, als das NumericUpDown.Value an das Formular Redner-Tester übergeben wurde.

Spitzen Sie Ihren Bleistift — Lösung

Drei dieser Anweisungen lassen sich nicht kompilieren, weil sie versuchen, zu große Daten oder Daten des falschen Typs in eine Variable zu stecken. Kreisen Sie sie ein.

```
short y = 78000;
```

Der Typ short speichert Zahlen von -32.767 bis 32.768. Die Zahl ist zu groß!

```
byte tage = 365;
```

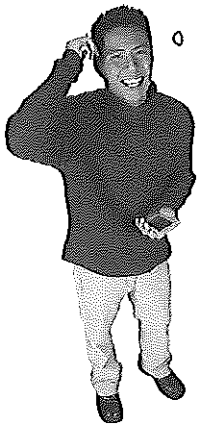
In einen byte passen nur Werte bis 256. Für diese Zahl bräuchten Sie einen short.

```
bool fertig = ja;
```

Einem bool können Sie nur die Werte »true« oder »false« zuweisen.

Wandeln Sie einen zu großen Wert um, passt C# ihn automatisch an

Gerade haben Sie gesehen, dass ein `decimal` in einen `int` umgewandelt werden kann. Es ist sogar so, dass *jede* Zahl in *jede andere* Zahl umgewandelt werden kann. Aber das heißt nicht, dass der ***Wert*** bei der Umwandlung intakt bleibt. Wandeln Sie eine `int`-Variable, die auf 365 gesetzt ist, in einen `byte` um, ist 365 zu groß für den **`byte`**. Aber anstelle einer Fehlermeldung wird der Wert einfach **überschlagen**: Beispielsweise hat 256, in einen `byte` umgewandelt, den Wert 0, 257 den Wert 1, 258 den Wert 2 usw. bis 365, das zu **109** wird. Und sind Sie wieder bei 255 angekommen, »überschlägt« der umgewandelte Wert wieder zu null.



In meinen Dialogfenstern habe ich ständig Zahlen und Strings kombiniert, seit ich in Kapitel 2 Schleifen kennengelernt habe! Habe ich mich da die ganze Zeit schon auf Umwandlungen verlassen?

Ja! Der +-Operator wandelt um.

Dort haben Sie den +-Operator verwendet, der **automatisch eine Reihe von Umwandlungen für Sie durchführt** – aber er ist darin ziemlich gewandt. Wenn Sie mit + eine Zahl oder einen Booleschen Wert zu einem String hinzuaddieren, wandelt er den Wert automatisch in einen String um. Verwenden Sie + (oder *, / oder –) mit unterschiedlichen Typen, **wird der kleinere Typ automatisch in den größeren umgewandelt**. Hier ist ein Beispiel:

```
int meinInt = 36;
float meinFloat = 16.4F;
meinFloat = meinInt + meinFloat;
```

Weisen Sie einem float einen Zahlwert zu, müssen Sie ans Ende der Zahl F anhängen, um dem Compiler zu sagen, dass es ein float und kein double ist.

Da ein `int` in einen `float` passt, ein `float` aber nicht in einen `int`, wandelt der +-Operator `meinInt` in einen **`float`** um, bevor `meinFloat` hinzuaddiert wird.

Wandeln Sie selbst um!

Wie die Umwandlung Zahlen »überschlägt«, ist kein Geheimnis – das können Sie selbst. Starten Sie den Windows-Rechner, wechseln Sie in den Wissenschaftlich-Modus und berechnen Sie 365 Mod 256 (über den »Mod«-Button, der eine Modulo-Berechnung durchführt). Sie erhalten 109.

Spitzen Sie Ihren Bleistift

Typen können nicht immer in andere Typen umgewandelt werden.

Erzeugen Sie ein neues Projekt mit einem Button und stecken Sie in seine Methode diese Anweisungen. Beim Erstellen des Programms erhalten Sie Massen an Fehlern. Streichen Sie die Anweisungen durch, die zu Fehlern führen. So sehen Sie, welche Typen umgewandelt werden können und welche nicht!

```
int einInt = 10;
byte einByte = (byte)einInt;
double einDouble = (double)einByte;
bool einBool = (bool)einDouble;
string einString = "false";
einBool = (bool)einString;
einString = (string)einInt;
einString = einInt.ToString();
einBool = (bool)einByte;
einByte = (byte)einBool;
short einShort = (short)einInt;
char einChar = 'x';
einString = (string)einChar;
long einLong = (long)einInt;
decimal einDecimal = (decimal)einLong;
einString = einString + einInt +
einByte + einDouble + einChar;
```

Einige Umwandlungen macht C# automatisch

Es gibt zwei wichtige Konvertierungen, die keine explizite Umwandung verlangen. Die erste wird immer automatisch durchgeführt, wenn Sie arithmetische Operatoren verwenden, wie in diesem Beispiel:

```
long l = 139401930;
short s = 516;
double d = l - s;
d = d / 123.456;
MessageBox.Show("Die Antwort ist " + d);
```

Der --Operator zieht den short von dem long ab und der =-Operator wandelt das Ergebnis in einen double um.

Der +-Operator kann den decimal in einen string umwandeln.

Außerdem konvertiert C# Typen automatisch für Sie, wenn Sie den +-Operator einsetzen, um Strings **zu verketten** (was einfach nur heißt, einen String an das Ende eines anderen anzuhängen, wie Sie es bei den Dialogfenstern gemacht haben). Wenn Sie mit dem +-Operator einen String mit etwas verketten, das einen anderen Typ hat, wandelt er Zahlen automatisch in Strings um. Hier ist ein Beispiel. Die ersten beiden Zeilen sind in Ordnung, aber die dritte lässt sich nicht kompilieren:

```
long x = 139401930;
MessageBox.Show("Die Antwort ist " + x);
MessageBox.Show(x);
```

Der C#-Compiler spuckt einen Fehler aus, der irgendetwas von einem ungültigen Argument sagt (Argument ist ein anderer Name für einen Methodenparameter). Das liegt daran, dass der Parameter für **`MessageBox.Show()`** ein `string` ist und dieser Code einen `long`, also den falschen Typ für diese Methode, übergibt. Aber Sie können ihn sehr leicht in einen String umwandeln, indem Sie seine **`ToString()`**-Methode aufrufen. Diese Methode ist ein Member jedes Werttyps und Objekts. (Alle Klassen, die Sie selbst erstellen, haben eine `ToString()`-Methode, die den Namen der Klasse liefert.) Auf diese Weise können Sie `x` in etwas konvertieren, das **`MessageBox.Show()`** verwenden kann:

```
MessageBox.Show(x.ToString());
```



Spitzen Sie Ihren Bleistift
Lösung

Typen können nicht immer in andere Typen umgewandelt werden. Erzeugen Sie ein neues Projekt mit einem Button und stecken Sie in seine Methode diese Anweisungen. Beim Erstellen des Programms erhalten Sie Massen an Fehlern. Streichen Sie die Anweisungen durch, die zu Fehlern führen. So sehen Sie, welche Typen umgewandelt werden können und welche nicht!

```
int einInt = 10;
byte einByte = (byte)einInt;
double einDouble = (double)einByte;
bool einBool = (bool)einDouble;
string einString = "false";
einBool = (bool)einString;
einString = (string)einInt;
einString = einInt.ToString();
einBool = (bool)einByte;
einByte = (byte)einBool;
short einShort = (short)einInt;
char einChar = 'x';
einString = (string)einChar;
long einLong = (long)einInt;
decimal einDecimal = (decimal)
einLong;
einString = einString + einInt +
einByte + einDouble + einChar;
```



In C# gibt es ungefähr 77 **reservierte Wörter**. Diese Wörter werden durch den C#-Compiler reserviert und dürfen nicht als Variablennamen verwendet werden. Wenn Sie mit dem Buch durch sind, werden Sie sie alle gut kennen. Hier sind einige, die Sie schon verwendet haben. Schreiben Sie, was Ihrer Meinung nach diese Wörter in C# tun.

`namespace`

`for`

`class`

`public`

`else`

`new`

`using`

`if`

`while`

→ Antworten auf Seite 161.

Beim Aufruf einer Methode müssen die Variablen den Typen der Parameter entsprechen

Versuchen Sie, **`MessageBox.Show(123)`** aufzurufen, übergeben Sie MessageBox.Show() ein Literal (123) statt eines Strings. Die IDE lässt Sie Ihr Programm nicht erstellen. Stattdessen meldet sie Ihnen einen Fehler: »1-Argument: kann nicht von ›int‹ in ›string‹ konvertiert werden.«

Aber MessageBox.Show() ist nicht die einzige Methode, die Ihnen Compiler-Fehler meldet, wenn Sie versuchen, ihr eine Variable zu übergeben, deren Typ dem Parameter nicht entspricht. Das tun *alle* Methoden, sogar die, die Sie selbst schreiben. Das können Sie testen, indem Sie diese absolut zulässige Methode in eine Klasse eingeben:

```
public int MeineMethode(bool jaNein) {
    if (jaNein) {
        return 45;
    } else {
        return 61;
    }
}
```

Zur Erinnerung – der Code, der diese Methode aufruft, muss den Parameter nicht als Variable namens jaNein angeben. Er muss einfach nur einen Booleschen Wert oder eine beliebige Boolesche Variable angeben. jaNein heißt er lediglich innerhalb der Methode.

Liefert Ihnen der Compiler einen »Ungültige Argumente«-Fehler, haben Sie versucht, eine Methode mit Werten aufzurufen, deren Typen den Methodenparametern nicht entsprechen.

Sie funktioniert wunderbar; solange Sie ihr übergeben, was sie erwartet (einen `bool`) – rufen Sie `MeineMethode(true)` oder `MeineMethode(false)` auf, kompiliert das Programm problemlos.

Aber was ist, wenn Sie stattdessen einen int oder einen string übergeben? Die IDE liefert Ihnen einen ähnlichen Fehler wie den, den Sie erhalten haben, als Sie `MessageBox.Show()` 123 übergeben hatten. Übergeben Sie jetzt einen Booleschen Wert, aber versuchen Sie, den Rückgabewert einem short zuzuweisen oder an **`MessageBox.Show()`** zu übergeben. Auch das geht nicht – die Methode liefert einen `int`, keinen **`short`** und auch nicht den **`string`**, den **`MessageBox.Show()`** erwartet.

Einer Variablen, einem Parameter oder einem Feld mit dem Typ object können Sie alles zuweisen.

if-Anweisungen prüfen immer auf Wahrheit

Ist Ihnen aufgefallen, dass wir die if-Anweisung folgendermaßen geschrieben haben:

```
if (jaNein) {
```

Wir mussten nicht explizit »if (jaNein == true)« sagen. Grund dafür ist, dass eine if-Anweisung immer prüft, ob etwas wahr ist. Wollen Sie prüfen, ob etwas falsch ist, nutzen Sie das ! (ein Ausrufezeichen bzw. den NICHT-Operator). »if (!jaNein)« ist das Gleiche wie »if (jaNein == false)«. In unseren Codebeispielen werden wir von jetzt ab in der Regel »if (jaNein)« oder »if (!jaNein)« verwenden und nicht explizit prüfen, ob ein Boolescher Wert true oder false ist.



Lösungen zu den Übungen

Hier ist der Code für den ersten Teil der Übung.

```
public partial class Form1 : Form
{
    int tachoAnfangsstand;
    int tachoEndstand;
    double gefahreneKM;
    double erstattungsSatz = 0.39;
    double erstattungsBetrag;
    public Form1() {
        InitializeComponent();
    }
    private void button1_Click(object sender, EventArgs e){
      tachoAnfangsstand = (int)numericUpDown1.Value;
      tachoEndstand = (int)numericUpDown2.Value;
        if (tachoAnfangsstand <= tachoEndstand){
            gefahreneKM = tachoEndstand -= tachoAnfangsstand;
            erstattungsBetrag = gefahreneKM *= erstattungsSatz;
            label4.Text = erstattungsBetrag + "€";
        } else {
           MessageBox.Show(
              "Der Anfangsstand muss kleiner als der Endstand sein.",
              "Berechnung nicht möglich");
        }
    }
}
```

`int` funktioniert wunderbar für ganze Zahlen. Diese Zahl könnte bis 999.999 gehen. Ein `short` oder ein `byte` würde also nicht reichen.

Ja, der Punkt ist richtig. Das und nicht das Komma ist der Dezimaltrenner, den C# verwendet – genau wie jede andere Programmiersprache. Darauf müssen Sie achten, wenn Sie in Ihrem Code Gleitkommazahlen literal angeben.

Haben Sie daran gedacht, den `decimal`-Wert des NumericUpDown-Steuerelements in einen `int` umzuwandeln?

Dieser Block soll berechnen, wie viele Kilometer zurückgelegt wurden, und diese dann mit dem Erstattungssatz multiplizieren.

Wir haben hier eine alternative Form der Methode MessageBox.Show() verwendet. Wir haben ihr zwei Parameter gegeben, der erste ist die anzuzeigende Nachricht und der zweite der Text der Titelleiste.

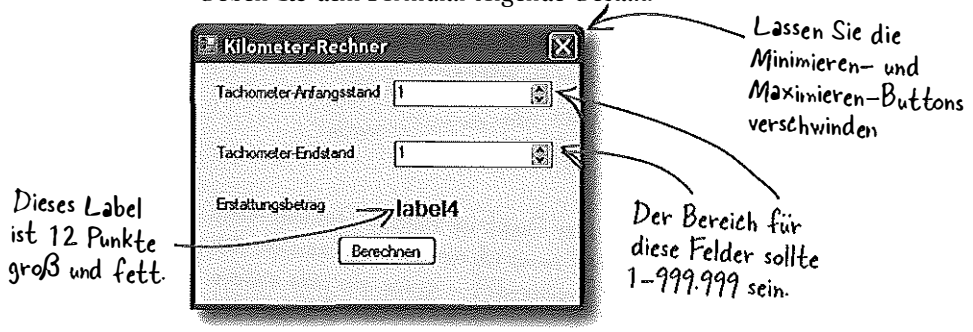
Dieser Button scheint zu funktionieren, birgt eigentlich aber ein ziemlich großes Problem. Sehen Sie es?



Erzeugen Sie einen Reisekostenerstattungsrechner für Geschäftsreisen. Der Benutzer sollte den Anfangs- und Endstand des Tachometers eingeben können. Auf Basis dieser beiden Zahlen wird der Rechner die angefallenen Kilometer berechnen und daraus ermitteln, wie hoch der Erstattungsbetrag ist, wenn für jeden angefallenen Kilometer 0,39 € gezahlt wird.

1 Beginnen Sie mit einem neuen Windows-Projekt.

Geben Sie dem Formular folgende Gestalt:



Klicken Sie doppelt auf den Button, wenn Sie das Formular fertig haben, um dem Projekt etwas Code hinzuzufügen.

2 Erzeugen Sie die Variablen, die Sie für den Rechner benötigen.

Stecken Sie die Variablen in die Klassendefinition am Anfang von `Form1`. Sie brauchen zwei Variablen für ganze Zahlen, die den Tachometer-Anfangsstand und -Endstand speichern. Nennen Sie diese `tachoAnfangsstand` and `tachoEndstand`. Außerdem brauchen Sie drei Zahlen, die Nachkommastellen aufnehmen können. Nehmen Sie jeweils den Typ `double` und nennen Sie diese `gefahreneKM`, `erstattungsSatz` und `erstattungsBetrag`. Setzen Sie den Wert für `erstattungsSatz` auf `0.39`.

3 Bringen Sie den Rechner zum Funktionieren.

Fügen Sie der Methode `button1_Click()` Code zu folgenden Zwecken hinzu:

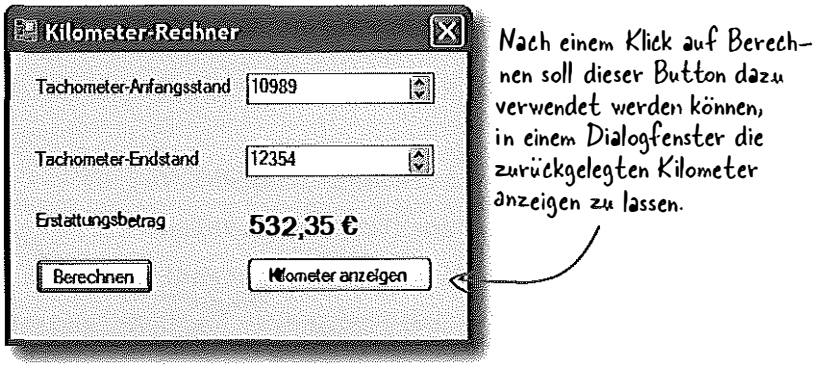
- ★ Stellen Sie sicher, dass die Zahl im Feld Tachometer-Anfangsstand kleiner als die Zahl im Feld Tachometer-Endstand ist. Lassen Sie andernfalls ein Dialogfenster mit der Meldung »Der Anfangsstand muss kleiner als der Endstand sein.« anzeigen. Geben Sie diesem Dialog den Titel »Keine Berechnung möglich«.
- ★ Ziehen Sie mit folgenden Zeilen den Anfangsstand vom Endstand ab und multiplizieren Sie ihn dann mit dem Erstattungssatz:

```
gefahreneKM = tachoEndstand -= tachoAnfangsstand;
erstattungsBetrag = gefahreneKM *= erstattungsSatz;
label4.Text = erstattungsBetrag + "€";
```

4 Führen Sie die Anwendung aus.

Prüfen Sie, ob der Rechner korrekte Werte liefert. Probieren Sie, den Anfangsstand größer als den Endstand zu machen, und schauen Sie, ob Ihr Dialogfenster angezeigt wird.

1 Fügen Sie dem Formular jetzt einen weiteren Button hinzu.
Richten Sie ihn so ein, dass im Formular die Anzahl der zurückgelegten Kilometer angezeigt wird, nachdem der Erstattungsbetrag berechnet wurde.



Klicken Sie, wenn Sie das Formular fertig haben, doppelt auf den Button Kilometer anzeigen, um dem Projekt etwas Code hinzuzufügen.

2 Eine Zeile Code sollte ausreichen.
Wir müssen das Formular im Prinzip nur dazu bringen, die Variable `gefahreneKM` anzuzeigen, oder? Das sollte sich eigentlich mit dieser Zeile erledigen lassen:

```
private void button2_Click(object sender, EventArgs e) {
        Messagebox.Show(gefahreneKM + " KM",
                        "Zurückgelegte Kilometer");
}
```

3 Führen Sie das Programm aus.
Geben Sie ein paar Werte ein und schauen Sie, was passiert.

4 Hmm, irgendetwas ist nicht in Ordnung ...
Die Kilometeranzahl entspricht immer dem Erstattungsbetrag. Warum?

Einen Operator mit = kombinieren

Sehen Sie sich einmal genau den Operator an, den wir verwendet haben, um den Tachometer-Anfangsstand vom Tachometer-Endstand abzuziehen (-=). Das Problem ist, dass er nicht nur subtrahiert, sondern der Variablen links des Minuszeichens auch einen Wert zuweist. Das Gleiche passiert in der Zeile, in der wir die Anzahl der zurückgelegten Kilometer mit der Erstattungsrate multiplizieren. Wir müssen -= und *= durch nur - bzw. * ersetzen.

```
private void button1_Click(object sender, EventArgs e)
{
  tachoAnfangsstand = (int)numericUpDown1.Value;
  tachoEndstand = (int)numericUpDown2.Value;
    if (tachoAnfangsstand <= tachoEndstand) {
        gefahreneKM = tachoEndstand -= tachoAnfangsstand;
        erstattungsBetrag = gefahreneKM *= erstattungsSatz;
        label4.Text = erstattungsBetrag + " €";
    } else {
       MessageBox.Show(
          "Der Anfangsstand muss kleiner als der Endstand sein.",
          "Berechnung nicht möglich");
    }
}
```

Das nennt man einen zusammengesetzten Operator. Er zieht tachoAnfangsstand von tachoEndstand ab, weist den neuen Wert aber auch gleichzeitig tachoEndstand und gefahreneKM zu.

Das ist besser – jetzt verändert der Code tachoEndstand und gefahreneKM nicht mehr.

```
gefahreneKM = tachoEndstand - tachoAnfangsstand;
erstattungsBetrag = gefahreneKM * erstattungsSatz;
```

Können Ihnen gute Variablen hier helfen? Unbedingt! Sehen Sie sich genau an, was die einzelnen Variablen vermeintlich tun. Der Name `gefahreneKM` gibt Ihnen bereits eine Menge Hinweise – Sie wissen, dass es diese Variable ist, die das Formular falsch anzeigt, und haben eine gute Vorstellung davon, wie dieser Wert berechnet werden sollte. Das können Sie also nutzen, wenn Sie Ihren Code durchsuchen, um den Fehler aufzuspüren. Es wäre viel schwerer, das Problem zu finden, sähen die fehlerhaften Zeilen stattdessen so aus:

```
mT = eM -= sM;
aO = mT *= rR;
```

Variablen mit Namen wie diese sind im Grunde nutzlos, wenn es darum geht, Ihnen etwas über ihren Zweck mitzuteilen.

Auch für Objekte braucht man Variablen

Bislang haben wir Objekte von den anderen Typen getrennt betrachtet. Aber Objekte sind einfach nur eine andere Art von Daten, die Ihr Code genau so behandelt wie Zahlen, Strings und Boolesche Werte.

Einen int verwenden

1. Schreiben Sie eine Anweisung, die einen int deklariert.

```
int einInt;
```

2. Weisen Sie der Variablen einen Wert zu.

```
einInt = 3761;
```

3. Nutzen Sie den Wert in Ihrem Code.

```
while (i < einInt) {
```

Ein Objekt verwenden

1. Schreiben Sie eine Anweisung, die ein Objekt deklariert.

```
Hund rex;
```

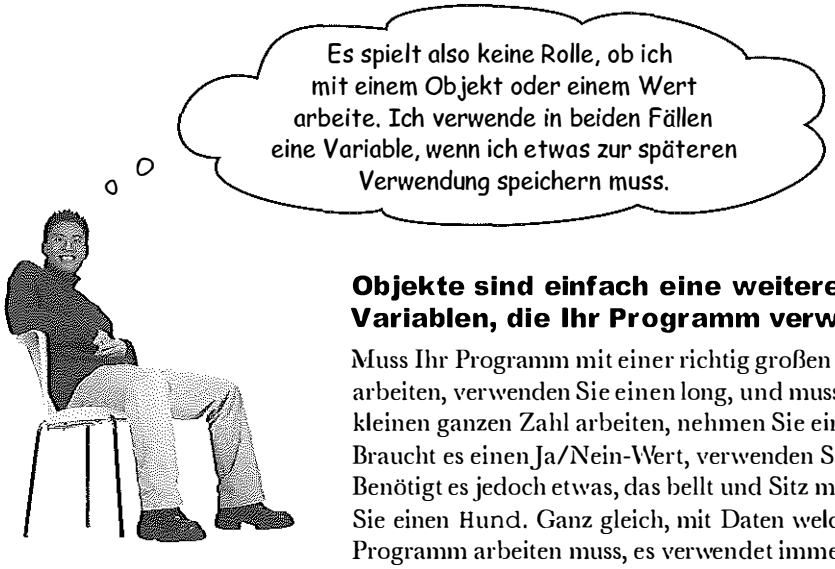
Haben Sie eine Klasse wie Hund, nutzen Sie diese in der Variablendeklaration als Typ.

2. Weisen Sie der Variablen ein Objekt zu.

```
rex = new Hund();
```

3. Prüfen Sie eins der Felder des Objekts.

```
while (rex.Zufrieden) {
```



Objekte sind einfach eine weitere Art von Variablen, die Ihr Programm verwenden kann.

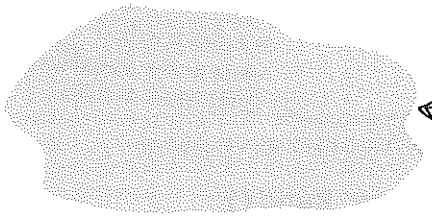
Muss Ihr Programm mit einer richtig großen ganzen Zahl arbeiten, verwenden Sie einen long, und muss es mit einer kleinen ganzen Zahl arbeiten, nehmen Sie einen short. Braucht es einen Ja/Nein-Wert, verwenden Sie einen boolean. Benötigt es jedoch etwas, das bellt und Sitz macht, verwenden Sie einen Hund. Ganz gleich, mit Daten welcher Art Ihr Programm arbeiten muss, es verwendet immer eine Variable.

Mit Referenzvariablen auf Objekte verweisen

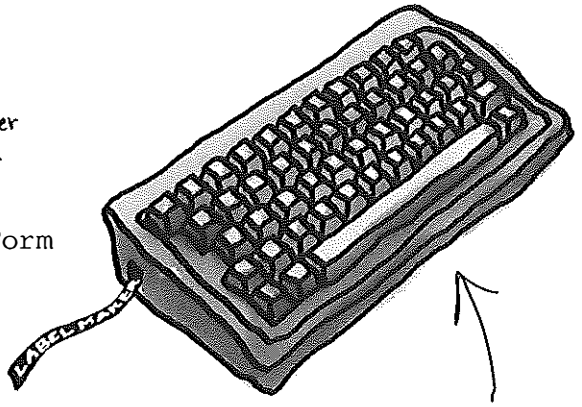
Das wird als Instantiierung des Objekts bezeichnet.

Um ein neues Objekt zu erstellen, verwenden Sie Code wie **`new Typ`**. Aber das ist nicht genug. Obgleich dieser Code ein neues **`Typ`**-Objekt auf dem Heap erstellt, gibt er Ihnen noch keine Möglichkeit, auf dieses Objekt *zuzugreifen*. ***Sie benötigen eine Referenz auf dieses Objekt***. Sie erzeugen deswegen eine **Referenzvariable**: eine Variable des Typs **`Typ`** mit einem Namen wie **`tim`**. **`tim`** ist also eine Referenz auf das neue **`Typ`**-Objekt, das Sie erstellt haben. Jedes Mal, wenn Sie diesen bestimmten Typ verwenden möchten, können Sie ihn mit der Referenzvariablen namens **`tim`** referenzieren.

Wenn Sie eine Variable haben, die einen Objekttyp hat, ist es eine Referenzvariable: eine Referenz auf ein bestimmtes Objekt. Werfen Sie einen Blick darauf:



Hier ist der Heap vor der Ausführung Ihres Codes.



Das Erstellen einer Referenz ist, als erstelle man ein Etikett mit einer Etikettiermaschine – statt es an Ihr Zeug zu kleben, nutzen Sie es, um ein Objekt zu etikettieren, damit Sie später darauf verweisen können.

Diese Variable heißt tim, sie wird ein Objekt des Typs Typ referenzieren.

```
public partial class Form1 : Form
{
    Typ tim;

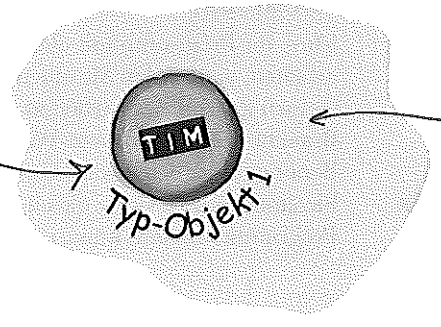
    public Form1()
    {
        InitializeComponent();

        tim = new Typ();
    }
```

Das ist die Referenzvariable ...

... und das ist das Objekt, auf das tim jetzt verweist.

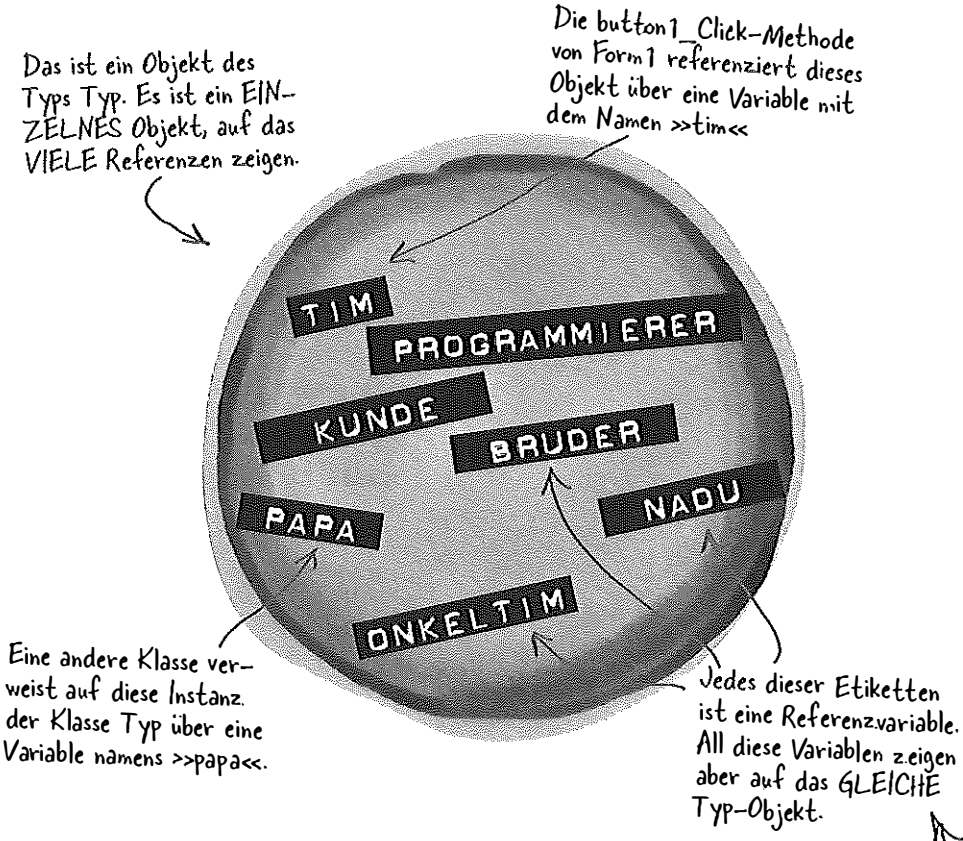
Hier ist der Heap nach Ausführung Ihres Codes. Es gibt ein Objekt, auf das die Variable tim verweist.



Dieses Typ-Objekt kann NUR über die Referenzvariable tim referenziert werden.

Referenzen sind wie Etiketten für Ihr Objekt

In der Küche haben Sie wahrscheinlich eine Dose für Salz und eine für Zucker. Würden Sie die Etiketten darauf tauschen, könnte das dazu führen, dass Sie das eine oder andere Essen versauen – den selbst wenn die Etiketten vertauscht werden, bleibt der Inhalt gleich. ***Referenzen sind wie Etiketten.*** Sie können Etiketten herumreichen, an unterschiedliche Dinge heften, aber das **Objekt** stellt die Methoden und Daten, nicht die Referenz selbst.



Muss Ihr Code mit Objekten im Speicher arbeiten, nutzt er eine Referenz, d.h. eine Variable, deren Typ die Klasse des Objekts ist, auf das sie verweist. Eine Referenz ist wie ein Etikett, das Ihr Code nutzt, um mit einem bestimmten Objekt zu reden.

Auf ein Objekt verweist man nie direkt. Code wie `Typ.GeldGeben()` können Sie beispielsweise nicht schreiben, wenn `Typ` Ihr Objekttyp ist. Der C#-Compiler weiß nicht, von welchem `Typ` Sie hier reden, da es auf dem Heap mehrere Instanzen von `Typ` geben könnte. Sie brauchen also eine Referenzvariable `tim`, der Sie eine spezifische Instanz zuweisen können, z.B. `Typ tim = new Typ()`.

Jetzt können Sie Methoden wie `tim.GeldGeben()` verwenden. `tim` verweist auf eine bestimmte Instanz der Klasse `Typ`, und der C#-Compiler weiß genau, welche Instanz er verwenden soll. Wie Sie oben gesehen haben, können ***mehrere Etiketten auf die gleiche Instanz zeigen***. Sie könnten also `Typ papa = tim` sagen und dann `papa.GeldGeben()` aufrufen. Auch das geht – Tims Sohn macht das jeden Tag!

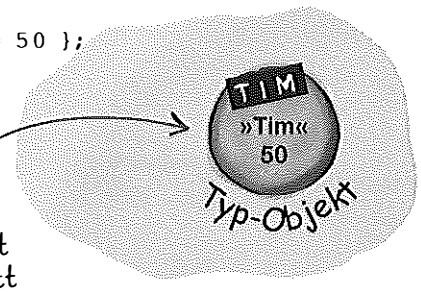
Gibt es keine Referenzen mehr, holt die Müllabfuhr Ihr Objekt

Trägt ein Objekt kein einziges Etikett mehr, kann kein Programm mehr auf dieses Objekt zugreifen. Das heißt, dass C# dieses Objekt für die Müllabfuhr, die wir, wie in Programmiererkreisen üblich, **Garbage Collection** nennen werden, vormerkt. So bezeichnet man den Mechanismus, über den C# nicht referenzierte Objekte loswird und den Speicher freigibt, den diese Objekte für Ihr Programm belegt haben.

Ein Objekt bleibt nur so lange auf dem Heap, wie es Referenzen darauf gibt. Verschwindet die letzte Referenz, verschwindet auch das Objekt.

1 **Hier ist etwas Code, der ein Objekt erzeugt.**

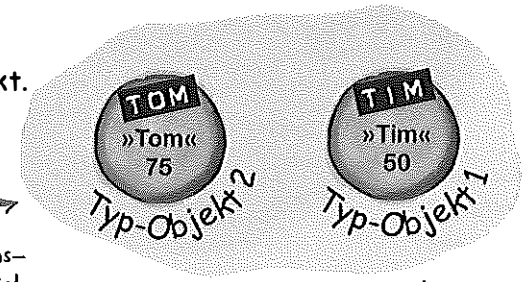
```
Typ tim = new Typ()
 { Name = "Tim", Geld = 50 };
```



Im ersten Teil der Zeile steht die Deklaration, die das Etikett erzeugt. Der zweite Teil erzeugt das Objekt und klebt das Etikett darauf.

2 **Erzeugen wir jetzt ein zweites Objekt.**

```
Typ tom = new Typ()
 { Name = "Tom", Geld = 75 };
```



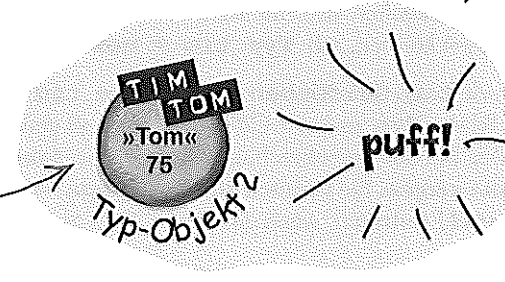
Jetzt haben wir zwei Typ-Instanzen und zwei Referenzvariablen: eine für jede Typ-Instanz.

Aber es gibt keine Referenz auf das erste Typ-Objekt mehr ...

3 **Ändern wir die Referenz auf das erste Objekt und lassen wir sie auf das zweite zeigen.**

```
tim = tom;
```

Jetzt zeigt tim auf das gleiche Objekt wie tom.



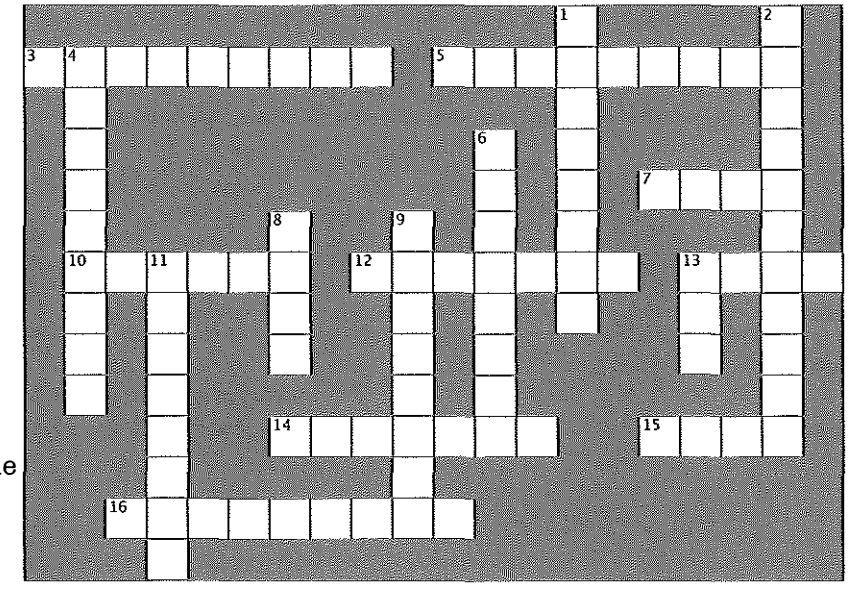
... C# merkt es deswegen für die Garbage Collection vor, die es löscht. Und schwupps, weg ist es!



Typ-Kreuzworträtsel

Machen Sie eine Pause, lehnen Sie sich zurück und geben Sie Ihrer rechten Gehirnhälfte etwas zu tun. Es ist das übliche Kreuzworträtsel. Alle Lösungswörter stammen aus diesem Kapitel.

Wenn Sie fertig sind, können Sie umblättern, um sich dem Rest dieses Kapitels zu widmen.



Waagerecht

3. Die Deklaration und die _____ einer Variablen können zu einer Anweisung zusammengefasst werden.
5. Variablennamen wie dieGroßeZahl und rexDerHund nutzen diese Art der Groß-/Kleinschreibung.
7. Der Typ, der nur ein einzelnes Zeichen speichert.
10. Der Typ, der die größten Zahlen speichern kann.
12. Das, was (int) in folgender Codezeile macht:
x = (int) y;
13. Eine spezielle Referenzvariable, die auf die aktuelle Instanz verweist.
14. Zeigen auf ein Objekt keine Referenzen mehr, wird es von der _____-Collection aus dem Heap entfernt.
15. Der zweite Teil einer Variablendeklaration.
16. Das tun Sie, wenn Sie mit dem +-Operator zwei Strings zusammenkleben.

Senkrecht

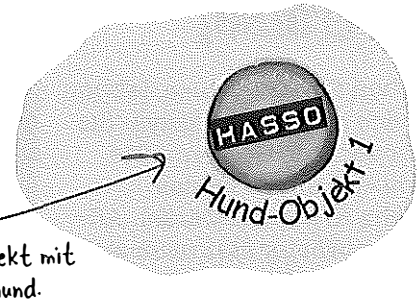
1. Eine Variable, die auf ein Objekt zeigt.
2. »namespace«, »for«, »while«, »using« und »new« sind alles Beispiele für _____ Wörter.
4. Das machen die arithmetischen Operatoren für Sie mit Typen automatisch.
6. Jedes Objekt hat diese Methode, die es in einen String umwandelt.
8. Wird das für eine Variable nicht gesetzt, lässt sich Ihr Programm nicht kompilieren.
9. Was Ihr Programm nutzt, um mit Daten zu arbeiten, die im Speicher sind.
11. Die vier Typen für ganze Zahlen, die nur positive Zahlen speichern.
13. Die Deklaration einer Variablen beginnt immer damit.

Antworten auf Seite 162.

Mehrere Referenzen und ihre Nebenwirkungen

Wenn Sie beginnen, Ihre Referenzvariablen herumzureichen, müssen Sie aufpassen. Oft scheint es, als würden Sie eine Variable einfach auf ein anderes Objekt zeigen lassen. Aber das kann dazu führen, dass dabei alle Referenzen auf ein Objekt gelöscht werden. Das ist nicht notwendigerweise falsch, aber auch nicht immer das, was Sie eigentlich wollten. Werfen Sie einen Blick darauf:

1
```
Hund hasso = new Hund();
hasso.Rasse = "Windhund";
```

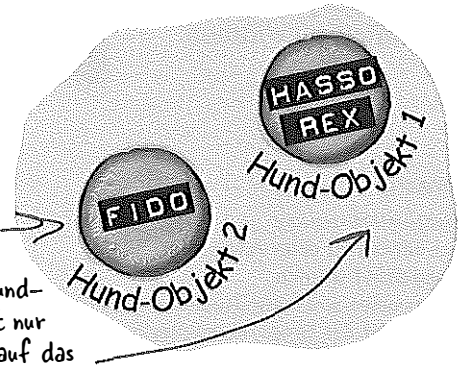


hasso ist ein Hund-Objekt mit dem Rasse-Wert Windhund.

Objekte: 1

Referenzen: 1

2
```
Hund fido = new Hund();
fido.Rasse = "Beagle";
Hund rex = hasso;
```

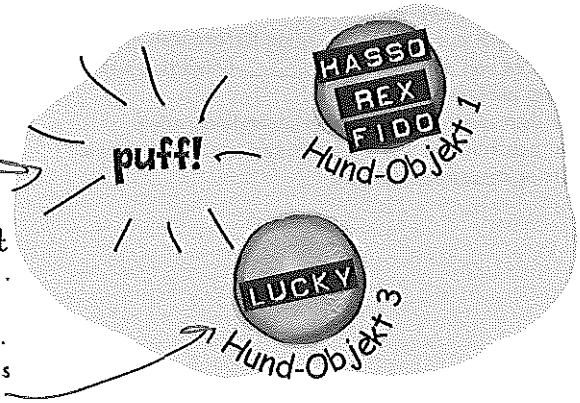


fido ist ein weiteres Hund-Objekt. rex dagegen ist nur eine weitere Referenz auf das erste Objekt.

Objekte: 2

Referenzen: 3

3
```
Hund lucky = new Hund();
lucky.Rasse = "Dachs";
fido = hasso;
```



lucky ist das dritte Objekt. Aber fido zeigt jetzt auf das Objekt 1. Es gibt keine Referenzen auf Objekt 2 mehr. Für das Programm ist es erledigt.

Objekte: 2

Referenzen: 4

Spitzen Sie Ihren Bleistift

Jetzt sind Sie an der Reihe. Hier ist ein langer Codeblock. Ermitteln Sie, wie viele Objekte und Referenzen es nach den einzelnen Phasen gibt. Zeichnen Sie auf der rechten Seite ein Bild der Objekte und Etiketten auf dem Heap.

1
```
Hund hasso = new Hund();
hasso.Rasse = "Windhund";
Hund rantanplan = new Hund();
Hund fido = new Hund();
Hund quentin = Fido;
```

Objekte:_____

Referenzen:_____

2
```
Hund rex = new Hund();
rex.Rasse = "Dackel";
rex = hasso;
```

Objekte:_____

Referenzen:_____

3
```
Hund lucky = new Hund();
lucky.Rasse = "Beagle";
Hund charlie = fido;
fido = hasso;
```

Objekte:_____

Referenzen:_____

4
```
rantanplan = lucky;
Hund otto = new Hund();
otto.Rasse = "Mops";
```

Objekte:_____

Referenzen:_____

5
```
charlie = otto;
lucky = rantanplan;
```

Objekte:_____

Referenzen:_____



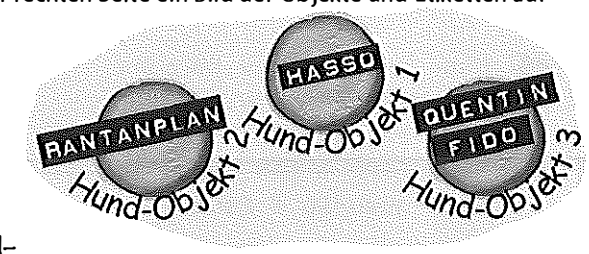
Spitzen Sie Ihren Bleistift
Lösung

Jetzt sind Sie an der Reihe. Hier ist ein langer Codeblock. Ermitteln Sie, wie viele Objekte und Referenzen es nach den einzelnen Phasen gibt. Zeichnen Sie auf der rechten Seite ein Bild der Objekte und Etiketten auf dem Heap.

1
```
Hund hasso = new Hund();
hasso.Rasse = "Windhund";
Hund rantanplan = new Hund();
Hund fido = new Hund();
Hund quentin = fido;
```

Objekte: 3

Referenzen: 4



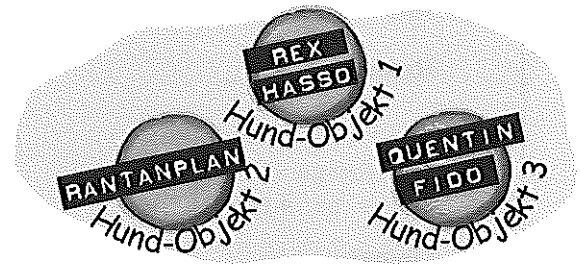
2
```
Hund rex = new Hund();
rex.Rasse = "Dackel";
rex = hasso;
```

Es wird ein neues Hund-Objekt erzeugt, aber rex ist die einzige Referenz darauf. Nach rex = hasso ist dieses Objekt wieder verschwunden.

Objekte: 3

Referenzen: 5

Hier wird ein neues Hund-Objekt erstellt, das erhalten bleibt, weil die Referenz darauf nicht überschrieben wird.

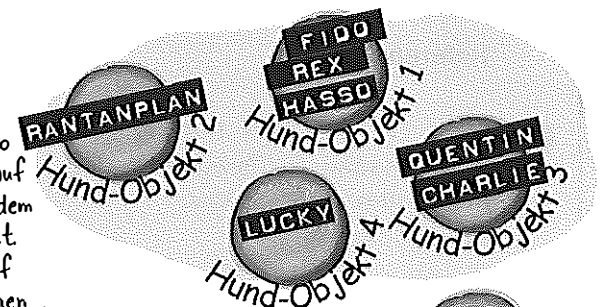


3
```
Hund lucky = new Hund();
lucky.Rasse = "Beagle";
Hund charlie = fido;
fido = hasso;
```

Charlie wurde gleich fido gesetzt, als fido noch auf Objekt 3 verwies. Nachdem fido gleich hasso gesetzt wurde, verweist fido auf Objekt 1, charlie hingegen weiter auf Objekt 3.

Objekte: 4

Referenzen: 7



4
```
rantanplan = lucky;
Hund otto = new Hund();
otto.Rasse = "Mops";
```

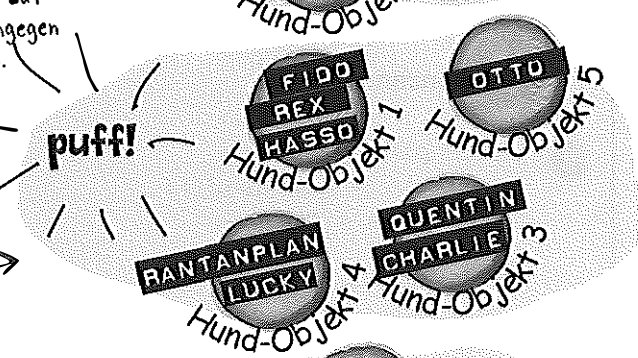
Objekt 2 verliert seine letzte Referenz und verschwindet.

puff!

Objekte: 4

Referenzen: 8

Weil lucky jetzt auf das gleiche Objekt wie rantanplan verweist, gibt es keine Referenz auf das alte Objekt von rantanplan mehr.

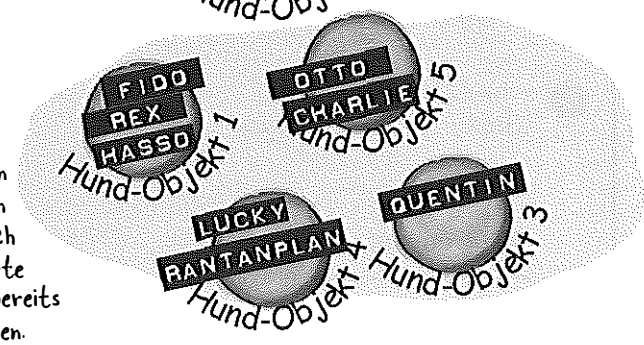


5
```
charlie = otto;
lucky = rantanplan;
```

Objekte: 4

Referenzen: 8

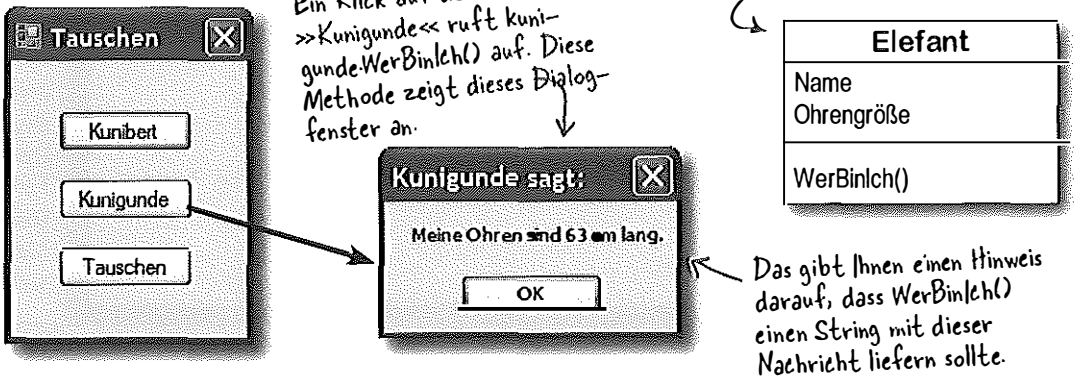
Hier werden nur Referenzen verschoben, aber keine neuen Objekte erzeugt. lucky gleich rantanplan zu setzen änderte nichts, da beide Variablen bereits auf das gleiche Objekt zeigen.





Erstellen Sie ein Programm mit einer Elefant-Klasse. Erzeugen Sie zwei Elefant-Instanzen und tauschen Sie dann die Referenzvariablen, die auf sie zeigen, ***ohne*** dass eine der Elefant-Instanzen dabei ein Fall für die Müllabfuhr wird.

1 **Erstellen Sie ein neues Windows-Forms-Anwendung-Projekt.**
Geben Sie dem Formular diese Gestalt:



2 **Erzeugen Sie die Klasse Elefant.**
Fügen Sie dem Projekt die Klasse `Elefant` hinzu. Sehen Sie sich das Klassendiagramm für `Elefant` an – Sie brauchen ein `int`-Feld namens `Ohrengröße` und ein `string`-Feld namens `Name`. (Achten Sie darauf, dass beide public sind.) Fügen Sie dann die Methode `WerBinIch()` hinzu, die ein Dialogfenster anzeigt, das Ihnen den Namen und die Ohrengröße des Elefanten sagt.

3 **Erzeugen Sie zwei Elefant-Instanzen und -Referenzen.**
Fügen Sie `Form1` (direkt unter der Klassendeklaration) zwei `Elefant`-Felder namens `kunibert` und `kunigunde` hinzu. Initialisieren Sie diese mit dem korrekten Namen und einer Ohrengröße. Hier sind die **`Elefant`**-Objektinitialisierer für Ihr Formular:

```
kunigunde = new Elefant() { Name = "Kunigunde", Ohrengröße = 63 };
kunibert = new Elefant() { Name = "Kunibert", Ohrengröße = 81 };
```

4 **Machen Sie die Buttons "Kunibert" und "Kunigunde" funktionsfähig.**
Lassen Sie den Kunibert-Button `kunibert.WerBinIch()` und den Kunigunde-Button `kunigunde.WerBinIch()` aufrufen.

5 **Richten Sie den Tauschen-Button ein.**
Jetzt kommt der schwere Teil. Bringen Sie den Tauschen-Button dazu, die beiden Referenzen zu ***tauschen***, damit die Variablen `kunibert` und `kunigunde` durch einen Klick auf Tauschen die Objekte tauschen und eine »Objekte getauscht«-Meldung angezeigt wird. Testen Sie das Programm, indem Sie auf Tauschen und dann auf die beiden anderen Buttons klicken. Nach dem ersten Klick auf Tauschen sollte der Button Kunibert das Dialogfenster für Kunigunde anzeigen. Nach dem nächsten Klick auf Tauschen sollte alles wieder beim Alten sein.

C# löscht alle Objekte, auf die es keine Referenzen mehr gibt. Ein kleiner Hinweis: Wenn Sie ein Glas Bier in ein Glas schütten wollen, in dem im Moment Wasser ist, brauchen Sie ein drittes Glas ...



Lösungen zu den Übungen

Erstellen Sie ein Programm mit einer Elefant-Klasse. Erzeugen Sie zwei Elefant-Instanzen, und tauschen Sie dann die Referenzvariablen, die auf sie zeigen, **ohne** dass eine der Elefant-Instanzen dabei ein Fall für die Müllabfuhr wird.

Das ist der Code für die Klasse Elefant in der Datei Elefant.cs, die wir dem Projekt hinzugefügt haben. Vergessen Sie am Anfang des Formulars die >>using System.Windows.Forms;<<-Zeile nicht. Ohne funktioniert die MessageBox-Anweisung nicht.

```
using System.Windows.Forms;

class Elefant {

  public int Ohrengröße;
  public String Name;

  public void WerBinIch() {
     MessageBox.Show("Meine Ohren sind " + Ohrengröße + " cm lang.",
        Name + " sagt");
  }
}
```

Hier ist der Code für die Klasse Form1 aus Form1.cs.

```
public partial class Form1 : Form {

    Elefant kunigunde;
    Elefant kunibert;

    public Form1()
    {
        InitializeComponent();
        kunigunde = new Elefant()
             { Name = "Kunigunde", Ohrengröße = 63 };
        kunibert = new Elefant()
             { Name = "Kunibert", Ohrengröße = 81 };
    }

    private void button1_Click(object sender, EventArgs e) {
        kunibert.WerBinIch();
    }

    private void button2_Click(object sender, EventArgs e) {
        kunigunde.WerBinIch();
    }

    private void button3_Click(object sender, EventArgs e) {
        Elefant halter;
        halter = kunibert;
        kunibert = kunigunde;
        kunigunde = halter;
        MessageBox.Show("Objekte getauscht");
    }
}
```

Für diese Referenz gibt es keine new-Anweisung, weil wir keine weitere Elefant-Instanz brauchen.

Lassen Sie einfach kunibert auf kunigunde zeigen, gibt es keine Referenz auf Kunibert mehr. Sein Objekt geht also verloren. Deswegen brauchen Sie die Referenzhalter, um das Kunibert-Objekt festzuhalten, wenn der Referenz kunibert das Objekt Kunigunde zugewiesen wird.



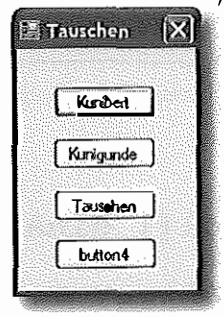
Kopf-Nuss

Warum haben wir die Methode Tauschen() nicht in die Klasse Elefant eingefügt?

Zwei Referenzen bedeuten ZWEI Möglichkeiten, die Daten eines Objekts zu ändern

Sie können nicht nur alle Referenzen auf ein Objekt verlieren, wenn Sie mehrere Referenzen haben, Sie können ein Objekt auch unbeabsichtigt ändern. Anders gesagt: Eine Referenz auf ein Objekt kann das Objekt ***ändern***, während die andere Referenz auf das Objekt ***keine Ahnung*** hat, das sich etwas geändert hat. Aufgepasst:

Tun Sie das!



1. Spendieren Sie dem Formular noch einen Button.

2. Das ist der Code für diesen Button. Können Sie sich denken, was passiert, wenn Sie darauf klicken?

```
private void button4_Click(object sender, EventArgs e)
{
    kunibert = kunigunde;
    kunibert.Ohrengröße = 4321;
    kunibert.WerBinIch();
}
```

Diese Anweisung setzt die Ohrengröße des Objekts, auf das kunibert verweist, auf 4321.

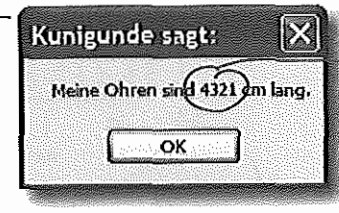
Nachdem dieser Code ausgeführt wurde, zeigen kunibert und kunigunde auf das GLEICHE Objekt.

Sie rufen die Methode WerBinIch() auf der Referenz kunibert auf.

Aber kunibert zeigt auf das gleiche Ding wie kunigunde.



3. Klicken Sie dann auf den neuen Button. Moment mal ... das ist doch die Meldung für Kunigunde! Haben wir nicht die WerBinIch()-Methode von Kunibert aufgerufen?



Das ist die Meldung für Kunigunde.

Aber die Ohrengröße haben wir doch per Referenz kunibert gesetzt! Wieso das?



kunibert und kunigunde sind jetzt austauschbar. Änderungen an einer der Referenzen ändern das Objekt, auf das BEIDE zeigen ... es gibt keinen Unterschied zwischen kunibert und kunigunde mehr, da beide auf das GLEICHE Objekt zeigen.

Beachten Sie, dass die Daten NICHT überschrieben werden – es ändern sich nur die Referenzen.

Ein Sonderfall: Arrays

Wenn Sie eine Menge Daten des gleichen Typs festhalten müssen, beispielsweise eine Liste mit Größen oder eine Rotte von Hunden, können Sie das mit einem **Array** machen. Ein Array zeichnet aus, dass es eine **Sammlung von Variablen** ist, die als ein Objekt behandelt wird. Über ein Array können Sie mehrere Datenteile speichern und ändern, ohne dass Sie die einzelnen Variablen separat nachhalten müssen. Um ein Array zu erzeugen, deklarieren Sie es genau wie jede andere Variable mit einem Namen und einem Typ:

Sie hätten die Deklaration der Variablen einArray mit der Initialisierung verbinden können – wie bei jeder anderen Variablen. Dann sähe das so aus:

```
bool[] einArray = new bool[15];
```

Sie deklarieren ein Array über den Typ plus eckige Klammern.

```
bool[] einArray;
einArray = new bool[15];
einArray[4] = true;
```

Dieses Array enthält 15 Elemente.

Ein Array wird mit dem Schlüsselwort new erzeugt, weil es ein Objekt ist. Eine Array-Variable ist also ebenfalls eine Referenzvariable.

Diese Zeile setzt den Wert des fünften Elements von einArray auf true. Es ist das fünfte, weil das erste einArray[0] ist, das zweite einArray[1] usw.

Nutzen Sie die Elemente in einem Array, als wären es gewöhnliche Variablen

Hier ist ein Beispiel für Code, der ein Array deklariert und füllt – und es zeigt, was auf dem Heap passiert, wenn Sie das tun. Das erste Element im Array hat den **Index** null.

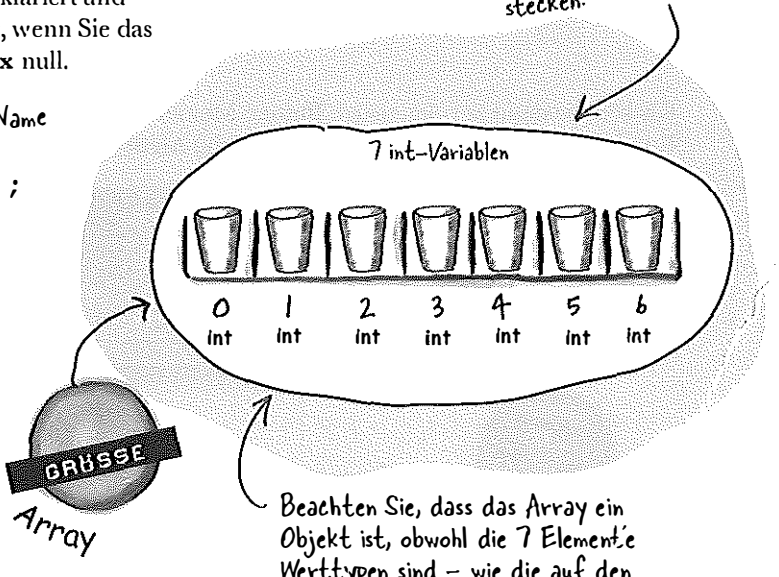
Der Typ jedes Elements in diesem Array.

Name

```
int[] größe;
größe = new int[7];
größe[0] = 68;
größe[1] = 70;
größe[2] = 63;
größe[3] = 60;
größe[4] = 58;
größe[5] = 72;
größe[6] = 74;
```

Sie referenzieren die Elemente, aber jedes funktioniert im Prinzip wie eine gewöhnliche Variable.

Im Speicher wird das Array als ein Speicherhappen gespeichert, obwohl darin mehrere int-Variablen stecken.



Beachten Sie, dass das Array ein Objekt ist, obwohl die 7 Elemente Werttypen sind – wie die auf den ersten beiden Seiten dieses Kapitels.

Arrays können auch einen Haufen Referenzvariablen speichern

Ein Array mit Objektreferenzen können Sie auf gleiche Weise erzeugen wie ein Array mit Zahlen oder Strings. Arrays kümmern sich nicht um den Typ der darin gespeicherten Variablen. Der bleibt Ihnen überlassen. Sie können also problemlos ein Array mit `int`s wie auch ein Array mit `Ente`-Objekten haben.

Hier ist Code, der ein Array mit 7 **Hund**-Variablen erzeugt. Die Zeile, die das Array initialisiert, erzeugt nur die Referenzvariablen. Da es nur zwei `new Hund()`-Zeilen gibt, werden nur zwei richtige Instanzen der Klasse **Hund** erstellt.

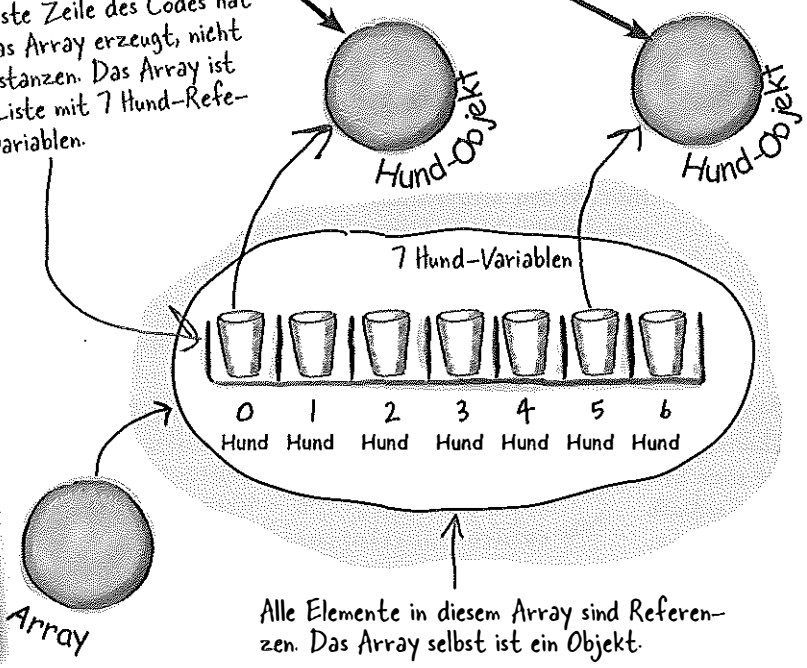
```
Hund[] hunde = new Hund[7];
hunde[5] = new Hund();
hunde[0] = new Hund();
```

Diese Zeile deklariert die Variable hunde, die ein Array mit Referenzen auf Hund-Objekte speichert, und erzeugt dann ein Array mit 7 Elementen.

Diese beiden Zeilen erzeugen neue Instanzen von Hund und stecken sie zu den Indizes 0 und 5.

Die beim Setzen oder Abrufen eines Array-Elements in den eckigen Klammern angegebene Zahl nennt man Index. Das erste Element im Array hat den Index null.

Die erste Zeile des Codes hat nur das Array erzeugt, nicht die Instanzen. Das Array ist eine Liste mit 7 Hund-Referenzvariablen.



Die Länge von Arrays

Wie viele Elemente in einem Array stecken, können Sie über seine Length-Eigenschaft ermitteln. Haben Sie ein Array namens größe, können Sie größe.Length verwenden, um seine Länge zu ermitteln. Enthält es 7 Elemente, wird Ihnen 7 gemeldet – das bedeutet, dass die Elemente im Array die Indizes 0 bis 6 tragen.

Willkommen beim Sandwich-Discount vom Strammen Max

Der Stramme Max hat einen Haufen Fleisch, diverses Brot und mehr Soßen, als man aufzählen kann. Aber was er nicht hat, ist eine Speisekarte! Können Sie ein Programm erstellen, das ihm jeden Tag eine neue *zufällige* Speisekarte erstellt?

Tun Sie das!

MenüMacher
Zufallszahl Fleisch Soßen Brot
MenüEintragHolen()

1 Fügen Sie einem neuen Projekt die Klasse MenüMacher hinzu.

Zur Erstellung einer Speisekarte benötigen Sie Zutaten. Arrays wären perfekt für diese Listen. Außerdem brauchen wir eine Möglichkeit, zufällig Zutaten auszuwählen, um diese zu einem Sandwich zu kombinieren. Das .NET Framework bietet die eingebaute Klasse **Random**, die Zufallszahlen generiert. Unsere Klasse hat also vier Felder: das Feld Zufallszahl mit einer Referenz auf ein Random-Objekt und drei `string`-Arrays für Fleisch, Soßen und Brot.

Die Klasse hat drei Felder zur Speicherung der drei String-Arrays. Diese nutzt sie, um zufällige Speisekarteneinträge zu erzeugen.

Das Feld namens Zufallszahl hält eine Referenz auf ein Random-Objekt. Ein Aufruf seiner Next()-Methode erzeugt Zufallszahlen.

```
public class MenüMacher {
    public Random Zufallszahl;

    string[] Fleisch = { "Roastbeef", "Salami", "Pute", "Speck", "Braten" };

    string[] Soßen = { "Süßer Senf", "Scharfer Senf",
                "Ketchup", "Mayo", "Aioli", "Remoulade" };

    string[] Brot = { "Graubrot", "Weißbrot", "Toast", "Pumpernickel",
                "Ciabatta", "Brötchen" };
}
```

Denken Sie daran, beim Zugriff auf die Array-Elemente eckige Klammern zu verwenden. Der Wert von Brot[2] ist »Toast«.

2 Schreiben Sie eine MenüEintragHolen()-Methode, die ein Zufalls-Sandwich erzeugt.

Der Zweck der Klasse ist die Zusammenstellung von Sandwiches. Fügen wir also eine entsprechende Methode hinzu. Sie nutzt die **`Next()`**-Methode des `Random`-Objekts, um zufällig Fleisch, Soße und Brot aus den Arrays auszuwählen. Übergeben Sie `Next()` einen `int`-Parameter, liefert die Methode eine Zufallszahl, die kleiner als dieser Parameter ist. Heißt Ihr `Random`-Objekt `Zufallszahl`, liefert `Zufallszahl.Next(7)` also eine Zufallszahl zwischen 0 und 6.

Woher wissen wir nun, welchen Parameter wir der `Next()`-Methode übergeben müssen? Na, das ist einfach – wir übergeben einfach die `Length`-Eigenschaft der einzelnen Arrays. Diese liefert die Anzahl der Elemente im jeweiligen Array.

Die Methode MenüEintragHolen() liefert einen String, der ein Sandwich enthält, das aus Zufallselementen der drei Arrays gebildet wird.

```
public string MenüEintragHolen() {
    string zufallsFleisch = Fleisch[Zufallszahl.Next(Fleisch.Length)];
    string zufallsSoße = Soßen[Zufallszahl.Next(Soßen.Length)];
    string zufallsBrot = Brot[Zufallszahl.Next(Brot.Length)];
    return zufallsFleisch + " mit " + zufallsSoße + " auf " + zufallsBrot;
}
```

Die Methode steckt ein beliebiges Element aus dem Array Fleisch in die Variable zufallsFleisch, indem sie der Next()-Methode des Random-Objekts Fleisch.Length übergibt. Das Array Fleisch enthält 5 Elemente. Next(5) liefert also eine Zufallszahl zwischen 0 und 4.



Wie es funktioniert ...

Die Methode Zufallszahl.Next(7) liefert eine Zufallszahl kleiner 7. Fleisch.Length liefert die Anzahl von Elementen in Fleisch. `Zufallszahl.Next(Fleisch.Length)` liefert Ihnen also eine Zufallszahl, die größer gleich null, aber kleiner als die Anzahl von Elementen im Array Fleisch ist.

```
Fleisch[Zufallszahl.Next(Fleisch.Length)]
```

Fleisch ist ein Array mit Strings. Es enthält fünf Elemente mit den Indizes 0 bis 4. Fleisch[0] entspricht also >>Roastbeef<< und Fleisch[3] entspricht >>Speck<<.



8 Erstellen Sie Ihr Formular.

Fügen Sie dem Formular sechs Labels, `label1` bis `label6`, hinzu. Ergänzen Sie dann Code, der die `Text`-Eigenschaft jedes Labels über ein `MenüMacher`-Objekt setzt. Sie müssen das Objekt mit einer neuen Instanz der Klasse Random initialisieren. Hier ist der Code:

```
public Form1() {
    InitializeComponent();

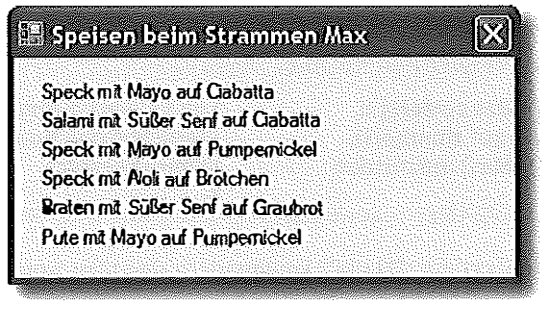
    MenüMacher menü = new MenüMacher() { Zufallszahl = new Random() };

    label1.Text = menü.MenüEintragHolen();
    label2.Text = menü.MenüEintragHolen();
    label3.Text = menü.MenüEintragHolen();
    label4.Text = menü.MenüEintragHolen();
    label5.Text = menü.MenüEintragHolen();
    label6.Text = menü.MenüEintragHolen();
}
```

Initialisieren Sie das Zufallszahl-Feld des MenüMacher-Objekts über einen Objektinitialisierer mit einer neuen Instanz der Klasse Random.

Jetzt ist alles eingerichtet, um über die Methode MenüEintragHolen() sechs verschiedene zufällige Sandwiches abzurufen.

Bei der Ausführung des Programms zeigen die sechs Labels sechs verschiedene zufällig zusammengestellte Sandwiches.



Objekte nutzen Referenzen, um miteinander zu reden

Bislang haben Sie gesehen, wie Formulare mit Objekten reden, indem Sie auf ihnen Methoden aufrufen oder ihre Eigenschaften abfragen. Über Referenzen können Objekte aber auch untereinander Methoden aufrufen. Eigentlich kann ein Formular nichts tun, was Objekte nicht auch tun können, weil **ein Formular nur ein anderes Objekt ist**. Wenn Objekte miteinander reden, ist das Schlüsselwort **`this`** nützlich. Jedes Mal, wenn ein Objekt das Schlüsselwort `this` verwendet, verweist es auf sich selbst – es ist eine Referenz, die auf das Objekt zeigt, das es aufruft.

1 Hier ist eine Methode, die einem Elefanten das Sprechen beibringt.

Fügen wir der Klasse Elefant eine Methode hinzu. Ihr erster Parameter ist eine Nachricht von einem Elefanten. Der zweite Parameter ist der Elefant, der hier spricht:

```
public void SprichMitMir(string nachricht, Elefant werHierSpricht) {
    MessageBox.Show(werHierSpricht.Name + " sagt: " + nachricht);
}
```

So sieht es aus, wenn die Methode aufgerufen wird:

```
Elefant kunibert = new Elefant() { Name = "Kunibert", Ohrengröße = 81 };
Elefant kunigunde = new Elefant() { Name = "Kunigunde", Ohrengröße = 63 };
kunibert.SprichMitMir("Hallo", kunigunde);
```

Wir haben Kuniberts SprichMitMir()-Methode aufgerufen und ihr zwei Parameter übergeben: »Hallo« und eine Referenz auf das Kunigunde-Objekt. Der Parameter werHierSpricht wird genutzt, um auf die Name-Eigenschaft des Elefanten zuzugreifen, der an SprichMitMir() übergeben wurde.

2 Hier ist eine Methode, die eine andere Methode aufruft.

Fügen wir der Klasse Elefant jetzt die Methode SprichMit() hinzu. Sie nutzt ein besonderes Schlüsselwort: **`this`**. Das ist eine Referenz, **über die ein Objekt über sich selbst reden kann**.

```
public void SprichMit(Elefant redeMit, string nachricht) {
    redeMit.SprichMitMir(nachricht, this);
}
```

Diese Methode von Elefant ruft die SprichtMit()-Methode eines anderen Elefanten auf. Über sie können Elefanten miteinander reden.

Schauen wir uns genauer an, wie das funktioniert.

```
kunigunde.SprichMit(kunibert, "Hallo");
```

Wird Kunigundes `SprichMit()`-Methode aufgerufen, nutzt sie den Parameter `redeMit`, um Kuniberts `SprichMitMir()`-Methode aufzurufen.

```
redeMit.SprichMitMir(nachricht, this);
```

Kunigunde nutzt `redeMit` (eine Refrenz auf Kunibert), um SprichMitMir() aufzurufen

`this` wird durch eine Referenz auf das Kunigunde-Objekt ersetzt

```
kunibert.SprichMitMir(nachricht, [Referenz auf Kunigunde]);
```



Das ist, als wäre Kunibert mit ("Hallo", kunigunde) aufgerufen worden:

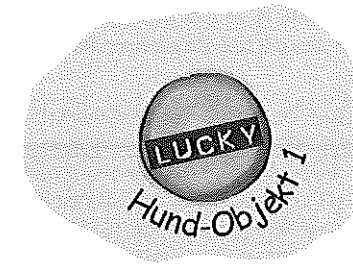
Wohin noch kein Objekt gekommen ist

Es gibt ein weiteres wichtiges Schlüsselwort, das Sie im Zusammenhang mit Objekten verwenden werden. Wenn Sie eine neue Referenz erstellen und diese auf nichts setzen, hat sie dennoch einen Wert. Anfangs ist sie auf **null** gesetzt. Das heißt, dass sie auf nichts zeigt.

```
Hund fido;
```

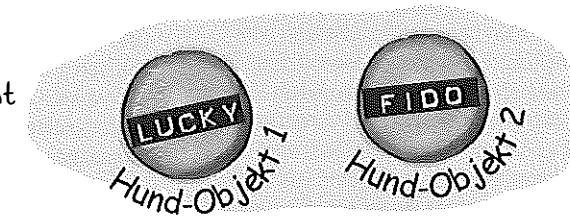
Noch gibt es nur ein Objekt. Die fido-Referenz ist auf null gesetzt.

```
Hund lucky = new Hund();
```



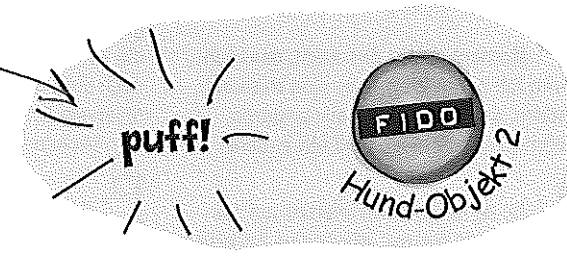
Jetzt zeigt fido auf ein Objekt und ist nicht länger null.

```
fido = new Hund();
```



Setzen wir lucky auf null, zeigt nichts mehr auf das Objekt, das dann Futter für den Garbage Collector wird.

```
lucky = null;
```



Es gibt keine Dummen Fragen

F: Noch mal, bitte – mein Formular ist ein Objekt?

A: Ja! Deswegen beginnt Ihr Code mit einer Klassendeklaration. Öffnen Sie den Code für Ihr Formular und schauen Sie ihn sich an. Öffnen Sie dann Program.cs und blicken Sie in die Methode Main() – dort werden Sie »`new Form1()`« finden.

F: Warum sollte ich `null` verwenden?

A: In gewöhnlichen Programmen wird null auf mehrere Weisen verwendet. Am häufigsten für Tests:

```
if (kunibert == null)
```

Dieser Test liefert `true`, wenn die `kunibert`-Referenz auf null gesetzt ist.

Außerdem wird das Schlüsselwort `null` verwendet, wenn man erzwingen will, dass sich der Garbage Collector ein Objekt schnappt. Haben Sie eine Referenz auf ein Objekt, das Sie nicht mehr benötigen, können Sie die Referenz auf `null` setzen, um es für die Garbage Collection zu markieren (es sei denn, es gibt noch eine weitere Referenz darauf).

F: Sie reden ständig von der Garbage Collection. Wer sammelt da denn den Müll ein?

A: Erinnern Sie sich daran, dass wir Ende des ersten Kapitels von der **Common Language Runtime (oder CLR)** gesprochen haben? Das ist die virtuelle Maschine, die alle .NET-Programme ausführt. Eine *virtuelle Maschine* ist ein Mechanismus, über den man laufende Programme vom Rest des Systems isoliert. Unter anderem verwalten virtuelle Maschinen den Speicher, den sie nutzen. Das bedeutet, dass sie alle ihre Objekte nachhalten, herausfinden, wenn die letzte Referenz auf ein Objekt verschwindet, und den Speicher freigeben, den es eingenommen hat.

Es gibt keine Dummen Fragen

F: Mir ist immer noch nicht ganz klar, wie das mit den Referenzen funktioniert.

A: Über Referenzen nutzen Sie die Methoden und Felder eines Objekts. Erzeugen Sie eine Referenz auf ein Hund-Objekt, beispielsweise rex, können Sie diese Referenz nutzen, um auf Methoden zuzugreifen, die Sie für das Hund-Objekt erstellt haben. Haben Sie (nicht statische) Methoden namens Hund.Bellen() oder Hund.Betteln(), können Sie auf diese über die Referenz rex zugreifen, indem Sie rex.Bellen() oder rex.Betteln() verwenden. Sie könnten über die Referenz auch Informationen in den Feldern des Objekts ändern. Über rex.Rasse könnten Sie beispielsweise das Feld Rasse ändern.

F: Moment. Heißt das nicht, dass Ich, wenn ich einen Wert über eine Referenz ändere, diesen auch für alle anderen Referenzen auf dieses Objekt ändere?

A: Ja. Ist hasso eine Referenz auf das gleiche Objekt wie rex, würde eine Änderung von hasso.Rasse in »Beagle« auch rex.Rasse in ››Beagle« ändern.

F: Noch mal zu diesen Werttypen. Warum kann Ich hinreichend kleine Werte eines großen Datentyps nicht In einen kleineren Datentyp umwandeln?

A: Gut. Die Sache ist, dass Variablen Ihren Zahlen eine Größe zuweisen, die von der tatsächlichen Zahl unabhängig ist. Geben Sie einer Variablen den Typ `long`, reserviert C#, auch wenn die Zahl sehr klein ist (5 beispielsweise), so viel Speicher, wie für den größten möglichen `long`-Wert erforderlich ist. Und das ist vernünftig, weil sich Variablen ändern und andere Werte annehmen können.

C# geht davon aus, dass Sie wissen, was Sie tun, und einer Variablen keinen Typ geben, den sie nicht braucht. Selbst wenn die Zahl jetzt nicht groß ist, kann sie das nach einigen Berechnungen werden, und C# reserviert hinreichend Speicher, um damit klarzukommen.

F: Sagen Sie mir noch einmal, was dieses »`this`.« macht?

A: `this` ist eine spezielle Variable, die Sie nur in einem Objekt verwenden können. In einer Klasse nutzen Sie `this`, um auf ein Feld oder eine Methode dieser speziellen Instanz zu verweisen. Das ist besonders nützlich, wenn Sie mit einer Klasse arbeiten, deren Methoden Methoden anderer Klassen aufrufen. Ein Objekt kann es nutzen, um eine **Referenz auf sich selbst** an ein anderes Objekt zu senden. Ruft rex eine von hassos Methoden mit `this` als Parameter auf, gibt er hasso eine Referenz auf rex.

Code in einer Klasse, von der Objekte instantiiert werden sollen, kann das Schlüsselwort this als Referenz auf die aktuelle Instanz verwenden.

Punkt für Punkt

- Es gibt **Werttypen** für Zahlen, die Zahlen unterschiedlicher Größe aufnehmen. Der größte Ganzzahltyp ist `long` und der kleinste `byte` (bis 255).
- Bei der Deklaration einer Variablen wird IMMER ein Typ angegeben. Die Deklaration kann mit der Initialisierung kombiniert werden.
- Werttypen haben eine Größe. Werte größerer Typen können nicht in Variablen kleinerer Typen gesteckt werden, egal wie groß die Daten tatsächlich sind.
- Bei literalen Werten gibt das Suffix F einen float (15.6F) und das Suffix M einen decimal (36.12M) an.
- Der Compiler verhindert, dass Sie zwei Variablen unterschiedlichen Typs gleichsetzen, es sei denn, Sie geben einen Cast (eine Umwandlung) an.
- Die Sprache reserviert einige Wörter, die Sie nicht für die Namen von Variablen verwenden dürfen. Das sind Wörter wie `for`, `while`, `using`, `new` und andere, die in der Sprache besondere Bedeutung haben.
- Referenzen sind wie Etiketten. An ein Objekt können Sie so viele heften, wie Sie möchten.
- Gibt es auf ein Objekt keine Referenz mehr, kommt der Garbage Collector.

Spitzen Sie Ihren Bleistift

Hier sind ein Array mit Elefant-Objekten und eine Schleife, die alle durchläuft und den mit den größten Ohren sucht. Geben Sie den Wert von größteOhren.Ohrengröße **nach** jedem Durchlauf der `for`-Schleife an.

```
private void button1_Click(object sender, EventArgs e)
{
  Elefant[] elefanten = new Elefant[7];
  elefanten[0] = new Elefant() { Name = "Kunibert", Ohrengröße = 81 };
  elefanten[1] = new Elefant() { Name = "Kunigunde", Ohrengröße = 63 };
  elefanten[2] = new Elefant() { Name = "Lukas", Ohrengröße = 84 };
  elefanten[3] = new Elefant() { Name = "Lucille", Ohrengröße = 61 };
  elefanten[4] = new Elefant() { Name = "Lothar", Ohrengröße = 88 };
  elefanten[5] = new Elefant() { Name = "Linda", Ohrengröße = 67 };
  elefanten[6] = new Elefant() { Name = "Hubert", Ohrengröße = 89 };

  Elefant größteOhren = elefanten[0];

  for (int i = 1; i < elefanten.Length; i++)
  {
    if (elefanten[i].Ohrengröße > größteOhren.Ohrengröße)
    {
      größteOhren = elefanten[i];
    }
  }
  MessageBox.Show(größteOhren.Ohrengröße.ToString());
}
```

Wir erzeugen ein Array mit 7 Elefant-Referenzen.

Jedes Array beginnt mit dem Index 0. Der erste Elefant im Array ist also elefanten[0].

Diese Zeile lässt die Referenz größteOhren auf den Elefanten zeigen, auf den elefanten[i] zeigt.

Vorsicht – diese Schleife beginnt mit dem zweiten Element des Arrays (beim Index 1) und läuft sechs Mal, bis `i` gleich der Länge des Arrays ist.

Durchlauf 1 größteOhren.Ohrengröße = _____

Durchlauf 2 größteOhren.Ohrengröße = _____

Durchlauf 3 größteOhren.Ohrengröße = _____

Durchlauf 4 größteOhren.Ohrengröße = _____

Durchlauf 5 größteOhren.Ohrengröße = _____

Durchlauf 6 größteOhren.Ohrengröße = _____



Spitzen Sie Ihren Bleistift
Lösung

Hier sind ein Array mit Elefant-Objekten und eine Schleife, die alle durchläuft und den mit den größten Ohren sucht. Geben Sie den Wert von größteOhren.Ohrengröße **nach** jedem Durchlauf der `for`-Schleife an.

```
private void button1_Click(object sender, EventArgs e)
{
  Elefant[] elefanten = new Elefant[7];
  elefanten[0] = new Elefant() { Name = "Kunibert", Ohrengröße = 81 };
  elefanten[1] = new Elefant() { Name = "Kunigunde", Ohrengröße = 63 };
  elefanten[2] = new Elefant() { Name = "Lukas", Ohrengröße = 84 };
  elefanten[3] = new Elefant() { Name = "Lucille", Ohrengröße = 61 };
  elefanten[4] = new Elefant() { Name = "Lothar", Ohrengröße = 88 };
  elefanten[5] = new Elefant() { Name = "Linda", Ohrengröße = 67 };
  elefanten[6] = new Elefant() { Name = "Hubert", Ohrengröße = 89 };

  Elefant größteOhren = elefanten[0];
  for (int i = 1; i < elefanten.Length; i++)
  {
    if (elefanten[i].Ohrengröße > größteOhren.Ohrengröße)
    {
      größteOhren = elefanten[i];
    }
  }
  MessageBox.Show(größteOhren.Ohrengröße.ToString());
}
```

Haben Sie daran gedacht, dass die Schleife mit dem zweiten Element des Arrays beginnt? Warum ist das wohl so?

Durchlauf 1 größteOhren.Ohrengröße = 81

Durchlauf 2 größteOhren.Ohrengröße = 84

Durchlauf 3 größteOhren.Ohrengröße = 84

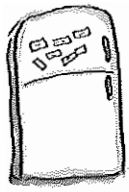
`größteOhren` hält das Element fest, das bei den bisherigen Schleifendurchläufen als das mit den größten Ohren erkannt wurde.

Durchlauf 4 größteOhren.Ohrengröße = 88

Die `for`-Schleife beginnt mit dem zweiten Elefanten und vergleicht ihn mit dem Elefanten, auf den `größteOhren` zeigt. Sind seine Ohren größer, lässt sie `größteOhren` stattdessen auf diesen Elefanten zeigen. Dann geht sie zum nächsten weiter und wieder zum nächsten ... am Ende der Schleife zeigt `größteOhren` auf den Elefanten mit den größten Ohren.

Durchlauf 5 größteOhren.Ohrengröße = 88

Durchlauf 6 größteOhren.Ohrengröße = 89



Code-Magneten

Der Code für einen Button ist völlig durcheinandergeraten. Können Sie die Schnipsel so zusammenstellen, dass sie eine funktionierende Methode ergeben, die die unten gezeigte Ausgabe erzeugt?

```
int y = 0;
```

```
refNum = index[y];
```

```
inseln[0] = "Bermuda";
inseln[1] = "Fidschi";
inseln[2] = "Azoren";
inseln[3] = "Kuba";
```

```
int refNum;

while (y < 4) {
```

```
ergebnis += inseln[refNum];
```

```
MessageBox.Show(ergebnis);
```

```
index[0] = 1;
index[1] = 3;
index[2] = 0;
index[3] = 2;
```

```
}
```

```
}
```

```
String[] inseln = new String[4];
```

```
ergebnis += "\nInsel = ";
```

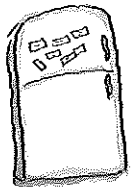
```
int[] index = new int[4];
```

```
y = y + 1;
```




```
private void button1_Click (object sender, EventArgs e)
{
```

```
String ergebnis = "";
```



Code-Magneten, Lösung

Der Code für einen Button ist völlig durcheinandergeraten. Können Sie die Schnipsel so zusammenstellen, dass sie eine funktionierende Methode ergeben, die die unten gezeigte Ausgabe erzeugt?

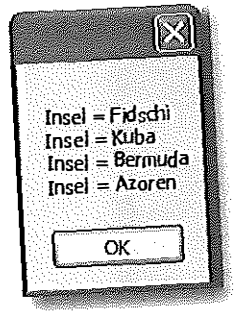
```
private void button1_Click (object sender, EventArgs e)
{
    String ergebnis = "";
    int[] index = new int[4];
    index[0] = 1;
    index[1] = 3;
    index[2] = 0;
    index[3] = 2;
    String[] inseln = new String[4];
    inseln[0] = "Bermuda";
    inseln[1] = "Fidschi";
    inseln[2] = "Azoren";
    inseln[3] = "Kuba";
    int y = 0;
    int refNum;
    while (y < 4) {
        refNum = index[y];
        ergebnis += "\nInsel = ";
        ergebnis += inseln[refNum];
        y = y + 1;
    }
    MessageBox.Show(ergebnis);
}
```

Hier wird das Array `index[]` initialisiert.

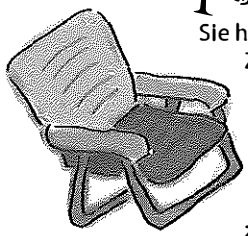
Hier wird das Array `inseln[]` initialisiert

Der String `ergebnis` wird aufgebaut, indem die Zeilen mit dem `+=`-Operator aneinandergehängt werden.

Diese while-Schleife zieht einen Wert aus dem Array `index[]` und nutzt ihn als Index in das Array `inseln[]`.

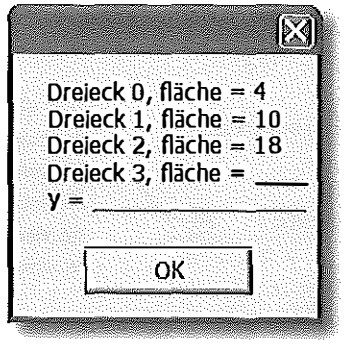


Pool-Puzzle



Sie haben die ***Aufgabe***, die leeren Zeilen im Code mit den Codeschnipseln aus dem Pool zu füllen. Einzelne Schnipsel **können** mehrfach verwendet werden, und Sie werden nicht alle Schnipsel benötigen. Das ***Ziel*** ist es, eine Klasse zu erstellen, die sich kompilieren lässt und die gezeigte Ausgabe erzeugt.

Ausgabe



Bonusfrage!

Zusatzpunkte gibt es, wenn es Ihnen gelingt, die beiden in der Ausgabe verbleibenden Lücken mit Schnipseln aus dem Pool zu füllen.

```
class Dreieck
{
   double Fläche;
   int Höhe;
   int Breite;
   public static void Main(String[] args)
   {
      string ergebnisse = "";

      _____.
      _____________________________
      while ( _____ )
      {

         _____ _____ _____
         _____.Höhe = (x + 1) * 2;
         _____.Breite = x + 4;

         _____ _____ _____
         ergebnisse += "Dreieck " + x + ", Fläche";
         ergebnisse += " = " + _____.Fläche + "\n";

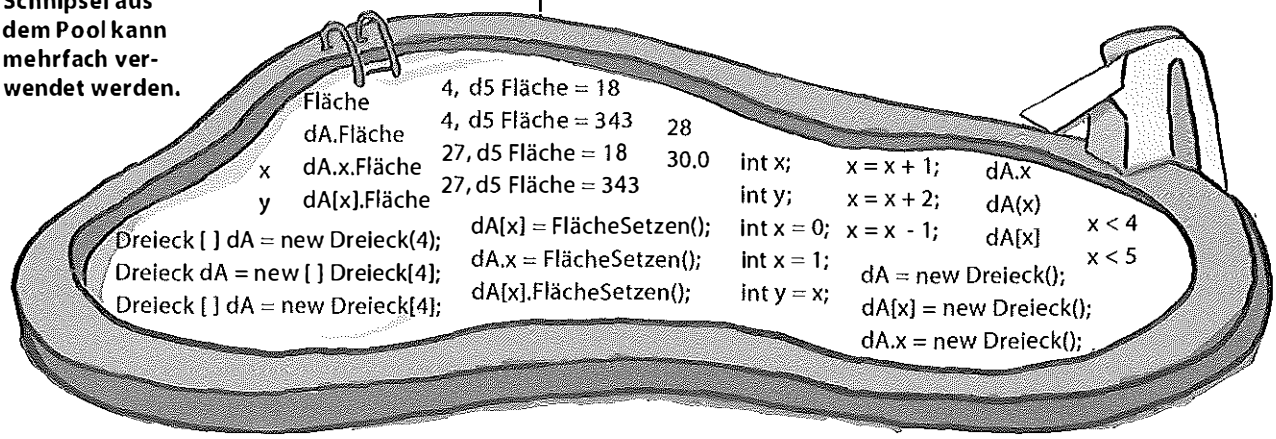
         _____ _____
      }

      _____ _____
      x = 27;
      Dreieck d5 = dA[2];
      dA[2].Fläche = 343;
      ergebnisse += "y = " + y;
      MessageBox.Show(ergebnisse +
         ", d5 Fläche = " + d5.Fläche);
   }
   void FlächeSetzen()
   {
      _____ _____ = (Höhe * Breite) / 2;
   }
}
```

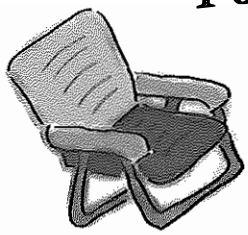
Hier ist der Einstiegspunkt für die Anwendung. Nehmen Sie an, sie steht in einer Datei mit den erforderlichen >>using<<-Zeilen.

Tipp: FlächeSetzen() ist NICHT statisch. Schauen Sie in Kapitel 3 nach, wenn Sie eine Auffrischung zum Schlüsselwort static brauchen.

Hinweis: Jeder Schnipsel aus dem Pool kann mehrfach verwendet werden.



Pool-Puzzle, Lösung



Sehen Sie, dass diese Klasse den Einstiegspunkt enthält, gleichzeitig aber eine Instanz von sich selbst erzeugt? Das ist in C# vollkommen legal.

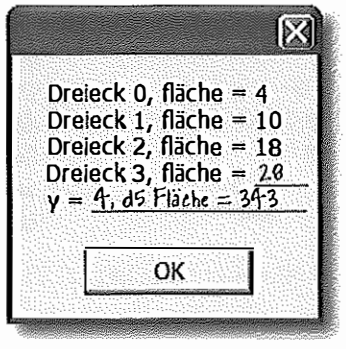
```
class Dreieck
{
    double Fläche;
    int Höhe;
    int Breite;
    public static void Main(String[] args)
    {
        string ergebnisse = "";
        int x = 0;
        Dreieck[] dA = new Dreieck[4];
        while ( x < 4 )
        {
            dA[x] = new Dreieck();
            dA[x].Höhe = (x + 1) * 2;
            dA[x].Breite = x + 4;
            dA[x].FlächeSetzen();
            ergebnisse += "Dreieck " + x + ", Fläche";
            ergebnisse += " = " + dA[x].Fläche + "\n";
            x = x + 1;
        }
        int y = x;
        x = 27;
        Dreieck d5 = dA[2];
        dA[2].Fläche = 343;
        ergebnisse += "y = " + y;
        MessageBox.Show(ergebnisse +
            ", d5 Fläche = " + d5.Fläche);
    }
    void FlächeSetzen()
    {
        Fläche = (Höhe * Breite) / 2;
    }
}
```

Nach dieser Zeile haben wir ein Array mit vier Dreieck-Referenzen – aber es enthält noch keine Objekte.

Die while-Schleife erzeugt vier Instanzen von Dreieck, indem sie vier Mal new verwendet.

Die Methode FlächeSetzen() nutzt die Felder Höhe und Breite, um das Feld Fläche zu setzen. Da es keine statische Methode ist, kann sie nur auf einer Instanz von Dreieck aufgerufen werden.

Bonusantwort





Lösungen zu den Übungen

in C# gibt es ungefähr 77 **reservierte Wörter**. Diese Wörter werden durch den C#-Compiler reserviert und dürfen nicht als Variablennamen verwendet werden. Wenn Sie mit dem Buch durch sind, werden Sie sie alle gut kennen. Hier sind einige, die Sie schon verwendet haben. Schreiben Sie, was Ihrer Meinung nach diese Wörter in C# tun.

namespace

Namensräume sichern, dass die Namen, die Sie in Ihren Programmen verwenden, nicht mit denen im .NET Framework oder anderen in Ihrem Programm verwendeten externen Klassen kollidieren. Alle Klassen und Methoden in einem Programm sind in einem Namensraum.

for

Eine Schleife auf Basis von drei Ausdrücken. Im ersten wird die verwendete Variable deklariert, die dann im zweiten anhand einer Testbedingung geprüft wird. Die dritte Anweisung macht etwas mit dem Wert.

class

In einer Klasse definieren Sie ein Objekt. Klassen haben Eigenschaften und Methoden. Eigenschaften sind das, was eine Klasse weiß, Methoden sind das, was eine Klasse tut.

public

Eine öffentliche Klasse kann von jeder anderen Klasse im Projekt verwendet werden. Öffentliche Methoden und Eigenschaften können außerhalb der Klasse verwendet werden, in der sie deklariert werden.

else

Code, der mit else beginnt, wird ausgeführt, wenn die dazugehörige if-Anweisung fehlschlägt.

new

Wird verwendet, um eine neue Instanz einer Klasse zu erstellen.

using

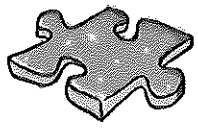
Damit werden alle Namensräume aufgeführt, die in einem Programm verwendet werden. using ermöglicht Ihnen, Code aus dem .NET Framework und anderen vordefinierten Klassen anderer Hersteller sowie aus Klassen zu verwenden, die Sie selbst geschrieben haben.

if

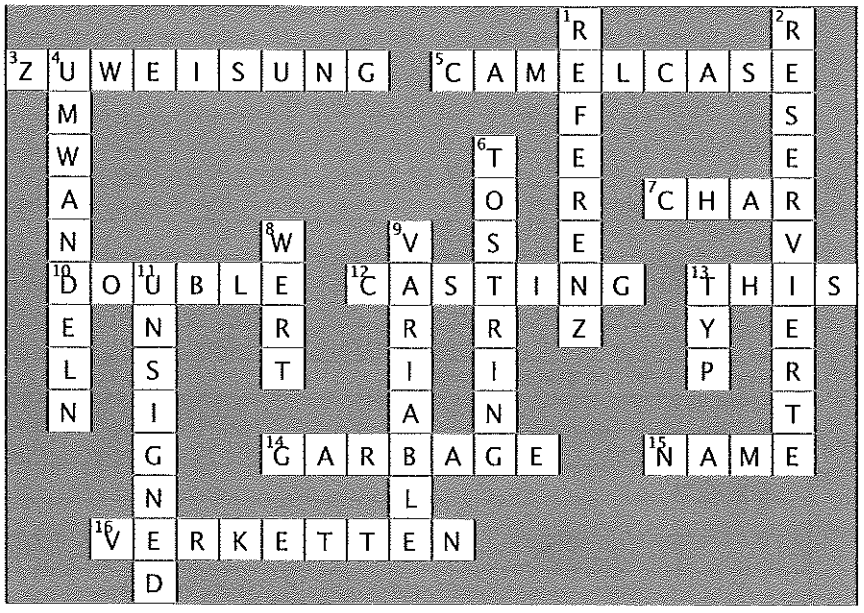
Eine Möglichkeit, in einem Programm eine Bedingungsanweisung einzurichten. Sie sagt, dass eine Gruppe von Anweisungen ausgeführt werden soll, wenn eine Testbedingung wahr ist, und eine andere, wenn sie nicht wahr ist.

while

while-Schleifen sind Schleifen, die ausgeführt werden, solange ihre Testbedingung wahr ist.



Typ-Kreuzworträtsel, Lösung



Name: Datum:

C#-Workshop

Ein Tag beim Rennen

Dieser Workshop gibt Ihnen eine Spezifikation für ein Programm, das Sie erstellen sollen, um die Fertigkeiten zum Einsatz zu bringen, die Sie in den letzten Kapiteln erworben haben.

Dieses Projekt ist umfangreicher als alle, die Ihnen bisher begegnet sind. Lesen Sie sich also alles erst einmal durch, bevor Sie beginnen, und gönnen Sie sich etwas Zeit. Es macht nichts, wenn Sie hängen bleiben – es passiert hier nichts Neues. Sie können also im Buch weiterlesen und später zu diesem Workshop zurückkehren.

Ein paar Entwurfsdetails haben wir bereits für Sie eingesetzt. Und wir haben sichergestellt, dass Sie alle Teile haben, die Sie benötigen ... und sonst nichts.

Ob Sie die Arbeit zu Ende führen, liegt bei Ihnen.

Die Grafiken für diesen Workshop können Sie unter *http://www.headfirstlabs.com/books/hfcsharp/#code* herunterladen. Lösungsvorschläge finden Sie dort unter der Überschrift »Check out the labs«.

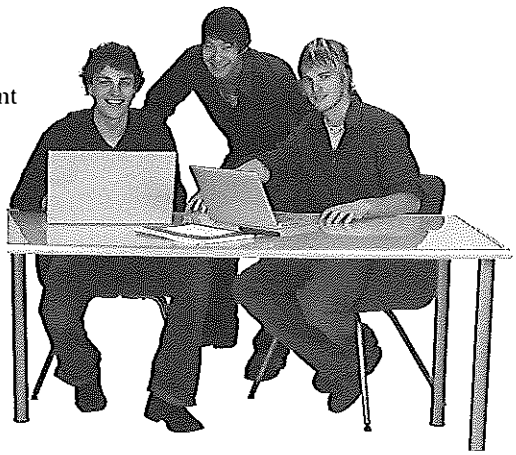
Die Spezifikation: Erstellen Sie eine Rennbahn-Simulation

Tim, Tom und Jan verbringen ihre Zeit mit Vorliebe auf der Rennbahn, sind es allerdings leid, dabei ständig ihr gesamtes Geld zu verlieren. Sie sollen den Dreien einen Simulator aufbauen, der ihnen hilft, den Sieger zu bestimmen, *bevor* sie ihr Geld setzen. Und wenn Sie dabei gute Arbeit leisten, erhalten Sie einen Anteil an ihren Profiten.

Sie werden Folgendes für sie aufbauen ...

Die Jungs

Tim, Tom und Jan wollen auf Hunderennen wetten. Tim beginnt mit 50 Euro, Tom mit 75 Euro und Jan mit 45 Euro. Vor jedem Rennen entscheiden sie, ob sie wetten wollen und wie viel sie setzen möchten. Ihren Einsatz können die Jungs bis unmittelbar vor dem Start eines Rennens ändern ... ist das Rennen aber gestartet, sind alle Einsätze endgültig.



Das Wettbüro

Das Wettbüro hält nach, wie viel Geld die Jungs jeweils haben und was für eine Wette platziert wurde. Der Mindestwetteinsatz beträgt 5 Euro. Das Wettbüro akzeptiert nur eine Wette pro Rennen und pro Person.

Das Wettbüro prüft, ob der Spieler genug Geld hat, um seinen Einsatz zu decken – damit kein Spieler eine Wette platzieren kann, zu deren Deckung nicht genügend Geld zur Verfügung steht.



Willkommen in Krausis Wettbüro

Mindestwetteinsatz: 5 €

Je Rennen eine Wette pro Person

Haben Sie genug Geld dabei?

Wetten

Bei jeder Wette heißt es: »doppelt oder nichts« – entweder verdoppelt der Gewinner seinen Einsatz, oder er verliert ihn. Der Mindestwetteinsatz beträgt 5 €, und jeder kann bis zu 15 € auf einen Hund setzen. Gewinnt der Hund, erhält der Wetter das Doppelte seines Einsatzes (nachdem das Rennen abgeschlossen ist). Verliert er, verschwindet das Geld von seinem Konto.

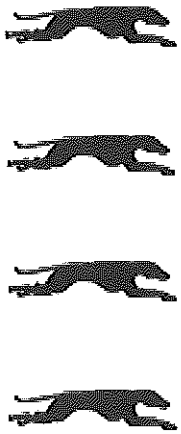
Angenommen, jemand wettet 10 €. Gewinnt sein Hund, wird sein Kontostand um 10 € erhöht (weil er die ursprünglich gesetzten 10 € behält und weitere 10 € für den Sieg dazukommen). Verliert er, vermindert sich sein Kontost nd um 10 €.

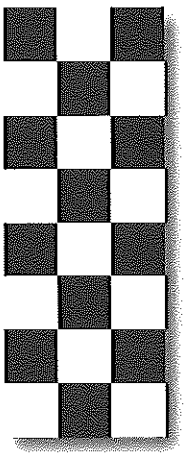
Alle Wetten: »doppelt oder nichts«
Mindestwetteinsatz: 5 €
Bis zu 15 € pro Hund
Sieg: Einsatzverdoppelung
Niederlage: Einsatzverlust

Das Rennen

Vier Hunde laufen auf einer geraden Rennbahn. Sieger ist der Hund, der als erster die Ziellinie überquert. Das Rennen ist vollkommen zufällig, es gibt keine Handicaps oder Wettquoten. Außerdem steigt die Wahrscheinlichkeit des Siegs bei einem Hund nicht, wenn er in der Vergangenheit erfolgreich war.

Möchten Sie ein Handicap-System einbauen, dürfen Sie das natürlich tun! Zum Vergnügen Code zu schreiben ist eine gute Übung.





Klingt gut? Weitere Details folgen gleich ...

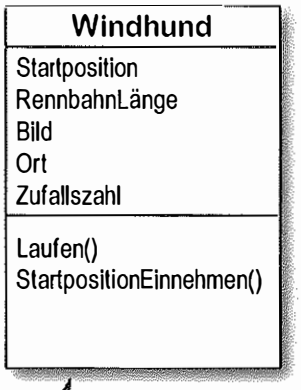
Sie brauchen drei Klassen und ein Formular

Für dieses Projekt erstellen Sie drei Klassen sowie das GUI für den Simulator. Sie sollten ein Array mit drei `Spieler`-Objekten erstellen, das die drei Spieler und ihre Erträge nachhält, und ein Array mit vier `Windhund`-Objekten, die das eigentliche Rennen laufen. Jede Instanz von `Spieler` sollte zusätzlich ein eigenes `Wette`-Objekt haben, das seine Wette festhält und am Ende des Rennens das Geld auszahlt (oder abzieht).

Wir haben für Sie die Klassenbeschreibung und einige Codeschnipsel geschrieben, die Ihnen als Ausgangsbasis dienen sollen. Sie müssen alles fertigstellen.

Am Anfang der Klassen Windhund und Spieler müssen Sie »using System.Windows.Forms« einfügen, am Anfang von Windhund zusätzlich noch »using System.Windows.Drawing;«, da sie Point nutzt.

Wir haben für Sie das Gerüst der Klasse geschrieben, die Sie erstellen müssen. Ihre Aufgabe ist es, die Methoden zu füllen.



Sehen Sie, wie das Klassendiagramm dem Code entspricht?

```
public class Windhund {
  public int Startposition; // wo das Bild beginnt
  public int RennbahnLänge; // die Länge der Rennstrecke
  public PictureBox Bild = null; // das Bild
  public int Ort = 0; // der Ort auf der Rennstrecke
  public Random Zufallszahl; // eine Instanz von Random

  public bool Laufen() {
    // Rückt zufällig 1, 2, 3 oder 4 Kästchen vor und
    // aktualisiert die Position des Bilds im Formular.
    // Liefert true, wenn das Rennen gewonnen wurde.
  }

  public void StartpositionEinnehmen() {
    // Setzt den Ort auf die Startposition zurück.
  }
}
```

Unsere Kommentare sollen Ihnen sagen, was zu tun ist.

Machen Sie sich nicht zu viele Gedanken ... manchmal reicht es, eine einzige Variable zu setzen.

Der Objektinitialisierer für Windhund ist ziemlich einfach. Achten Sie nur darauf, dass Sie eine Referenz auf die richtige PictureBox im Formular an die einzelnen Windhund-Objekte übergeben.

Ihr Objekt kann Dinge in Ihrem Formular steuern ...

Die Klasse `Windhund` hält während des Rennens die Position auf der Strecke nach. Außerdem aktualisiert sie den Ort der `PictureBox`, die den Hund auf der Strecke symbolisiert. Jede Instanz von `Windhund` nutzt ein Feld namens `Bild`, um die `PictureBox` auf dem Formular zu steuern, die das Bild des Hundes enthält. Die Variable `distanz` soll den Weg enthalten, den der Hund vorrückt. Dieser Code aktualisiert dann die Position von `Bild`, indem sie dem X-Wert `distanz` hinzufügt:

Sie müssen sicherstellen, dass Sie den Objektinitialisierern der Windhund-Objekte die richtigen PictureBox-Objekte übergeben.

```
Point p = Bild.Location;
p.X += distanz;
Bild.Location = p;
```

Ruft die aktuelle Position des Bilds ab ...

... fügt der X-Koordinate den Wert hinzu, um den vorgerückt wird ...

... und aktualisiert dann die Bild-Position im Formular.

Spieler
Name MeineWette Geld MeinRadioButton MeinLabel
LabelAktualisieren() WetteAbgeben() WetteLöschen() Einkassieren()

```
public class Spieler {
   public string Name; // der Name des Spielers
   public Wette MeineWette; // eine Instanz von Wette
   public int Geld; // wie viel Geld der Spieler hat


   // die GUI-Steuerelemente für den Spieler auf dem Formular
   public RadioButton MeinRadioButton; // ein Radiobutton
   public Label MeinLabel; // ein Label

   public void LabelAktualisieren() {
     // Setzt das Label auf die Beschreibung der Wette und das
     // für den Radiobutton auf den Kontostand("Tim hat 43 €").
   }

   public void WetteLöschen() { } // die Wette zurücksetzen

   public bool WetteAbgeben(int betrag, int hund) {
     // Setzt eine Wette und speichert sie in MeineWette.
     // Liefert true, wenn der Spieler genug Geld hat.
   }

   public void Einkassieren(int sieger) { } // auszahlen lassen
}
```

Fügen Sie hier Ihren Code ein.

Denken Sie daran, dass Wetten durch Instanzen von Wette repräsentiert werden.

Achten Sie bei der Initialisierung der Spieler-Objekte darauf, dass Sie das Feld MeineWette auf `null` setzen und die Methode LabelAktualisieren() aufrufen, sobald die Initialisierung abgeschlossen ist.

Über diese Klasse werden die Spielerwetten repräsentiert.

Der Schlüssel ist hier der Rückgriff auf das Wette-Objekt – lassen Sie es die Arbeit verrichten

Der Objektinitialisierer für Wette setzt einfach Betrag, Hund und Wetter.

Wette
Betrag Hund Wetter
BeschreibungAbrufen Auszahlen

```
public class Wette {
   public int Betrag; // der gesetzte Geldbetrag
   public int Hund; // die Nummer des gewetteten Hundes
   public Spieler Wetter; // der wettende Spieler

   public string BeschreibungAbrufen() {
     // Liefert einen String, der den Wetter, den gesetzten Betrag
     // und den gewetteten Hund angibt("Tim wettet 8 € auf
     // Hund 4"). Ist der Betrag null, wurde nicht gewettet
     // ("Tim hat nicht gewettet").
   }

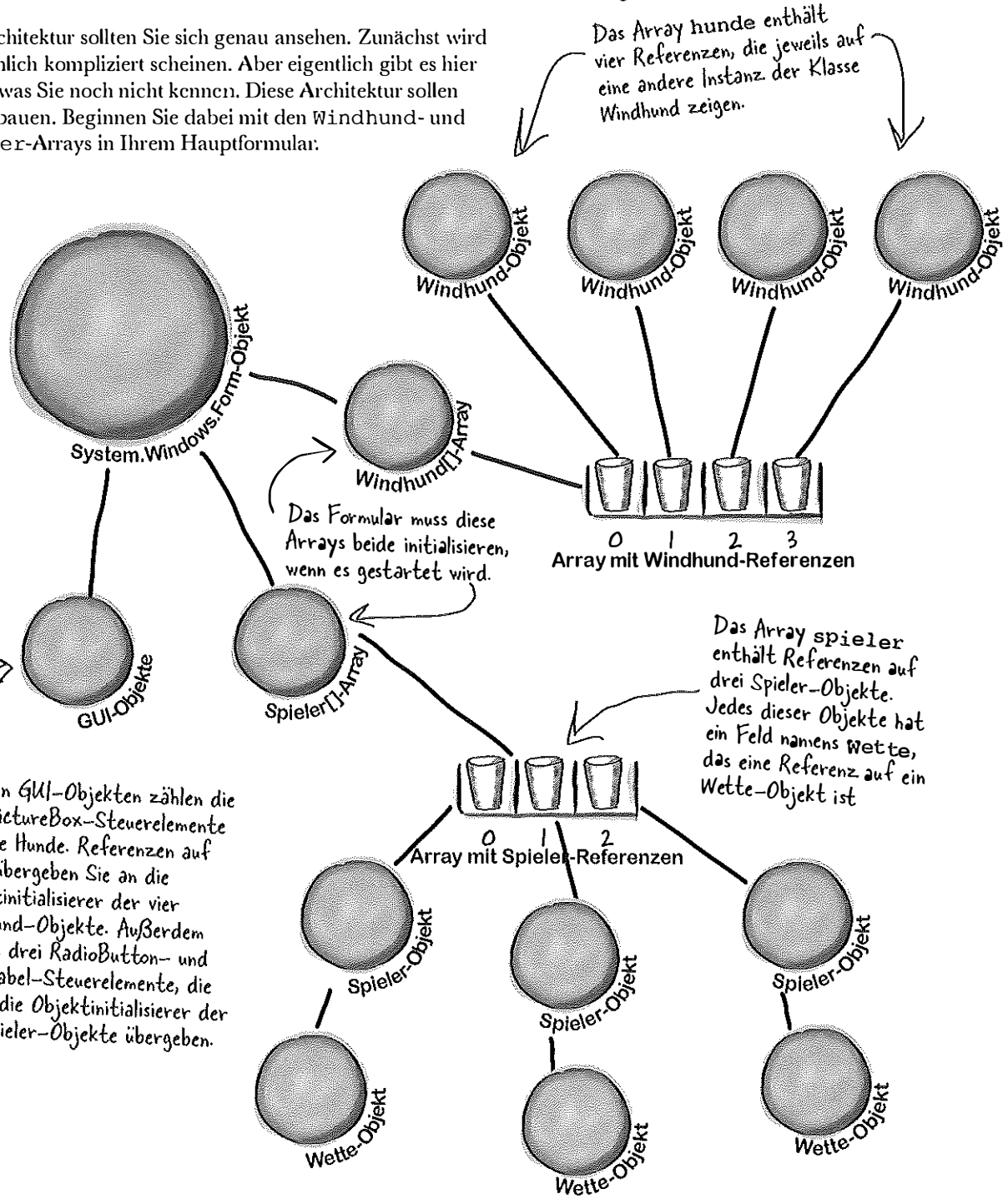
   public int Auszahlen(int sieger) {
     // Der Parameter ist der Sieger des Rennens. Gewann der Hund,
     // wird der gewettete Betrag zurückgeliefert, andernfalls
     // die Negation des gewetteten Betrags.
   }
}
```

Das ist übliche Programmierpraxis: einen String oder eine Meldung aus mehreren Datenteilen zusammenbauen.

Tipp: Wette instantiieren Sie im Spieler-Code. Spieler nutzt das Schlüsselwort **this**, um eine Referenz auf sich selbst an den Initialisierer von Wette zu übergeben.

Hier ist die Architektur Ihrer Anwendung

Die Architektur sollten Sie sich genau ansehen. Zunächst wird sie ziemlich kompliziert scheinen. Aber eigentlich gibt es hier nichts, was Sie noch nicht kennen. Diese Architektur sollen Sie aufbauen. Beginnen Sie dabei mit den `Windhund`- und `Spieler`-Arrays in Ihrem Hauptformular.

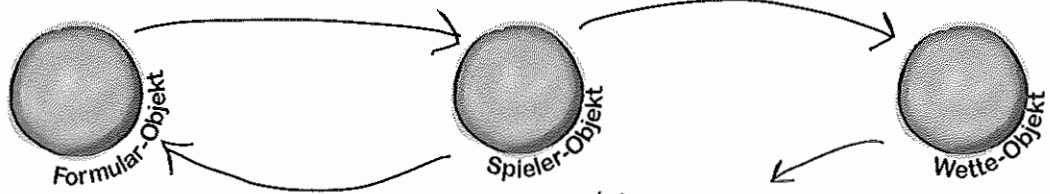


Gibt ein Spieler eine Wette ab, erzeugt er ein neues Wette-Objekt.

Erst lässt das Formular Spieler 2 eine Wette von 7 € auf Hund 3 abgeben …

… Spieler 2 erzeugt also eine neue Instanz von Wette und nutzt dabei das Schlüsselwort `this`, um dem Wette-Objekt den Spieler mitzuteilen …

```
Spieler[2].WetteAbgeben(7, 3)  MeineWette = new Wette() {Betrag = 7, Hund = 3, Wetter = this}
```



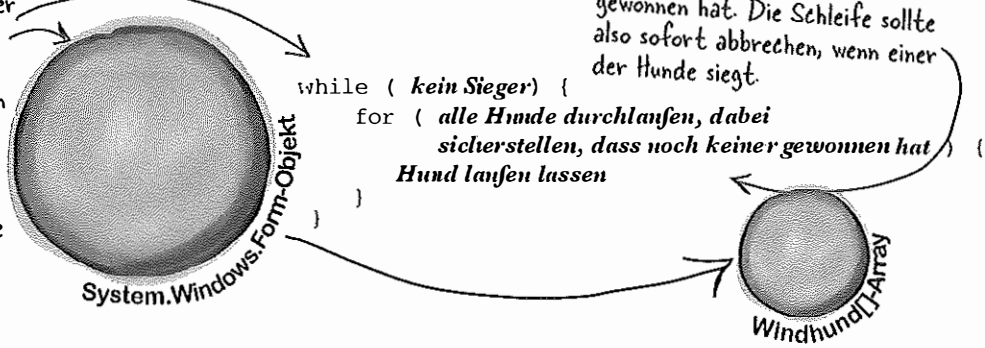
… und da der Spieler genug Geld hat, um die Wette abzugeben, liefert WetteAbgeben() true.

Das Formular sagt den Hunden, dass sie laufen sollen, bis es einen Sieger gibt.

Sagt der Benutzer dem Formular, dass das Rennen gestartet werden soll, startet das Formular eine Schleife, die die über die Strecke laufenden Hunde animiert.

Die Laufen()-Methode der Hunde prüft, ob der Hund das Rennen gewonnen hat. Die Schleife sollte also sofort abbrechen, wenn einer der Hunde siegt.

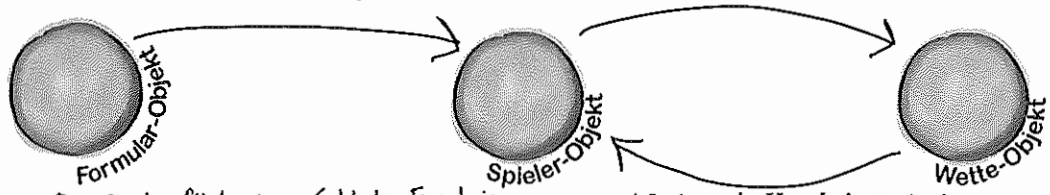
```
while ( kein Sieger) {
    for ( alle Hunde durchlaufen, dabei
          sicherstellen, dass noch keiner gewonnen hat ) {
        Hund laufen lassen
    }
}
```



Das Wette-Objekt prüft, ob es auszahlen muss.

Das Wettbüro im Formular sagt den Spielern, welcher Hund gewonnen hat, damit diese eventuelle Gewinne auf seine Wette einstreichen können.

```
Spieler[2].Einkassieren(siegerHund)        MeineWette.Auszahlen(siegerHund)
```



Der Spieler fügt seinem Geld das Ergebnis von Wette.Auszahlen() hinzu. Gewann der Hund, muss diese Methode also Betrag liefern, andernfalls -Betrag.

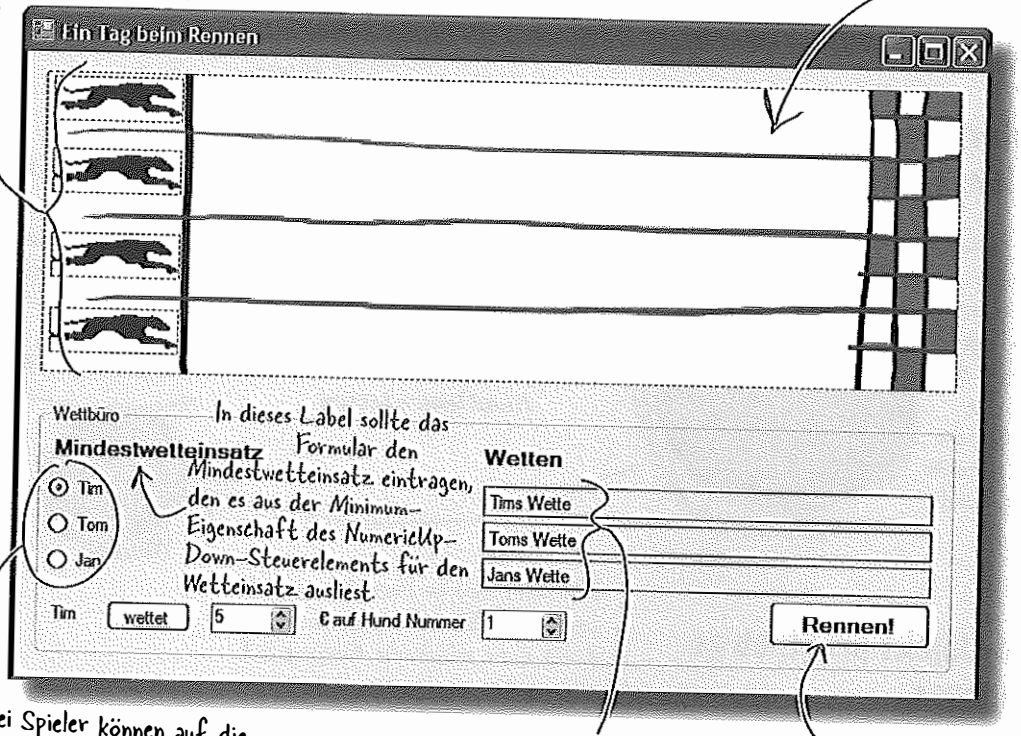
```
if ( mein Hund siegte ) {
    return Betrag;
} else {
    return -Betrag;
}
```

So sollte Ihr GUI aussehen

Das GUI für die Anwendung »Ein Tag beim Rennen« besteht aus einem zweigeteilten Formular. Die obere Hälfte nimmt die Rennstrecke ein: ein `PictureBox`-Steuerelement für die Rennstrecke und vier weitere für die Hunde. Die untere Hälfte des Formulars enthält das Wettbüro, in dem drei Spieler (Tim, Tom und Jan) auf den Ausgang des Rennens wetten können.

Jeder der vier Hunde hat ein eigenes PictureBox-Steuerelement. Bei der Initialisierung der vier Windhund-Objekte erhält das jeweilige Bild-Feld eine Referenz auf eins dieser Objekte. Diese Referenz übergeben Sie (mit der Länge der Rennstrecke und der Startposition) an den Objektinitialisierer für den Windhund.

Die Length-Eigenschaft des PictureBox-Steuerelements für die Rennstrecke bestimmt die Länge der Rennstrecke in den Windhund-Objekten, die sie nutzen, um zu prüfen, ob sie das Rennen gewonnen haben.



Alle drei Spieler können auf die Rennen wetten. Aber es gibt nur einen Wettschalter. Sie müssen ihre Wetten also nacheinander abgeben. Diese Radiobuttons wählen aus, wer aktuell am Wettschalter steht.

Gibt ein Spieler eine Wette ab, überschreibt er eventuell zuvor abgegebene Wetten. Die aktuelle Wette wird in diesen Label-Steuerelementen angezeigt. Für die Labels wurde AutoSize auf False und BorderStyle auf FixedSingle gesetzt.

Sind alle Wetten abgegeben, startet ein Klick auf diesen Button das Rennen.

Die Grafiken können Sie unter http://examples.oreilly.de/german_examples/hfcsharpger/ herunterladen.

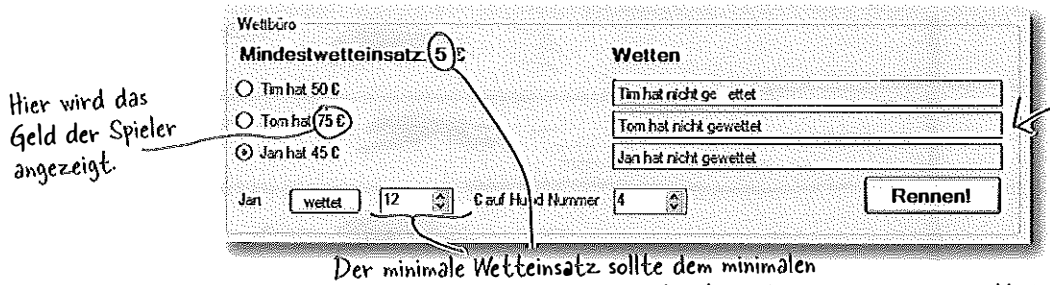
Wetten abgeben

Nutzen Sie die Steuerelemente in der GroupBox Wettbüro, um für die Spieler zu wetten. Es gibt drei separate Phasen:

Gibt ein Spieler eine Wette ab, aktualisiert ein Spieler dieses Label über die Referenz MeinLabel. Über die Referenz MeinRadioButton aktualisiert er ebenfalls seinen Geldbetrag.

1 Es wurden noch keine Wetten abgegeben.

Beim Programmstart oder nach Beendigung eines Rennens wurden im Wettbüro noch keine Wetten abgegeben. Neben dem Namen der einzelnen Spieler sehen Sie, wie viel Geld sie jeweils haben.



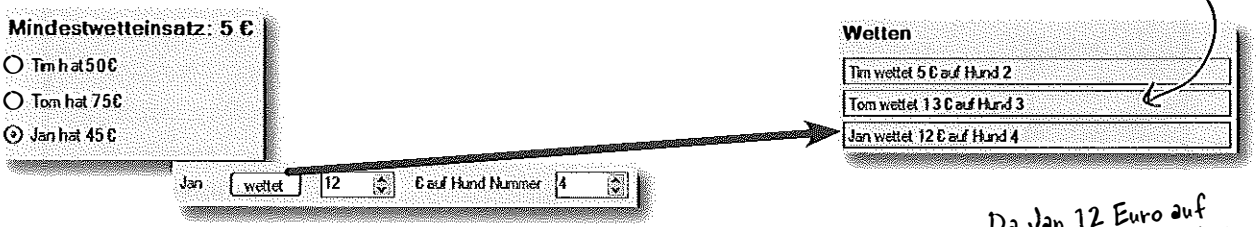
Hier wird das Geld der Spieler angezeigt.

Der minimale Wetteinsatz sollte dem minimalen Betrag im Wette-Steuerelement entsprechen.

Nachdem Tom gewettet hat, aktualisiert sein Spieler-Objekt das Label und den Radiobutton.

2 Jeder Spieler gibt seine Wette ab.

Um eine Wette abzugeben, wählen Sie den Radiobutton des Spielers an, geben dann einen Betrag und einen Hund an und klicken auf den Button wettet. Die `WetteAbgeben()`-Methode des Spielers aktualisiert Radiobutton und Label.



Da Jan 12 Euro auf den Siegerhund gesetzt hat, erhöht sich sein Kontostand um 12 €. Die beiden anderen Spieler verlieren ihre Einsätze.

3 Nach dem Rennen wird abgerechnet.

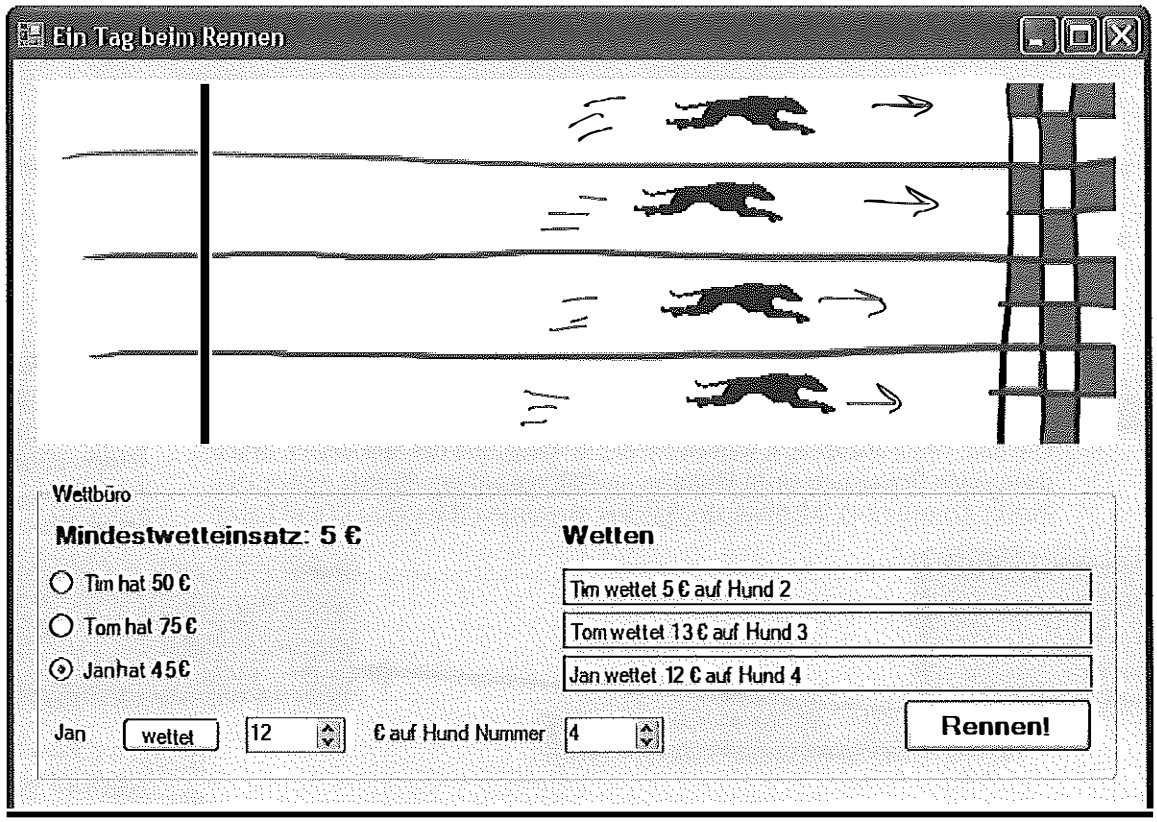
Ist das Rennen beendet und gibt es einen Sieger, rufen alle `Spieler`-Objekte ihre `Einkassieren()`-Methode auf und fügen ihrem Konto ihren Wettgewinn hinzu oder ziehen von ihm ihren Wetteinsatz ab.



Das fertige Produkt

Die Anwendung »Ein Tag beim Rennen« ist fertig, wenn die Spieler ihre Wetten abgeben und das Hunderennen beobachten können.

Während des Rennens laufen die vier Hunde über die Rennstrecke, bis einer von ihnen das Rennen gewinnt.



Während des Rennens können keine Wetten abgegeben werden ... und achten Sie auch darauf, dass kein neues Rennen gestartet werden kann, solange die Hunde laufen!

Die fertige Programmdatei und die Grafiken für die vier Hunde und die Rennstrecke können Sie unter http://examples.oreilly.de/german_examples/hfcsharpger herunterladen.

Aber den Quellcode finden Sie dort nicht! Im wahren Leben kriegen Sie auch keine Lösung für Programmierprobleme. Hier haben Sie die Möglichkeit, Ihre C#-Fertigkeiten richtig zu testen und zu prüfen, wie viel Sie gelernt haben.

5 Kapselung

Alles zu zeigen ist nicht immer richtig

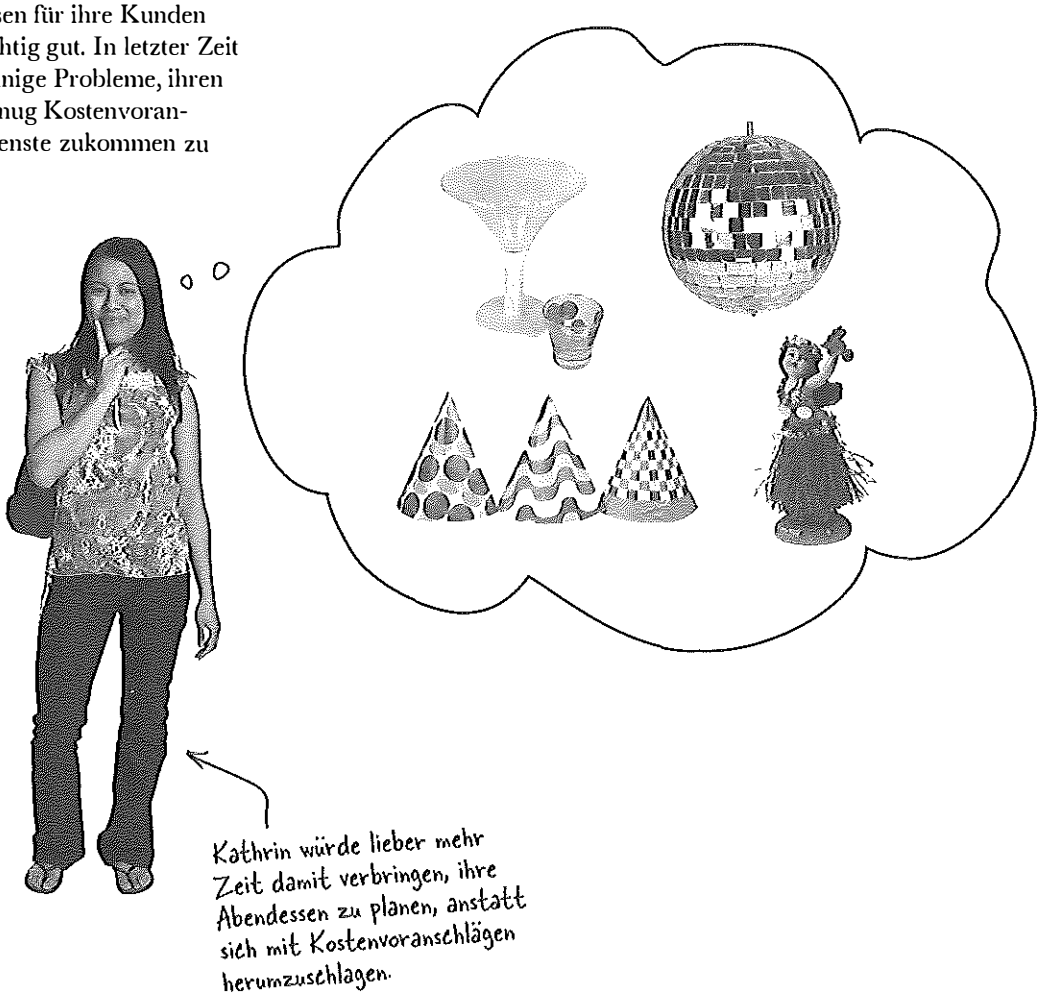


Denken Sie auch, dass man manches nicht gleich allen zeigen muss?

Manchmal geht es Ihren Objekten ebenfalls so. Genau so, wie Sie nicht jeden Ihr Tagebuch oder Ihre Kontoauszüge lesen lassen, lassen gute Objekte ***andere*** Objekte nicht in ihre Felder blicken. In diesem Kapitel werden Sie die Macht der **Kapselung** kennenlernen. Sie **werden die Daten Ihrer Objekte privat machen** und Methoden hinzufügen, **die schützen, wie auf diese Daten zugegriffen wird.**

Kathrin ist Event-Managerin

Sie plant Abendessen für ihre Kunden und ist darin so richtig gut. In letzter Zeit hat sie allerdings einige Probleme, ihren Kunden schnell genug Kostenvoranschläge für ihre Dienste zukommen zu lassen.



Meldet sich ein neuer Kunde bei Kathrin, weil er ein Abendessen ausrichten lassen will, muss sie in Erfahrung bringen, wie viele Gäste kommen, was es zu trinken gibt und wie die Dekoration beschaffen sein soll. Dann nutzt sie eine recht komplizierte Berechnung, die auf einem Flussdiagramm basiert, das sie seit Jahren einsetzt, um die absoluten Kosten zu ermitteln. Das Blöde ist, dass sie ziemlich lang braucht, um ihre Diagramm durchzuarbeiten und ihr potenziellen Kunden die Wartezeit nutzen, um auch bei anderen Event-Managern anzufragen.

Sie haben die Aufgabe, für sie einen C#-basierten Kostenrechner zu erstellen und so ihr Geschäft zu retten. Denken Sie einfach an das Abendessen, das sie für Sie schmeißen wird, wenn Ihnen das gelingt.

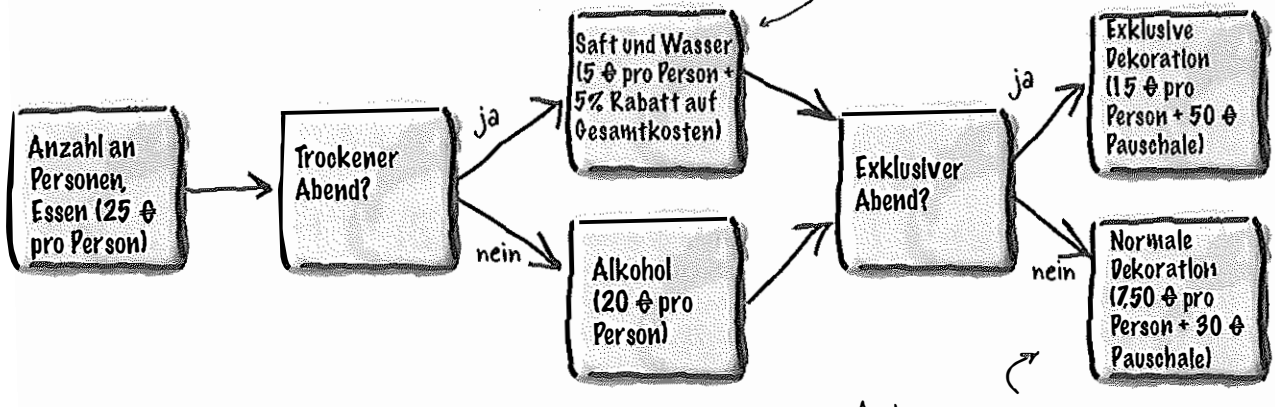
Was der Kostenrechner macht

Kathrin zählt einige der Grundlagen des Systems auf, das sie nutzt, um die Kosten für ein Event zu ermitteln. Hier ist ein Auszug aus dem, was sie aufgeschrieben hat:

Kathrins Party-Planer – Kostenvoranschlag für ein Abendessen

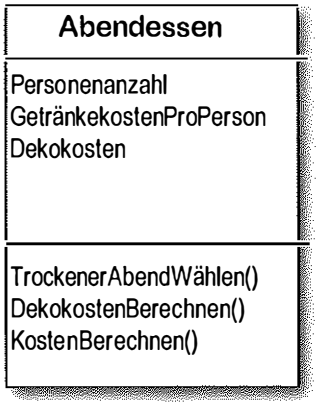
- Pro Person auf der Gästeliste werden 25 € für das Essen veranschlagt.
- Bei Getränken haben die Kunden die Wahl. Zu den meisten Abendessen gibt es alkoholische Getränke. Das kostet 20 € pro Person. Sie bietet aber auch Abende ohne Alkohol. Kathrin bezeichnet das als »Trockener Abend«. Wird statt Alkohol Wasser und Saft serviert, kostet das nur 5 € pro Person. Da der Trockene Abend ihr weniger Arbeit bereitet, gibt sie den Kunden zusätzlich 5% Rabatt auf das gesamte Abendessen.
- Es gibt zwei Dekorationsoptionen. Beschränkt sich der Kunde auf normale Dekoration, kostet das 7,50 € pro Person plus eine Dekorationspauschale von 30 €. Stattdessen kann der Kunde auch die umfangreicheren Dekorationen des »Exklusiven Abends« wählen – das kostet 15 € pro Person zuzüglich einer einmaligen Dekorationspauschale von 50 €.

Hier ist ein anderer Blick auf die gleiche Kostenaufstellung, die jetzt aber zu einem kleinen Flussdiagramm umgestellt wurde, damit Sie besser sehen, wie es funktioniert:





Erstellen Sie ein Programm, das Kathrins Kostenvoranschlagsproblem löst.



Hier ist das Klassendiagramm für die Abendessen-Klasse, die Sie erstellen müssen.

1. Erzeugen Sie ein neues Windows-Forms-Anwendung-Projekt. Fügen Sie ihm eine Klassendatei namens `Abendessen.cs` hinzu und erstellen Sie die Klasse Abendessen auf Basis des Klassendiagramms. Es enthält drei Methoden: `DekokostenBerechnen()`, `TrockenerAbendWählen()` und `KostenBerechnen()`. Nutzen Sie `decimal` für die beiden Kostenfelder und `int` für die Personenanzahl. Denken Sie daran, **jedem Literal**, das Sie einer decimal-Variablen zuweisen, **ein M anzuhängen** (`10.0M`).

2. Da das Programm die Essenskosten nicht ändert, können wir diese als ***Konstante*** deklarieren. Konstanten ähneln Variablen, ihr Wert kann aber nicht geändert werden. Das ist die Deklaration, die Sie brauchen:

```
public const int EssenskostenProPerson = 25;
```

3. Blättern Sie eine Seite zurück, um zu prüfen, ob Sie die Logik für die Methoden verstanden haben. Nur eine liefert einen Wert (einen `decimal`) – die beiden anderen sind `void`. Die Methode `DekokostenBerechnen()` ermittelt die Dekorationskosten für die Anzahl der geladenen Personen. Nutzen Sie die Methode `KostenBerechnen()`, um die Gesamtkosten zu berechnen, indem Sie die Dekorationskosten, die Getränkekosten und die Essenskosten pro Person zusammenrechnen. Ist ein Trockener Abend gewünscht, müssen Sie in `KostenBerechnen()` nach Ermittlung der Gesamtkosten noch den Rabatt abziehen.

4. Fügen Sie Ihrem Formular diesen Code hinzu:

Das Feld abendessen deklarieren Sie im Formular. Dann fügen Sie unter InitializeComponent() die folgenden vier Zeilen ein.

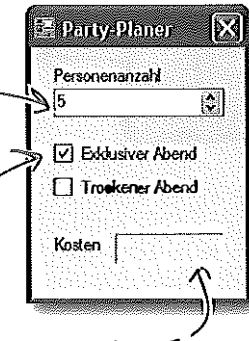
```
Abendessen abendessen;
public Form1() {
  InitializeComponent();
  abendessen = new Abendessen() { Personenanzahl = 5 };
  abendessen.TrockenerAbendWählen(false);
  abendessen.DekokostenBerechnen(true);
  AbendessenKostenAnzeigen();
}
```

Die Methode TrockenerAbendWählen() nutzt einen bool-Parameter (trockenerAbend), um das Feld GetränkekostenProPerson zu aktualisieren, wenn der Kunde einen trockenen Abend wünscht.

5. So sollte das Formular aussehen. Nutzen Sie die Eigenschaften des NumericUpDown-Steuerelements, um die maximale Personenanzahl auf 20, die minimale auf 1 und die standardmäßige auf 5 zu setzen. Entfernen Sie auch die Minimieren- und Maximieren-Buttons.

Setzen Sie Value auf 5, Minimum auf 1 und Maximum auf 5.

Bei Exklusiver Abend sollte Checked auf true gesetzt sein.



Das ist nur ein Label, dessen Text-Eigenschaft auf "", BorderStyle-Eigenschaft auf Fixed3D und AutoSize-Eigenschaft auf false gesetzt ist.

Der Klasse Abendessen müssen Sie >>using System.Windows.Forms;<< nicht hinzufügen, weil sie weder MessageBox.Show() noch irgendetwas anderes aus diesem Namensraum des .NET Framework nutzt.

6 Dieses Formular nutzt keinen Button, sondern aktualisiert das Kosten-Label automatisch, sobald eine Checkbox oder das NumericUpDown-Steuerelement genutzt wird. Zunächst erstellen Sie im Formular eine Methode zur Anzeige der Kosten.

Diese Methode wird von allen anderen Methoden im Formular aufgerufen. Über sie aktualisieren Sie das Kosten-Label mit dem neuen Wert, wenn sich etwas ändert.

Fügen Sie `Form1()` diese Methode hinzu. Sie wird aufgerufen, wenn auf das NumericUpDown-Steuerelement geklickt wird:

Fügen Sie dem Formular diese Methode hinzu – sie berechnet die Essenskosten erneut und steckt sie ins Kosten-Label.

```
private void AbendessenKostenAnzeigen()
{
  decimal kosten = abendessen.KostenBerechnen(checkBox2.Checked);
  kostenLabel.Text = kosten.ToString("c");
}
```

Das ist true, wenn die Checkbox Trockener Abend ausgewählt ist.

Ändern Sie den Namen des Labels zur Anzeige der Kosten in kostenLabel.

Wird ToString >>c<< übergeben, sagt dies, dass der Wert als Währungsangabe formatiert werden soll.

7 Jetzt verknüpfen wir das NumericUpDown-Element mit dem Personenanzahl-Feld der Klasse Abendessen und zeigen im Formular die Kosten an. Klicken Sie doppelt auf das NumericUpDown-Element –, die IDE fügt Ihrem Code dann einen ***Event-Handler*** hinzu. Das ist eine Methode, die jedes Mal ausgeführt wird, wenn sich das Steuerelement ändert. Es setzt die Anzahl der Personen bei der Party. Füllen Sie ihn so:

Bei einem Doppelklick auf einen Button fügt die IDE Code ein, der bei einem Klick auf den Button ausgeführt wird; auch das ist ein Event-Handler.

```
private void numericUpDown1_ValueChanged(
                                    object sender, EventArgs e)
{
    abendessen.Personenanzahl = (int) numericUpDown1.Value;
    AbendessenKostenAnzeigen();
}
```

Sie müssen numericUpDown.Value in einen int umwandeln, weil es eine decimal-Eigenschaft ist.

Oje – der Code birgt Probleme. Fällt Ihnen etwas auf? Machen Sie sich keine Gedanken, wenn nicht. Das schauen wir uns in ein paar Minuten an!

Der Wert, den Sie aus dem Formular an diese Methode senden, ist exklusivBox.Checked. Er wird der Methode in der Klasse als Parameter übergeben.

Das sind beides Zweizeiler. Die erste Zeile ruft die Methode aus der Klasse auf, um die Kosten zu berechnen, und die zweite zeigt die Gesamtkosten im Formular an.

8 Klicken Sie doppelt auf die Checkbox Exklusiver Abend und achten Sie darauf, dass zuerst `DekokostenBerechnen()` und dann `AbendessenKostenAnzeigen()` aufgerufen wird. Klicken Sie dann doppelt auf die Checkbox Trockener Abend und sorgen Sie dafür, dass zuerst die `TrockenerAbendWählen()`-Methode der Klasse Abendessen und dann die Methode `AbendessenKostenAnzeigen()` aufgerufen wird.



Lösungen zu den Übungen

Hier ist der Code für Abendessen.cs.

Indem wir für EssenskostenProPerson eine Konstante verwenden, sichern wir, dass der Wert nicht verändert werden kann. Es macht den Code besser lesbar – es ist klar, dass sich dieser Wert nie ändert.

```
public class Abendessen {
    const int EssenskostenProPerson = 25;
    public int Personenanzahl;
    public decimal GetränkekostenProPerson;
    public decimal Dekokosten = 0;

    public void TrockenerAbendWählen(bool trockenerAbend) {
        if (trockenerAbend) {
            GetränkekostenProPerson = 5.00M;
        } else {
            GetränkekostenProPerson = 20.00M;
        }
    }

    public void DekokostenBerechnen(bool exklusiv) {
        if (exklusiv)
        {
            Dekokosten = (Personenanzahl * 15.00M) + 50M;
        } else {
            Dekokosten = (Personenanzahl * 7.50M) + 30M;
        }
    }
    public decimal KostenBerechnen(bool trockenerAbend) {
        decimal gesamtkosten = Dekokosten +
               ((GetränkekostenProPerson + EssenskostenProPerson)
                   * Personenanzahl);

        if (trockenerAbend) {
            return gesamtkosten * .95M;
        } else {
            return gesamtkosten;
        }
    }
}
```

Bei der Erstellung des Formulars wird der Initialisierer genutzt, um Personenanzahl zu setzen. Dann werden TrockenerAbendWählen() und DekokostenBerechnen() aufgerufen, um die anderen Felder zu berechnen.

Wir haben >>if (exklusiv)<< statt >>if (exklusiv == true)<< verwendet, weil >>exklusiv<< bereits ein bool-Wert ist.

Wir mussten Klammern verwenden, damit die Berechnung korrekt ausgeführt wird.

Das wendet den 5-%-Rabatt auf die gesamten Abendessen-Kosten an, falls ein Trockener Abend gewählt wurde.

Für die Preise mussten wir `decimal` nutzen, weil das der Datentyp ist, der für Geldbeträge gedacht ist. Achten Sie nur darauf, dass Sie nach jedem Literal »M« angeben – wenn Sie 35,26 speichern wollen, müssen Sie also `35.26M` schreiben.

```
public partial class Form1 : Form {
    Abendessen abendessen;
    public Form1() {
        InitializeComponent();
        abendessen = new Abendessen() { Personenanzahl = 5 };
        abendessen.DekokostenBerechnen(exklusivBox.Checked);
        abendessen.TrockenerAbendWählen(trockenBox.Checked);
        AbendessenKostenAnzeigen();
    }

    private void exklusivBox_CheckedChanged(object sender, EventArgs e) {
        abendessen.DekokostenBerechnen(exklusivBox.Checked);
        AbendessenKostenAnzeigen();
    }

    private void trockenBox_CheckedChanged(object sender, EventArgs e) {
        abendessen.TrockenerAbendWählen(trockenBox.Checked);
        AbendessenKostenAnzeigen();
    }

    private void numericUpDown1_ValueChanged(object sender, EventArgs e) {
        abendessen.Personenanzahl = (int)numericUpDown1.Value;
        AbendessenKostenAnzeigen();
    }

    private void AbendessenKostenAnzeigen() {
        decimal kosten = abendessen.KostenBerechnen(trockenBox.Checked);
        kostenLabel.Text = kosten.ToString("c");
    }
}
```

Wir rufen AbendessenKostenAnzeigen auf, um das Label, das die Kosten anzeigt, zu initialisieren, sobald das Formular geladen wird.

Änderungen an den Checkboxen setzen die bool-Parameter trockenerAbend und exklusiv für die Methoden TrockenerAbendWählen() bzw. DekokostenBerechnen() auf true oder false.

Wir haben unseren Checkboxen die Namen »trockenBox« und »exklusivBox« gegeben, damit Sie sehen können, was in den Event-Handler-Methoden passiert.

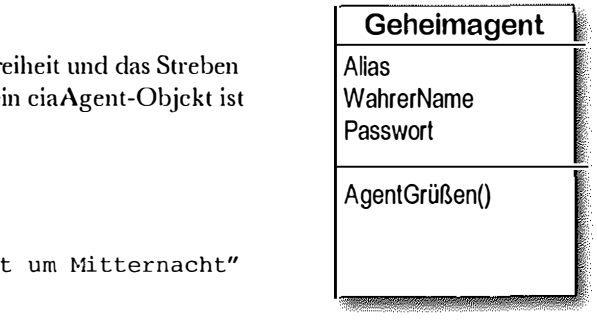
Die Abendessen-Kosten müssen neu berechnet und angezeigt werden, wenn sich die Anzahl oder der Status einer der Checkboxen ändert.

String-Formatierung

Sie haben bereits gesehen, wie Sie eine Variable mit ihrer ToString()-Methode in einen String umwandeln. Übergeben Sie ToString() »c«, liefert es die Ausgabe im lokalen Währungsformat. Sie können auch »f3« angeben, um sie als Dezimalzahl mit drei Nachkommastellen auszugeben, »0« (das ist eine Null), um ihn in eine ganze Zahl umzuwandeln, »0%« für einen Prozentwert und »n«, um eine Zahl mit Tausendertrenner zu erhalten. Nehmen Sie sich einen Augenblick Zeit und schauen Sie, wie das in Ihrem Programm jeweils aussieht.

Nutzen Sie Kapselung, um den Zugriff auf die Methoden und Felder Ihrer Klassen zu steuern

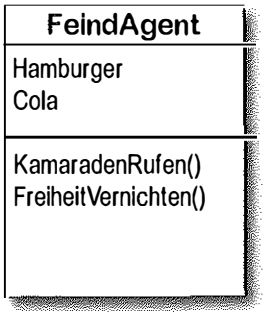
Machen Sie Ihre Felder und Methoden öffentlich, kann jede andere Klasse auf sie zugreifen. Alles, was Ihre Klasse macht und weiß, ist ein offenes Buch für alle anderen Klassen in Ihrem Programm ... und gerade haben Sie gesehen, dass das dazu führen kann, dass sich Ihr Programm auf eine Weise verhält, die Sie nie erwartet hätten. Mit Kapselung können Sie steuern, was in Ihren Klassen Sie teilen und was Sie privat halten möchten. Schauen wir, wie das funktioniert.



1. Der Superspion Ganz Geheim verteidigt Leben, Freiheit und das Streben nach Glück in geheimer Mission in Diktatorien. Sein ciaAgent-Objekt ist eine Instanz der Klasse Geheimagent.

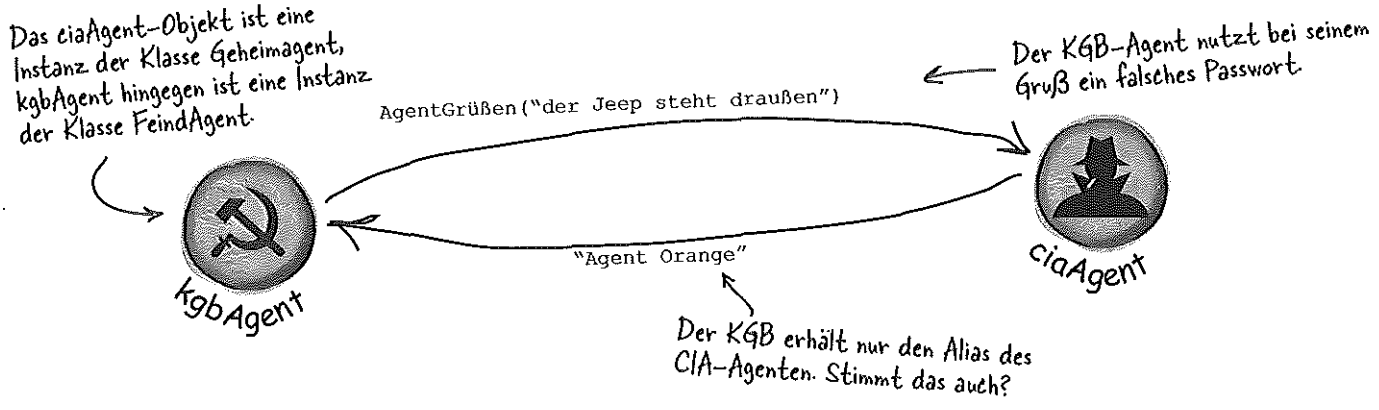



```
WahrerName: "Ganz Geheim"
Alias: "Agent Orange"
Passwort: "die Krähe fliegt um Mitternacht"
```



2. Agent Geheim hat einen Plan, der ihm helfen soll, feindlichen Agenten zu entgehen. Er verwendet eine `AgentGrüßen()`-Methode, die als Parameter ein Passwort erwartet. Erhält sie das falsche Passwort, wird nur sein Alias, Agent Orange, angezeigt.

3. Das scheint ein narrensicherer Weg, die Identität des Agenten zu schützen, oder? Verfügt das aufrufende Agent-Objekt nicht über das richtige Passwort, ist der Name des Agenten sicher.

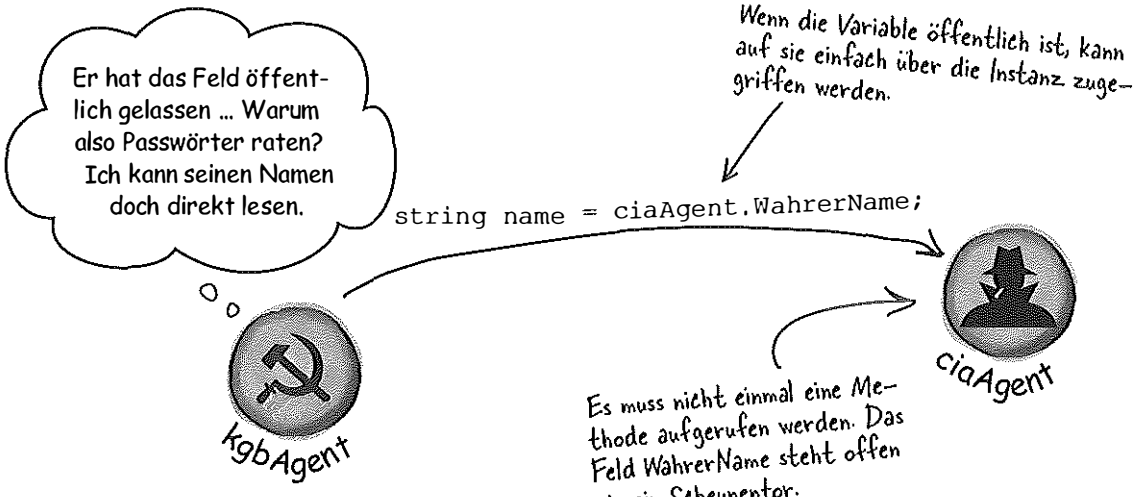


Aber ist das Feld WahrerName WIRKLICH geschützt?

Solange der KGB keine Passwörter von CIA-Agenten kennt, sind die wahren Name der CIA sicher. Oder? Aber was ist mit der Deklaration für das Feld WahrerName:

Machen Sie Ihre Variablen public, d.h. öffentlich, bedeutet das, dass auf sie von außerhalb der Klasse zugriffen werden kann, sogar um sie zu verändern.

```
public string WahrerName;
```



Agent Geheim kann **private** deklarierte Felder nutzen, um seine Identität vor anderen Spion-Objekten verborgen zu halten. Wird das Feld `wahrerName` privat gemacht, kommt man nur noch daran, ***indem man Methoden aufruft, die Zugriff auf die privaten Teile der Klasse haben***. Die Bemühungen des KGB-Agenten werden vereitelt!

Das kgbAgent-Objekt kann nicht auf die privaten Felder von ciaAgent zugreifen, weil es eine Instanz einer anderen Klasse ist.

Ersetzen Sie einfach public durch private, und schon sind Ihrer Felder vor dem Rest der Welt verborgen.

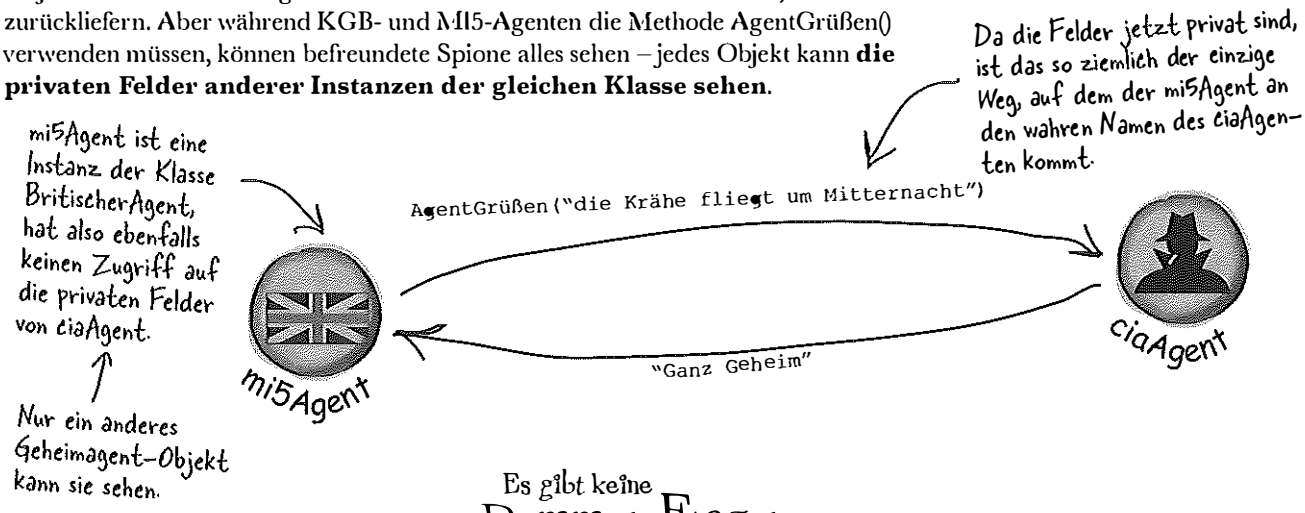
```
private string wahrerName;
```

Halten Sie Ihre Felder und Methoden privat, sichert das, dass kein äußerer Code unerwartete Änderungen an den Werten vornehmen kann, die Sie verwenden.

Sie müssen auch sicherstellen, dass das Feld, das das Passwort speichert, privat ist. Andernfalls kann der Feindagent darauf zugreifen.

Auf private Felder und Methoden kann nur aus der Klasse selbst heraus zugegriffen werden

An Daten, die in privaten Feldern eines Objekts gespeichert sind, kommt ein anderes Objekt nur auf einem Weg heran: über öffentliche Felder und Methoden, die die Daten zurückliefern. Aber während KGB- und MI5-Agenten die Methode AgentGrüßen() verwenden müssen, können befreundete Spione alles sehen – jedes Objekt kann **die privaten Felder anderer Instanzen der gleichen Klasse sehen.**



Es gibt keine Dummen Fragen

F: Auf private Daten muss Ich also über öffentliche Methoden zugreifen. Und was ist, wenn mir eine Klasse mit privaten Feldern keinen Weg bietet, an Daten zu kommen, die mein Objekt benötigt?

A: Dann können Sie auf die Daten von außerhalb des Objekts nicht zugreifen. Beim Schreiben einer Klasse sollten Sie immer darauf achten, dass Sie anderen Objekten Mittel geben, an Daten heranzukommen, die diese benötigen. Private Felder sind ein wichtiger Teil der Kapselung, aber eben nur ein Teil der Geschichte. Gute Kapselung heißt auch, dass man anderen Objekten vernünftige und leicht zu verwendende Wege bietet, an die Daten heranzukommen, die sie benötigen, ohne ihnen dabei einen Zugriff zu gestatten, über den sie Daten kapern könnten, die Ihre Klasse benötigt.

F: Warum sollte man ein Feld so abschotten, dass andere Klassen darauf nicht zugreifen können?

A: Gelegentlich muss eine Klasse Informationen festhalten, die sie für ihre Operationen benötigt, die aber kein anderes Objekt sehen muss. Hier ist ein Beispiel. Wenn Computer Zufallszahlen generieren, nutzen sie besondere Werte, die als *Seed* (Saat) bezeichnet werden. Jede Instanz von `Random` enthält ein Array mit mehreren Dutzend Zahlen. Über diese stellt sie sicher, dass `Next()` immer eine Zufallszahl liefert. Erstellen Sie eine neue Instanz von `Random`, können Sie dieses Array nicht sehen. Das liegt daran, dass Sie es nicht benötigen – aber wenn Sie darauf zugreifen könnten, könnten Sie eventuell Werte hineinstecken, die bewirken, dass die gelieferten Zahlen nicht mehr zufällig sind. Die Seed-Werte wurden also vollständig vor Ihnen gekapselt.

F: Mir ist gerade aufgefallen, dass alle Event-Handler, die Ich genutzt habe, das Schlüsselwort `private` verwenden. Warum sind diese privat?

A: Weil C#-Formulare so eingerichtet sind, dass nur die Steuerelemente auf dem Formular Event-Handler anstoßen können. Geben Sie vor einer Methode das Schlüsselwort `private` an, kann auf diese nur innerhalb der Klasse zugegriffen werden. Fügt die IDE Ihrem Programm eine Event-Handler-Methode hinzu, deklariert sie diese als privat, damit andere Formulare oder Objekte nicht darauf zugreifen können. Aber es gibt keine Regel, die besagt, dass Event-Handler privat sein müssen. Das können Sie sogar selbst ausprobieren – klicken Sie nur mal doppelt auf einen Button und ändern Sie dann die Event-Handler-Deklaration in `public`. Der Code lässt sich weiterhin kompilieren und ausführen.

Spitzen Sie Ihren Bleistift



Hier ist eine Klasse mit ein paar privaten Feldern. Kreisen Sie unten die Anweisungen ein, die sich **nicht kompilieren lassen**, wenn sie von außerhalb der Klasse über **eine Instanz namens einSuperKoch** aufgerufen werden.

```
public class SuperKoch
{
      public string Keksrezept;
      private string geheimzutat;
      private const int bestellmengeFürGuteKunden = 60;
      public int Temperatur;
      private string zutatenlieferant;

      public string RezeptAbrufen (int bestellMenge)
      {
            if (bestellMenge >= bestellmengeFürGuteKunden)
            {
                  return Keksrezept + " " + geheimzutat;
            }
            else
            {
                  return Keksrezept;
            }
      }
}
```

1. `string ofenTemp = einSuperKoch.Temperatur;`

2. `string lieferant = einSuperKoch.zutatenlieferant;`

3. `int bestellmengeFürGuteKunden = 94;`

4. `einSuperKoch.geheimzutat = "Kardamom";`

5. `einSuperKoch.Keksrezept = "3 Eier, 2 1/2 Tassen Mehl, 1 Msp. Salz, 1 Msp. Vanille und 1 1/2 Tassen Zucker vermischen. Zehn Minuten bei 200°C backen. Lecker!";`

6. `string rezept = einSuperKoch.RezeptAbrufen(56);`

7. Wie sieht der Wert von Rezept aus, nachdem alle vorangehenden Anweisungen ausgeführt wurden, die sich kompilieren lassen?

.....

.....

Spitzen Sie Ihren Bleistift
Lösung

Hier ist eine Klasse mit ein paar privaten Feldern. Kreisen Sie unten die Anweisungen ein, die sich **nicht kompilieren lassen**, wenn sie von außerhalb der Klasse über **eine Instanz namens einSuperKoch** aufgerufen werden.

```
public class SuperKoch
{
      public string Keksrezept;
      private string geheimzutat;
      private const int bestellmengeFürGuteKunden = 60;
      public int Temperatur;
      private string zutatenlieferant;

      public string RezeptAbrufen (int bestellMenge)
      {
            if (bestellMenge >= bestellmengeFürGuteKunden)
            {
                  return Keksrezept + " " + geheimzutat;
            }
            else
            {
                  return Keksrezept;
            }
      }
}
```

An die geheimzutat kommt man nur, indem man eine ganze Masse Kekse bestellt. Externer Code kann auf das Feld nicht direkt zugreifen.

```
1. string ofenTemp = einSuperKoch.Temperatur;
```

1 lässt sich nicht kompilieren, weil Sie einem String nicht einfach einen int zuweisen können.

```
2. string lieferant = einSuperKoch.zutatenlieferant;
```

```
3. int bestellmengeFürGuteKunden = 94;
```

```
4. einSuperKoch.geheimzutat = "Kardamom";
```

2 und 4 lassen sich nicht kompilieren, weil zutatenlieferant und geheimzutat privat sind.

```
5. einSuperKoch.Keksrezept = "3 Eier, 2 1/2 Tassen Mehl, 1 Msp. Salz,
   1 Msp. Vanille und 1 1/2 Tassen Zucker vermischen. Zehn Minuten bei
   200°C backen. Lecker!";

6. string rezept = einSuperKoch.RezeptAbrufen(56);
```

7. Wie sieht der Wert von Rezept aus, nachdem alle vorangehenden Anweisungen ausgeführt wurden, die sich kompilieren lassen?

>>3 Eier, 2 1/2 Tassen Mehl, 1 Msp. Salz, 1 Msp. Vanille und 1 1/2 Tassen Zucker vermischen.
Zehn Minuten bei 200°C backen. Lecker!<<

Ein paar Gedanken zur Kapselung von Klassen

★ **Überlegen Sie, welche Felder missbraucht werden können.**
Was kann passieren, wenn sie nicht ordentlich gesetzt werden?

★ **Ist alles in Ihrer Klasse öffentlich?**
Hat Ihre Klasse nur öffentliche Felder und Methoden, müssen Sie wahrscheinlich noch etwas über die Kapselung nachdenken.

★ **Was passiert, wenn das Setzen bestimmter Felder zusätzliche Verarbeitungsschritte oder Berechnungen verlangt?**
Das sind die wichtigsten Kandidaten für eine Kapselung. Schreibt jemand später eine Methode, die den Wert in einem dieser Felder ändert, kann das Probleme bei der Arbeit bewirken, die das Programm erledigen soll.



★ **Machen Sie Felder und Methoden nur dann öffentlich, wenn es erforderlich ist.**
Wenn Sie keinen Grund haben, etwas als public zu deklarieren, tun Sie es auch nicht. Sie könnten sich die Arbeit erschweren, wenn Sie alle Felder in Ihrem Programm öffentlich machen – aber machen Sie sie auch nicht alle privat. Denken Sie zu Anfang ausgiebig darüber nach, welche Felder wirklich öffentlich sein müssen und welche nicht, kann Ihnen das später Zeit sparen.

Kapselung hält Ihre Daten rein

Manchmal ändert sich der Wert in einem Feld, während Ihr Programm seiner Arbeit nachgeht. Teilen Sie dem Programm nicht explizit mit, diesen Wert zurückzusetzen, kann es passieren, dass Sie Ihre Berechnungen mit veralteten Werten durchführen. In dem Fall sollten Sie Ihr Programm bestimmte Anweisungen ausführen lassen, wenn sich ein Feld ändert – beispielsweise Kathrins Programm die Kosten jeweils neu berechnen lassen, wenn sich die Personenanzahl ändert. Das Problem können wir umgehen, indem wir die Daten mit privaten Feldern kapseln. Wir bieten eine Methode an, um den Wert des Felds abzurufen, und eine weitere, um ihn zu setzen und dabei alle notwendigen Berechnungen durchzuführen.

Ein kurzes Kapselungsbeispiel

Eine Bauer-Klasse nutzt ein Feld, um die Anzahl an Kühen zu speichern, und multipliziert diese mit einer Zahl, um zu berechnen, wie viele Säcke Viehfutter erforderlich sind, um die Kühe zu füttern:

```
class Bauer
{
      private int anzahlKühe;
}
```

Dieses Feld machen wir besser privat, damit es keiner ändern kann, ohne auch SäckeFutter zu ändern – sind diese beiden Werte nicht mehr synchron, kann das zu Fehlern führen!

Erstellen Sie ein Formular, in dem der Benutzer die Anzahl an Kühen in ein numerisches Feld eingeben kann, müssen Sie den Wert im Feld `AnzahlKühe` ändern können. Dazu können Sie eine Methode erstellen, die den Wert dieses Felds an das Formular liefert.

```
public int AnzahlKüheAbrufen()
{
      return anzahlKühe;
}
public void AnzahlKüheSetzen(int neueAnzahlKühe)
{
      anzahlKühe = neueAnzahlKühe;
      SäckeFutter = anzahlKühe * FutterMultiplikator;
}
```

Wir fügen eine Methode hinzu, die anderen Klassen eine Möglichkeit gibt, die Anzahl an Kühen abzurufen.

Diese beiden Dinge bewirken dasselbe!

Für die privaten Felder nutzen wir camelCase und für die öffentlichen PascalCase.

Und hier ist eine Methode, mit der die Anzahl von Kühen gesetzt werden kann, die gleichzeitig sicherstellt, dass das Feld SäckeFutter ebenfalls geändert wird. Jetzt bleiben diese beiden Felder immer synchron.

Eigenschaften vereinfachen die Kapselung

C# bietet besondere Methoden, die die Kapselung von Daten zu einem Kinderspiel machen. Sie können **Eigenschaften** verwenden, besondere Methoden, die ausgeführt werden, wenn der Wert eines Felds gesetzt oder abgerufen werden soll. Dieses Feld wird dann als Unterstützungsfeld bezeichnet.

Wir nennen das private Feld in anzahlKühe um (beachten Sie das kleine »a«). Es wird das Unterstützungsfeld für die Eigenschaft AnzahlKühe.

Eigenschaften nutzen Sie, indem Sie sie mit einer gewöhnlichen Felddeklaration kombinieren. Hier ist die Deklaration für AnzahlKühe.

```
private int anzahlKühe;

public const int FeedMultiplier = 30;

public int AnzahlKühe
{
    get
    {
        return anzahlKühe;
    }
    set
    {
        anzahlKühe = value;
        SäckeFutter = anzahlKühe * FutterMultiplikator;
    }
}
```

Das ist ein Getter oder Get-Akzessor. Das ist eine Methode, die jedes Mal ausgeführt wird, wenn die Eigenschaft AnzahlKühe gelesen wird. Ihr Rückgabewert entspricht dem Typ der Variablen – in diesem liefert es den Wert des privaten Felds anzahlKühe.

Das ist ein Setter oder Set-Akzessor, der jedes Mal aufgerufen wird, wenn die Eigenschaft AnzahlKühe gesetzt wird. Selbst wenn es scheint, als habe die Methode einen Parameter, hat sie dennoch einen, der value heißt und den Wert enthält, auf den das Feld gesetzt wurde.

Nutzen können Sie Getter und Setter wie Felder. Hier ist der Code für einen Button, der die Anzahl an Kühen setzt und dann den Wert von SäckeFutter abruft:

```
private void button1_Click(object sender, EventArgs e) {
    Bauer einBauer = new Bauer();
    einBauer.AnzahlKühe = 10;

    int wieVieleSäcke = einBauer.SäckeFutter;

    einBauer.AnzahlKühe = 20;
    wieVieleSäcke = einBauer.SäckeFutter;
}
```

Setzt diese Zeile AnzahlKühe auf 10, setzt der Setter das private Feld anzahlKühe und aktualisiert dann das öffentliche Feld SäckeFutter.

Da der Setter für AnzahlKühe SäckeFutter aktualisiert hat, können Sie diesen Wert jetzt abrufen.

Obwohl der Code AnzahlKühe wie ein Feld behandelt, führt er den Setter aus und übergibt ihm 20. Fragt er das Feld SäckeFutter ab, führt er den Getter aus, der 300 liefert.

Eine Anwendung zum Testen der Klasse Bauer erstellen

Erzeugen Sie eine neue Windows-Forms-Anwendung, die wir nutzen können, um die Klasse Bauer zu testen und uns anzuschauen, wie Eigenschaften funktionieren. Wir werden die Methode **`Console.WriteLine()`** verwenden, um die Ergebnisse in das Ausgabefenster der IDE zu schreiben.



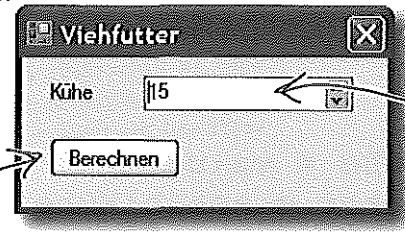
1 Fügen Sie dem Projekt die Klasse Bauer hinzu:

```
public class Bauer {
    public int SäckeFutter;
    public const int FutterMultiplikator = 30;

    private int anzahlKühe;
    public int AnzahlKühe {
        (Fügen Sie hier die Akzessoren von der letzten Seite ein)
    }
}
```

2 Erstellen Sie dieses Formular:

Geben Sie dem Button den Namen Berechnen – er nutzt die öffentlichen Bauer-Daten, um eine Zeile in die Ausgabe zu schreiben.



Setzen Sie für das NumericUpDown-Steuerelement Value auf 15, Minimum auf 5 und Maximum auf 300.

3 Hier ist der Code für das Formular. Er nutzt `Console.WriteLine()`, um seine Ausgabe an das Ausgabefenster zu senden (das Sie öffnen können, indem Sie im Menü Ansicht »Ausgabe« wählen). `WriteLine()` können Sie mehrere Parameter übergeben – der erste ist der zu schreibende String. Schließen Sie »`{0}`« in den String ein, ersetzt `WriteLine()` das mit dem zweiten Parameter. »`{1}`« ersetzt es mit dem dritten, »`{2}`« mit dem vierten usw.

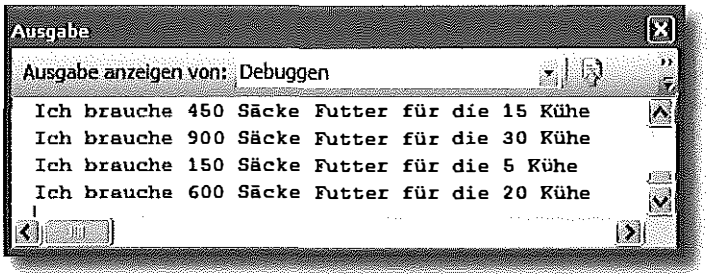
```
public partial class Form1 : Form {
    Bauer bauer;
    public Form1() {
        InitializeComponent();
        bauer = new Bauer() { AnzahlKühe = 15 };
    }
    private void numericUpDown1_ValueChanged(object sender, EventArgs e) {
        bauer.AnzahlKühe = (int)numericUpDown1.Value;
    }
    private void berechnen_Click(object sender, EventArgs e) {
        Console.WriteLine("Ich brauche {0} Säcke Futter für die {1} Kühe",
            bauer.SäckeFutter, bauer.AnzahlKühe);
    }
}
```

Senden Sie mit der Methode Console.WriteLine() eine Textzeile an das Ausgabefenster der IDE.

WriteLine() ersetzt »{0}« mit dem Wert des zweiten Parameters und »{1}« mit dem des dritten Parameters.

Automatische Eigenschaften

Es scheint, als würde der Viehfutterrechner richtig gut funktionieren. Probieren Sie ihn aus –, starten Sie das Programm und klicken Sie auf den Button. Ändern Sie dann die Kuhanzahl auf 30 und klicken Sie erneut auf den Button. Machen Sie das Gleiche für 5 Kühe und dann für 20 Kühe. Hier ist das Ausgabefenster:



Aber es gibt ein Problem. Fügen Sie einen Button hinzu, der diese Anweisung ausführt:

```
bauer.SäckeFutter = 5;
```

Führen Sie das Programm erneut aus. Es funktioniert korrekt, solange Sie nicht den neuen Button betätigen. Klicken Sie auf den neuen Button und dann erneut auf Berechnen. Die Ausgabe sagt jetzt, dass Sie 5 Säcke Futter brauchen – egal wie viele Kühe Sie haben!



Aufgepasst

Automatische Eigenschaften sind ein C#-3.0-Feature.

*Arbeiten Sie immer noch mit Visual Studio 2005 und C# 2.0, funktioniert dieser Code nicht. Wir empfehlen Ihnen, Visual Studio 2008 Express zu verwenden. Sie können es **frei** herunterladen.*

Die Klasse Bauer vollständig kapseln

Das Problem ist, dass Ihre Klasse **nicht vollständig gekapselt ist**. AnzahlKühe haben Sie mit einer Eigenschaft gekapselt, aber SäckeFutter ist immer noch öffentlich. Dieses Problem gibt es oft. Es ist sogar so häufig, dass C# ein automatisiertes Mittel bietet, es zu beheben. Ändern Sie das öffentliche Feld SäckeFutter einfach in eine ***automatische Eigenschaft***. Diese können Sie mit der IDE sehr leicht erstellen:

Der Codeschnipsel prop-Tab-Tab fügt Ihrem Code eine automatische Eigenschaft hinzu.

1. Entfernen Sie aus der Klasse Bauer das Feld SäckeFutter. Gehen Sie mit dem Cursor an den Ort, an dem sich das Feld befand, tippen Sie **`prop`** ein und drücken Sie dann zweimal die Tabulatortaste. Die IDE fügt Ihrem Code dann diese Zeile hinzu:

```
public int MyProperty { get; set; }
```

2. Drücken Sie auf Tab, springt der Cursor zu `MyProperty`. Ändern Sie den Namen in `SäckeFutter`:

```
public int SäckeFutter { get; set; }
```

Jetzt haben Sie eine Eigenschaft statt eines Felds. Das funktioniert genau so, als hätten Sie ein Unterstützungsfeld verwendet (wie das private Feld `AnzahlKühe` hinter der öffentlichen Eigenschaft `AnzahlKühe`).

3. Das behebt das Problem noch nicht. Aber es gibt eine einfache Reparatur – machen Sie die **Eigenschaft schreibgeschützt**:

```
public int SäckeFutter { get; private set; }
```

Versuchen Sie, Ihren Code neu zu kompilieren – Sie erhalten einen Fehler auf der Zeile in dem Button, der `SäckeFutter` setzt. Er sagt Ihnen, dass der **Setter privat ist**. Jetzt ist die Klasse Bauer besser gekapselt!

Was ist, wenn wir den FutterMultiplikator ändern möchten?



Wir haben den Viehfutterrechner so aufgebaut, dass er für den Futtermuliplikator einen `const`-Wert verwendet. Was aber ist, wenn wir die Klasse Bauer in anderen Programmen verwenden möchten, die andere Futtermultiplikatoren benötigen? Sie haben gesehen, dass eine schlechte Kapselung zu Problemen führen kann, wenn ein Feld für andere Klassen zugreifbar ist. Deswegen sollten Sie **nur die Felder und Methoden öffentlich machen, bei denen das notwendig ist**. Da der Viehfutterrechner `FutterMultiplikator` nie aktualisiert, darf anderen Klassen nicht gestattet werden, ihn zu ändern. Machen wir daraus also eine schreibgeschützte automatische Eigenschaft.

1. Entfernen Sie diese Zeile aus Ihrem Programm:

```
public const int FutterMultiplikator = 30;
```

Nutzen Sie `prop-Tab-Tab`, um eine schreibgeschützte automatische Eigenschaft einzufügen:

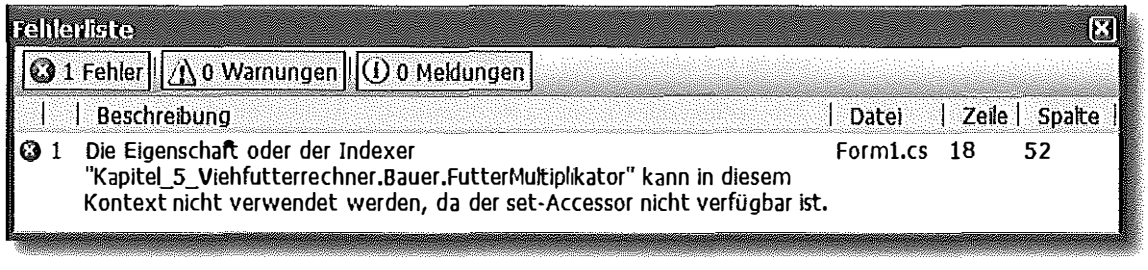
```
public int FutterMultiplikator { get; private set; }
```

2. Nehmen Sie diese Änderungen an Ihrem Code vor und führen Sie ihn dann aus. Oh, oh – irgendetwas stimmt nicht! SäckeFutter **liefert immer 0 Säcke**!

Moment. Eigentlich ist das doch klar: `FutterMultiplikator` wurde nie initialisiert. Es begann mit dem Default-Wert null und ändert sich nicht. Eine Multiplikation mit der Anzahl von Kühen liefert Ihnen immer noch null. Fügen wir also einen Objektinitialisierer hinzu:

```
public Form1() {
    InitializeComponent();
    bauer = new Bauer() { AnzahlKühe = 15, FutterMultiplikator = 30 };
```

Hm ... jetzt lässt sich das Programm nicht mehr kompilieren! Sie sollten folgenden Fehler sehen:



Nur öffentliche Felder und Eigenschaften können mit Objektinitialisierern initialisiert werden. Wie können wir also sicherstellen, dass unser Objekt ordentlich initialisiert wird, wenn einige der zu initialisierenden Felder privat sind?

Initialisieren Sie private Felder mit einem Konstruktor

Müssen Sie ein Objekt initialisieren, bei dem einige der zu initialisierenden Felder privat sind, schafft das ein Objektinitialisierer nicht. Glücklicherweise gibt es eine spezielle Methode, die Sie Ihrer Klasse hinzufügen können. Diese wird als **Konstruktor** bezeichnet. Hat eine Klasse einen Konstruktor, ist dieser das, was **als Allererstes ausgeführt wird,** wenn die Klasse mit dem Schlüsselwort `new` initialisiert wird. Sie können dem Konstruktor Parameter übergeben, um ihm die Werte zu liefern, die initialisiert werden müssen. Aber der Konstruktor **hat keinen Rückgabewert**, weil Sie ihn nicht direkt aufrufen. Sie übergeben ihn als Parameter an die Anweisung `new`. Und Sie wissen bereits, dass `new` das Objekt zurückliefert – ein Konstruktor hat also keine Möglichkeit, einen Wert zurückzuliefern.

Einer Klasse fügen Sie einen Konstruktor hinzu, indem Sie ihr eine Methode hinzufügen, die den gleichen Namen wie die Klasse hat, aber keinen Rückgabewert.

1 Fügen Sie der Klasse Bauer einen Konstruktor hinzu.

Dieser Konstruktor hat nur zwei Zeilen, aber in denen geschieht eine ganze Menge. Gehen wir es schrittweise durch. Wir wissen bereits, dass wir für die Klasse die Anzahl Kühe und einen Futtermultiplikator benötigen. Fügen wir diese also dem Konstruktor als Parameter hinzu. Dann setzen wir die beiden Eigenschaften.

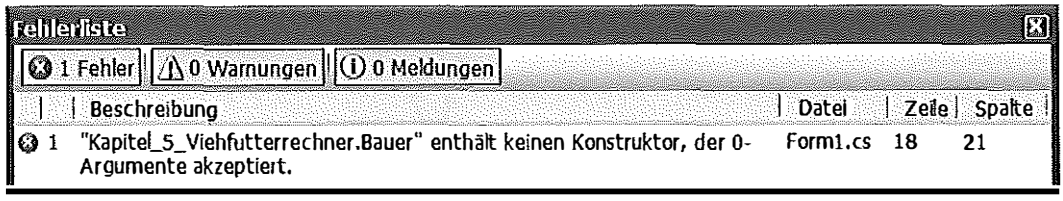
Beachten Sie, dass nach dem »public« kein »void« oder »int« und auch kein anderer Typ folgt, weil Konstruktoren keinen Rückgabewert haben.

```
public Bauer(int anzahlKühe, int futterMultiplikator) {
    FutterMultiplikator = futterMultiplikator;
    this.AnzahlKühe = anzahlKühe;
}
```

Zunächst setzen wir den Futtermultiplikator, weil er gesetzt sein muss, bevor wir den AnzahlKühe-Setter aufrufen.

Setzen wir einfach das private Feld anzahlKühe, wird der Setter für AnzahlKühe nie aufgerufen. Das Setzen von this.AnzahlKühe stellt sicher, dass er aufgerufen wird.

Diesen Fehler erhalten Sie, wenn Ihr Konstruktor-Parameter erwartet, beim Aufruf aber keine angegeben werden.



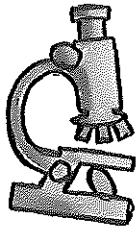
2 Ändern Sie das Formular so, dass es den Konstruktor nutzt.

Jetzt müssen Sie nur noch das Formular so ändern, dass die `new`-Anweisung, die das `Bauer`-Objekt erzeugt, statt des Objektinitialisierers den Konstruktor verwendet.

```
public Form1() {
    InitializeComponent();
    bauer = new Bauer(15, 30);
}
```

Sie wissen bereits, dass das Formular ein Objekt ist. Es hat also ebenfalls einen Konstruktor! Genau das ist diese Methode – beachten Sie, dass sie (genau wie die Klasse) Form1 heißt und keinen Rückgabewert hat.

Hier ruft die new-Anweisung den Konstruktor auf. Das sieht genau wie jede andere new-Anweisung aus nur dass Parameter an die Konstruktormethode übergeben werden. Achten Sie bei der Eingabe auf das IntelliSense-Fenster – das sieht genau so aus, wie bei jeder anderen Methode.



Konstruktoren Unter dem Mikroskop

Werfen wir einen genaueren Blick auf den Bauer-Konstruktor, damit wir verstehen, was hier wirklich passiert.

Weil Konstruktoren nichts zurückliefern, gibt es keinen Rückgabetyp.

Dieser Konstruktor hat zwei Parameter, die genau wie gewöhnliche Parameter funktionieren. Der erste gibt die Anzahl Kühe an, der zweite den Futtermultiplikator.

```
public Bauer(int anzahlKühe, int futtermultiplikator) {
    FutterMultiplikator = futtermultiplikator;
    this.AnzahlKühe = anzahlKühe;
}
```

Der Futtermultiplikator muss zuerst gesetzt werden, weil die zweite Anweisung, die den Setter von AnzahlKühe aufruft, für den FutterMultiplikator einen Wert haben muss, damit SäckeFutter gesetzt werden kann.

In Konstruktoren werden Sie dem Schlüsselwort `this` oft begegnen. Das Feld wird explizit über die Instanz referenziert, da Felder häufig von Parametern gleichen Namens verdeckt werden.

Es gibt keine Dummen Fragen

F: Kann man auch einen Konstruktor ohne Parameter haben?

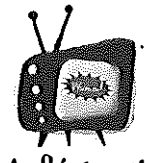
A: Ja. Es ist sogar recht verbreitet, dass Klassen Konstruktoren ohne Parameter haben. Und Sie haben dafür auch schon ein Beispiel gesehen – **den Konstruktor Ihres Formulars**. Werfen Sie einen Blick in ein neu erzeugtes Windows-Formular und suchen Sie seine Konstruktordeklaration:

```
public Form1() {
  InitializeComponent();
}
```

Das ist der Konstruktor für Ihr Form-Objekt. Er erwartet keine Parameter, muss aber eine Menge erledigen. Nehmen Sie sich eine Minute Zeit und öffnen Sie `Form1.Designer.cs`. Suchen Sie die Methode `InitializeComponent()`, indem Sie auf das Pluszeichen neben »Vom Windows Form-Designer generierter Code« klicken.

Diese Methode initialisiert alle Steuerelemente im Formular und setzt alle ihre Eigenschaften. Ziehen Sie ein neues Steuerelement auf Ihr Formular im Form-Designer der IDE und setzen im Eigenschaftenfenster ein paar Eigenschaften, sehen Sie, dass diese Änderungen in der Methode `InitializeComponent()` widergespiegelt werden.

Die Methode `InitializeComponent()` wird im Konstruktor des Formulars aufgerufen, damit alle Steuerelemente initialisiert werden, sobald das Form-Objekt erzeugt wird. (Denken Sie daran, dass jedes Formular, das angezeigt wird, einfach nur ein Objekt ist, das Methoden nutzt, die das .NET Framework im System.Windows.Forms-Namensraum zur Anzeige von Fenstern, Buttons und anderen Steuerelementen anbietet.)



Aufgepasst

Hat ein Methodenparameter den gleichen Namen wie ein Feld, maskiert es das Feld.

Haben Sie bemerkt, dass der Konstruktorparameter `anzahlKühe` genau so aussieht wie das Unterstützungsfeld hinter der Eigenschaft AnzahlKühe? Möchten Sie im Konstruktor das Unterstützungsfeld verwenden, müssten Sie »`this.`« verwenden – `anzahlKühe` verweist auf den Parameter, und über `this.anzahlKühe` greifen Sie auf das private Feld zu.

Es gibt keine Dummen Fragen

F: Wozu sollte man in Gettern oder Settern eine komplizierte Logik brauchen? Sie dienen doch nur dazu, ein Feld zu erstellen.

A: Weil Sie manchmal wissen, dass Sie jedes Mal, wenn ein Feld gesetzt wird, eine Berechnung oder eine Aktion ausführen müssen. Denken Sie an Kathrins Problem – sie bekam Ärger, weil das Formular die Methoden zur Neuberechnung der Dekorationskosten nicht ausführte, nachdem die Personenanzahl in der Klasse Abendessen neu gesetzt wurde. Würden wir das Feld durch einen Setter ersetzen, könnten wir sicherstellen, dass der Setter die Dekorationskosten neu berechnet. (Und genau das werden Sie in ein paar Seiten auch tun!)

F: Moment – was ist denn dann der Unterschied zwischen einer Methode und einem Setter oder Getter?

A: Es gibt keinen! Getter und Setter sind besondere Arten von Methoden – für andere Objekte sehen sie wie Felder aus, und sie werden aufgerufen, wenn das Feld gesetzt oder gelesen wird. Der Wert, den Getter liefern, hat immer den gleichen Typ wie das Feld, und Setter erwarten immer genau einen Parameter namens `value`, der den gleichen Typ hat wie das Feld. Ach, und statt »Getter und Setter« können Sie einfach »Eigenschaft« sagen.

F: Können in einer Eigenschaft BELIEBIGE Arten von Anweisungen stehen?

A: Sicher. Alles, was Sie in einer Methode tun können, können Sie auch in einer Eigenschaft tun. Sie können andere Methoden aufrufen, auf andere Felder zugreifen, sogar Objekte und Instanzen erzeugen. Aber sie werden nur aufgerufen, wenn auf eine Eigenschaft zugegriffen wird. Es ergibt also keinen Sinn, in ihnen Anweisungen anzugeben, die nichts mit dem Abrufen oder Setzen der Eigenschaft zu tun haben.

F: Wenn ein Setter immer einen Parameter namens value erwartet, warum enthält seine Deklaration dann keine Klammer mit »`int value`«, wie man das bei jeder anderen Methode hätte, die einen Parameter namens value erwartet?

A: Weil C# so konzipiert wurde, dass Sie keine Informationen angeben müssen, die der Compiler nicht benötigt. Der Parameter wird deklariert, ohne dass Sie ihn explizit eintippen müssen. Das ist nicht viel, wenn Sie nur ein oder zwei Eigenschaften eintippen – aber wenn es ein paar Hundert werden, kann das wirklich Zeit sparen (vom Fehlerverhindern ganz zu schweigen).

Jeder Setter hat *immer* genau einen Parameter namens `value`, und der Typ dieses Parameters entspricht immer dem Typ der Eigenschaft C# hat alle erforderlichen Informationen zu Typ und Parameter, sobald Sie »`set {`« eintippen. Mehr müssen Sie nicht schreiben, und C# wird Sie nicht dazu zwingen, mehr zu tippen, als unbedingt notwendig ist.

F: Einen Augenblick – muss ich meinem Konstruktor deswegen keinen Rückgabewert hinzufügen?

A: Genau! Ihr Konstruktor muss keinen Rückgabewert haben, weil *jeder* Konstruktor immer `void` ist. Es wäre redundant, würde man Sie zwingen, vor jedem Konstruktor »`void`« anzugeben, deswegen müssen Sie es nicht.

F: Kann ich Get ohne Set oder Set ohne Get haben?

A: Ja! Wenn Sie einen Getter, aber keinen Setter haben, erzeugen Sie eine schreibgeschützte Eigenschaft. Die Klasse Geheimagent könnte eine solche beispielsweise für den Namen verwenden:

```
string name =
       "Agent Orange";
public string Name {
   get { return name; }}
```

Und wenn Sie eine Eigenschaft mit Setter, aber ohne Getter erstellen, kann Ihr Unterstützungsfeld nur geschrieben, aber nicht gelesen werden. Die Klasse Geheimagent könnte das für eine Passwort-Eigenschaft verwenden, die andere Spione schreiben, aber nicht lesen können:

```
public string Passwort {
 set {
  if (value == geheimCode) {
    name = "Ganz Geheim";
  }
}
```

Beide Techniken sind praktisch, wenn Sie mit Kapselung arbeiten.

Eigenschaften (Getter und Setter) sind besondere Methoden, die ausgeführt werden, wenn auf ein Feld zugegriffen wird.



Spitzen Sie Ihren Bleistift

Schauen Sie sich die Getter und Setter hier an. Das Formular, das diese Klasse verwendet, hat eine neue Instanz von DigitalRechnung namens letzterMonat und ruft per Button-Klick die Methode MonatsrechnungAbrufen() auf. Schreiben Sie den Wert auf, den die Variable rechnungsbetrag nach Ausführung der jeweiligen Codeauszüge hat.

```
class DigitalRechnung
{
    private int payPerViewFilme;
    public int AnzahlKanäle;
    public const int Grundgebühr = 45;

    public int PayPerViewFilme
    {
        get
        {
            return payPerViewFilme;
        }
        set
        {
            payPerViewFilme = value;
        }
    }

    public int MonatsrechnungAbrufen()
    {
        int payPerViewBetrag = PayPerViewFilme * 4;
        int kanalBetrag = AnzahlKanäle * 2;
        int gesamtkosten = payPerViewBetrag + kanalBetrag + Grundgebühr;
        return gesamtkosten;
    }
}
```

1.
```
letzterMonat.PayPerViewFilme = 5;
letzterMonat.AnzahlKanäle = 2;
int rechnungsbetrag = letzterMonat.MonatsrechnungAbrufen();
MessageBox.Show ("Sie schulden " + rechnungsbetrag);
```

Was ist der Wert von rechnungsbetrag?

2.
```
letzterMonat.payPerViewFilme = 5;
letzterMonat.AnzahlKanäle = 2;
int rechnungsbetrag = letzterMonat.MonatsrechnungAbrufen();
MessageBox.Show ("Sie schulden " + rechnungsbetrag);
```

Was ist der Wert von rechnungsbetrag?

3.
```
letzterMonat.PayPerViewFilme = 7;
int rechnungsbetrag = letzterMonat.MonatsrechnungAbrufen();
MessageBox.Show ("Sie schulden " + rechnungsbetrag);
```

Was ist der Wert von rechnungsbetrag?

Es gibt keine Dummen Fragen

F: Mir ist aufgefallen, dass Sie für einige Felder Namen mit Großbuchstaben verwendet haben, für andere aber Namen mit Kleinbuchstaben. Ist das wichtig?

A: Ja – es ist wichtig für Sie. Aber für den Compiler ist es egal. C# kümmert sich nicht darum, welche Namen Sie Ihren Variablen geben. Aber wenn Sie seltsame Namen wählen, kann das die Lesbarkeit ihres Codes verschlechtern. Manchmal kann es verwirrend sein, wenn sich Variablennamen nur in der Groß-/Kleinschreibung des ersten Buchstabens unterscheiden.

Hier sind ein paar Tipps zu Variablennamen, die Sie auf Kurs halten. Das sind keine festgeschriebenen Regeln – dem Compiler ist es vollkommen gleich, ob Ihre Variable Groß- oder Kleinbuchstaben verwendet –, aber es sind gute Richtlinien, die helfen, Ihren Code lesbarer zu machen.

1. Wenn Sie ein privates Feld deklarieren, sollten Sie cameiCase verwenden (mit einem Kleinbuchstaben beginnen).

2. Öffentliche Eigenschaften und Methoden soliten PascalCase verwenden (mit einem Großbuchstaben beginnen).

3. Methodenparameter sollten cameiCase verwenden.

4. Einige Methoden, insbesondere Konstruktoren, haben gelegentlich Parameter mit dem gleichen Namen wie Felder. Wenn das passiert, **maskiert** der Parameter das Feld. Das bedeutet, dass Anweisungen in der Methode, die diesen Namen verwenden, den Parameter referenzieren und nicht das Feld. Nutzen Sie das Schlüsselwort `this`, um dieses Problem zu beheben – fügen Sie es der Variablen hinzu, um dem Compiler zu sagen, dass Sie das Feld und nicht den Parameter meinen.

Spitzen Sie Ihren Bleistift

Dieser Code birgt Probleme. Schreiben Sie auf, was Ihrer Meinung nach falsch ist und wie Sie es ändern würden.

```
class KaugummiAutomat {
      private int kaugummis;

      Private int preis;
      public int Preis
      {
        get
        {
            return preis;
        }
      }
      public KaugummiAutomat(int kaugummis, int preis)
      {
            kaugummis = this.kaugummis;
            preis = Preis;
      }
      public string KaugummiAusgeben(int preis, int eingefügteMünzen)
      {
            if (this.eingefügteMünzen >= preis) { // Feld prüfen
                  kaugummis -= 1;
                  return "Hier ist Ihr Kaugummi";
            } else {
                  return "Werfen Sie mehr Münzen ein";
            }
      }
}
```

Spitzen Sie Ihren Bleistift — Lösung

Schreiben Sie den Wert der Variablen rechnungsbetrag nach Ausführung des Codes unten auf.

1.
```
letzterMonat.PayPerViewFilme = 5;
letzterMonat.AnzahlKanäle = 2;
int rechnungsbetrag = letzterMonat.MonatsrechnungAbrufen();
MessageBox.Show ("Sie schulden " + rechnungsbetrag);
```
Was ist der Wert von rechnungsbetrag? **69**

2.
```
letzterMonat.payPerViewFilme = 5;
letzterMonat.AnzahlKanäle = 2;
int rechnungsbetrag = letzterMonat.MonatsrechnungAbrufen();
MessageBox.Show ("Sie schulden " + rechnungsbetrag);
```
Was ist der Wert von rechnungsbetrag? **kompiliert nicht**

3.
```
letzterMonat.PayPerViewFilme = 7;
int rechnungsbetrag = letzterMonat.MonatsrechnungAbrufen();
MessageBox.Show ("Sie schulden " + rechnungsbetrag);
```
Was ist der Wert von rechnungsbetrag? **77**

Spitzen Sie Ihren Bleistift — Lösung

Dieser Code birgt Probleme. Schreiben Sie auf, was Ihrer Meinung nach falsch ist und wie Sie es ändern würden.

Der `preis` in Kleinbuchstaben verweist auf den Konstruktorparameter, nicht auf das Feld. Diese Zeile setzt den PARAMETER auf den Wert, der vom Getter für Preis geliefert wird, aber Preis wurde überhaupt noch nicht gesetzt! Er macht also nichts Vernünftiges. Ändern Sie den Konstruktorparameter in Preis mit großem P, funktioniert die Zeile ordentlich.

Das Schlüsselwort >>this<< steht vor dem falschen >>kaugummis<<. `this.kaugummis` verweist auf die Eigenschaft, `kaugummis` auf den Parameter.

```
public KaugummiAutomat(int kaugummis, int preis)
{
    kaugummis = this.kaugummis;
    preis = Preis;
}
```

Der Parameter maskiert das private Feld preis, und der Kommentar sagt, dass die Anweisung den Wert des Unterstützungsfelds preis prüfen soll.

```
public string KaugummiAusgeben(int preis, int eingefügteMünzen)
{
    if (this.eingefügteMünzen >= preis) { // Feld prüfen
        kaugummis -= 1;
        return "Hier ist Ihr Kaugummi";
    } else {
        return "Werfen Sie mehr Münzen ein";
    }
}
```

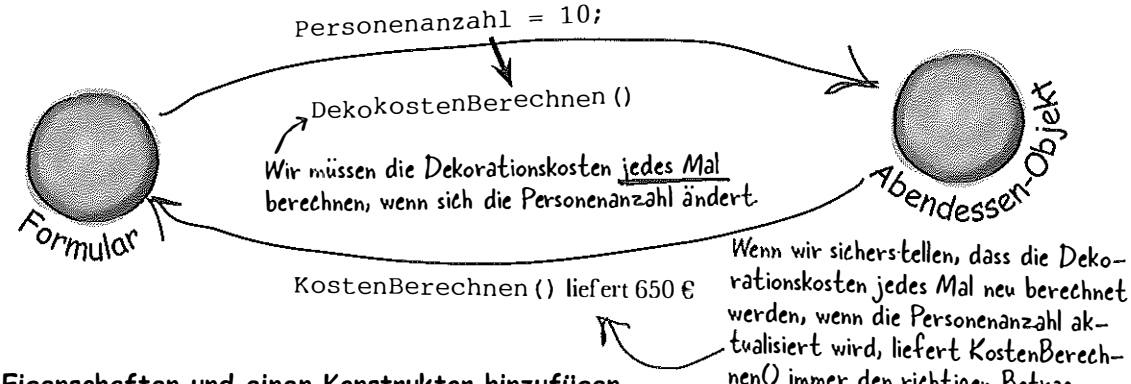
Das Schlüsselwort >>this<< steht vor einem Parameter. Dort gehört es nicht hin. Es sollte vor `preis` stehen, weil dieses Feld von dem Parameter maskiert wird.



Nutzen Sie das, was Sie über Eigenschaften und Konstruktoren gelernt haben, um Kathrins Party-Planer zu reparieren.

1 Wie man den Party-Planer repariert ...

Wenn wir die Abendessen-Klasse reparieren möchten, müssen wir sicherstellen, dass die Methode DekokostenBerechnen() jedes Mal aufgerufen wird, wenn sich die Personenanzahl ändert.



2 Eigenschaften und einen Konstruktor hinzufügen.

Um Kathrins Problem in den Griff zu bekommen, müssen Sie sicherstellen, dass die Klasse Abendessen ordentlich gekapselt ist. Sie beginnen damit, dass Sie **Personenanzahl in eine Eigenschaft umwandeln**, die immer DekokostenBerechnen() aufruft, wenn sie gesetzt wird. Dann **fügen Sie einen Konstruktor hinzu**, der sicherstellt, dass die Instanz ordentlich initialisiert wird. Schließlich **ändern Sie das Formular**, damit es den neuen Konstruktor nutzt. Machen Sie das richtig, müssen Sie am Formular keine weiteren Änderungen vornehmen.

- ★ Erstellen Sie eine neue Eigenschaft für `Personenanzahl`, deren Setter `DekokostenBerechnen()` aufruft. Sie benötigt ein Unterstützungsfeld namens `personenanzahl`.
- ★ Der Setter für `Personenanzahl` muss einen Wert haben, der als Parameter an die Methode `DekokostenBerechnen()` übergeben wird. Fügen Sie ein privates `bool`-Feld namens `exklusiverAbend` hinzu, das jedes Mal gesetzt wird, wenn `DekokostenBerechnen()` aufgerufen wird.
- ★ Fügen Sie einen Konstruktor hinzu, der Instanzen einrichtet. Er braucht die drei Parameter Personenanzahl, TrockenerAbend und ExklusiverAbend. Zur Initialisierung von Abendessen-Objekten ruft das Formular zwei Methoden auf – verschieben Sie diese in den Konstruktor:

```
abendessen.DekokostenBerechnen(exklusivBox.Checked);
abendessen.TrockenerAbendWählen(trockenBox.Checked);
```

- ★ Hier ist der Konstruktor für das Formular – alles andere im Formular bleibt gleich:

```
public Form1() {
    InitializeComponent();
    abendessen = new Abendessen((int)numericUpDown1.Value,
                                trockenBox.Checked, exklusivBox.Checked);
    AbendessenKostenAnzeigen();
}
```



LÖSUNGEN ZU DEN ÜBUNGEN

Nutzen Sie das, was Sie über Eigenschaften und Konstruktoren gelernt haben, um Kathrins Party-Planer zu reparieren.

```
public class Abendessen {
    const int EssenskostenProPerson = 25;

    private int personenanzahl;
    public int Personenanzahl {
        get { return personenanzahl; }
        set {
            personenanzahl = value;
            DekokostenBerechnen(exklusiverAbend);
        }
    }
    private bool exklusiverAbend;

    public decimal GetränkekostenProPerson;
    public decimal Dekokosten = 0;

    public Abendessen(int personenanzahl, bool trockenerAbend, bool exklusiverAbend) {
        Personenanzahl = personenanzahl;
        this.exklusiverAbend= exklusiverAbend;
        TrockenerAbendWählen(trockenerAbend);
        DekokostenBerechnen(exklusiverAbend);
    }

    public void TrockenerAbendWählen(bool trockenerAbend) {
        if (trockenerAbend) {
            GetränkekostenProPerson = 5.00M;
        } else {
            GetränkekostenProPerson = 20.00M;
        }
    }

    public void DekokostenBerechnen(bool exklusiv) {
        exklusiverAbend= exklusiv;
        if (exklusiv) {
            Dekokosten = (Personenanzahl * 15.00M) + 50M;
        } else {
            Dekokosten = (Personenanzahl * 7.50M) + 30M;
        }
    }

    public decimal KostenBerechnen(bool trockenerAbend) {
        decimal gesamtkosten = Dekokosten
           + ((GetränkekostenProPerson + EssenskostenProPerson) * Personenanzahl);

        if (trockenerAbend) {
            return gesamtkosten * .95M;
        } else {
            return gesamtkosten;
        }
    }
}
```

Da personenanzahl jetzt privat ist, hat das Formular keine Möglichkeit, es zu ändern, ohne die Dekorationskosten neu zu berechnen. Das behebt den Fehler, der Kathrin fast einen Ihrer Kunden gekostet hätte.

Indem Sie eine Eigenschaft nutzen, können Sie sicherstellen, dass die Dekorationskosten immer neu berechnet werden, wenn sich die Personenanzahl ändert.

Geben Sie acht, wie Sie >>this<< verwenden. Sie würden es benötigen, um das private Feld >>personenanzahl<< und den gleichnamigen Parameter auseinander zu halten.

Achten Sie darauf, dass Sie die exklusive Dekoration in einem Feld speichern, das der Setter von Personenanzahl verwenden kann.

>>exklusiverAbend<< müssen Sie stets >>this<< voranstellen, weil der Parameter exklusiverAbend das private Feld gleichen Namens verdeckt.

6 Vererbung

Der Stammbaum Ihres Objekts

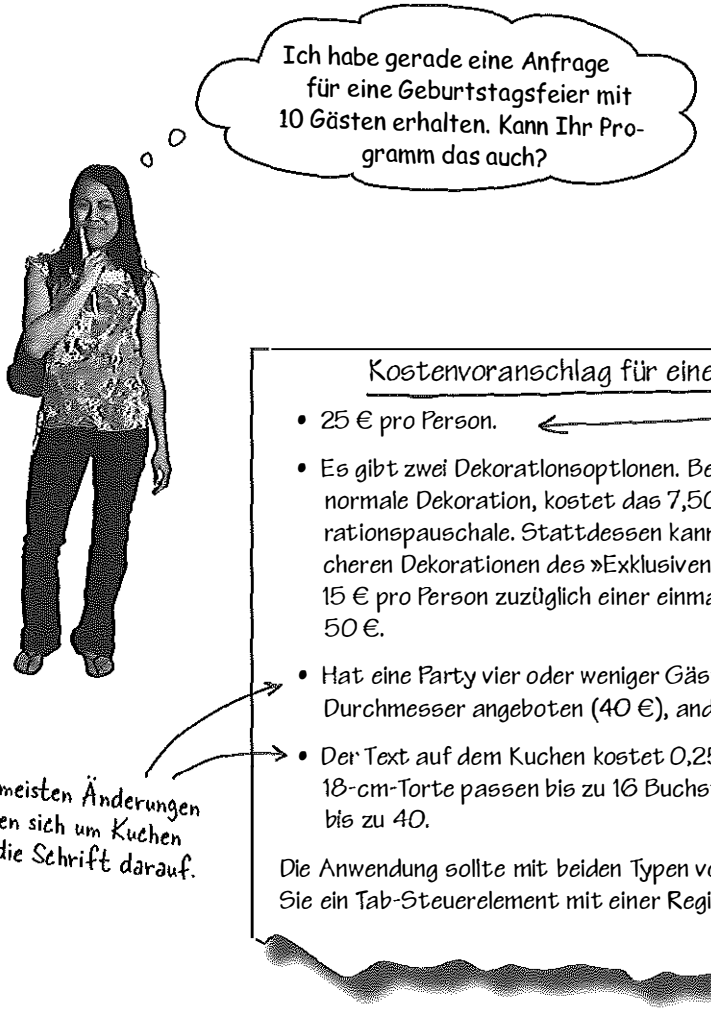


Es gibt Momente, da *WILL man wie die eigenen Eltern sein.*

Ist Ihnen schon einmal ein Objekt über den Weg gelaufen, das ***fast*** genau das macht, was ***Ihr*** Objekt machen soll? Und haben Sie sich gewünscht, Sie ***bräuchten einfach nur ein paar Dinge zu ändern*** und das Objekt wäre perfekt? Genau das ist der Grund dafür, dass **Vererbung** eins der mächtigsten und wichtigsten Konzepte in C# ist. Bevor Sie mit diesem Kapitel durch sind, werden Sie gelernt haben, wie Sie von einer Klasse **ableiten**, um ihr Verhalten zu bekommen, dabei aber **die Flexibilität** bewahren, Änderungen an diesem Verhalten vorzunehmen. Sie **vermeiden doppelten Code, modellieren die wahre Welt** präziser nach und erhalten Code, der **leichter zu warten ist.**

Kathrin plant auch Geburtstagsfeiern

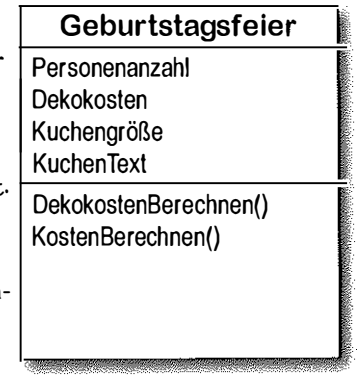
Seitdem Sie Kathrins Programm ans Laufen gebracht haben, arbeitet sie permanent damit. Aber sie nutzt es nicht nur für Abendveranstaltungen. Sie plant auch Geburtstagsfeiern, die allerdings etwas anders veranschlagt werden. Sie müssen also das Programm für sie auf Geburtstage erweitern.



Wir brauchen eine Geburtstagsfeier-Klasse

Das Programm so zu modifizieren, dass die Kosten von Kathrins Geburtstagsfeiern berechnet werden können, bedeutet, dass wir dem Programm eine neue Klasse hinzufügen und das Formular verändern müssen, damit wir beide Arten von Partys verarbeiten können.

All das werden wir in einer Minute tun – aber erst müssen wir einen Überblick darüber gewinnen, was diese Aufgabe alles einschließt.



Das müssen wir tun:

1. **Erzeugen Sie die neue Geburtstagsfeier-Klasse.**
   Unsere neue Klasse muss die Kosten berechnen, sich um die Dekoration kümmern und die Länge der Beschriftung für den Kuchen prüfen.

2. **Fügen Sie Ihrem Formular ein TabControl hinzu.**
   Die Registerkarten auf dem Formular haben Ähnlichkeit mit dem GroupBox-Steuerelement, das Sie verwendet haben, um in Workshop 1 zu zeigen, wer wieviel Geld im Wettbüro gesetzt hat. Klicken Sie einfach auf die Registerkarte, die Sie anzeigen möchten, und ziehen Sie Steuerelemente darauf.

3. **Geben Sie der ersten Registerkarte einen Namen und ziehen Sie die Abendessen-Steuerelemente darauf.**
   Sie ziehen die Steuerelemente, die sich mit Abendessen befassen, auf eine neue Registerkarte. Diese funktionieren wie zuvor, werden aber nur angezeigt, wenn die Abendessen-Registerkarte ausgewählt wird.

4. **Geben Sie der zweiten Registerkarte einen Namen und fügen Sie ihr die neuen Geburtstagsfeier-Steuerelemente hinzu.**
   Die Schnittstelle für die Bearbeitung von Geburtstagsfeiern fügen Sie genau so hinzu wie sonst auch.

5. **Verknüpfen Sie die Klasse Geburtstagsfeier mit den Steuerelementen.**
   Jetzt müssen Sie den Formularfeldern nur noch eine Geburtstagsfeier-Referenz hinzufügen und Ihren Steuerelementen den Code beigeben, der ihn die Methoden und Eigenschaften der Klasse nutzen lässt.

Es gibt keine Dummen Fragen

F: Warum können wir nicht einfach eine neue Instanz von Abendessen erstellen, wie wir es bei Mark gemacht haben, als er in seinem Navi-Programm drei Routen vergleichen wollte?

A: Weil Sie, wenn Sie eine weitere Instanz von Abendessen hinzufügen, nur zusätzliche Abendessen planen können. Zwei Instanzen der gleichen Klasse kümmern sich um zwei unterschiedliche Happen der gleichen Art von Daten. Aber wenn Sie **unterschiedliche Arten von Daten** speichern müssen, brauchen Sie dazu **unterschiedliche Klassen**.

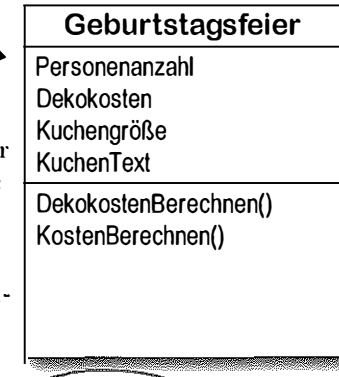
F: Woher weiß Ich, was Ich in die neuen Klassen packen muss?

A: Bevor Sie damit beginnen können, eine neue Klasse aufzubauen, müssen Sie wissen, welche Aufgabe sie lösen soll. Deswegen mussten wir mit Kathrin reden – sie wird mit dem Programm arbeiten. Gut, dass Sie sich viele Notizen gemacht haben! Ihre Klassenmethoden, Felder und Eigenschaften entwerfen Sie, indem Sie über das Verhalten der Klasse (was sie ***tun muss***) und ihren Zustand (was sie ***wissen muss***) nachdenken.



Fügen Sie Kathrins Party-Planer Geburtstagsfeiern hinzu.

Achten Sie darauf, dass Sie für die Felder und Eigenschaften für Geldbeträge decimal verwenden.



1 **Fügen Sie Ihrem Programm die Klasse Geburtstagsfeier hinzu.** Was bei der Eigenschaft Personenanzahl und den Dekokosten-Methoden zu tun ist, wissen Sie bereits – diese entsprechen ihren Gegenstücken in der Klasse Party. Beginnen Sie die Erstellung der neuen Klasse damit, dass Sie diese hinzufügen. Ergänzen Sie dann das restliche Verhalten:

Ist der Kuchentext zu lang für den Kuchen, schneidet der Set-Akzessor ihn auf die Länge des Unterstützungsfelds ab. Sie müssen sicherstellen, dass der Text jedes Mal neu in das Textfeld geladen wird, wenn sich die Personenanzahl ändert (für den Fall, dass ein größerer oder kleinerer Kuchen verwendet wird).

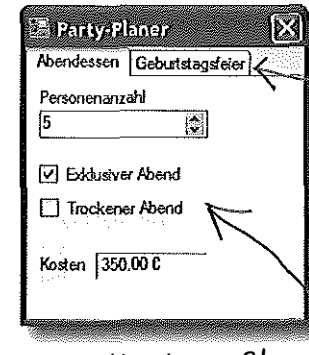
- Fügen Sie das öffentliche int-Feld Kuchengröße hinzu. Erstellen Sie die private Methode kuchengrößeBerechnen(), die die Kuchengröße je nach Personenanzahl auf 18 oder 26 setzt. Rufen Sie diese aus dem Konstruktor und dem Personenanzahl-Setter auf.
- Sie brauchen die string-Eigenschaft KuchenText, um den Text für den Kuchen zu speichern. Dafür geben wir Ihnen den Code.

KuchenText sollte von keinen anderen Methoden geändert werden können.

- Der KuchenText-Setter prüft die Kuchengröße, weil die Anzahl der möglichen Buchstaben auf einem Kuchen von seiner Größe abhängig ist. Dann nutzt er `value.Length`, um die String-Länge zu prüfen. Ist der String zu lang, wird das Feld nicht gesetzt und stattdessen ein Fenster eingeblendet, das sagt: »Text zu lang für 26-cm-Kuchen« (oder 18-cm-Kuchen).
- Strings haben die Methode **`Substring()`**, die einen Teil des Strings liefert. Diese nutzt KuchenText, um die Schriftlänge zu beschneiden – das ist auch erforderlich, wenn eine Änderung der Personenanzahl zu einer Änderung der Kuchengröße führt.
- Fügen Sie abschließend die Methode KostenBerechnen() hinzu. Addieren Sie statt Dekorations- und Getränkekosten (wie beim Abendessen) die Dekorations- und Kuchenkosten zusammen.

Zwischen den Tabs wechseln Sie, indem Sie darauf klicken. Nutzen Sie die Eigenschaft TabCollection, um den Text für die Tabs zu ändern. Klicken Sie auf den »...«-Button daneben und wählen Sie die Text-Eigenschaften der Tabs.

2 **Fügen Sie dem Formular die Registerkarten hinzu.** Ziehen Sie ein TabControl aus der Toolbox auf Ihr Formular und passen Sie seine Größe an die des Formulars an. Ändern Sie den Text der Tabs über die Eigenschaft TabPages: Im Eigenschaftenfenster erscheint neben dieser Eigenschaft ein »...«-Button. Klicken Sie darauf, öffnet die IDE ein Fenster, in dem Sie die Eigenschaften der Tabs bearbeiten können. Setzen Sie die Text-Eigenschaft der Tabs auf »Abendessen« und »Geburtstagsfeier«.



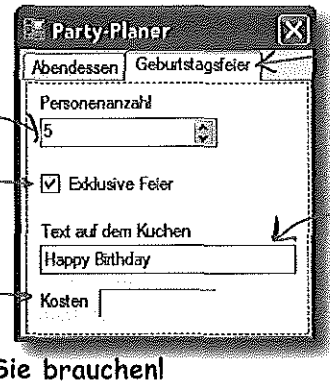
3 **Geben Sie dem ersten Tab einen Namen und schieben Sie die Abendessen-Steuerelemente darauf.** Wenn Sie die Steuerelemente für die Abendessen auf den neuen Tab ziehen, funktionieren diese weiter wie zuvor, sind aber nur sichtbar, wenn dieser Tab ausgewählt ist.

Wurden die Abendessen-Steuerelemente auf den Tab gezogen, sind sie nur noch sichtbar, wenn der entsprechende Tab ausgewählt ist.

4 Erstellen Sie die Schnittstelle für die Geburtsfeiern.

Das GUI für Geburtstagsfeiern enthält ein NumericUpDown-Steuerelement für die Personenanzahl, eine CheckBox für die exklusiven Dekorationen und ein Label mit einem 3-D-Rahmen für die Kosten. Dann fügen Sie noch eine TextBox für den Kuchentext hinzu.

Wie der Abendessen-Tab nutzt auch dieser NumericUpDown-, CheckBox- und Label-Steuerelemente. Geben Sie ihnen die Namen anzahlGeburtstag, exklusiverGeburtstag und geburtstagsKosten.



Klicken Sie auf den Tab Geburtstagsfeier und ergänzen Sie die neuen Steuerelemente.

Fügen Sie eine TextBox namens kuchenText für den Text auf dem Kuchen hinzu (und ein Label darüber, damit der Benutzer weiß, wozu sie dient). Geben Sie ihr über die Eigenschaft Text den Default-Wert »Herzlichen Glückwunsch«.

5 Diese Eigenschaft werden Sie brauchen!

Hier ist der Code für die Eigenschaft Geburtstagsfeier.KuchenText – sie wird hilfreich sein:

```
private string kuchenText = "";
public string KuchenText {
    get { return this.kuchenText; }
    set {
        int maxLänge;
        if (Kuchengröße == 8)
            maxLänge = 16;
        else
            maxLänge = 40;
        if (value.Length > maxLänge) {
            MessageBox.Show("Text zu lang für " + Kuchengröße + "cm-Kuchen");
            if (maxLänge > this.kuchenText.Length)
                maxLänge = this.kuchenText.Length;
            this.kuchenText = kuchenText.Substring(0, maxLänge);
        }
        else
            this.kuchenText = value;
    }
}
```

Diese Eigenschaft ist etwas komplizierter als die, die Sie bisher gesehen haben. Sie prüft, ob der Text zu lang für den Kuchen ist. Dazu nutzt sie die Variable maxLänge, um die maximale Länge zu speichern. Ist der Text zu lang, wird eine Fehlermeldung geliefert und das Unterstützungsfeld auf die richtige Länge abgeschnitten, damit er wieder in die TextBox geladen werden kann.

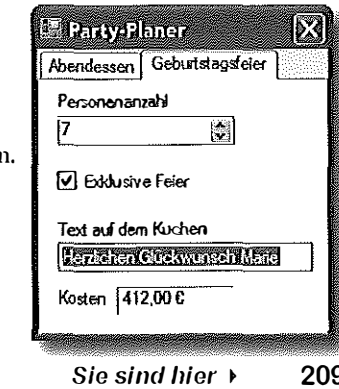
Sehen Sie, dass wir einige der geschweiften Klammern weggelassen haben? Enthält ein Codeblock nur eine Anweisung, benötigen Sie diese nicht.

Strings bieten die Methode Substring(), die einen Teil des Strings liefert. Diese schneidet ihn auf die erlaubte Länge ab. Sie müssen ihn deswegen neu in die TextBox laden, wenn sich Text oder Kuchengröße ändert.

6 Packen Sie alles zusammen.

Jetzt ist alles da. Sie müssen also nur noch etwas Code schreiben, damit die Steuerelemente funktionieren.

- ★ Fügen Sie dem Formular ein Geburtstagsfeier-Objekt hinzu und instantiieren Sie es.
- ★ Fügen Sie dem Event-Handler des NumericUpDown-Steuerelements Code hinzu, der die Personenanzahl-Eigenschaft des Objekts setzt.
- ★ Sorgen Sie dafür, dass die Checkbox für die exklusive Dekoration funktioniert.
- ★ Fügen Sie eine GeburtstagsfeierKostenAnzeigen()-Methode hinzu, über die alle Event-Handler das Label mit den Kosten aktualisieren.



7 Führen Sie es aus.

Schauen Sie, ob das Programm funktioniert wie erwartet. Testen Sie, ob eine Meldung angezeigt wird, wenn der Text für den Kuchen zu lang ist, und prüfen Sie, ob der Preis immer stimmt. Wenn alles funktioniert, sind Sie fertig!



Fügen Sie Kathrins Party-Planer Geburtstagsfeiern hinzu.

```
public partial class Form1 : Form {
    Abendessen abendessen;
    Geburtstagsfeier geburtstagsfeier;
    public Form1() {
        InitializeComponent();
        abendessen = new Abendessen((int)numericUpDown1.Value,
                                    trockenBox.Checked, exklusivBox.Checked);
        AbendessenKostenAnzeigen();

        geburtstagsfeier = new Geburtstagsfeier((int)anzahlGeburtstag.Value,
                                exklusiverGeburtstag.Checked, kuchenText.Text);
        GeburtstagsfeierKostenAnzeigen();
    }

    // Die Event-Handler für exklusivBox, trockenBox und numericUpDown1
    // die Methode AbendessenKostenAnzeigen() entsprechen denen in der
    // Abendessen-Übung am Ende von Kapitel 5.


    private void anzahlGeburtstag_ValueChanged(object sender, EventArgs e) {
        geburtstagsfeier.Personenanzahl = (int)anzahlGeburtstag.Value;
        GeburtstagsfeierKostenAnzeigen();
    }

    private void exklusiverGeburtstag_CheckedChanged(object sender,
                                                     EventArgs e) {
        geburtstagsfeier.DekokostenBerechnen(exklusiverGeburtstag.Checked);
        GeburtstagsfeierKostenAnzeigen();
    }

    private void kuchenText_TextChanged(object sender, EventArgs e) {
        geburtstagsfeier.KuchenText = kuchenText.Text;
        GeburtstagsfeierKostenAnzeigen();
    }

    private void GeburtstagsfeierKostenAnzeigen() {
        kuchenText.Text = geburtstagsfeier.KuchenText;
        decimal kosten = geburtstagsfeier.KostenBerechnen();
        geburtstagsKosten.Text = kosten.ToString("c");
    }
}
```

Die Geburtstagsfeier-Instanz wird genau wie die Abendessen-Instanz im Formularkonstruktor initialisiert.

Die Event-Handler der CheckBox und des NumericUp-Down sind die gleichen wie die für das Abendessen.

Wie das Formular mit dem Kuchentext umgeht, kann so einfach sein, weil die Klasse Geburtstagsfeier gut gekapselt ist. Das Formular muss nur die Eigenschaften des Objekts setzen, und das Objekt kümmert sich dann um den Rest.

Die gesamte Logik für die Prüfung des Texts und der Personenanzahl ist in die Setter von Personenanzahl und KuchenText eingebaut. Das Formular muss die Werte also nur noch setzen und anzeigen.

```
using System.Windows.Forms;

public class Geburtstagsfeier {
    public const int EssenskostenProPerson = 25;

    public decimal Dekokosten = 0;
    private bool exklusiveDekoration;
    public int Kuchengröße;

    public Geburtstagsfeier(int personenanzahl, bool exklusiveDekoration, string kuchenText) {
        this.personenanzahl = personenanzahl;
        this.exklusiveDekoration = exklusiveDekoration;
        kuchengrößeBerechnen();
        this.KuchenText = kuchenText;
        DekokostenBerechnen(exklusiveDekoration);
    }

    private void kuchengrößeBerechnen() {
        if (Personenanzahl <= 4)
            Kuchengröße = 8;
        else
            Kuchengröße = 16;
    }

    private string kuchenText = "";
    public string KuchenText {
        get { return this.kuchenText; }
        set {
            int maxLänge;
            if (Kuchengröße == 8)
                maxLänge = 16;
            else
                maxLänge = 40;
            if (value.Length > maxLänge) {
                MessageBox.Show("Text zu lang für einen " + Kuchengröße + "cm-Kuchen");
                if (maxLänge > this.kuchenText.Length)
                    maxLänge = this.kuchenText.Length;
                this.kuchenText = kuchenText.Substring(0, maxLänge);
            }
            else
                this.kuchenText = value;
        }
    }
```

Wird das Geburtstagsfeier-Objekt initialisiert, muss es die Personenanzahl, die Dekorationsart und den Text auf dem Kuchen kennen, damit direkt die richtigen Kuchenkosten berechnet werden können, wenn KostenBerechnen() aufgerufen wird.

Der Konstruktor ruft den Setter auf, um den Kuchentext zu setzen. Dazu muss zunächst die Kuchengröße berechnet werden, damit der Setter prüfen kann, ob der Text für den Kuchen zu lang ist

Der Konstruktor setzt die Eigenschaft und führt dann die Berechnung aus.

Die Methode kuchengrößeBerechnen() setzt das Feld Kuchengröße. Es wird vom Personenanzahl-Setter und der Methode KostenBerechnen() aufgerufen.

Die Eigenschaft KuchenText sichert, dass der Text auf dem Kuchen nie zu lang für die Kuchengröße ist. Ihr Setter prüft die Kuchengröße und nutzt dann die Length-Eigenschaft des Unterstützungsfelds, um sicherzustellen, dass er nicht zu lang ist. Ist der Text zu lang, schneidet er den String auf die richtige Länge ab.



Fügen Sie Kathrins Party-Planer Geburtstagsfeiern hinzu.

Lösungen zu den Übungen

Wir nutzen decimal, weil wir mit Preisen und Geldbeträgen arbeiten.

```
public decimal KostenBerechnen() {
    decimal GesamtKosten = Dekokosten + (EssenskostenProPerson * Personenanzahl);
    decimal Kuchenkosten;
    if (Kuchengröße == 8)
        Kuchenkosten = 40M + KuchenText.Length * .25M;
    else
        Kuchenkosten = 75M + KuchenText.Length * .25M;
    return GesamtKosten + Kuchenkosten;
}
```

Die Methode KostenBerechnen() hat große Ähnlichkeit mit der aus Abendessen, fügt aber die Kuchenkosten statt der Getränkekosten hinzu.

```
private int Personenanzahl;
public int Personenanzahl {
    get { return personenanzahl; }
    set {
        personenanzahl = value;
        DekokostenBerechnen(exklusiveDekoration);
        kuchengrößeBerechnen();
        this.KuchenText = kuchenText;
    }
}
```

Den KuchenText-Setter die Textlänge abschneiden zu lassen ist nur die Hälfte der Lösung. Die andere Hälfte besteht darin sicherzustellen, dass der KuchenText-Setter jedes Mal ausgeführt wird, wenn sich die Personenanzahl ändert.

Ändert sich die Personenanzahl, berechnet die Klasse zunächst die Kuchengröße neu und nutzt dann den KuchenText-Setter, um den Text zurechtzuschneiden – verwandelt sich eine 10-Personen-Feier in eine 4-Personen-Feier, wird ein 36-Buchstaben-Text so zugeschnitten, dass er auf den kleineren Kuchen passt.

Diese Methode ist genau wie die in der Klasse Abendessen.

```
public void DekokostenBerechnen(bool exklusiv) {
    exklusiveDekoration = exklusiv;
    if (exklusiv)
        Dekokosten = (Personenanzahl * 15.00M) + 50M;
    else
        Dekokosten = (Personenanzahl * 7.50M) + 30M;
}
}
```

Bei Ein-Zeilen-Blöcken sind geschweifte Klammern optional

Die Blöcke von if-Anweisungen oder while-Schleifen enthalten häufig nur eine einzige Anweisung. Passiert das häufig, kann das dazu führen, dass Sie viele geschweifte Klammern haben – und das kann ziemlich hässlich aussehen! C# hilft Ihnen, dieses Problem zu vermeiden, indem es Ihnen gestattet, die geschweiften Klammern wegzulassen, wenn es nur eine Anweisung gibt. Folgende Syntaxformen sind also vollkommen in Ordnung:

```
for (int i = 0; i < 10; i++)
    AufgabeAusführen(i);

if (einWert == 36)
    einWert *= 5;
```

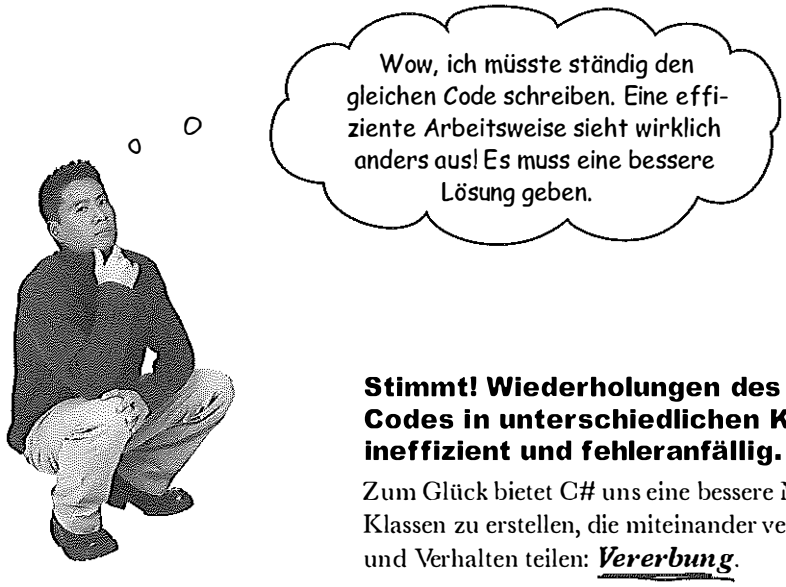
Eine Sache noch ... können Sie eine Gebühr von 100 € für Partys mit mehr als 12 Personen hinzufügen?

Seitdem Kathrin Ihr Programm einsetzt, hat sie so viel zu tun, dass sie es sich leisten kann, bei ihren größeren Kunden etwas mehr zu verlangen. Was müssten Sie tun, um Ihrem Programm eine zusätzliche Gebühr hinzuzufügen?

- Sie müssen Abendessen.KostenBerechnen() so ändern, dass die Personenanzahl geprüft und dem Rückgabewert 100 hinzugefügt wird, wenn sie größer als 12 ist.
- Sie müssen genau das Gleiche für Geburtstagsfeier.KostenBerechnen() erledigen.

Nehmen Sie sich einen Moment Zeit und überlegen Sie, wie Sie den Klassen Abendessen und Geburtstagsfeier diese Gebühr hinzufügen. Was für Code würden Sie schreiben? Wo müssten Sie ihn hinstecken?

Kein Problem ... aber was ist, wenn es drei ähnliche Klassen gibt? Oder vier? Oder zwölf? Und was ist, wenn Sie diesen Code später warten und weitere Änderungen an ihm vornehmen müssten? Was ist, wenn Sie ***die gleiche Änderung*** an fünf, sechs ***eng verwandten*** Klassen vornehmen müssten?



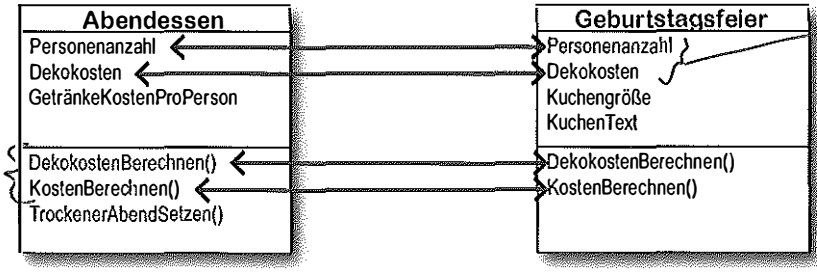
Stimmt! Wiederholungen des gleichen Codes in unterschiedlichen Klassen ist ineffizient und fehleranfällig.

Zum Glück bietet C# uns eine bessere Möglichkeit, Klassen zu erstellen, die miteinander verwandt sind und Verhalten teilen: ***Vererbung***.

Nutzen Ihre Klassen Vererbung, müssen Sie Ihren Code nur einmal schreiben

Es ist kein Zufall, dass Ihre Klassen Abendessen und Geburtstagsfeier eine Menge Code gemeinsam haben. Wenn man C#-Programme schreibt, verwendet man oft Klassen, die Dinge aus der richtigen Welt repräsentieren – und diese Dinge sind in der Regel miteinander verwandt. Ihre Klassen haben **ähnlichen Code**, weil die Dinge aus der richtigen Welt, die sie repräsentieren – eine Geburtstagsfeier und ein Abendessen –, **ähnliche Verhalten** haben.

Kathrin muss die Kosten ihrer Partys herausfinden, egal welcher Art die Partys sind.



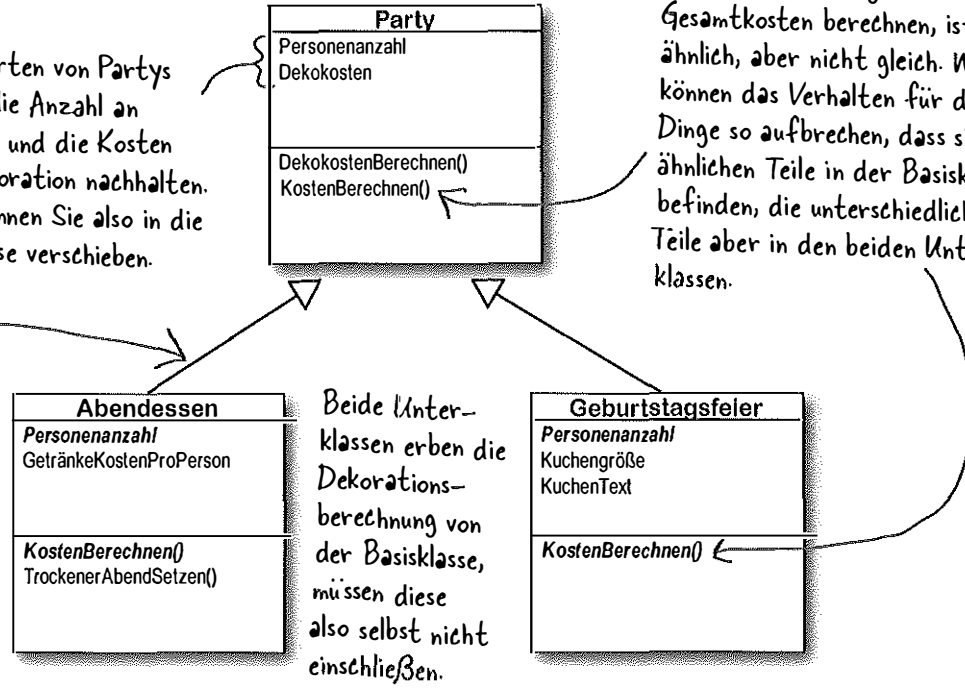
Eine Geburtstagsfeier geht mit der Personenanzahl und den Dekorationskosten fast auf die gleiche Weise um wie ein Abendessen.

Abendessen und Geburtstagsfeiern sind beides Partys

Wenn Sie zwei Klassen haben, die spezifischere Fälle von etwas Allgemeinerem sind, können Sie es so einrichten, dass sie von der gleichen Klasse **erben**. Tun Sie das, sind beides **Unterklassen** der gleichen **Basisklasse**.

Beide Arten von Partys müssen die Anzahl an Personen und die Kosten der Dekoration nachhalten. Diese können Sie also in die Basisklasse verschieben.

Wie die beiden Partys mit der Personenanzahl umgehen und die Gesamtkosten berechnen, ist ähnlich, aber nicht gleich. Wir können das Verhalten für diese Dinge so aufbrechen, dass sich die ähnlichen Teile in der Basisklasse befinden, die unterschiedlichen Teile aber in den beiden Unterklassen.

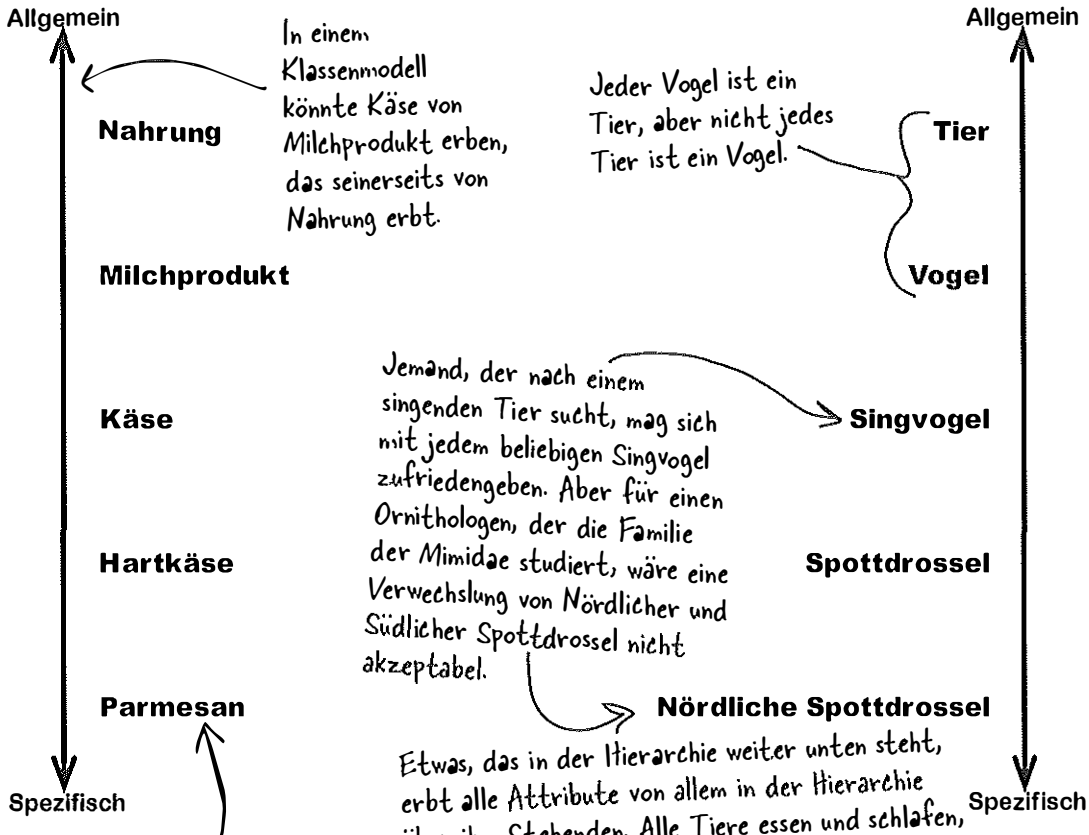


Dieser Pfeil im Klassendiagramm bedeutet, dass die Klasse Abendessen von der Klasse Party erbt.

Beide Unterklassen erben die Dekorationsberechnung von der Basisklasse, müssen diese also selbst nicht einschließen.

Erstellen Sie Ihr Klassenmodell, indem Sie allgemein beginnen und immer spezifischer werden

C#-Programme nutzen Vererbung, weil sie die Beziehungen der von ihnen modellierten Dinge imitieren, die diese in der wirklichen Welt haben. Im wahren Leben stehen Dinge oft in einer **Hierarchie**, die vom Allgemeineren zum Spezifischeren geht. Ihre Klassen haben eine eigene **Klassenhierarchie**, die das Gleiche macht. In Ihrem Klassenmodell **erben** Klassen, die in der Hierarchie weiter unten stehen, von denen darüber.



er-ben, Verb
Ein Attribut von Eltern oder Vorfahren erhalten. *Sie wünschte, ihr Baby würde ihre großen braunen Augen **erben**, nicht die kleinen blauen ihres Mannes.*

Wie würden Sie einen Zoo-Simulator entwerfen?

Löwen, Tiger und Bären ... meine Güte! Und dann gibt es ja auch noch Nilpferde, Wölfe und die eine oder andere Katze. Ihre Aufgabe ist es, ein Programm zu entwerfen, das einen Zoo simuliert. (Freuen Sie sich nicht zu früh – wir werden keinen Code schreiben, sondern nur die Klassen entwerfen, die die Tiere repräsentieren.)

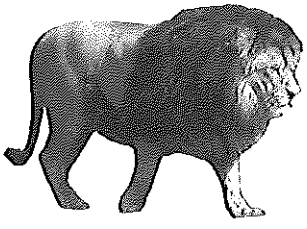
Sie haben eine Liste mit einigen der Tiere erhalten, die im Programm repräsentiert werden sollen, aber nicht mit allen. Wir wissen, dass jedes Tier durch ein Objekt repräsentiert wird und dass sich die Objekte im Simulator herumbewegen und Dinge tun werden, für die diese spezifische Art Tier programmiert ist.

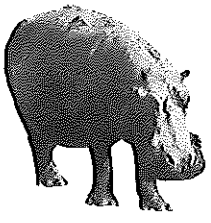
Noch wichtiger ist, dass das Programm für andere Entwickler leicht zu warten sein soll. Das bedeutet, dass diese dazu in der Lage sein müssen, später eigene Klassen hinzuzufügen, wenn sie den Simulator um weitere Tiere erweitern wollen.

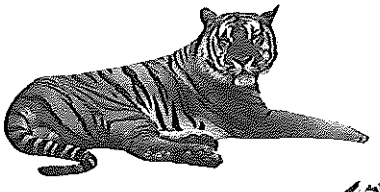
Was also ist der erste Schritt? Na, bevor wir über **bestimmte** Tiere sprechen können, müssen wir herausfinden, welche **allgemeinen** Dinge sie gemeinsam haben, abstrakte Charakteristiken, die ***alle*** Tiere haben. Dann können wir diese Charakteristiken in eine Klasse einbauen, von der alle Tierklassen erben können.

1 Suchen Sie nach Dingen, die diese Tiere gemeinsam haben.

Schauen Sie sich diese sechs Tiere an. Was haben ein Löwe, ein Nilpferd, ein Tiger, eine Katze, ein Wolf und ein Dalmatiner gemeinsam? Wie sind sie miteinander verwandt? Sie müssen ihre Beziehungen herausfinden, damit Sie ein Klassenmodell entwickeln können, das alle einschließt.

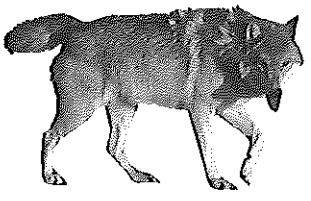












Nutzen Sie Vererbung, um doppelten Code in Unterklassen zu vermeiden

Sie haben bereits eingesehen, dass doppelter Code schlecht ist. Er ist schwer zu warten und wird irgendwann zu Kopfschmerzen führen. Wählen wir also Felder und Methoden für eine Basisklasse Tier, **die Sie nur einmal schreiben müssen**, damit alle Tier-Unterklassen diese erben können. Beginnen wir mit den öffentlichen Feldern:

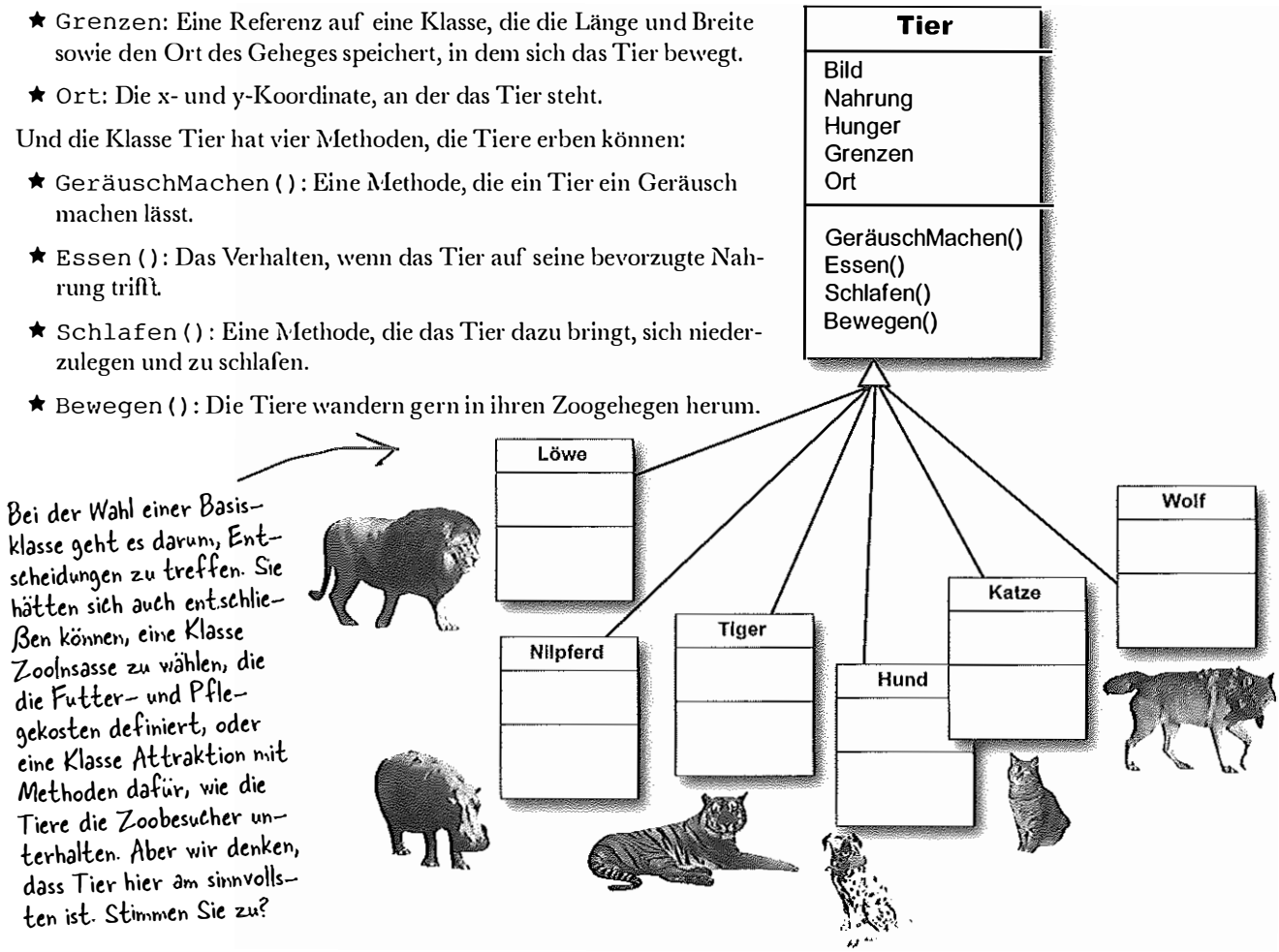
- ★ `Bild`: Ein Bild, das Sie in eine PictureBox stecken können.
- ★ `Nahrung`: Der Typ Nahrung, den dieses Tier zu sich nimmt. Im Augenblick gibt es nur zwei Werte: Fleisch und Gras.
- ★ `Hunger`: Ein `int`, der den Grad des Hungers eines Tiers repräsentiert. Er ändert sich in Abhängigkeit davon, wann das Tier isst (und wie viel).
- ★ `Grenzen`: Eine Referenz auf eine Klasse, die die Länge und Breite sowie den Ort des Geheges speichert, in dem sich das Tier bewegt.
- ★ `Ort`: Die x- und y-Koordinate, an der das Tier steht.

Und die Klasse Tier hat vier Methoden, die Tiere erben können:

- ★ `GeräuschMachen()`: Eine Methode, die ein Tier ein Geräusch machen lässt.
- ★ `Essen()`: Das Verhalten, wenn das Tier auf seine bevorzugte Nahrung trifft.
- ★ `Schlafen()`: Eine Methode, die das Tier dazu bringt, sich niederzulegen und zu schlafen.
- ★ `Bewegen()`: Die Tiere wandern gern in ihren Zoogehegen herum.

❽ Erstellen Sie eine Basisklasse, um den Tieren alles zu geben, was sie gemeinsam haben.

Die Felder, Eigenschaften und Methoden in der Basisklasse geben allen Tieren, die von ihr erben, einen gemeinsamen Zustand und ein gemeinsames Verhalten. Da es alles Tiere sind, ist es sinnvoll, der Basisklasse den Namen Tier zu geben.



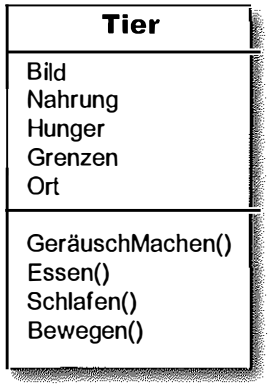
Unterschiedliche Tiere machen unterschiedliche Geräusche

Löwen brüllen, Hunde bellen – und *unseres* Wissens machen Nilpferde überhaupt keine Geräusche. Jede der Klassen, die von Tier erben, hat eine `GeräuschMachen()`-Methode, aber jede dieser Methoden arbeitet auf andere Weise und hat anderen Code. Wenn eine Unterklasse das Verhalten einer der Methoden ändert, die sie geerbt hat, sprechen wir davon, dass sie diese Methode **überschreibt**.

Überlegen Sie, was Sie überschreiben müssen

Wenn eine Unterklasse das Verhalten einer geerbten Methode ändert, nennen wir das **überschreiben**. Jedes Tier muss essen. Aber wahrscheinlich nimmt ein Hund nur kleine Bissen Fleisch, während ein Nilpferd große gewaltige Mengen an Gras isst. Wie sähe also der Code für dieses Verhalten aus? Hund und Nilpferd würden beide die Methode `Essen()` überschreiben. Die Methode von Nilpferd würde bei jedem Aufruf beispielsweise 20 Pfund Heu verschlingen. Die `Essen()`-Methode von Hund würde hingegen den Nahrungsvorrat des Zoos um eine 400-Gramm-Dose Hundefutter verringern.

Wenn Sie eine Unterklasse haben, die von einer Basisklasse erbt, muss sie alle Verhalten dieser Basisklasse erben ... aber Sie können sie in der Unterklasse so modifizieren, dass sie nicht auf gleiche Weise ausgeführt werden. Das ist das, worum es beim Überschreiben geht.







Nur weil sich eine Eigenschaft oder Methode in der Basisklasse Tier befindet, heißt das nicht, dass jede Unterklasse sie auf die gleiche Weise nutzen soll ... oder überhaupt!

8

Finden Sie heraus, was die einzelnen Tier-Klassen anders machen – oder überhaupt nicht.

Hat eine Tierart ein Verhalten, das keine der anderen Tierarten hat? Hunde essen Futter, die Essen()-Methode von Hund muss also die Methode `Tier.Essen()` überschreiben. Nilpferde schwimmen, also braucht Nilpferd eine `Schwimmen()`-Methode, die keine der anderen Tierklassen hat.



KOPF-NUSS

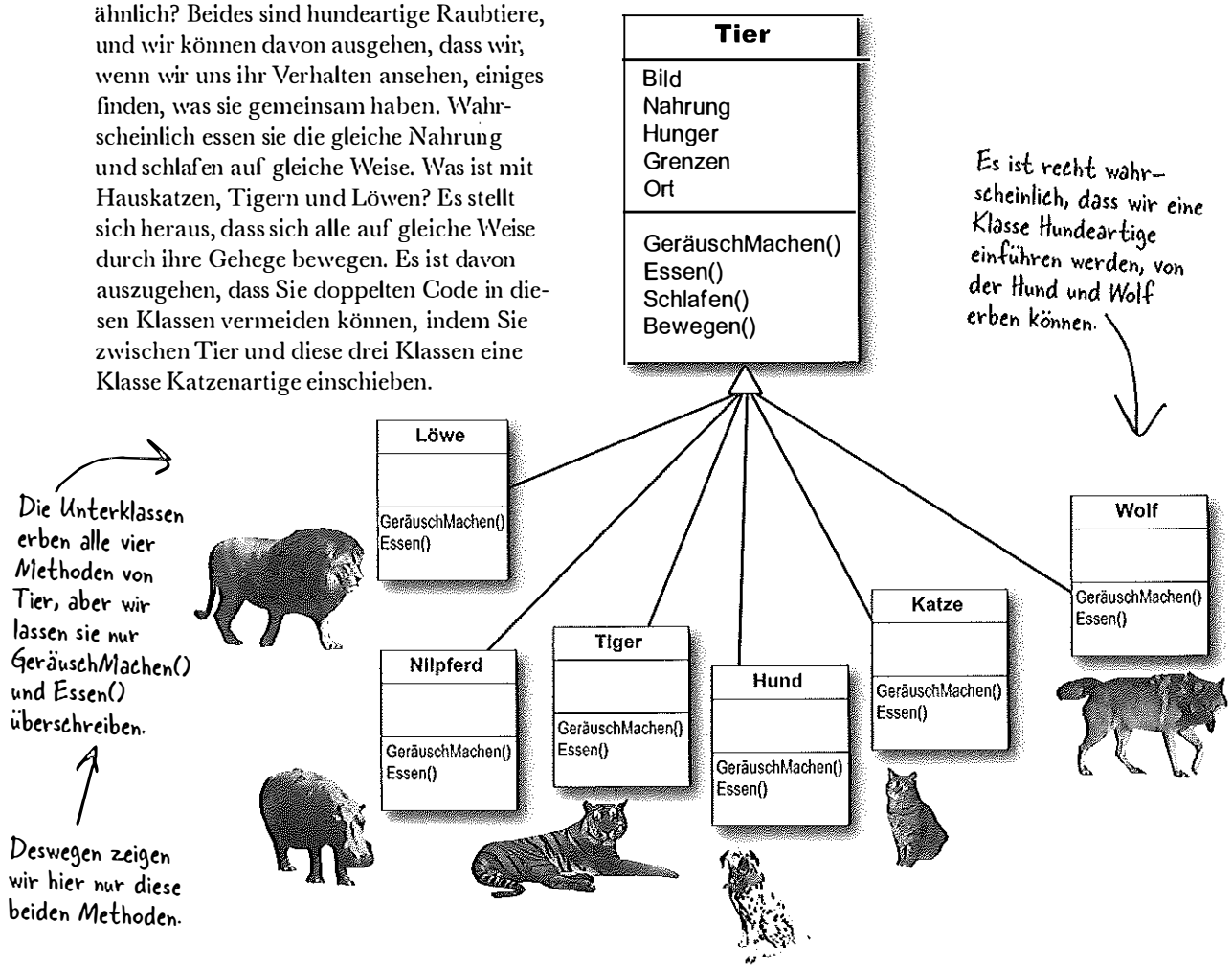
Wir wissen bereits, dass einige Tiere die Methoden **`GeräuschMachen()`** und **`Essen()`** überschreiben werden. Welche Tiere werden **`Schlafen()`** oder **`Bewegen()`** überschreiben? Tun es überhaupt welche? Was ist mit den Eigenschaften – welche Tiere überschreiben einige Eigenschaften?

Denken Sie daran, wie die Tiere gruppiert sind

Parmesan ist eine Art Käse, Käse ein Milchprodukt und das wiederum eine Art Nahrung. Ein gutes Klassenmodell für Nahrung würde das widerspiegeln. Glücklicherweise gibt C# uns ein einfaches Mittel, das zu tun. Sie können eine Kette von Klassen erstellen, die voneinander erben, die mit der obersten Basisklasse beginnt und sich nach unten vorarbeitet. Sie könnten eine Klasse Nahrung mit einer Unterklasse Milchprodukt haben, wobei Milchprodukt als Basisklasse für die Klasse Käse dient, die eine Unterklasse namens Hartkäse hat, von der Parmesan erbt.

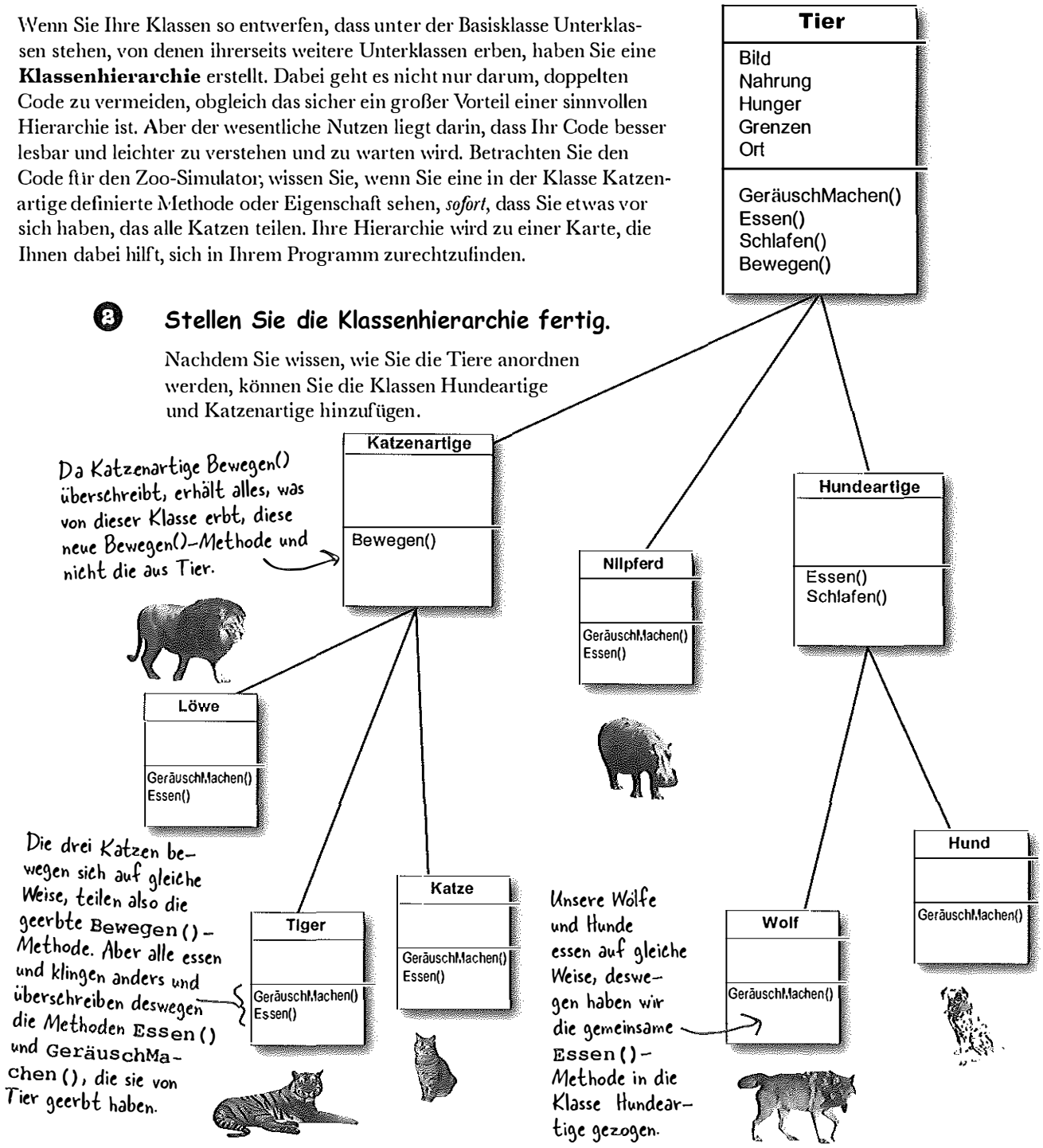
4 Suchen Sie nach Klassen, die vieles gemeinsam haben.

Sind Wölfe und Hunde nicht ziemlich ähnlich? Beides sind hundeartige Raubtiere, und wir können davon ausgehen, dass wir, wenn wir uns ihr Verhalten ansehen, einiges finden, was sie gemeinsam haben. Wahrscheinlich essen sie die gleiche Nahrung und schlafen auf gleiche Weise. Was ist mit Hauskatzen, Tigern und Löwen? Es stellt sich heraus, dass sich alle auf gleiche Weise durch ihre Gehege bewegen. Es ist davon auszugehen, dass Sie doppelten Code in diesen Klassen vermeiden können, indem Sie zwischen Tier und diese drei Klassen eine Klasse Katzenartige einschieben.



Die Klassenhierarchie erstellen

Wenn Sie Ihre Klassen so entwerfen, dass unter der Basisklasse Unterklassen stehen, von denen ihrerseits weitere Unterklassen erben, haben Sie eine **Klassenhierarchie** erstellt. Dabei geht es nicht nur darum, doppelten Code zu vermeiden, obgleich das sicher ein großer Vorteil einer sinnvollen Hierarchie ist. Aber der wesentliche Nutzen liegt darin, dass Ihr Code besser lesbar und leichter zu verstehen und zu warten wird. Betrachten Sie den Code für den Zoo-Simulator; wissen Sie, wenn Sie eine in der Klasse Katzenartige definierte Methode oder Eigenschaft sehen, *sofort*, dass Sie etwas vor sich haben, das alle Katzen teilen. Ihre Hierarchie wird zu einer Karte, die Ihnen dabei hilft, sich in Ihrem Programm zurechtzufinden.



8 Stellen Sie die Klassenhierarchie fertig.

Nachdem Sie wissen, wie Sie die Tiere anordnen werden, können Sie die Klassen Hundeartige und Katzenartige hinzufügen.

Jede Unterklasse erweitert Ihre Basisklasse

Sie sind nicht auf die Methoden beschränkt, die eine Unterklasse von ihrer Basisklasse erbt ... aber das wissen Sie bereits! Schließlich haben Sie ja schon die ganze Zeit Ihre eigenen Klassen aufgebaut. Leiten Sie eine Klasse von einer anderen Klasse ab, ***erweitern*** Sie diese, indem Sie ihr weitere Felder, Eigenschaften und Methoden für speziellere Aufgaben hinzufügen. Möchten Sie der Klasse Hund eine `Apportieren()`-Methode hinzufügen, ist das ganz in Ordnung. Sie erbt oder überschreibt nichts – nur Hund-Objekte erhalten diese Methode, sie kommt nicht in Wolf, Hundeartige, Tier, Nilpferd oder irgendeine andere Klasse.

Hi-e-r-ar-chie, Nomen
Eine Anordnung oder Klassifikation, in der Gruppen oder Dinge unter- und übereinander angeordnet werden. *Der aktuelle Direktor des Unternehmens hatte sich von der Poststelle bis an die Spitze der Unternehmens**hierarchie** emporgearbeitet.*

macht ein neues Hund-Objekt `Hund rex = new Hund();`

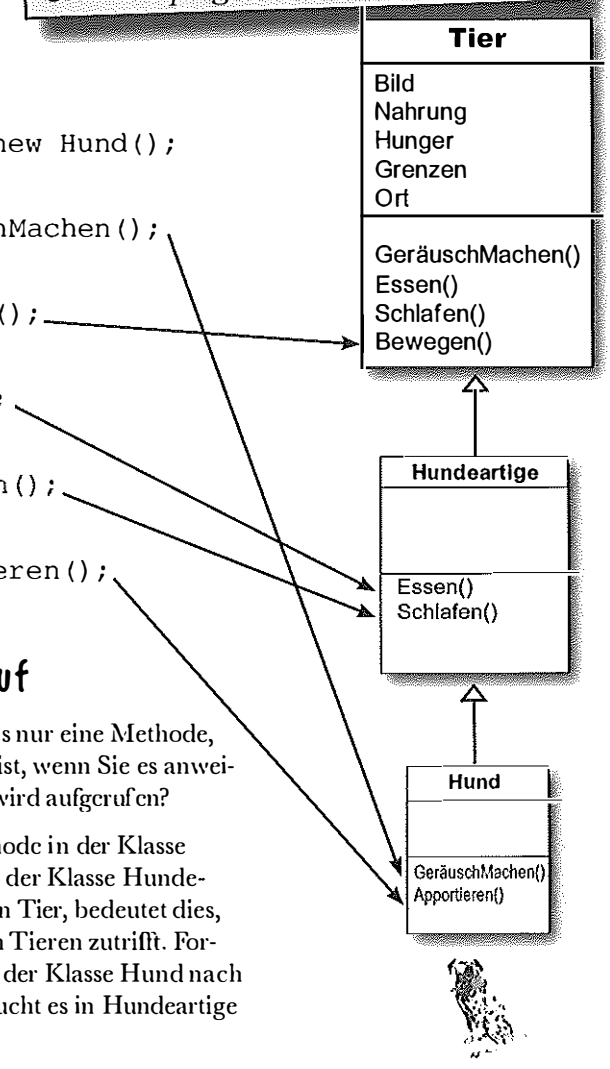
ruft die Version in Hund auf `rex.GeräuschMachen();`

ruft die Version in Tier auf `rex.Bewegen();`

ruft die Version in Hundeartige auf `rex.Essen();`

ruft die Version in Hundeartige auf `rex.Schlafen();`

ruf die Version in Hund auf `rex.Apportieren();`



C# ruft immer die spezifischste Methode auf

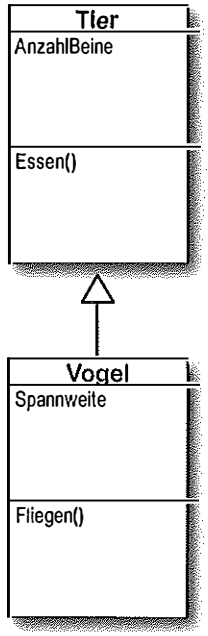
Sagen Sie Ihrem Hund-Objekt, dass es sich bewegen soll, gibt es nur eine Methode, die aufgerufen werden kann – die in der Klasse Tier. Aber was ist, wenn Sie es anweisen, Laut zu geben? Welche `GeräuschMachen()`-Methode wird aufgerufen?

Eigentlich ist es nicht so schwer, das herauszufinden. Eine Methode in der Klasse Hund zeigt Ihnen, wie ein Hund das macht. Ist die Methode in der Klasse Hundeartige, sagt Sie ihnen, wie es Hundeartige machen. Und ist sie in Tier, bedeutet dies, dass dieses Verhalten so allgemein ist, dass es auf alle Arten von Tieren zutrifft. Fordern Sie einen Hund auf, Laut zu geben, sucht C# zunächst in der Klasse Hund nach einem *spezifischen* Verhalten von Hunden. Besitzt Hund keins, sucht es in Hundeartige und anschließend in Tier.

Erweiterung geben Sie mit einem Doppelpunkt an

Leiten Sie eine Klasse ab, nutzen Sie einen **Doppelpunkt (:)**, um sie von einer Basisklasse erben zu lassen. Das macht die neue Klasse zu einer Unterklasse und gibt ihr **alle Felder, Eigenschaften und Methoden** der Klasse, die sie erweitert.

Erbt eine Unterklasse von einer Basisklasse, erhält sie automatisch alle Felder, Eigenschaften und Methoden der Basisklasse.




```
public class Tier
{
    public int AnzahlBeine;
    public void Essen() {
        // Code, der das Tier essen lässt.
    }
}
```

Die Klasse Vogel nutzt einen Doppelpunkt, um von der Klasse Tier abzuleiten. Das bedeutet, dass sie alle Felder, Eigenschaften und Methoden von Tier erbt.

```
public class Vogel : Tier
{
    public double Spannweite;
    public void Fliegen() {
        // Code, der den Vogel fliegen lässt.
    }
}
```

Eine Klasse leiten Sie ab, indem Sie ans Ende der Klassendeklaration einen Doppelpunkt anhängen und dahinter den Namen der zu erweiternden Klasse angeben.

```
public buttonl_Click(object sender, EventArgs e) {
    Vogel tweety = new Vogel();
    tweety.Spannweite = 7.5;
    tweety.Fliegen();
    tweety.AnzahlBeine = 2;
    tweety.Essen();
}
```

tweety ist eine Instanz von Vogel, hat also die Vogel-Methoden und -Felder.

Da die Klasse Vogel von Tier erbt, hat jede Instanz von Vogel auch die Felder und Methoden, die in der Klasse Tier definiert werden.

Es gibt keine Dummen Fragen

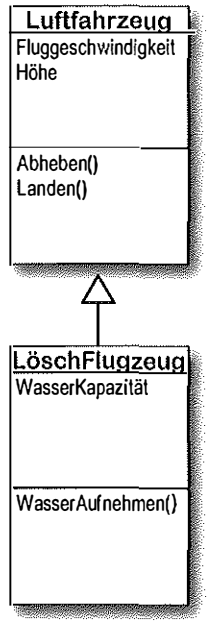
F: Warum zeigt der Pfeil von der Unterklasse zur Basisklasse? Sähe das Diagramm nicht besser aus, wenn der Pfeil in die andere Richtung zeigen würde?

A: Das sähe vielleicht besser aus, wäre aber nicht so genau. Wenn Sie eine Klasse von einer anderen ableiten, bauen Sie dieses Verhalten in die Unterklasse ein – die Basisklasse bleibt unverändert. Und aus Perspektive der Basisklasse ist das auch sinnvoll.

Fügen Sie eine Klasse ein, die eine andere erweitert, verändern Sie das Verhalten der Basisklasse nicht. Sie weiß noch nicht einmal, dass sie von dieser neuen Klasse erweitert wird. Ihre Methoden, Felder und Eigenschaften bleiben intakt. Aber die Unterklasse ändert ihr Verhalten. Jede Instanz der Unterklasse erhält automatisch alle Eigenschaften, Felder und Methoden der Basisklasse – und all das wird allein durch den Doppelpunkt bewirkt. Deswegen sollten Sie den Pfeil in Ihrem Diagramm so auszeichnen, dass er ein Teil der Unterklasse ist und auf die Basisklasse zeigt, die er erweitert.

Spitzen Sie Ihren Bleistift

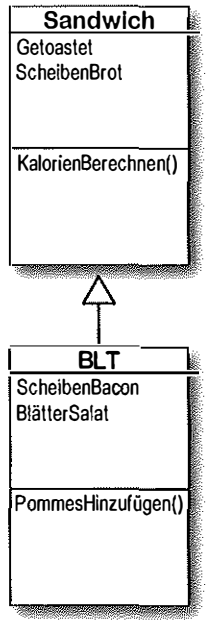
Werfen Sie einen Blick auf diese Klassenmodelle und -deklarationen und kreisen Sie dann die Anweisungen ein, die nicht funktionieren.




```
public class Luftfahrzeug {
    public double Fluggeschwindigkeit;
    public double Höhe;
    public void Abheben() { ... };
    public void Landen() { ... };
}

public class LöschFlugzeug : Luftfahrzeug {
    public double WasserKapazität;
    public void WasserAufnehmen() { ... };
}

public void LöschAuftrag() {
    LöschFlugzeug einLöschFlugzeug = new LöschFlugzeug();
    new LöschFlugzeug.WasserKapazität = 500;
    Luftfahrzeug.Höhe = 0;
    einLöschFlugzeug.Abheben();
    einLöschFlugzeug.Fluggeschwindigkeit = 192.5;
    einLöschFlugzeug.WasserAufnehmen();
    Luftfahrzeug.Landen();
}
```



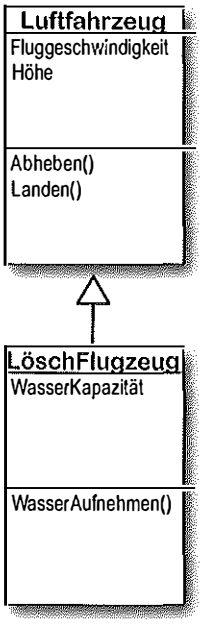

```
public class Sandwich {
    public boolean Getoastet;
    public int ScheibenBrot;
    public int KalorienBerechnen() { ... }
}

public class BLT : Sandwich {
    public int ScheibenBacon;
    public int BlätterSalat;
    public int PommesHinzufügen() { ... }
}

public BLT BLTBestellen() {
    BLT einSandwich = new BLT();
    BLT.Getoastet = true;
    Sandwich.ScheibenBrot = 3;
    einSandwich.PommesHinzufügen();
    einSandwich.ScheibenBacon += 5;
    MessageBox.Show("Mein Sandwich hat "
       + einSandwich.KalorienBerechnen + " Kalorien".);
    return einSandwich;
}
```

Spitzen Sie Ihren Bleistift
Lösung

Werfen Sie einen Blick auf diese Klassenmodelle und -deklarationen und kreisen Sie dann die Anweisungen ein, die nicht funktionieren.



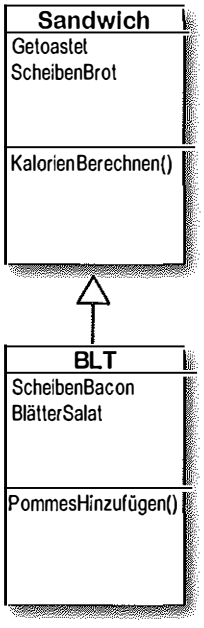

```
public class Luftfahrzeug {
    public double Fluggeschwindigkeit;
    public double Höhe;
    public void Abheben() { ... };
    public void Landen() { ... };
}

public class LöschFlugzeug : Luftfahrzeug {
    public double WasserKapazität;
    public void WasserAufnehmen() { ... };
}

public void LöschAuftrag() {
    LöschFlugzeug einLöschFlugzeug = new LöschFlugzeug();
    new LöschFlugzeug.WasserKapazität = 500;
    Luftfahrzeug.Höhe = 0;
    einLöschFlugzeug.Abheben();
    einLöschFlugzeug.Fluggeschwindigkeit = 192.5;
    einLöschFlugzeug.WasserAufnehmen();
    Luftfahrzeug.Landen();
}
```

So wird das Schlüsselwort new nicht verwendet.

Diese Anweisungen nutzen den Klassennamen statt der Instanz. einLöschFlugzeug.




```
public class Sandwich {
    public boolean Getoastet;
    public int ScheibenBrot;
    public int KalorienBerechnen() { ... }
}

public class BLT : Sandwich {
    public int ScheibenBacon;
    public int BlätterSalat;
    public int PommesHinzufügen() { ... }
}

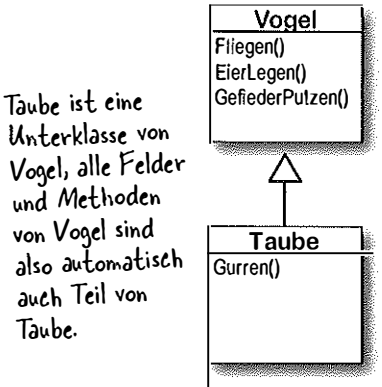
public BLT BLTBestellen() {
    BLT einSandwich = new BLT();
    BLT.Getoastet = true;
    Sandwich.ScheibenBrot = 3;
    einSandwich.PommesHinzufügen();
    einSandwich.ScheibenBacon += 5;
    MessageBox.Show("Mein Sandwich hat "
      + einSandwich.KalorienBerechnen + " Kalorien".);
    return einSandwich;
}
```

Diese Eigenschaften sind Teil der Instanz, aber die Anweisungen versuchen, sie fälschlich über den Klassennamen aufzurufen.

KalorienBerechnen() ist eine Methode, wird hier aber ohne die Klammern nach dem Methodennamen angegeben.

Wir wissen, dass Vererbung die Felder, Eigenschaften und Methoden der Basisklasse den Unterklassen hinzufügt ...

Vererbung ist einfach, wenn Ihre Unterklasse ***alle*** Felder, Eigenschaften und Methoden der Basisklasse benötigt.



Taube ist eine Unterklasse von Vogel, alle Felder und Methoden von Vogel sind also automatisch auch Teil von Taube.

```
public class Vogel {
   public void Fliegen() {
    // Code, der den Vogel fliegen lässt.
   }

   public void EierLegen() { ... };

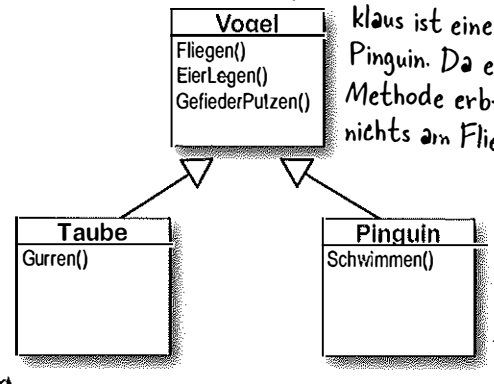
   public void GefiederPutzen() { ... };
}


public class Taube : Vogel {
   public void Gurren() { ... }
}


public class Pinguin : Vogel {
   public void Schwimmen() { ... }
}
```

... aber manche Vögel fliegen nicht!

Was tun Sie, wenn Ihre Basisklasse eine Methode hat, die Ihre Unterklasse ***modifizieren*** muss?



klaus ist eine Instanz von Pinguin. Da er die Fliegen()-Methode erbt, hindert ihn nichts am Fliegen.

```
public void VogelSimulator() {
        Taube hanne = new Taube();
        Pinguin klaus = new Pinguin();
        hanne.Fliegen();
        hanne.Gurren();
        klaus.Fliegen();
}
```

Taube und Pinguin erben beide von Vogel, erhalten also jeweils die Methoden Fliegen(), EierLegen(), und GefiederPutzen().

Pinguin-Objekte sollten nicht fliegen können! Aber die Klasse Pinguin erbt dieses Verhalten von Vogel, und Sie haben plötzlich fliegende Pinguine. Was sollen wir tun?

Tauben fliegen, legen Eier und putzen ihr Gefieder, es gibt also keine Probleme, wenn die Klasse Taube von Vogel erbt.



KOPF-NUSS

Wäre das Ihr Code für einen Vogel-Simulator, was würden Sie tun, um zu verhindern, dass Pinguine fliegen können?

Eine Unterklasse kann Methoden überschreiben, um geerbte Methoden zu ändern oder zu ersetzen

Manchmal haben Sie eine Klasse, die Sie gern die *meisten* Verhalten einer Basisklasse erben lassen möchten, aber *nicht alle*. Wenn Sie die Verhalten überschreiben möchten, die eine Klasse geerbt hat, können Sie diese Methoden **überschreiben**.

1. **Fügen Sie der Basisklassenmethoden das Schlüsselwort virtual hinzu.** Eine Unterklasse kann nur Methoden überschreiben, die mit dem Schlüsselwort **virtual** markiert sind. Dieses sagt C#, dass es Überschreibungen der Methode zulassen soll.

Fügen Sie der Methode Fliegen() das Schlüsselwort virtual hinzu, sagt das C#, dass Unterklassen sie überschreiben dürfen.

```
public class Vogel {
    public virtual void Fliegen() {
        // Code, der den Vogel fliegen lässt.
    }
}
```

2. **Fügen Sie der abgeleiteten Klasse eine Methode gleichen Namens hinzu.** Sie müssen genau die gleiche Signatur verwenden – d.h. den gleichen Rückgabewert und die gleichen Parameter–, und Sie müssen in der Deklaration das Schlüsselwort **override** verwenden.

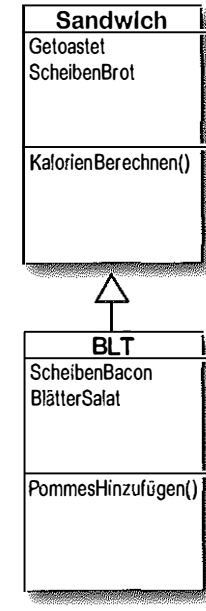
Um Fliegen() zu überschreiben, müssen Sie der Unterklasse eine identische Methode hinzufügen und das Schlüsselwort override angeben.

```
public class Pinguin : Vogel {
    public override void Fliegen() {
        MessageBox.Show("Pinguine fliegen nicht!");
    }
}
```

Überschreiben Sie eine Methode, muss die neue Methode die gleiche Signatur haben wie die Methode in der Basisklasse, die sie überschreibt. Hier heißt das, dass sie Fliegen heißen und void zurückliefern muss und keine Parameter haben darf.

Nutzen Sie das Schlüsselwort override, um einer Unterklasse eine Methode hinzuzufügen, die eine geerbte Methode ersetzt. Damit das möglich ist, muss die Methode in der Basisklasse als virtual markiert sein.

Überall dort, wo Sie die Basisklasse verwenden können, können Sie stattdessen ein Unterklasse verwenden



Zu den nützlichsten Dingen, die Sie mit Vererbung anstellen können, gehört es, dass Sie eine Unterklasse anstelle der Basisklasse verwenden können, von der sie erbt. Nimmt Ihre `Rezept()`-Methode ein `Käse`-Objekt, können Sie ihr stattdessen eine Instanz der Klasse `Parmesan` übergeben, die von `Käse` erbt. `Rezept()` hat allerdings nur Zugriff auf die Felder, Eigenschaften und Methoden, die Teil der Klasse `Käse` sind – die Methode hat keinen Zugriff auf Dinge, die `Parmesan`-spezifisch sind.

1. Nehmen wir an, wir haben eine Methode zur Analyse von Sandwich-Objekten:

```
public void SandwichAnalyse(Sandwich exemplar) {
    int kalorien = exemplar.KalorienBerechnen();
    DiätPlanAktualisieren(kalorien);
    BrotBerechnen(exemplar.ScheibenBrot, exemplar.Getoastet);
}
```

2. Sie könnten der Methode ein Sandwich übergeben – aber Sie könnten ihr auch ein BLT übergeben. Da BLT eine *Art* Sandwich ist, haben wir die Klasse so eingerichtet, dass Sie von Sandwich erbt.

```
public button1_Click(object sender, EventArgs e) {
    BLT einBLT = new BLT();
    SandwichAnalyse(einBLT);
}
```

Das sehen wir uns im nächsten Kapitel ausführlicher an.

3. Sie können jederzeit im Klassendiagramm nach ***unten*** gehen – eine Referenzvariable kann immer eine Instanz einer ihrer Subklassen zugewiesen werden. Aber Sie können im Klassendiagramm nie nach ***oben*** gehen.

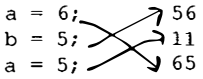
```
public button2_Click(object sender, EventArgs e) {
    Sandwich einSandwich = new Sandwich();
    BLT einBLT = new BLT();
    Sandwich irgendeinSandwich = einBLT;
    BLT anderesBLT = einSandwich;     // <--- KOMPILIERT NICHT!!!
}
```

Sie können einBLT einer Sandwich-Variablen zuweisen, weil BLT eine Art Sandwich ist.

Aber einSandwich kann nicht einer BLT-Variablen zugewiesen werden, weil nicht jedes Sandwich ein BLT ist! Deswegen führt diese letzte Zeile zu einem Fehler.



vermischte Nachrichten



Unten ist ein kurzes C#-Programm abgedruckt. Ein Block des Programms fehlt! Ihre Herausforderung ist es, die Code-Kandidaten (auf der linken Seite) den Ausgaben zuzuordnen – was in den Fenstern steht, die das Programm einblendet –, die Sie sehen würden, wenn die Blöcke eingefügt würden. Nicht alle Ausgabezeilen werden verwendet, und einige können mehrfach verwendet werden. Ziehen Sie Linien, die die Kandidaten-Codeblöcke mit den jeweiligen Ausgaben verbinden.

Anweisungen:

1. **Füllen Sie die vier Lücken im Code aus.**
2. **Ordnen Sie den Code-Kandidaten die jeweilige Ausgabe zu.**

```
public class A {
  public int ivar = 7;
  public __________ string m1() {
    return "m1 von A, ";
  }
  public string m2() {
    return "m2 von A, ";
  }
  public __________ string m3() {
    return "m3 von A, ";
  }
}

public class B : A {
  public __________ string m1() {
    return "m1 von B, ";
  }
}
```

```
public class C : B {
  public __________ string m3() {
    return "m3 von C, " + (ivar + 6);
  }
}
public class Mixed5 {
  public static void Main(string[] args) {
    A a = new A();
    B b = new B();
    C c = new C();
    A a2 = new C();
    string q = "";
    ____________________
    |                  |
    ____________________
    System.Windows.Forms.MessageBox.Show(q);
  }
}
```

Hier ist der Einstiegspunkt für das Programm – es zeigt kein Formular an, sondern blendet nur eine Meldung ein.

Hier kommen die Codeblöcke hin. (drei Zeilen)

Code-Kandidaten:

```
q += b.m1();
q += c.m2();
q += a.m3();

q += c.m1();
q += c.m2();
q += c.m3();

q += a.m1();
q += b.m2();
q += c.m3();

q += a2.m1();
q += a2.m2();
q += a2.m3();
```

Ausgaben:

```
m1 von A, m2 von A, m3 von C, 6

m1 von B, m2 von A, m3 von A,

m1 von A, m2 von B, m3 von A,

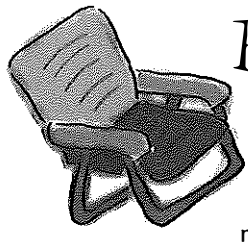
m1 von B, m2 von A, m3 von C, 13

m1 von B, m2 von C, m3 von A,

m1 von B, m2 von A, m3 von C, 6

m1 von A, m2 von A, m3 von C, 13
```

(Geben Sie das nicht einfach in die IDE ein – Sie lernen eine Menge mehr, wenn Sie das auf Papier ausarbeiten!)



Pool-Puzzle

Sie haben die ***Aufgabe***, die leeren Zeilen im Code mit den Codeschnipseln aus dem Pool zu füllen. Einzelne Schnipsel **können** mehrfach verwendet werden, und Sie werden nicht alle Schnipsel benötigen. Das ***Ziel*** ist es, eine Klasse zu erstellen, die sich kompilieren lässt und die gezeigte Ausgabe erzeugt. Lassen Sie sich nicht täuschen – das ist schwerer, als es scheint.

```
public class Ruderboot..................... {
   public ..............BootRudern() {
      return "Pull weg";
   }
}
```

```
public class.................. {
   private int.................. ;
  ..............void.....................(..............) {
      länge = 1;
   }
   public int LängeAbrufen() {
      ..................... ;
   }
   public ..................... Bewegen() {
      return   "................. ";
   }
}
```

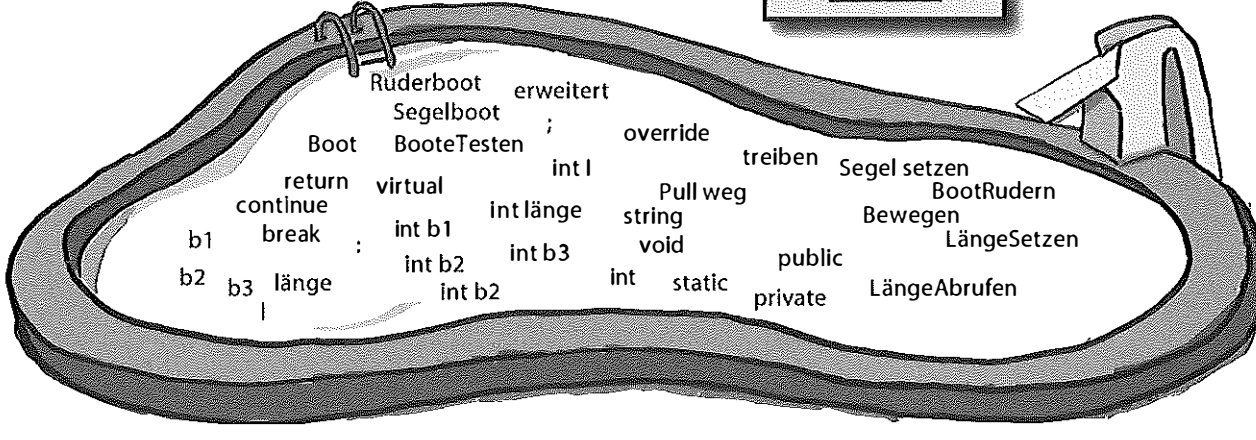
```
public class BooteTesten {
   ..................... Main(){
     ..............xyz = "";
     ..............b1 = new Boot();
      Segelboot b2 = new.................. ();
      Ruderboot.......... = new Ruderboot();
      b2.LängeSetzen(32);
      xyz = b1. ..........();
      xyz += b3. ..........();
      xyz += ..........Bewegen();
      System.Windows.Forms.MessageBox.Show(xyz);
   }
}
```

Tipp: Das ist der Einstiegspunkt für das Programm.

```
public class ..................: Boot {
   public ..............................() {
      return "..................";
   }
}
```

Ausgabe:







LÖSUNGEN ZU DEN ÜBUNGEN

vermischte Nachrichten

```
a = 6;   → 56
b = 5;   → 11
a = 5;   → 65
```

```
public class A {
  public virtual string m1() {
...
  public virtual string m3() {
}
```

```
public class B : A {
  public override string m1() {
...
public class C : B {
  public override string m3() {
```

```
q += b.m1();
q += c.m2();
q += a.m3();
```
→ m1 von A, m2 von A, m3 von C, 6

```
q += c.m1();
q += c.m2();
q += c.m3();
```
→ m1 von B, m2 von A, m3 von A,

```
q += a.m1();
q += b.m2();
q += c.m3();
```
→ m1 von A, m2 von B, m3 von C, 6

```
q += a2.m1();
q += a2.m2();
q += a2.m3();
```
→ m1 von B, m2 von A, m3 von C, 13

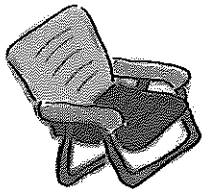
→ m1 von B, m2 von C, m3 von A,

→ m1 von A, m2 von B, m3 von A,

→ m1 von B, m2 von A, m3 von C, 6

→ m1 von A, m2 von A, m3 von C, 13

Pool-Puzzle, Lösung



```
public class Ruderboot : Boot {
   public string BootRudern() {
      return "Pull weg";
   }
}
public class Boot {
   private int länge;
   public void LängeSetzen ( int l ) {
      länge = l;
   }
   public int LängeAbrufen() {
      return länge;
   }
   public virtual string Bewegen() {
      return " treiben ";
   }
}
```

```
public class BooteTesten {
  public static void Main(){
    string xyz = "";
    Boot b1 = new Boot();
    Segelboot b2 = new Segelboot ();
    Ruderboot b3 = new Ruderboot();
    b2.LängeSetzen(32);
    xyz = b1.Bewegen();
    xyz += b3.Bewegen();
    xyz += b2.Bewegen();
    System.Windows.Forms.MessageBox.Show(xyz);
  }
}
public class Segelboot : Boot {
  public override string Bewegen() {
    return " Segel setzen ";
  }
}
```

Es gibt keine Dummen Fragen

F: Was ist mit dem Einstiegspunkt, den Sie im Pool-Puzzle hervorgehoben haben – bedeutet das, dass ich ein Programm ohne ein Form1-Formular haben kann?

A: Ja. Erstellen Sie ein neues Windows-Form-Anwendung-Projekt, erzeugt die IDE für Sie alle Dateien für das Projekt. Dazu zählen die Dateien Program.cs (die eine statische Klasse mit einem Einstiegspunkt enthält) und Form1.cs (die ein leeres Formular namens Form1 enthält).

Probieren Sie Folgendes: Erstellen Sie statt eines Windows-Form-Anwendung-Projekts ein leeres Projekt, indem Sie »Leeres Projekt« statt »Windows-Form-Anwendung« wählen, wenn Sie in der IDE ein neues Projekt erstellen. Fügen Sie ihm dann im Projektmappen-Explorer eine Klasse hinzu und geben Sie alles aus der Pool-Puzzle-Übung ein. Da Ihr Programm eine MessageBox verwendet, müssen Sie einen **Verweis** hinzufügen, indem Sie im Projektmappen-Explorer auf »Verweise« klicken, »Verweis hinzufügen« auswählen und dann auf der Registerkarte .NET System.Windows.Forms suchen. (Das ist eine andere Sache, die die IDE automatisch für Sie erledigt, wenn Sie eine Windows-Form-Anwendung erstellen.) Wählen Sie dann im Projekt-Menü »Eigenschaften« und wählen Sie den Projekttyp »Windows-Form-Anwendung«.

Führen Sie es aus ... Sie werden die Ergebnisse sehen! Glückwunsch, Sie haben gerade Ihr erstes C#-Programm aus dem Nichts geschaffen.

↑

Wenn Sie eine kleine Auffrischung zur Main()-Methode und zum Einstiegspunkt brauchen, blättern Sie zurück zu Kapitel 2!

F: Kann ich von der Klasse erben, die den Einstiegspunkt enthält?

A: Ja. Der Einstiegspunkt ***muss*** eine statische Methode sein, aber diese Methode ***muss sich nicht*** in einer statischen Klasse befinden. (Erinnern Sie sich: Das Schlüsselwort `static` bedeutet, dass die Klasse nicht instantiiert werden kann, sondern ihre Methoden verfügbar sind, sobald das Programm startet. Im Pool-Puzzle-Programm können Sie BooteTesten. Main() also von jeder anderen Methode aus aufrufen, ohne eine Referenzvariable zu deklarieren oder mithilfe der `new`-Anweisung ein Objekt zu instantiieren.)

F: Ich verstehe immer noch nicht, warum man diese Methoden »virtuell« nennt– mir scheinen die doch ziemlich real!

A: Die Bezeichnung »virtuell« hat etwas damit zu tun, wie .NET hinter den Kulissen mit virtuellen Methoden umgeht. Es nutzt einen Mechanismus, der als virtuelle Methodentabelle (oder vtable) bezeichnet wird. Das ist eine Tabelle, über die .NET nachhält, welche Methoden geerbt werden und welche überschrieben. Machen Sie sich keine Gedanken – Sie müssen nicht wissen, wie das funktioniert, um mit virtuellen Methoden zu arbeiten!

F: Was meinen Sie damit, dass man im Klassendiagramm nur nach oben und nicht nach unten gehen kann?

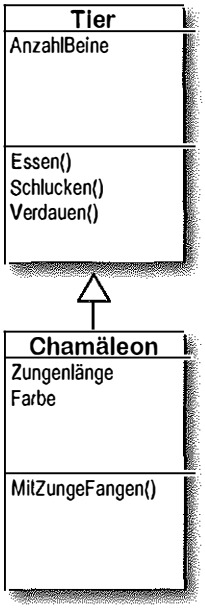
A: Wenn Sie ein Diagramm haben, in dem eine Klasse über einer anderen steht, ist die Klasse, die weiter oben steht, *abstrakter* als die Klasse, die weiter unten steht. Spezifischere oder konkrete Klassen (wie Hemd oder Auto) erben von abstrakteren (wie Kleidung oder Fahrzeug). Betrachten Sie das auf diese Weise, werden Sie verstehen, dass ein Pkw, ein Minibus oder ein Motorrad in Ordnung ist, wenn Sie bloß irgendein Fahrzeug benötigen. Aber wenn Sie einen Pkw brauchen, bringt Ihnen ein Motorrad nichts.

Vererbung funktioniert ganz genau so. Haben Sie eine Methode mit einem Parameter des Typs `Fahrzeug` und eine `Motorrad`-Klasse, die von der Klasse `Fahrzeug` abgeleitet ist, können Sie der Methode eine Instanz von `Motorrad` übergeben. Aber nimmt die Methode ein `Motorrad` als Parameter, können Sie ihr kein `Fahrzeug`-Objekt übergeben, weil das auch eine `MiniBus`-Instanz sein könnte. Andernfalls weiß C# nicht, was es tun soll, wenn die Methode versucht, auf die Eigenschaft `Lenkergriffe` zuzugreifen!

Sie können einer Methode immer eine Instanz einer Unterklasse der Klasse übergeben, die sie als Parametertyp erwartet.

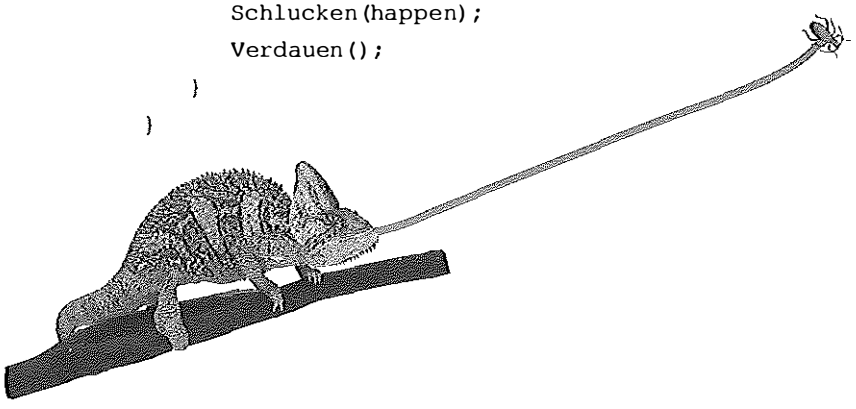
Basisklassenzugriff über das Schlüsselwort base

Wenn Sie eine Methode oder Eigenschaft aus der Basisklasse überschreiben, möchten Sie gelegentlich dennoch darauf zugreifen. Glücklicherweise können wir **base** nutzen, um auf jede Methode in der Basisklasse zuzugreifen.



1. Alle Tiere essen, die Klasse `Tier` hat also eine `Essen()`-Methode, die ein `Nahrung`-Objekt als Parameter übernimmt.

```
public class Tier {
   public virtual void Essen(Nahrung happen) {
      Schlucken(happen);
      Verdauen();
   }
}
```



2. Chamäleons essen, indem sie Nahrung mit ihren Zungen fangen. Die Klasse `Chamäleon` erbt also von `Tier`, überschreibt aber `Essen()`.

```
public class Chamäleon : Tier {
   public override void Essen(Nahrung happen) {
      MitZungeFangen(happen);
      Schlucken(happen);
      Verdauen();
   }
}
```

Wie jedes andere Tier muss das Chamäleon die Nahrung schlucken und verdauen. Müssen wir den Code deswegen wirklich doppelt haben?

3. Anstatt den Code zu verdoppeln, können wir das Schlüsselwort **base** einsetzen, um die Methode aufzurufen, die überschrieben wurde. Jetzt haben wir Zugriff auf die alte und die neue Version von `Essen()`.

```
public class Chamäleon : Tier {
   public override void Essen(Nahrung happen) {
      MitZungeFangen(happen);
      base.Essen(happen);
   }
}
```

Diese Zeile ruft die Essen()-Methode in der Basisklasse auf, die Chamäleon erweitert.

Hat die Basisklasse einen Konstruktor, braucht ihn die Unterklasse auch

Wenn Ihre Klasse einen Konstruktor hat, der Parameter erwartet, muss jede Klasse, die sie erweitert, diesen Konstruktor aufrufen. Der Unterklassenkonstruktor kann andere Parameter haben als der Basisklassenkonstruktor.

Hängen Sie diese zusätzliche Zeile an das Ende der Deklaration des Konstruktors Ihrer Unterklasse an, um C# zu sagen, dass der Basisklassenkonstruktor aufgerufen werden muss, wenn die Unterklasse instantiiert wird.

```
public class Unterklasse : Basisklasse {
    public Unterklasse(Parameterliste)
        : base(Parameterliste der Basisklasse) {
        // Erst wird der Basisklassenkonstruktor aufgerufen,
        // dann alle hier folgenden Anweisungen.
    }
}
```

Hier ist der Konstruktor für die Unterklasse.

Der Basisklassenkonstruktor wird vor dem Unterklassenkonstruktor ausgeführt

Aber verlassen Sie sich nicht einfach darauf – probieren Sie es aus!

✶Tun Sie das!

1 Erstellen Sie eine Basisklasse, deren Konstruktor eine Meldung anzeigt.
Fügen Sie dann einem Formular einen Button hinzu, der diese ***Basisklasse*** instantiiert und eine Meldung anzeigt:

```
public class EineBasisklasse {
    public EineBasisklasse(string basisParameter) {
        MessageBox.Show("Hier ist die Basisklasse: " + basisParameter);
    }
}
```

Das ist der Parameter, den der Basisklassenkonstruktor braucht.

2 Fügen Sie eine Unterklasse hinzu, ohne den Konstruktror aufzurufen.
Fügen Sie einem Formular dann einen Button hinzu, der diese ***Unterklasse*** instantiiert und eine Meldung anzeigt:

```
public class EineUnterklasse : EineBasisklasse{
    public EineUnterklasse(string basisParameter, int andererWert) {
        MessageBox.Show("Hier ist die Unterklasse: " + basisParameter
            + " und " + andererWert);
    }
}
```

Wählen Sie in der IDE Erstellen/Projektmappe erstellen, liefert dieser Code einen Fehler.

Achten Sie auf diesen etwas kryptischen Fehler. Er sagt, dass Ihre Unterklasse den Basisklassenkonstruktor nicht aufruft.

1 "EineBasisklasse" enthält keinen Konstruktor, der 0-Argumente akzeptiert.

3 Beheben Sie den Fehler, indem Sie den Basisklassenkonstruktor aufrufen.
Instantiieren Sie die Unterklasse und ***schauen Sie, in welcher Reihenfolge*** die beiden Meldungen erscheinen!

```
public class EineUnterklasse : EineBasisklasse{
    public EineUnterklasse(string basisParameter, int andererWert)
        : base(basisParameter)
    {
        // Der Rest bleibt gleich.
```

So senden wir der Basisklasse den Parameter, den ihr Konstruktor braucht.

So sagen Sie C#, dass der Basisklassenkonstruktor aufgerufen werden muss. Die Parameterliste zeigt, was diesem Konstruktor übergeben werden muss, damit der Fehler verschwindet!

Jetzt können Sie die Arbeit für Kathrin abschließen!

Als wir Kathrin verlassen haben, hatten wir ihrem Programm gerade Geburtstagsfeiern hinzugefügt. Sie sollten für sie eine zusätzliche Gebühr von 100 € für Partys mit mehr als 12 Personen einführen. Es schien, als müssten Sie den gleichen Code für beide Klassen separat schreiben. Nachdem Sie wissen, wie man Vererbung nutzt, können Sie beide Klassen von der gleichen Basisklasse erben lassen, die den gesamten Code enthält, den sich beide teilen.

Abendessen
Personenanzahl Dekokosten GetränkeKostenProPerson
DekokostenBerechnen() KostenBerechnen() TrockenerAbendSetzen()

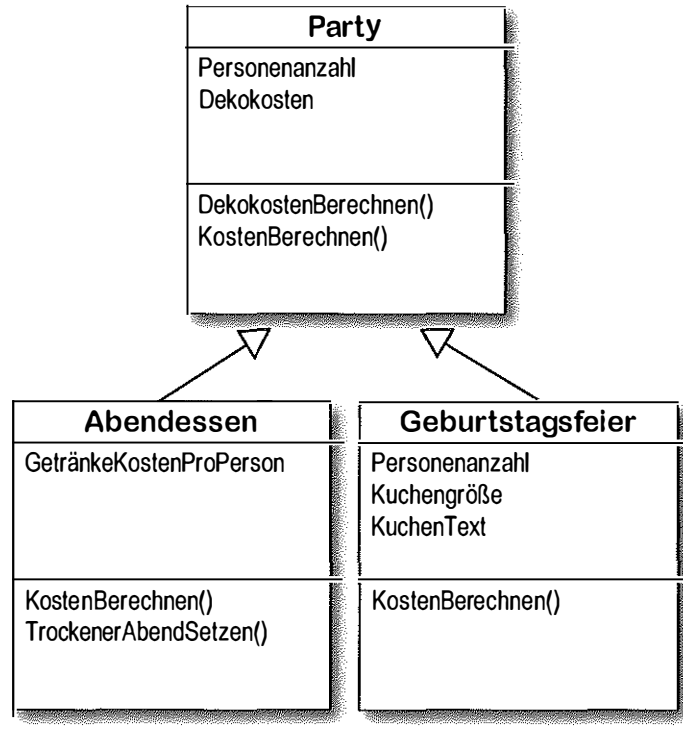
Geburtstagsfeier
Personenanzahl Dekokosten Kuchengröße KuchenText
DekokostenBerechnen() KostenBerechnen()



Wenn wir geschickt vorgehen, sollten wir die Klassen ändern können, ohne dass wir das Formular ändern müssen!

1. **Erstellen wir das neue Klassenmodell.**

Wir haben immer noch die Klassen Abendessen und Geburtstagsfeier, aber jetzt erben beide von der Klasse Party. Sie sollen weiterhin genau die gleichen Methoden, Eigenschaften und Felder haben, damit wir das Formular nicht ändern müssen. Aber einige der Methoden, Eigenschaften und Felder werden in die Basisklasse Party verschoben, und ein paar davon müssen wir eventuell überschreiben.



2 Erstellen Sie die Basisklasse Party.

Erstellen Sie die Klasse Party – achten Sie darauf, dass sie öffentlich ist. Sie müssen sich die Eigenschaften und Methoden im Klassendiagramm genau ansehen und herausfinden, was Sie aus Abendessen und Geburtstagsfeier in Party verschieben müssen.

- Verschieben Sie die Eigenschaften Personenanzahl und Dekokosten in Party, damit sie in Abendessen und Geburtstagsfeier gleich sind.
- Machen Sie das Gleiche mit den Methoden DekokostenBerechnen() und KostenBerechnen(). Sollten diese Methoden irgendwelche privaten Felder verwenden, müssen Sie auch diese verschieben. (Denken Sie daran, dass Unterklassen nur die als **public** deklarierten Felder sehen – verschieben Sie ein privates Feld in Party, haben Abendessen und Geburtstagsfeier keinen Zugriff mehr darauf.)
- Außerdem brauchen Sie einen Konstruktor. Sehen Sie sich die Konstruktoren für Geburtstagsfeier und Abendessen genau an – haben sie Gemeinsamkeiten, sollten diese in die Basisklasse verschoben werden.
- **Fügen Sie die Zusatzgebühr von 100 €** für Partys mit mehr als 12 Personen hinzu. Deswegen machen wir all das schließlich! Diese haben Geburtstagsfeiern und Abendessen gemeinsam. Sie gehört also in Party.

3 Lassen Sie Abendessen von Party erben.

Da Party jetzt eine Menge der Dinge macht, die Abendessen macht, können Sie die Überschneidungen entfernen und in Abendessen nur die Teile zurückbehalten, die für Abendessen spezifisch sind.

- Prüfen Sie, ob der Konstruktor ordentlich funktioniert. Macht er etwas, das der Party-Konstruktor nicht macht? Wenn ja, sollten Sie das belassen und dann alles andere dem Basisklassenkonstruktor überantworten.
- Die gesamte Logik für den trockenen Abend sollte in Abendessen bleiben.
- Sie müssen mindestens eine Methode überladen, die etwas erledigt, das für Abendessen eigentümlich ist. »Überladen« bedeutet, dass Methoden zwar den gleichen Namen, aber eine unterschiedliche Parameterliste haben (und wird in Kapitel 8 ausführlicher besprochen).

4 Lassen Sie Geburtstagsfeier von Party erben.

Machen Sie das Gleiche für Geburtstagsfeier – überlassen Sie alles, was nicht geburtstagsfeierspezifisch ist, der Basisklasse und lassen Sie alles, was geburtstagsfeierspezifisch ist, in Geburtstagsfeier.

- Was macht der Geburtstagsfeier-Konstruktor, was nicht Teil von Party ist?
- Um die Kuchenkosten müssen Sie sich innerhalb von Geburtstagsfeier kümmern. Dazu müssen Sie eine Methode und eine Eigenschaft überschreiben.
- Ja, Sie können auch Eigenschaften überschreiben! Das entspricht einfach dem Überschreiben einer Methode. Setzen Sie den Wert von `base.Personenanzahl`, wird der Eigenschafts-Setter in der Basisklasse aufgerufen. Sie müssen das Schlüsselwort `base` sowohl beim Abrufen als auch beim Setzen des Werts verwenden.



LÖSUNGEN ZU DEN ÜBUNGEN

Werfen Sie einen Blick darauf – Sie haben die Klassen Abendessen und Geburtstagsfeier so geändert, dass sie von der gemeinsamen Basisklasse Party abgeleitet sind. Dann konnten Sie die Änderungen an der Kostenrechnung vornehmen, die für das Hinzufügen der Gebühr von 100M erforderlich war, und mussten das Formular überhaupt nicht ändern. Klasse!

```
public class Party
{
    const int EssenskostenProPerson = 25;
    private bool exklusiveDekoration;
    public decimal Dekokosten = 0;

    public Party(int personenanzahl, bool exklusiveDekoration) {
        this.exklusiveDekoration = exklusiveDekoration;
        this.Personenanzahl = personenanzahl;
    }

    private int personenanzahl;
    public virtual int Personenanzahl {
        get { return personenanzahl; }
        set {
            personenanzahl = value;
            DekokostenBerechnen(exklusiveDekoration);
        }
    }

    public void DekokostenBerechnen(bool exklusiv) {
        exklusiveDekoration = exklusiv;
        if (exklusiv)
            Dekokosten = (Personenanzahl * 15.00M) + 50M;
        else
            Dekokosten = (Personenanzahl * 7.50M) + 30M;
    }

    public virtual decimal KostenBerechnen() {
        decimal GesamtKosten = Dekokosten + (EssenskostenProPerson * Personenanzahl);
        if (Personenanzahl > 12)
        {
            GesamtKosten += 100M;
        }
        return GesamtKosten;
    }
}
```

Dieser Code wurde einfach aus den Klassen Abendessen und Geburtstagsfeier in Party verschoben.

Der Party-Konstruktor macht alles, was zuvor in den Konstruktoren für Abendessen und Geburtstagsfeier stand.

Personenanzahl muss virtual deklariert werden, da Geburtstagsfeier sie überschreiben muss (damit eine Änderung der Personenanzahl zu einer Neuberechnung der Kuchengröße führt).

Die Dekorationskosten sind bei Abendessen und Geburtstagsfeiern gleich. Es ist also sinnvoll, diese in Party zu verschieben. So muss kein Code in mehreren Klassen verdoppelt werden.

Die Kostenberechnung muss in einer virtuellen Methode stecken, weil die Geburtstagsfeier sie überschreibt (und auch erweitert, indem sie die Basisklassenmethode überschreibt).

```
public class Geburtstagsfeier : Party {
    public int Kuchengröße;

    public Geburtstagsfeier(int personenanzahl, bool exklusiveDekoration, string kuchenText)
        : base(personenanzahl, exklusiveDekoration) {
        kuchengrößeBerechnen();
        this.KuchenText = kuchenText;
        DekokostenBerechnen(exklusiveDekoration);
    }

    private void kuchengrößeBerechnen() {
        if (Personenanzahl <= 4)
            Kuchengröße = 8;
        else
            Kuchengröße = 16;
    }

    private string kuchenText = "";
    public string KuchenText {
        get { return this.kuchenText; }
        set {
            int maxLänge;
            if (Kuchengröße == 8)
                maxLänge = 16;
            else
                maxLänge = 40;
            if (value.Length > maxLänge) {
                MessageBox.Show("Zu viele Buchstaben für " + Kuchengröße + "cm-Kuchen");
                if (maxLänge > this.kuchenText.Length)
                    maxLänge = this.kuchenText.Length;
                this.kuchenText = kuchenText.Substring(0, maxLänge);
            } else
                this.kuchenText = value;
        }
    }

    public override decimal KostenBerechnen() {
        decimal Kuchenkosten;
        if (Kuchengröße == 8)
            Kuchenkosten = 40M + KuchenText.Length * .25M;
        else
            Kuchenkosten = 75M + KuchenText.Length * .25M;
        return base.KostenBerechnen() + Kuchenkosten;
    }

    public override int Personenanzahl {
        get { return base.Personenanzahl; }
        set  {
            base.Personenanzahl = value;
            kuchengrößeBerechnen();
            this.KuchenText = kuchenText;
        }
    }
}
```

Für einen Großteil der Arbeit stützt sich dieser Konstruktor auf die Basisklasse. Dann ruft er genau wie der alte Geburtstagsfeier-Konstruktor kuchengrößeBerechnen() auf.

Die Methode kuchengrößeBerechnen() ist für Geburtstagsfeiern spezifisch, bleibt also in der Klasse Geburtstagsfeier.

Die Eigenschaft KuchenText verbleibt ebenfalls vollständig in Geburtstagsfeier.

Die Methode KostenBerechnen() muss ebenfalls überschrieben werden, da erst die Kosten des Kuchens berechnet und diese dann den Kosten hinzugefügt werden müssen, die in der KostenBerechnen()-Methode der Klasse Party ermittelt werden.

Die Eigenschaft Personenanzahl muss die in Party überschreiben, weil der Setter die Kuchengröße neu berechnen muss. Der Setter muss base.Personenanzahl aufrufen, damit der Setter in Party ebenfalls ausgeführt wird.

Fortsetzung auf Seite 238.



Lösungen zu den Übungen

Fortsetzung von S. 237

Hier ist die letzte Klasse in Kathrins Lösung. (Es gibt keine Änderungen am Formularcode.)

```
public class Abendessen : Party
{
    public decimal GetränkeKostenProPerson;

    public Abendessen(int personenanzahl, bool trockenerAbend,
                      bool exklusiveDekoration)
        : base(personenanzahl, exklusiveDekoration) {
        TrockenerAbendSetzen(trockenerAbend);
        DekokostenBerechnen(exklusiveDekoration);
    }

    public void TrockenerAbendSetzen(bool trockenerAbend) {
        if (trockenerAbend)
            GetränkeKostenProPerson = 5.00M;
        else
            GetränkeKostenProPerson = 20.00M;
    }

    public decimal KostenBerechnen(bool trockenerAbend) {
        decimal gesamtKosten = base.KostenBerechnen()
                             + (GetränkeKostenProPerson * Personenanzahl);

        if (trockenerAbend)
            return gesamtKosten * .95M;
        else
            return gesamtKosten;
    }
}
```

Dieses öffentliche Feld wird nur bei Abendessen benötigt, nicht bei Geburtstagsfeiern, bleibt also in dieser Klasse.

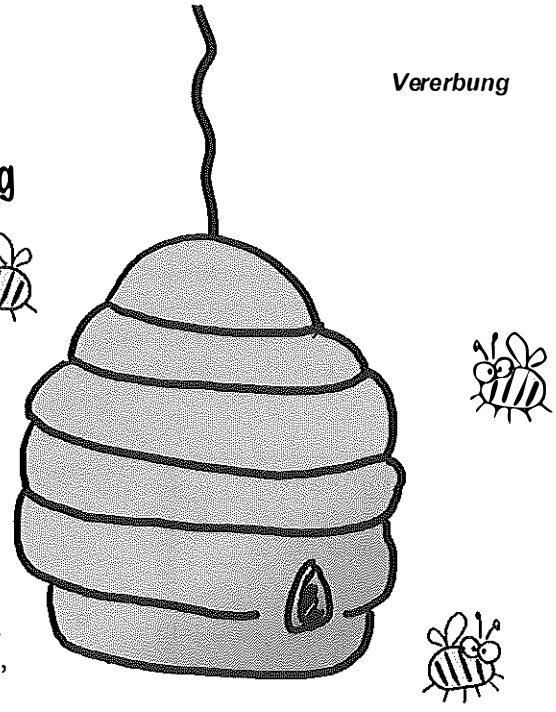
Der neue Konstruktor ruft den Party-Konstruktor und dann TrockenerAbendSetzen() auf, um seine Aufgaben zu erledigen.

Die Methode TrockenerAbendSetzen() bleibt genau gleich.

Abendessen muss KostenBerechnen() überladen – es nutzt die Grundkosten aus Party, fügt dann die Kosten für Getränke hinzu und zieht den Rabatt für den trockenen Abend ab.



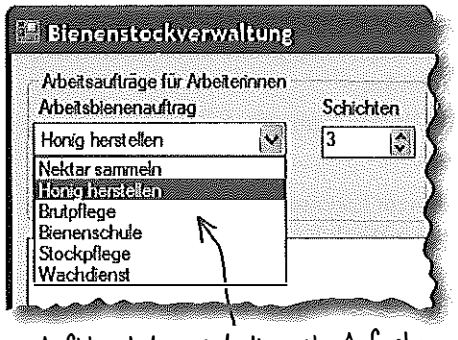
Bauen Sie eine Bienenstockverwaltung



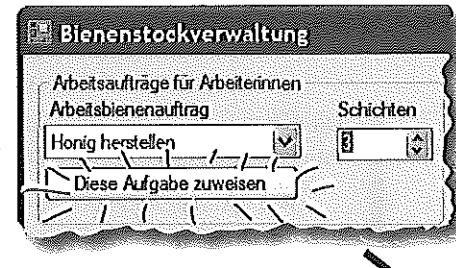
Eine Bienenkönigin braucht Ihre Hilfe! Ihr Bienenstock ist außer Kontrolle, und sie braucht ein Programm, das ihr hilft, ihn zu verwalten. Sie hat einen Bienenstock mit lauter Arbeitsbienen und einem Haufen Aufgaben, die für den Bienenstock erledigt werden müssen. Aber irgendwie hat sie den Überblick darüber verloren, welche Arbeiterin was macht, und ob sie ausreichend Arbeitskräfte hat, um die anstehenden Aufgaben zu bewältigen.

Sie haben die Aufgabe, eine Bienenstockverwaltung zu erstellen, die ihr hilft, ihre Arbeitsbienen zu steuern. Diese soll folgendermaßen funktionieren:

1 **Die Königin weist den Bienen Aufgaben zu.**
Es gibt sechs Aufgaben, die Arbeitsbienen erledigen müssen. Einige wissen, wie man Nektar sammelt und Honig herstellt, andere können den Bienenstock instand halten und auf Feinde achten. Ein paar Bienen können alle Aufgaben erfüllen, die im Bienenstock anfallen. Ihr Programm muss ihr also ein Mittel bieten, jeder Arbeiterin eine Aufgabe zuzuweisen, die sie bewältigen kann.



Diese Aufklappliste zeigt die sechs Aufgaben, die die Arbeiter erledigen können. Die Königin weiß, welche Jobs erledigt werden müssen, und kümmert sich eigentlich nicht darum, welche Arbeiterin diese erledigt. Sie wählt also einfach die Aufgabe aus, die erledigt werden muss – das Programm ermittelt dann, ob eine Arbeiterin frei ist, die die Aufgabe erledigen kann, und weist dieser die Aufgabe zu.

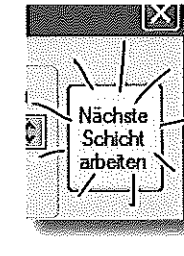


Die Bienen arbeiten in Schichten, und für die meisten Aufgaben sind mehrere Schichten erforderlich. Die Königin gibt also die Anzahl von Schichten ein, die für eine Aufgabe erforderlich sind und klickt auf den Button »Diese Aufgabe zuweisen«.

Ist eine Arbeiterin verfügbar, die die Aufgabe erfüllen kann, weist ihr das Programm die Aufgabe zu und teilt der Königin mit, dass sich darum gekümmert wird.



8 **Sind alle Aufgaben zugewiesen, ist es Zeit, an die Arbeit zu gehen.**
Nachdem die Königin die Arbeitseinteilung erledigt hat, sagt sie den Bienen, dass sie die nächste Schicht abarbeiten sollen, indem sie auf den Button »Nächste Schicht arbeiten« klickt. Das Programm erstellt dann einen Schichtbericht, der ihr sagt, welche Arbeiterin in dieser Schicht gearbeitet hat, welche Aufgabe sie erledigt hat und wie viele weitere Schichten sie daran arbeiten wird.



Erst bauen Sie das Grundsystem

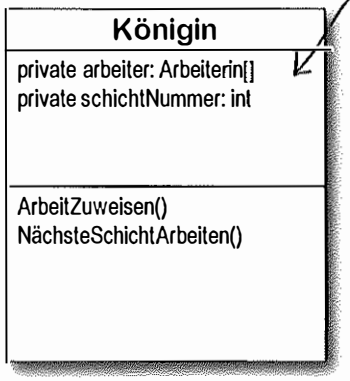




Dieses Projekt hat zwei Teile. Der erste Teil ist eine Art Rückblick, bei dem Sie das Grundsystem für die Verwaltung des Bienenstocks erstellen. Er hat die zwei Klassen Königin und Arbeiterin. Sie erstellen das Formular für das System und verknüpfen es mit den beiden Klassen. Und Sie stellen sicher, dass die Klassen ordentlich gekapselt sind, damit Sie sie leicht ändern können, wenn Sie zum zweiten Teil übergehen.



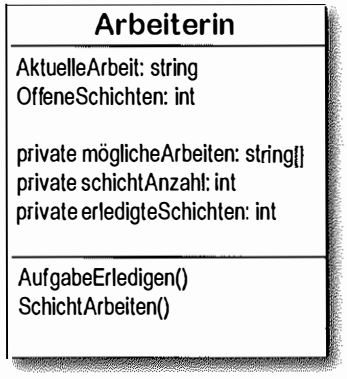
Manchmal führen Klassendiagramme private Felder und Typen auf.



Das Programm hat ein Königin-Objekt, das die zu erledigende Arbeit verwaltet.

- Die Königin nutzt ein Array mit Arbeiterin-Objekten, um die Arbeitsbienen und die ihnen zugewiesenen Aufgaben nachzuhalten. Es ist in einem privaten Arbeiterin[]-Feld namens `arbeiter` gespeichert.
- Das Formular ruft die Methode ArbeitZuweisen() auf, übergibt ihr einen String für die zu erledigende Arbeit und einen int für die Anzahl von Schichten. Sie liefert true, wenn sie eine Arbeiterin findet, der sie die Aufgabe zuweisen kann, oder false, wenn sie keine Arbeiterin für die Aufgabe findet.
- Der »Nächste Schicht arbeiten«-Button des Formulars ruft NächsteSchichtArbeiten() auf, schickt damit die Arbeiterinnen an die Arbeit und liefert einen Schichtbericht. Jede Arbeiterin wird aufgefordert, eine Schicht zu arbeiten. Dann wird der Status der Arbeiterin geprüft, damit dem Schichtbericht eine Zeile hinzugefügt werden kann.

Die Eigenschaften AktuelleArbeit und OffeneSchichten sind schreibgeschützt.



Die Königin nutzt ein Array mit Arbeiterin-Objekten, um die Arbeitsbienen und die ihnen zugewiesenen Aufgaben nachzuhalten.

- AktuelleArbeit ist eine schreibgeschützte Eigenschaft, die der Königin sagt, welche Arbeit die Arbeiterin erledigt (»Wachdienst«, »Stockpflege« usw.). Macht die Arbeiterin nichts, wird der leere String geliefert.
- Die Königin versucht, einer Arbeiterin über die Methode AufgabeErledigen() eine Arbeit zuzuweisen. Erledigt die Arbeiterin diese Aufgabe nicht bereits und ist das eine Arbeit, die sie beherrscht, nimmt sie die Aufgabe an und liefert true, andernfalls liefert sie false.
- Wird die Methode SchichtArbeiten() aufgerufen, arbeiten die Arbeitsbienen eine Schicht. Die Arbeiterin hält nach, wie viele Schichten für die aktuelle Aufgabe verbleiben. Ist die Aufgabe erledigt, wird String für die aktuelle Aufgabe zurückgesetzt, damit eine neue Aufgabe angenommen werden kann.

String.IsNullOrEmpty()

Da die Bienen die aktuelle Aufgabe als String speichern, können sie anhand der Eigenschaft AktuelleArbeit prüfen, ob sie bereits beschäftigt sind – diese entspricht dem leeren String, wenn sie auf die nächste Aufgabe warten. In C# können Sie das leicht mit String.IsNullOrEmpty(AktuelleArbeit) prüfen. Diese Anweisung liefert true, wenn AktuelleArbeit leer oder null ist, und andernfalls false.

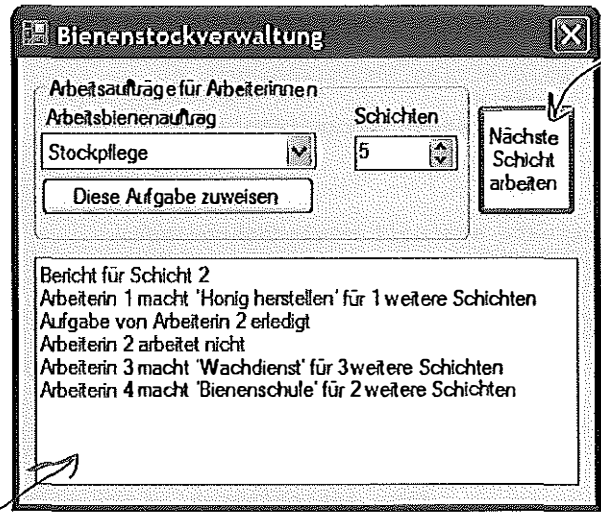


Eine Bienenkönigin braucht Ihre Hilfe! Wenden Sie an, was Sie zu Klassen und Objekten gelernt haben, um eine Bienenstockverwaltung aufzubauen, mit der sie ihre Arbeitsbienen anleiten kann.

1 Erstellen Sie das Formular.

Das Formular ist recht einfach – die gesamte Logik steckt in den Klassen Königin und Arbeiterin. Das Formular hat ein privates Königin-Feld, und zwei Buttons rufen ihre Methoden ArbeitZuweisen() und NächsteSchichtArbeiten() auf. Sie brauchen eine ComboBox für die Bienenaufgaben (seine Elemente finden Sie auf der vorangegangenen Seite), ein NumericUpDown, zwei Buttons und ein mehrzeiliges Textfeld für den Schichtbericht. Außerdem brauchen Sie den Formularkonstruktor unter der Abbildung.

Das ist eine ComboBox namens bienenAufgabe. Die Liste setzen Sie über die Eigenschaft Items. Setzen Sie die Eigenschaft DropDownStyle auf >>DropDownList<<, damit nur Elemente aus der Liste ausgewählt werden können. Verwenden Sie für die Schichten ein NumericUpDown namens schichten.



Der Button nächste-Schicht ruft die NächsteSchichtArbeiten()-Methode der Königin auf, die einen String mit dem Bericht zurückliefert.

Der Bericht hat folgende Form: Er beginnt mit einer Schichtnummer und meldet dann, was die einzelnen Bienen machen. Nutzen Sie die Escape-Sequenz >>\r\n<<, um in einen String einen Zeilenumbruch einzufügen.

Nennen Sie dieses Textfeld >>bericht<< und setzen Sie die Eigenschaft MultiLine auf true.

```
public Form1() {
    InitializeComponent();
    Arbeiterin[] arbeiter = new Arbeiterin[4];
    arbeiter[0] = new Arbeiterin(new string[] { "Nektar sammeln", "Honig herstellen" });
    arbeiter[1] = new Arbeiterin(new string[] { "Brutpflege", "Bienenschule" });
    arbeiter[2] = new Arbeiterin(new string[] { "Stockpflege", "Wachdienst" });
    arbeiter[3] = new Arbeiterin(new string[] { "Nektar sammeln", "Honig herstellen",
            "Brutpflege", "Bienenschule", "Stockpflege", "Wachdienst" });
    königin = new Königin(arbeiter);
}
```

Der Konstruktor von Arbeiterin erwartet als Parameter ein String-Array mit den Aufgaben, die die Arbeiterin erledigen kann.

Ihr Formular braucht ein Königin-Feld namens königin. Sie übergeben dem Königin-Konstruktor ein Array mit Arbeiterin-Objektreferenzen.

2 Erstellen Sie die Klassen Arbeiterin und Königin.

Was Sie über die Klassen Arbeiterin und Königin wissen müssen, wissen Sie bereits. Königin.ArbeitZuweisen() durchläuft das `arbeiter`-Array und versucht, jeder Arbeiterin über die Methode AufgabeErledigen() eine Aufgabe zuzuweisen. Das Arbeiterin-Objekt prüft anhand des String-Arrays `mögliche-Arbeiten`, ob es die Aufgabe erledigen kann. Kann sie es, setzt sie schichtAnzahl auf die Arbeitsdauer, AktuelleArbeit auf die Aufgabe und erledigteSchichten auf null. Wurde eine Schicht gearbeitet, wird erledigteSchichten um eins erhöht. Die schreibgeschützte Eigenschaft OffeneSchichten liefert `schicht-Anzahl - erledigteSchichten` – darüber prüft die Königin, wie viele Schichten noch anstehen.



Lösungen zu den Übungen

```
public class Arbeiterin {
    public Arbeiterin(string[] möglicheArbeiten) {
        this.möglicheArbeiten = möglicheArbeiten;
    }

    public int OffeneSchichten {
        get {
            return schichtAnzahl - erledigteSchichten;
        }
    }

    private string aktuelleAufgabe = "";
    public string AktuelleArbeit {
        get {
            return aktuelleAufgabe;
        }
    }

    private string[] möglicheArbeiten;
    private int schichtAnzahl;
    private int erledigteSchichten;

    public bool AufgabeErledigen(string job, int schichten) {
        if (!String.IsNullOrEmpty(aktuelleAufgabe))
            return false;
        for (int i = 0; i < möglicheArbeiten.Length; i++)
            if (möglicheArbeiten[i] == job) {
                aktuelleAufgabe = job;
                this.schichtAnzahl = schichten;
                erledigteSchichten = 0;
                return true;
            }
        return false;
    }

    public bool SchichtArbeiten() {
        if (String.IsNullOrEmpty(aktuelleAufgabe))
            return false;
        erledigteSchichten++;
        if (erledigteSchichten > schichtAnzahl) {
            erledigteSchichten = 0;
            schichtAnzahl = 0;
            aktuelleAufgabe = "";
            return true;
        }
        else
            return false;
    }
```

OffeneSchichten ist eine schreibgeschützte Eigenschaft, die berechnet, wie viele Schichten verbleiben.

AktuelleArbeit ist eine schreibgeschützte Eigenschaft, die der Königin sagt, welche Arbeit erledigt wird.

Der Konstruktor setzt nur das String-Array möglicheArbeiten. Er ist privat, weil die Königin der Arbeiterin nur den Auftrag geben und nicht selbst nachsehen soll, ob diese die Aufgabe auch erledigen kann.

Die Königin nutzt die Methode AufgabeErledigen(), um Arbeiterinnen Aufgaben zuzuweisen – die Arbeiterin prüft möglicheAufgaben, um zu sehen, ob sie die Aufgabe erledigen kann.

Wir prüfen mit dem NICHT-Operator !, ob der String NICHT null oder leer ist. Das ist genau so, als würden wir prüfen, ob etwas false ist.

Über SchichtArbeiten() weist die Königin die Arbeiterin an, die nächste Schicht zu arbeiten. Die Methode liefert true, wenn das die letzte Schicht für diese Aufgabe ist. So kann die Königin dem Bericht eine entsprechende Zeile hinzufügen.

Sehen Sie sich diese Logik genau an. Erst wird aktuelleAufgabe geprüft: Ist die Arbeiterin nicht beschäftigt, wird false geliefert und die Methode damit beendet. Ist sie es, wird erledigteSchichten erhöht und in einem Vergleich mit schichtAnzahl geprüft, ob die Aufgabe erfüllt ist. Ist das der Fall, liefert die Methode true, andernfalls false.

```
public class Königin {
    public Königin(Arbeiterin[] arbeiter) {
        this.arbeiter = arbeiter;
    }

    private Arbeiterin[] arbeiter;
    private int schichtNummer = 0;

    public bool ArbeitZuweisen(string job, int schichten) {
        for (int i = 0; i < arbeiter.Length; i++)
            if (arbeiter[i].AufgabeErledigen(job, schichten))
                return true;
        return false;
    }

    public string NächsteSchichtArbeiten() {
        schichtNummer++;
        string bericht = "Bericht für Schicht " + schichtNummer + "\r\n";
        for (int i = 0; i < arbeiter.Length; i++)
        {
            if (arbeiter[i].SchichtArbeiten())
                bericht += "Aufgabe von Arbeiterin " + (i + 1) + " erledigt\r\n";
            if (String.IsNullOrEmpty(arbeiter[i].AktuelleArbeit))
                bericht += "Arbeiterin " + (i + 1) + " arbeitet nicht\r\n";
            else
                if (arbeiter[i].OffeneSchichten > 0)
                    bericht += "Arbeiterin " + (i + 1) + " macht '"
                        + arbeiter[i].AktuelleArbeit + "' für "
                        + arbeiter[i].OffeneSchichten + " weitere Schichten\r\n";
                else
                    bericht += "Arbeiterin " + (i + 1) + " ist nach dieser Schicht mit '"
                        + arbeiter[i].AktuelleArbeit + "' fertig\r\n";
        }
        return bericht;
    }
}
```

Das Array arbeiter ist privat, weil es, nachdem es einmal zugewiesen wurde, von keiner anderen Klasse geändert werden soll ... oder auch gesehen werden soll, da nur die Königin Anweisungen gibt. Der Konstruktor setzt den Wert des Felds.

Bei der Zuweisung der Arbeit beginnt die Königin mit der ersten Arbeiterin, versucht, ihr Arbeit zuzuweisen, und geht zur nächsten weiter, wenn das nicht geht. ird eine Biene für die Aufgabe gefunden, endet die Methode und liefert true.

Die Methode NächsteSchichtArbeiten() weist alle Arbeiterinnen an, eine Schicht zu arbeiten, und hängt an den Bericht eine Zeile für den Status der Arbeiterin an.

Hier ist der restliche Code für das Formular (den Konstruktor kennen Sie ja).

```
Königin königin;

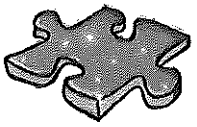
private void aufgabeZuweisen_Click(object sender, EventArgs e) {
    if (königin.ArbeitZuweisen(bienenAufgabe.Text, (int)shifts.Value) == false)
        MessageBox.Show("Keine Arbeiterin für die Aufgabe '"
            + bienenAufgabe.Text + "' verfügbar", "Die Königin spricht ...");
    else
        MessageBox.Show("Die Aufgabe '" + bienenAufgabe.Text + "' wird in "
            + shifts.Value + " Schichten erledigt", "Die Königin spricht ...");
}

private void nächsteSchicht_Click(object sender, EventArgs e) {
    bericht.Text = königin.NächsteSchichtArbeiten();
}
```

Das Formular nutzt das Feld königin als Referenz auf das Königin-Objekt, das seinerseits ein Array mit Referenzen auf Arbeiterin-Objekte hat.

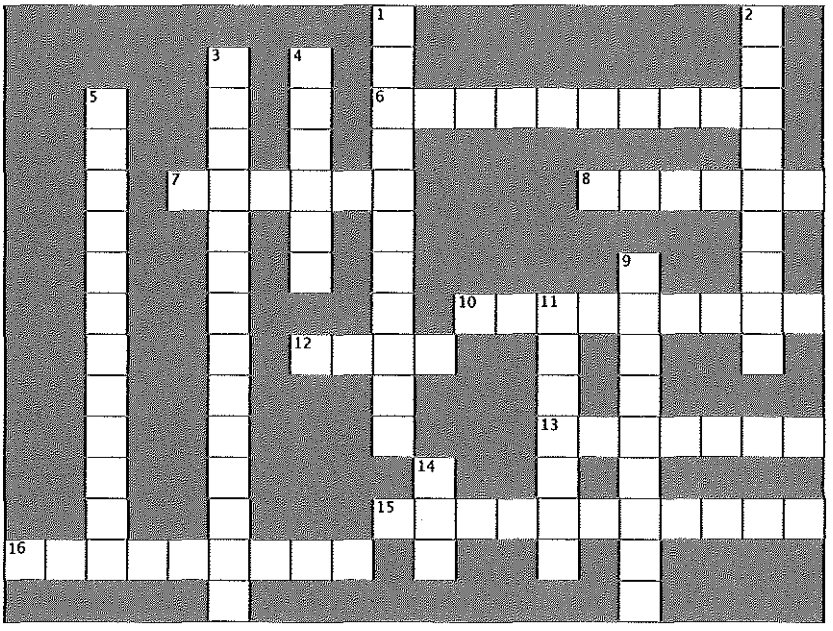
Der Button aufgabeZuweisen ruft ArbeitZuweisen() auf, um die Arbeit einer Arbeiterin zuzuweisen.

Der Button nächsteSchicht lässt die nächste Schicht arbeiten, den Bericht dafür generieren und im Textfeld anzeigen.



Objekt-Kreuzworträtsel

Gönnen Sie Ihrem Kopf eine kleine Pause, bevor Sie zum nächsten Teil dieser Übung weitergehen.



Waagerecht

6. Ein Steuerelement auf einem Formular, über das Sie Anwendungen mit Registerkarten erstellen können.
7. Eine besondere Methode, die aufgerufen wird, wenn Sie in ein Feld schreiben.
8. Der Einstiegspunkt muss immer eine Methode sein, die als _____ deklariert ist.
10. Dieses Konzept nutzen Sie, wenn Sie Teile einer Klasse vor dem Zugriff aus anderen Klassen schützen.
12. Dieses Schlüsselwort nutzen Sie in einer Klasse, um auf die Methoden und Felder der Klasse zuzugreifen, von der sie erbt.
13. Dieses Schlüsselwort müssen Sie in eine Methodendeklaration einschließen, damit Unterklassen sie überschreiben können.
15. Eine Unterklasse muss immer den Basisklassen-_____ aufrufen
16. Der Konstruktor in einer Unterklasse muss nicht die gleichen _____ haben wie der, den er von seiner Basisklasse erbt.

Senkrecht

1. Eine Klasse, die von einer anderen Klasse erbt, nennt man so.
2. Die Eigenschaft einer TextBox, die angibt, dass sie mehrere Zeilen darstellen soll.
3. Das machen Sie in einer Unterklasse, um eine Methode oder eine Eigenschaft einer Basisklasse zu ersetzen oder zu erweitern.
4. Man weiß nie, wann der _____ eines Felds ausgeführt wird. Deswegen sollte man sicherstellen, dass er keine Nebeneffekte hat.
5. Nutzen Sie Unterklassen, um etwas aus dem wahren Leben zu modellieren, das ein _____ Fall von etwas Abstrakterem oder Allgemeinerem ist.
9. Das ist ein wunderbares Werkzeug, um sicherzustellen, dass Sie in ähnlichen Klassen nicht mehrfach den gleichen Code haben.
11. Dieses Schlüsselwort nutzen Sie, damit andere Klassen auf eine Methode oder ein Feld nicht zugreifen können.
14. Der Basisklassenkonstruktor wird _____ dem Unterklassenkonstruktor ausgeführt.

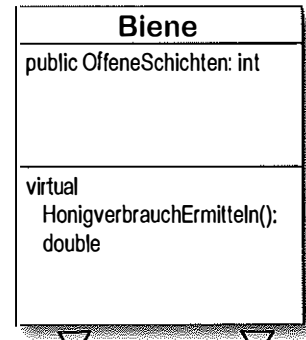
→ Antworten auf Seite 250.

Erweitern Sie die Bienenverwaltung mit Vererbung



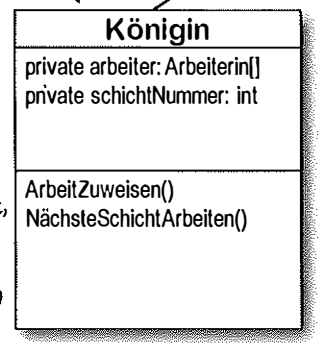
Nachdem Sie das Grundsystem eingerichtet haben, sollen Sie Vererbung nutzen, um nachzuhalten, wie viel Honig die einzelnen Bienen verbrauchen. Unterschiedliche Bienen verbrauchen unterschiedlich große Mengen Honig, und die Königin verbraucht am meisten von allen. Sie werden also das anwenden, was Sie zur Vererbung gelernt haben, um eine Biene-Klasse zu erzeugen, von der Königin und Arbeiterin erben.

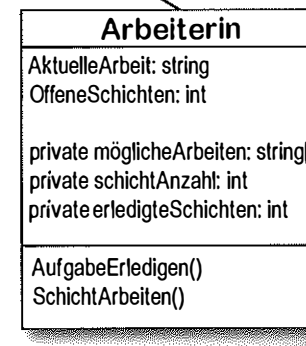
Die Klasse Biene hat ein grundlegendes Honigverbrauchsverhalten. Da der Honigverbrauch von der Anzahl der verbleibenden Schichten abhängt, verschieben wir die Eigenschaft OffeneSchichten in die Klasse Biene und deklarieren sie als virtual, damit Arbeiterin sie überschreiben kann.



Die Arbeiterin muss Biene nur erweitern und die Eigenschaft OffeneSchichten mit der überschreiben, die Sie bereits geschrieben haben.

Die Königin muss ihren Bericht anpassen und den Honigverbrauch einschließen. Das heißt, dass sie den Honigverbrauch der einzelnen Bienen abrufen muss – und da sie selbst ebenfalls Honig verbraucht, muss sie Biene erweitern und die virtuelle Methode HonigverbrauchErmitteln() überschreiben.

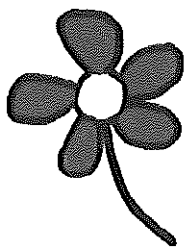




Vorhandenes Element hinzufügen

Bei mehrteiligen Übungen ist es gut, wenn Sie für den zweiten Teil ein neues Projekt beginnen. Dann können Sie bei Bedarf jederzeit zur ersten Projektmappe zurückkehren. Recht einfach geht das, indem Sie in der IDE im Projektmappen-Explorer mit der rechten Maustaste auf das Projekt klicken, Hinzufügen –> Vorhandenes Element wählen, und dann zum Ordner des alten Projekts navigieren und die gewünschen Dateien auswählen. Die IDE erstellt nun neue Kopien der Dateien und fügt sie dem neuen Projekt hinzu. Auf einige Dinge müssen Sie allerdings achten. Die IDE ändert den Namensraum nicht, diesen müssen Sie in den Dateien von Hand bearbeiten. Und wenn Sie ein Formular hinzufügen, müssen Sie die Designer-Datei (.Designer.cs) sowie die Ressourcen-Datei (.resx) hinzufügen und daran denken, auch darin den Namensraum zu ändern.







Wir sind noch nicht fertig! Die Königin muss festhalten, wie viel Honig der Stock für seine Arbeiterinnen verbraucht. Das ist eine wunderbare Gelegenheit, Ihre neuen Vererbungsfertigkeiten einzusetzen!

1 **Die Königin muss wissen, wie viel Honig der Stock verbraucht.**

Die Bienenbuchhalter haben der Königin gerade mitgeteilt, dass der Bienenstock nicht genug Honig produziert. Sie muss wissen, wie viel Honig sie und ihre Arbeiterinnen verbrauchen, damit sie entscheiden kann, ob sie Bienen von der Brutpflege abziehen und in die Honigproduktion stecken muss.

- ★ Da alle Bienen Honig essen, verbraucht der Bienenstock eine Menge Honig. Deswegen müssen sie permanent mehr davon produzieren.
- ★ Arbeitsbienen brauchen mehr Honig, wenn sie arbeiten. Den meisten Honig brauchen sie, wenn ihre Arbeit beginnt, damit sie ausreichend Energie haben, sie zu erledigen. Im weiteren Verlauf ihrer Arbeit verbrauchen sie immer weniger. Während der letzten Schicht benötigt die Biene 10 Einheiten Honig, in der vorletzten 11, der vorvorletzten 12 usw. Während eine Biene bei der Arbeit ist (d.h. OffeneSchichten größer als null ist), können Sie ihren Honigverbrauch also ermitteln, indem Sie OffeneSchichten 9 hinzuaddieren.
- ★ Macht eine Biene nichts (d.h., OffeneSchichten ist null), braucht sie nur 7,5 Honigeinheiten.
- ★ Das sind Zahlen für normale Bienen. Wiegt eine Biene mehr als 150 Milligramm, braucht sie 35% mehr Honig. Das gilt nur für die Arbeiterinnen, nicht für die Königin.
- ★ Königinnen brauchen viel Honig. Je mehr Arbeiterinnen eine Königin hat, umso mehr Honig braucht sie, da die Aufsicht über sie mehr Arbeit ist. Sie braucht so viel Honig, als müsste sie so viele Schichten arbeiten wie die Arbeiterin, die für ihre Arbeit die meisten Schichten offen hat.
- ★ Und sie braucht noch mehr Honig: Sie braucht 20 zusätzliche Honigeinheiten pro Schicht, wenn zwei oder weniger Arbeiterinnen arbeiten, und sogar 30, wenn es drei oder mehr sind. Der 35%-Regel unterliegt die Königin nicht, da Königinnen immer 275 Milligramm wiegen.
- ★ Die Zahlen zum Honigverbrauch müssen ans Ende jedes Schichtberichts angehängt werden.

8 **Erstellen Sie eine Biene-Klasse, um die Honigberechnungen abzuwickeln.**

Da Arbeitsbienen und Königin die Honigberechnung auf ähnliche Weise durchführen, können Sie doppelten Code vermeiden, indem Sie eine Basisklasse Biene erstellen, die Arbeiterin und Königin erweitern. Jede Biene muss ihr Gewicht kennen (damit sie weiß, ob sie auf ihren Honigverbrauch 35% aufschlagen muss).

- ★ Erstellen Sie eine HonigverbrauchErmitteln()-Methode, die die von einer Arbeitsbiene benötigte Honigmenge berechnet. Arbeitsbienen und Königin brauchen diese Berechnung, die Königin darüber hinaus noch weitere. Arbeitsbienen können die Methode also erben, und die Königin muss sie überschreiben.
- ★ Die Methode HonigverbrauchErmitteln() benötigt die Anzahl der verbleibenden Schichten. Fügen Sie also eine schreibgeschützte virtuelle Eigenschaft OffeneSchichten hinzu, die null liefert. Die OffeneSchichten-Eigenschaft der Arbeitsbienen wird diese überschreiben.
- ★ Die Honigberechnung muss das Gewicht der Biene kennen, der Biene-Konstruktor muss also das Gewicht der Biene als Parameter annehmen und in einem Feld speichern. Da dieses von keiner anderen Klasse benötigt wird, sollten Sie es privat machen.



Eine gute Faustregel ist es, Felder und Methoden standardmäßig privat zu lassen und nur öffentlich zu machen, wenn andere Klassen sie benötigen. So vermeiden Sie Fehler dadurch, dass Klassen die Eigenschaften oder Methoden einer anderen Klasse fehlerhaft gebrauchen.

Tipp: Sie können die etwas kryptische Fehlermeldung »Keine Überladung« nutzen! Lassen Sie die Klasse Arbeiterin von Biene erben und erstellen Sie dann das Projekt. Zeigt die IDE den Fehler an, klicken Sie doppelt darauf. Die IDE springt automatisch direkt in den Arbeiterin-Konstruktor. Wie praktisch!



3 Lassen Sie die Klasse Arbeiterin von Biene erben.

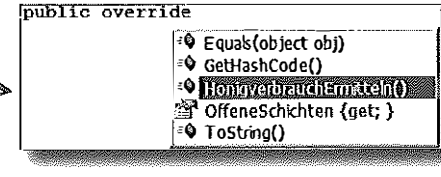
Wie bei Kathrin müssen Sie den Konstruktor so einrichten, dass er den Basisklassenkonstruktor aufruft. Sie müssen den Arbeiterin-Konstruktor so ändern, dass er als Parameter das Gewicht der Biene erwartet, und diesen Parameter dann an den Basisklassenkonstruktor weiterreichen. Fügen Sie anschließend der OffeneSchichten-Eigenschaft der Klasse Arbeiterin das Schlüsselwort `override` hinzu. Haben Sie das getan, kann jede Arbeiterin ihren Honigverbrauch für die Königin berechnen ... und Sie müssen keine weiteren Änderungen an der Klasse Arbeiterin vornehmen!

4 Lassen Sie die Klasse Königin von Biene erben.

Bei der Klasse Königin sind mehr Änderungen erforderlich als bei der Klasse Arbeiterin, da sie die Honigberechnung selbst durchführen und dem Schichtbericht hinzufügen muss.

- ★ Überschreiben Sie die Methode Biene.HonigverbrauchErmitteln() und fügen Sie die zusätzlichen Berechnungen der Königin hinzu. Sie muss herausfinden, ob zwei oder weniger Arbeiterinnen bei der Arbeit sind, damit sie weiß, ob sie 20 oder 30 Einheiten braucht. Dann fügt sie diese Zahl der Zahl Einheiten hinzu, die sie brauchen würde, wenn sie noch so viele Schichten arbeiten müsste wie die Arbeiterin mit den meisten verbleibenden Schichten.
- ★ Aktualisieren Sie die NächsteSchichtArbeiten()-Methode der Königin, um dem Bericht die Zeile für den Honigverbrauch hinzuzufügen. Verwenden Sie eine Schleife, um den Honigverbrauch der einzelnen Bienen zu addieren, und ermitteln Sie gleichzeitig die Biene mit dem größten Honigverbrauch – tun Sie das, **bevor** die Königin den einzelnen Arbeiterinnen sagt, dass sie die nächste Schicht arbeiten sollen (damit sie die Verbrauchszahlen für die aktuelle Schicht erhält). Sie addiert diese Werte zusammen, rechnet ihren eigenen Verbrauch hinzu und hängt ans Ende des Berichts dann folgende Zeile an: »Gesamthonigverbrauch: xxx Einheiten« (xxx ist dabei die Anzahl verbrauchter Einheiten).
- ★ Dann müssen Sie wie bei der Arbeiterin den Königin-Konstruktor aktualisieren.

Gehen Sie in die Klasse Königin und geben Sie »public override« ein – wenn Sie dann die Leertaste drücken, zeigt die IDE automatisch die Methoden an, die Sie überschreiben können. Wählen Sie die Methode, die Sie überschreiben möchten, fügt sie automatisch den Basismethodenaufruf ein.



8 Aktualisieren Sie das Formular und instantiieren Sie die Bienen richtig.

Da Sie die Konstruktoren von Königin und Arbeiterin geändert haben, müssen Sie auch ändern, wie sie aufgerufen werden. Beide Konstruktoren besitzen den neuen Parameter gewicht; Sie brauchen also das zu verwendende Gewicht:

- ★ Arbeitersbiene 1: 175mg; Arbeitsbiene 2: 114mg; Arbeitersbiene 3: 149mg; Arbeitersbiene 4: 155mg; Königin: 275mg

Das sind die einzigen Änderungen, die Sie am Formular vornehmen müssen.



LÖSUNGEN ZU DEN ÜBUNGEN

Hier ist die Klasse Biene. Sie führt die grundlegende Honigverbrauchsberechnung durch, die von den Klassen Arbeiterin und Königin verwendet wird.

```
public class Biene {
    public Biene(double gewicht) {
        this.gewicht = gewicht;
    }

    public virtual int OffeneSchichten {
        get { return 0; }
    }

    private double gewicht;

    public virtual double HonigverbrauchErmitteln() {
        double verbrauch;
        if (OffeneSchichten == 0)
            verbrauch = 7.5;
        else
            verbrauch = 9 + OffeneSchichten;
        if (gewicht > 150)
            verbrauch *= 1.35;
        return verbrauch;
    }
}
```

Die Klasse Biene hat einen Konstruktor, der das Feld gewicht setzt, und die Methode HonigverbrauchErmitteln(), die berechnet, wie viel Honig eine Arbeitsbiene verbraucht.

Steht für eine Biene noch eine Schicht aus, verbraucht sie 10 Einheiten, sind es 2, 11 usw. Hat sie keine Arbeit, verbraucht sie 7,5 Einheiten. Ist OffeneSchichten gleich null, hat die Biene keine Arbeit.

Wiegt die Biene mehr als 150mg, steigt der Verbrauch um 35%.

Vererbung hat es vereinfacht, den Code zu aktualisieren und den Klassen Königin und Arbeiterin das neue Honigverbrauchsverhalten hinzuzufügen. Mit doppeltem Code wären diese Änderungen viel schwerer gewesen.

Nur der Konstruktor des Formulars ändert sich – der Rest bleibt vollkommen gleich.

```
public Form1() {
  InitializeComponent();

  Arbeiterin[] arbeiter = new Arbeiterin[4];
  arbeiter[0] = new Arbeiterin(new string[] { "Nektar sammeln", "Honig herstellen" }, 175);
  arbeiter[1] = new Arbeiterin(new string[] { "Brutpflege", "Bienenschule" }, 114);
  arbeiter[2] = new Arbeiterin(new string[] { "Stockpflege", "Wachdienst" }, 149);
  arbeiter[3] = new Arbeiterin(new string[] { "Nektar sammeln", "Honig herstellen",
            "Brutpflege", "Bienenschule", "Stockpflege", "Wachdienst" }, 155);
  königin = new Königin(arbeiter);
}
```

Die einzige Änderung am Formular ist, dass den Konstruktoren der Arbeiterinnen das Gewicht hinzugefügt werden muss.

```
public class Arbeiterin : Biene {
    public Arbeiterin(string[] möglicheArbeiten, int gewicht)
        : base(gewicht) {
        this.möglicheArbeiten = möglicheArbeiten;
    }

    public override int OffeneSchichten {
    // ... der Rest der Klasse bleibt gleich ...
```

Bei Arbeiterin war es nur erforderlich, von Biene zu erben, den Konstruktor so anzupassen, dass ein gewicht-Parameter angenommen und an den Basiskonstruktor übergeben wird, und die Eigenschaft Biene.OffeneSchichten zu überschreiben, indem der Deklaration das Schlüsselwort override hinzugefügt wird.

An der Klasse Königin sind einige Änderungen erforderlich, beginnend mit dem Erben von Biene.

```
public class Königin : Biene {
    public Königin(Arbeiterin[] arbeiter)
        : base(275) {
        this.arbeiter = arbeiter;
    }
```

Die Königin wiegt 275mg. Dieses Gewicht übergibt ihr Konstruktor beim Aufruf des Basiskonstruktors.

NächsteSchichtArbeiten() erhält eine weitere Schleife die die HonigverbrauchErmitteln()-Methode der Arbeiterinnen aufruft. Dann wird die HonigverbrauchErmitteln()-Methode der Königin aufgerufen, um den Gesamtverbrauch zu ermitteln.

```
    public string NächsteSchichtArbeiten()
    {
        double gesamtVerbrauch = 0;
        for (int i = 0; i < arbeiter.Length; i++)
            gesamtVerbrauch += arbeiter[i].HonigverbrauchErmitteln();
        gesamtVerbrauch += HonigverbrauchErmitteln();

        // ... hier steht der ursprüngliche Code dieser Methode ...

        bericht += "Gesamthonigverbrauch: " + gesamtVerbrauch + " Einheiten";
        return bericht;
    }
```

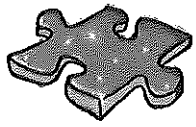
Am Ende von NächsteSchichtArbeiten() muss dem Bericht noch der Honigverbrauch hinzugefügt werden

Die Königin überschreibt Biene.HonigverbrauchErmitteln(), um ihre Honigberechnung durchzuführen. Sie sucht die Arbeiterin mit dem größten Verbrauch und fügt je nachdem, wie viele Bienen arbeiten, 20 oder 30 hinzu.

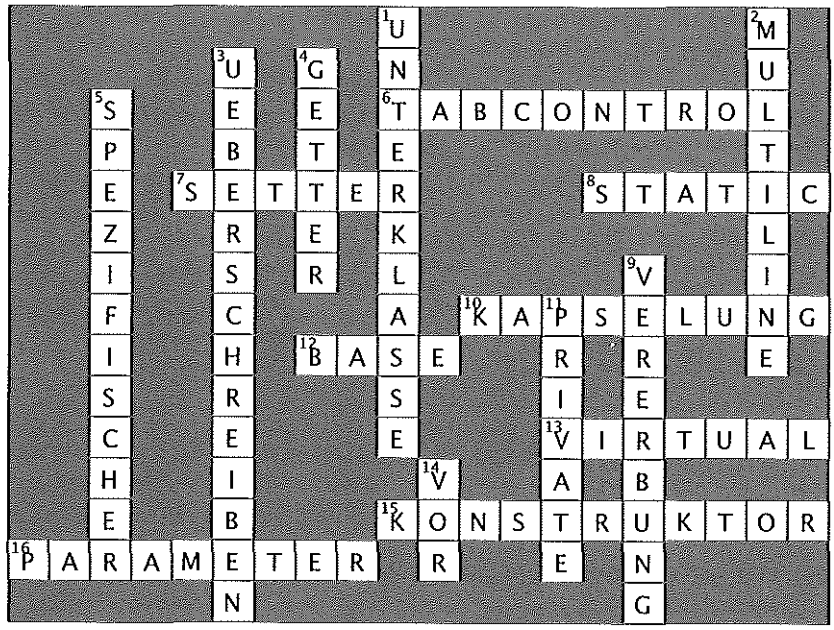
```
    public override double HonigverbrauchErmitteln() {
        double verbrauch = 0;
        double höchsterArbeiterVerbrauch = 0;
        int arbeitendeArbeiterinnen = 0;
        for (int i = 0; i < arbeiter.Length; i++) {
            if (arbeiter[i].HonigverbrauchErmitteln() > höchsterArbeiterVerbrauch)
                höchsterArbeiterVerbrauch = arbeiter[i].HonigverbrauchErmitteln();
            if (arbeiter[i].OffeneSchichten > 0)
                arbeitendeArbeiterinnen++;
        }
        verbrauch += höchsterArbeiterVerbrauch;
        if (arbeitendeArbeiterinnen >= 3)
            verbrauch += 30;
        else
            verbrauch += 20;
        return verbrauch;
    }
}
```

Diese Schleife betrachtet den Verbrauch aller Arbeiterinnen und sucht die mit dem größten.

Sind 3 oder mehr Arbeiterinnen aktiv, braucht die Königin 30 Einheiten Honig zusätzlich. Andernfalls braucht sie 20 weitere Einheiten.



Objekt-Kreuzworträtsel, Lösung



Klassen, die Versprechen halten

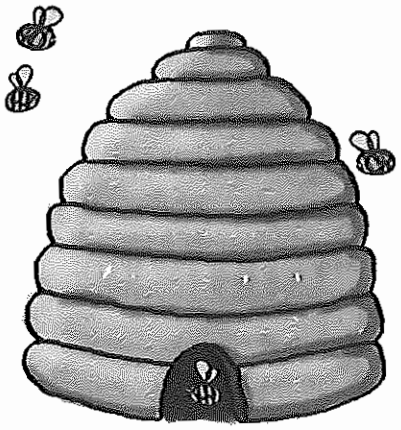


Taten sagen mehr als Worte.

Gelegentlich müssen Sie Ihre Objekte auf Basis dessen gruppieren, **was sie tun können,** nicht auf Basis der Klassen, von denen sie erben. Dann kommen **Schnittstellen** ins Spiel – diese ermöglichen Ihnen, mit jeder Klasse zu arbeiten, die bestimmte Aufgaben erledigen kann. Aber mit **großer Macht geht eine große Verantwortung einher**: Jede Klasse, die eine Schnittstelle implementiert, muss versprechen, **all ihre Verpflichtungen zu erfüllen** ... oder der Compiler tritt Ihnen vors Schienbein, Verstanden()?

Kehren wir zu den Bienen zurück

Die Firma Bicnomatic will die von Ihnen im letzten Kapitel erstellte Bienenstockverwaltung zu einem ausgewachsenen Bienenstocksimulator ausbauen. Hier ist ein Überblick über die Spezifikation für die neue Version des Programms:



Bienomatic Bienenstocksimulator

Zur besseren Darstellung des Bienenstocklebens müssen wir den Arbeitsbienen spezialisierte Fähigkeiten geben.

- Alle Bienen verbrauchen Honig und haben ein Gewicht.
- Königinnen weisen Arbeit zu, überwachen Schichtberichte und weisen Arbeiterinnen an, die nächste Schicht zu arbeiten.
- Alle Arbeitsbienen arbeiten in Schichten.
- Die Bienen vom Wachdienst müssen ihre Stachel schärfen, nach Feinden suchen und diese stechen können.
- Nektarsammler müssen Blumen suchen, Nektar sammeln und dann in den Stock zurückkehren.

Es sieht nicht so aus, als müssten sich die Klassen Biene ud Arbeiterin so sehr ändern. Wir können die Klassen erweitern, die wir bereits haben, um diese Funktionen zu erfüllen.

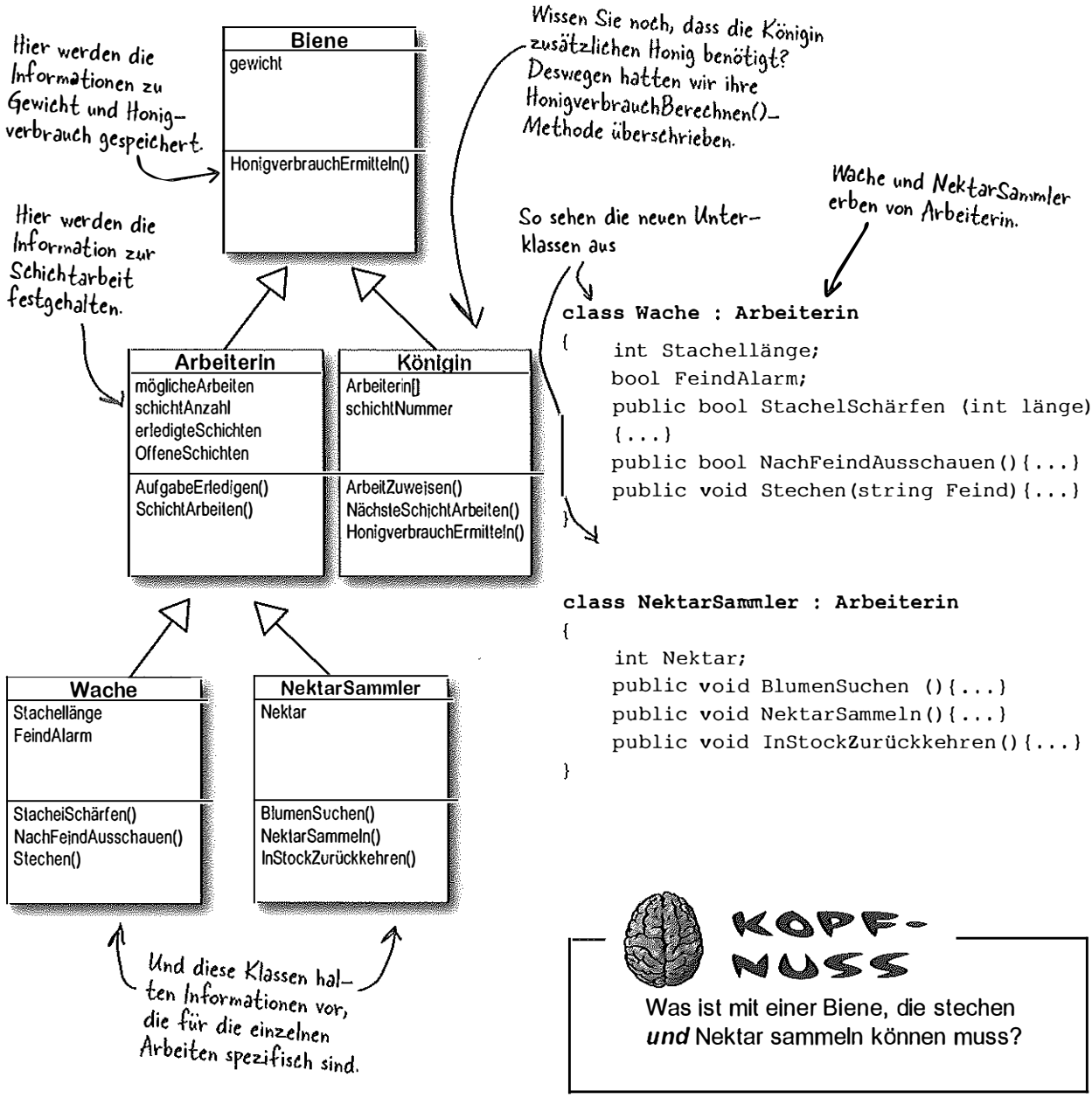
Es scheint, als müssten wir für die Arbeitsbienen, je nach Arbeitsauftrag, unterschiedliche Daten speichern.

Viele Dinge bleiben gleich

Die Bienen im neuen Bienenstocksimulator verbrauchen immer noch auf gleiche Weise Honig wie zuvor. Die Königin muss den Arbeiterinnen immer noch Arbeit zuweisen können und Schichtberichte einsehen, die zeigen, wer was tut. Wie zuvor arbeiten die Arbeiterinnen Schichten, die Aufgaben, die sie erfüllen, wurden allerdings etwas mehr ausgearbeitet.

Mit Vererbung können wir Klassen für die unterschiedlichen Arten von Bienen erstellen

Hier ist eine Klassenhierarchie mit Arbeiterin- und Königin-Klassen, die von Biene erben. Arbeiterin hat außerdem die Unterklassen Nektarsammler und Wache.




```
class Wache : Arbeiterin
{
    int Stachellänge;
    bool FeindAlarm;
    public bool StachelSchärfen (int länge)
    {...}
    public bool NachFeindAusschauen(){...}
    public void Stechen(string Feind){...}
}

class NektarSammler : Arbeiterin
{
    int Nektar;
    public void BlumenSuchen (){...}
    public void NektarSammeln(){...}
    public void InStockZurückkehren(){...}
}
```


KOPF-NUSS

Was ist mit einer Biene, die stechen ***und*** Nektar sammeln können muss?

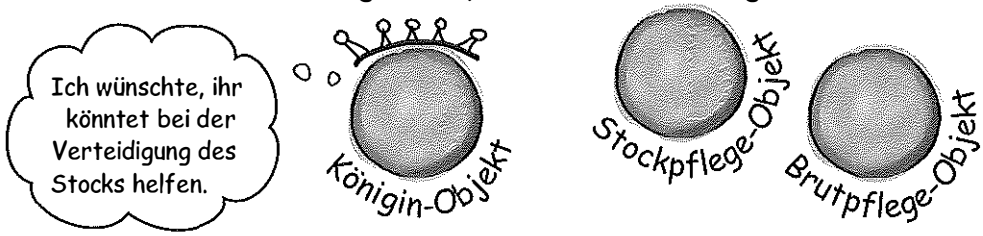
Eine Schnittstelle sagt einer Klasse, dass sie bestimmte Methoden und Eigenschaften implementieren muss

Eine Klasse kann nur von einer anderen Klasse erben. Es hilft uns also nicht, zwei separate Unterklassen für `Wache` und `NektarSammler` zu erstellen, wenn wir eine Biene haben, die **beide** Aufgaben erfüllen kann. Glücklicherweise gibt uns C# für diese Situationen **Schnittstellen**. Schnittstellen (oder Interfaces) geben Ihnen eine Möglichkeit, eine Gruppe von Methoden zu definieren, die eine Klasse haben ***muss***.

Eine Schnittstelle ***verlangt***, dass eine Klasse bestimmte Methoden hat. Sie tut das, indem sie ***den Compiler Fehler melden lässt***, wenn er nicht alle Methoden findet, die von einer Schnittstelle von den Klassen verlangt werden, die sie implementieren. Diese Methoden können direkt in der Klasse geschrieben sein oder von einer Basisklasse geerbt werden. Die Schnittstelle kümmert es nicht, wie die Methoden oder Eigenschaften dahin kommen, solange sie vorhanden sind, wenn der Code kompiliert wird.

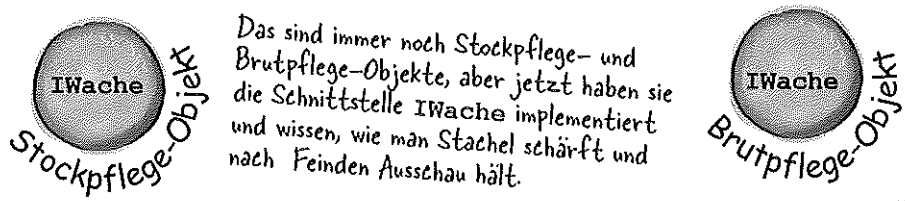
Sie nutzen eine Schnittstelle, um von einer Klasse zu verlangen, dass sie alle darin aufgeführten Methoden und Eigenschaften implementiert – tut sie es nicht, meldet der Compiler einen Fehler.

Die StockVerteidigen()-Methode der Königin kann nur Wache-Objekte anweisen, den Stock zu verteidigen. Gern würde sie auch andere Bienen trainieren, ihre Stachel einzusetzen, aber sie hat keine Möglichkeit, sie anzuweisen anzugreifen.



Erstellen wir eine Schnittstelle `IWache`, kann jede Biene, die sie implementiert und AlarmStufe, Stachellänge, NachFeindAusschauen() und StacheiSchärfen() bietet, den Stock verteidigen.

Indem Sie eine Klassendatei mit der Schnittstelle IWache hinzufügen und die Klasse Brutpflege so modifizieren, dass sie sie verwendet, geben Sie der Königin die Möglichkeit, auch Brutpflege-Objekte zur Stockverteidigung heranzuziehen.



Das sind immer noch Stockpflege- und Brutpflege-Objekte, aber jetzt haben sie die Schnittstelle `IWache` implementiert und wissen, wie man Stachel schärft und nach Feinden Ausschau hält.

Schnittstellennamen beginnen mit I

Wenn Sie Schnittstellen erstellen, sollten ihre Namen mit einem `I` (für Interface) beginnen. Es gibt keine Regel, die sagt, dass Sie das müssen, aber es verbessert die Lesbarkeit Ihres Codes. Wie sehr es Ihnen das Leben erleichtert, können Sie sich selbst ansehen. Gehen Sie in der IDE auf eine leere Zeile und tippen Sie >>I<< ein. IntelliSense blendet dann einen Haufen eingebauter .NET-Schnittstellen ein. Da diese alle mit `I` beginnen, ist sofort erkennbar, dass es Schnittstellen sind und keine Klassen.

Schnittstellen definieren Sie mit dem Schlüsselwort `interface`

Schnittstellen fügen Sie Ihrem Programm ganz ähnlich hinzu wie Klassen, nur dass Sie nie Methoden schreiben. Sie definieren lediglich den Rückgabetyp und die Parameter der Methode, aber statt mit einem Anweisungsblock in geschweiften Klammern schließen Sie die Zeile mit einem Semikolon ab.

Schnittstellen speichern keine Daten, Sie können also **keine Felder hinzufügen**. Aber Sie *können* Definitionen für Eigenschaften angeben. Grund dafür ist, dass Getter und Setter nur andere Arten von Methoden sind und es bei Schnittstellen darum geht, Klassen dazu zu zwingen, dass sie bestimmte Methoden mit bestimmten Namen, Typen und Parametern haben. Soll Ihre Schnittstelle also ein Feld mit einem bestimmten Namen und Typ haben, können Sie also **einfach eine Eigenschaft verwenden** – das bewirkt das Gleiche.

Eine Schnittstelle deklarieren Sie so, und deswegen wird dieses Konstrukt auch im deutschen Sprachraum häufig Interface genannt.

```
public interface IWache
{
    int AlarmStufe { get;}
    int Stachellänge { get; set;}
    bool NachFeindAusschauen();
    int StachelSchärfen(int Length);
}
```

Schnittstellen speichern keine Daten. Sie haben also keine Felder ... aber sie können Eigenschaften haben.

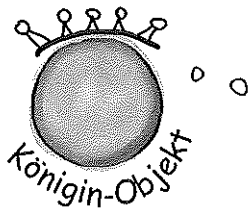
Jede Klasse, die diese Schnittstelle implementiert, muss eine StachelSchärfen()-Methode haben, die einen `int`-Parameter erwartet.

Jede Klasse, die diese Schnittstelle implementiert, muss all diese Methoden und Eigenschaften haben, sonst lässt sich das Programm nicht kompilieren.

```
public interface INektarSammler
{
    void BlumenSuchen();
    void NektarSammeln();
    void InStockZurückkehren();
}
```

Alles in einer Schnittstelle ist dazu gedacht, in anderen Klassen verwendet zu werden. Alle Methoden in einer Schnittstelle sind deswegen automatisch public.

Sie schreiben keinen Code für die Methoden in der Schnittstelle. Sie geben nur die Namen an. Den Code schreiben Sie in der Klasse, die sie implementiert.



Da auch ihr jetzt den Stock verteidigen könnt, sind wir viel sicherer.

Alles in einer öffentlichen Schnittstelle ist automatisch öffentlich, weil damit die öffentlichen Methoden und Eigenschaften der Klassen definiert werden, die sie implementieren.

Jetzt können Sie eine Klasse NektarWache erstellen, die beide Aufgaben erfüllt

Wie bei der Vererbung nutzen Sie den **Doppelpunkt-Operator**, um eine Schnittstelle zu deklarieren. Das funktioniert so: Das Erste hinter dem Doppelpunkt ist die Klasse, die erweitert wird, darauf folgt eine Liste der Schnittstellen – es sei denn, es wird keine Klasse erweitert, dann folgen einfach die Schnittstellen (in beliebiger Reihenfolge).

Die Klasse erbt von Arbeiterin und implementiert INektarSammler und IWache.

Eine Schnittstelle implementieren Sie, so wie Sie eine Klasse erben, mit dem Doppelpunkt-Operator.

```
class NektarWache : Arbeiterin, INektarSammler, IWache {
        public int AlarmStufe {
            get { return alarmStufe; }
        }

        public int Stachellänge {
            get { return stachellänge; }
            set {
                stachellänge = value;
            }
        }

        public bool NachFeindAusschauen() {...}
        public int StachelSchärfen(int länge)
        {...}
        public void BlumenSuchen() {...}
        public void NektarSammeln() {...}
        public void InStockZurückkehren() {...}
}
```

Sie können mehrere Schnittstellen angeben, indem Sie sie mit Kommata trennen.

Die NektarWache setzt das Hintergrundfeld für die Eigenschaft AlarmStufe in der Methode NachFeindAusschauen().

Die Biene zieht ihren Stachel ein, wenn keine Feinde anwesend sind, das Hintergrundfeld ändert mit der Zeit also seinen Wert.

Jeder Methode in der Schnittstelle entspricht eine Methode in der Klasse. Andernfalls ließe sie sich nicht kompilieren.

Erstellen Sie ein NektarWache-Objekt, beherrscht es sowohl die Aufgaben von NektarSammler-Objekten als auch die von Wache-Objekten.

Mit einer Klasse, die eine Schnittstelle implementiert, arbeiten Sie wie mit jeder anderen Klasse. Sie instantiieren sie mit `new` und nutzen ihre Methoden:

```
NektarWache bieneTom = new NektarWache();

bieneTom.NachFeindAusschauen();

bieneTom.BlumenSuchen();
```

Klassen, die Schnittstellen implementieren, müssen ALLE Methoden der Schnittstelle einschließen

Implementieren Sie eine Schnittstelle, heißt das, dass Sie in der Klasse jeweils eine Methode für alle Methoden und Eigenschaften haben müssen, die in der Schnittstelle deklariert werden – fehlt eine davon, lässt sie sich nicht kompilieren. Implementiert eine Klasse mehrere Schnittstellen, muss sie alle Methoden und Eigenschaften aller Schnittstellen anbieten, die sie implementiert.

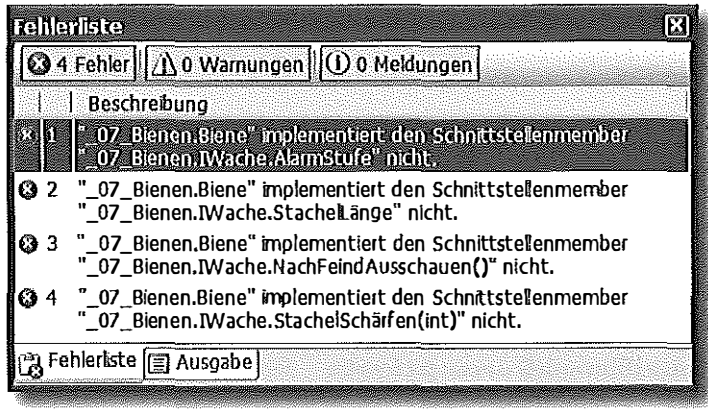
Tun Sie das!

1 **Erstellen Sie eine neue Anwendung und fügen Sie eine Datei namens IWache.cs hinzu.**
Geben Sie in die Klassendatei statt einer Klasse die Schnittstelle **`IWache`** von der letzten Seite ein.

2 **Fügen Sie dem Projekt eine Biene-Klasse hinzu.**
Fügen Sie noch keine Eigenschaften oder Methoden hinzu. Lassen Sie sie bloß IWache implementieren:

```
public class Biene : IWache {
```

3 **Versuchen Sie, das Programm zu kompilieren.**
Wählen Sie im Menü Erstellen »Projektmappe neu erstellen«. Oh – der Compiler lässt das nicht zu:



Sie werden einen dieser »implementiert nicht«-Fehler für jedes Schnittstellen-Member von IWache sehen, das in der Klasse nicht implementiert ist. Der Compiler besteht darauf, dass Sie alle Methoden in der Schnittstelle implementieren.

4 **Fügen Sie der Klasse Biene die Methoden und Eigenschaften hinzu.**
Fügen Sie die Methode `NachFeindAusschauen` und `StachelSchärfen` hinzu – sie müssen noch nichts tun, sie müssen sich nur kompilieren lassen. Ergänzen Sie dann den Getter für einen `int` namens AlarmStufe und die Getter und Setter für einen `int` namens Stachellänge. Jetzt lässt sich das Programm kompilieren!

Sammeln Sie Erfahrung im Umgang mit Schnittstellen

Schnittstellen sind wirklich leicht zu verwenden, aber man versteht sie am leichtesten, wenn man einfach etwas mit ihnen arbeitet. Erstellen Sie also eine neue Windows-Form-Anwendung, ziehen Sie einen Button auf das Formular und fangen Sie an.



1. Hier ist die Klasse `GroßerTyp` und der Code für einen Button, der ihn über einen Objektinitialisierer erstellt und seine ÜberSichSelbstReden()-Methode aufruft, also alles nichts Neues:

```
public class GroßerTyp {
    public string Name;
    public int Größe;

    public void ÜberSichSelbstReden() {
        MessageBox.Show("Ich heiße " + Name + " und bin "
            + Größe + " cm groß.");
    }
}

private void button1_Click(object sender, EventArgs e) {
    GroßerTyp großer = new GroßerTyp() { Größe = 74, Name = "Klaus" };
    großer.ÜberSichSelbstReden();
}
```

2. Erstellen wir eine IClown-Schnittstelle für die Klasse.

   Sie wissen bereits, dass alles in einer Schnittstelle öffentlich sein muss. Aber Sie müssen uns nicht blind vertrauen. Erstellen Sie ein neues Projekt und deklarieren Sie folgendermaßen eine Schnittstelle:

```
public interface IClown
```

   Versuchen Sie jetzt, innerhalb der Schnittstelle eine private Methode zu deklarieren:

```
private void Hupen();
```

   In einer Schnittstelle müssen Sie »public« nicht angeben, weil alle Eigenschaften und Methoden automatisch öffentlich sind.

   Wählen Sie in der IDE Erstellen→Projektmappe erstellen, sehen Sie diesen Fehler:

   ⊗ 1 Der private-Modifizierer ist für dieses Element nicht gültig.

   Löschen Sie jetzt den Zugriffsmodifizierer `private` – der Fehler verschwindet, und Ihr Programm lässt sich problemlos kompilieren.

3. Probieren Sie, bevor Sie zur nächsten Seite weitergehen, den Rest der Schnittstelle IClown zu erstellen und die Klasse GroßerTyp so anzupassen, dass sie diese Schnittstelle implementiert. Ihrem Projekt fügen Sie die Schnittstelle auf gleiche Weise hinzu wie eine Klasse: Klicken Sie im Projektmappen-Explorer rechts auf das Projekt und fügen Sie eine Klassendatei namens `IClown.cs` hinzu.

   Ihre neue IClown-Schnittstelle sollte eine `void`-Methode namens Hupen besitzen, die keine Parameter erwartet, und eine schreibgeschützte `string`-Eigenschaft namens MeinLustigesDing, die einen Getter, aber keinen Setter hat.

4 Hier ist die Schnittstelle – haben Sie es hinbekommen?

```
public interface IClown
{
    string MeinLustigesDing { get; }
    void Hupen();
}
```

Dies ist ein Beispiel für eine Schnittstelle, die einen Getter, aber keinen Setter hat. Denken Sie daran, dass Schnittstellen keine Felder enthalten können. Aber wenn Sie diese schreibgeschützte Eigenschaft in einer Klasse implementieren, sieht sie für andere Klassen wie ein Feld aus.

Modifizieren wir die Klasse GroßerTyp jetzt so, dass sie IClown implementiert. Denken Sie daran, dass auf den Doppelpunkt-Operator immer die Basisklasse folgt, von der geerbt wird (wenn es eine gibt), und dann eine Liste der zu implementierenden Schnittstellen, die jeweils durch Kommata getrennt werden. Da es keine Basisklasse und nur eine Schnittstelle gibt, sieht die Deklaration so aus:

GroßerTyp implementiert die Schnittstelle IClown.

```
public class GroßerTyp : IClown
```

Die IDE sagt Ihnen, dass GroßerTyp angibt, IClown zu implementieren, und Sie damit versprochen haben, alle Eigenschaften und Methoden in dieser Schnittstelle zu implementieren ... und dass Sie dieses Versprechen dann gebrochen haben!

Stellen Sie dann sicher, dass der Rest der Klasse identisch ist, einschließlich der beiden Felder und der Methode. Wählen Sie im Menü Erstellen »Projektmappe erstellen«, um das Programm zu kompilieren. Sie sehen zwei Fehler, unter anderem diesen:

```
"GroßerTyp" implementiert den Schnittstellen-
Member "IClown.Hupen()" nicht.
```

5 Diese Fehler verschwinden, sobald Sie alle Methoden und Eigenschaften hinzufügen, die die Schnittstelle deklariert. Legen Sie also mit der Implementierung der Schnittstelle los. Fügen Sie eine schreibgeschützte String-Eigenschaft namens MeinLustigesDing mit einem Getter hinzu, der immer den String »große Schuhe« liefert. Fügen Sie dann eine Hupen()-Methode hinzu, die ein Dialogfenster mit der Meldung »Hup-hup!« einblendet.

Das sieht so aus:

```
public string MeinLustigesDing {
    get { return "große Schuhe"; }
}

public void Hupen() {
    MessageBox.Show("Hup-hup!");
}
```

Die Schnittstelle verlangt bloß, dass eine Klasse, die sie implementiert, eine Eigenschaft namens MeinLustigesDing mit einem Getter hat. Sie können einen beliebigen Getter verwenden, auch einen, der jedes Mal den gleichen String liefert. Die meisten Getter machen mehr, aber das ist vollkommen ausreichend, wenn es das ist, was Sie brauchen.

Die Schnittstelle sagt nur, dass Sie eine public void-Methode namens Hupen haben müssen. Sie sagt nicht, was diese Methode tun muss. Sie kann alles machen, egal was. Der Code lässt sich kompilieren, solange eine Methode mit der richtigen Signatur vorhanden ist.

6 Jetzt lässt sich Ihr Code kompilieren! Aktualisieren Sie Ihren Button so, dass die Hupen()-Methode des GroßerTyp-Objekts aufgerufen wird.

Eine Schnittstelle können Sie nicht instantiieren, aber Sie können sie referenzieren

Nehmen Sie an, Sie hätten ein Objekt, das die Methode BlumenSuchen() besitzt. Das könnte ein beliebiges Objekt sein, das die Schnittstelle INektarSammler implementiert. Es könnte ein Arbeiterin-Objekt oder ein Roboter-Objekt oder auch ein Hund-Objekt sein ... es muss nur die Schnittstelle INektarSammler implementieren.

An diesem Punkt kommen **Schnittstellenreferenzen** ins Spiel. Diese können Sie einsetzen, um ein Objekt zu referenzieren, das die Schnittstelle implementiert, die Sie benötigen. So können Sie sichergehen, dass das Objekt immer die richtigen Methoden hat – selbst wenn Sie sonst nicht viel darüber wissen.

Versuchen Sie eine Schnittstelle zu implementieren, beschwert sich der Compiler.

Das funktioniert nicht ...

```
IWache dennis = new IWache();
```

```
1  Es konnte keine Instanz der abstrakten Klasse oder Schnittstelle
   "_07_Bienen.IWache" erstellt werden.
```

Sie können das Schlüsselwort `new` sinnvollerweise nicht mit Schnittstellen verwenden – die Methoden und Eigenschaften haben keine Implementierung. Wie könnten Sie wissen, wie sich ein Objekt verhält, das auf Grundlage einer Schnittstelle erstellt wurde?

... aber das schon:

Erinnern Sie sich daran, dass Sie eine BLT-Referenz an eine Klasse übergeben können, die ein Sandwich erwartet, weil BLT von Sandwich erbt? Das hier ist das Gleiche – NektarWache-Objekte können Sie in jeder Methode und Anweisung verwenden, die ein IWache-Objekt erwartet.

```
NektarWache fred = new NektarWache();
IWache willi = fred;
```

Auch wenn dieses Objekt eigentlich mehr kann, haben Sie über eine Schnittstellenreferenz nur Zugriff auf die Methoden in der Schnittstelle.

Die erste Zeile ist eine gewöhnliche `new`-Anweisung, die ein NektarWache-Objekt erzeugt und es über eine Referenzvariable namens Fred zugänglich macht.

Ab der zweiten Zeile wird es interessant, weil diese Zeile eine **neue Referenzvariable vom Typ `IWache` deklariert**. Diese Zeile mag zunächst etwas seltsam aussehen. Aber sehen Sie sich erst das hier an:

```
NektarWache maja = fred;
```

Was diese dritte Anweisung macht, wissen Sie – sie deklariert eine NektarWache-Referenz namens `maja` und weist ihr das zu, worauf die Referenz `fred` zeigt. Die `willi`-Anweisung nutzt IWache auf gleiche Weise.

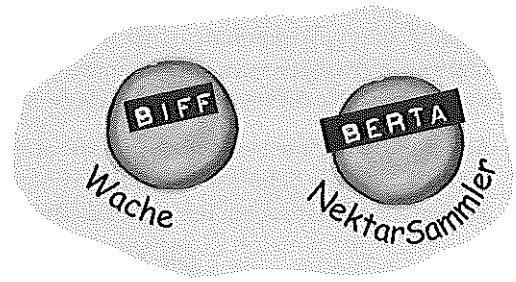


Was also ist passiert?

Da es lediglich eine `new`-Anweisung gibt, wurde **nur ein Objekt** erstellt. Die zweite Anweisung deklariert eine Referenz namens `willi`, die auf eine Instanz **jeder Klasse zeigen kann, die `IWache` implementiert**.

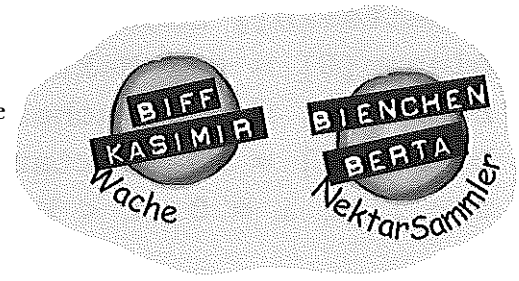
Schnittstellenreferenzen funktionieren wie Objektreferenzen

Sie wissen bereits, das Objektreferenzen auf dem Heap leben. Schnittstellenreferenzen sind nur eine andere Art, auf die gleichen Objekte zu verweisen, mit denen Sie bereits gearbeitet haben. Schauen Sie – es ist einfach!



1 Erzeugen Sie ein paar Bienen.
Das ist mittlerweile ganz vertrauter Kram.

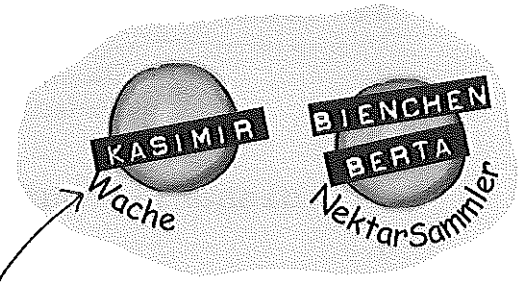
```
Wache biff = new Wache();
NektarSammler berta = new NektarSammler();
```



2 Deklarieren Sie IWache- und INektarSammler-Referenzen.
Schnittstellenreferenzen können Sie genau so verwenden wie jede andere Art von Referenz.

```
IWache kasimir = biff;
INektarSammler bienchen = berta;
```

Diese beiden Anweisungen nutzen Schnittstellen, um Referenzen auf bestehende Objekte zu erstellen. Einer Schnittstellenreferenz können Sie eine Instanz einer Klasse zuweisen, die die entsprechende Schnittstelle implementiert.



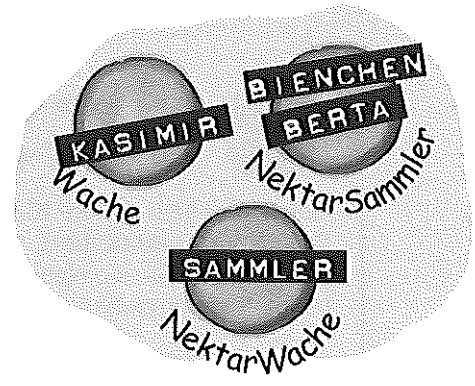
2 Eine Schnittstellenreferenz hält ein Objekt am Leben.
Ein Objekt verschwindet, wenn darauf keine Referenzen mehr zeigen. Aber es gibt keine Regel, die besagt, dass diese Referenzen den gleichen Typ haben müssen! Eine Schnittstellenreferenz hält ein Objekt genauso sicher fest wie eine Objektreferenz.

```
biff = null;
```

Das Objekt verschwindet nciht, weil `kasimir` immer noch darauf zeigt.

1 Einer Schnittstellenreferenz eine neue Instanz zuweisen.
Eine Objektreferenz *benötigen* Sie eigentlich nicht – Sie können ein neues Objekt erstellen und direkt einer Schnittstellenreferenz zuweisen.

```
INektarSammler sammler = new NektarWache();
```



Mit »is« können Sie ermitteln, ob eine Klasse eine bestimmte Schnittstelle implementiert

Manchmal müssen Sie herausfinden, ob eine bestimmte Klasse eine Schnittstelle implementiert. Angenommen, wir würden unsere Arbeitsbienen alle in einem Array namens bienen speichern. Wir können das Array den Typ Arbeiterin speichern lassen, da der Typ aller Arbeitsbienen die Klasse Arbeiterin oder eine ihrer Unterklassen ist.

Aber welche der Bienen können Nektar sammeln? Anders gesagt: Wir wollen wissen, ob die Klasse die Schnittstelle `INektarSammler` implementiert. Das Schlüsselwort **`is`** können wir einsetzen, um genau das herauszufinden.

Alle Arbeiterinnen befinden sich in einem Arbeiterin-Array. Wir nutzen »is«, um herauszufinden, was für eine Art Arbeitsbiene die einzelnen Bienen sind.

Wir haben ein Array mit Arbeiterin-Bienen und müssen ermitteln, welche wir zum Nektarsammeln ausschicken können. Also durchlaufen wir das Array und nutzen »is«, um herauszufinden, welche Objekte die richtigen Methoden und Eigenschaften für diese Aufgabe haben.

```
Arbeiterin[] bienen = new Arbeiterin[3];
bienen[0] = new NektarSammler();
bienen[1] = new Wache();
bienen[2] = new NektarWache();
for (int i = 0; i < bienen.Length; i++)
{
    if (bienen[i] is INektarSammler)
    {
        bienen[i].AufgabeErledigen("Nektarsammeln", 3);
    }
}
```

is funktioniert wie ein Gleichheitsoperator (==) für Typen.

Das ist, als würden Sie sagen: Tu das, wenn du INektarSammler implementierst.

Da wir jetzt wissen, dass die Biene ein Nektarsammler ist, können wir ihr die Aufgabe zuweisen, Nektar zu sammeln.



KOPF-NUSS

Haben Sie eine andere Klasse, die nicht von Arbeiterin erbt, aber die Schnittstelle INektarSammler ***implementiert***, dann auch sie die Aufgabe erfüllen! Aber da sie nicht von Arbeiterin erbt, kriegen Sie sie nicht in das Array mit den anderen Bienen. Fällt Ihnen ein Weg ein, dieses Problem zu umgehen und ein Array mit Bienen und dieser neuen Klasse zu erstellen?

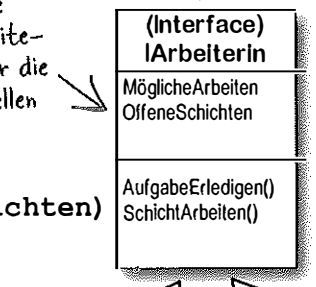
Eine Schnittstelle kann von einer anderen erben

Erbt eine Klasse von einer anderen, erhält sie alle Methoden und Eigenschaften der Basisklasse. **Schnittstellenvererbung** ist sogar noch einfacher. Da es in Schnittstellen keinen richtigen Methodenrumpf gibt, müssen Sie sich keine Gedanken darüber machen, Basisklassenkonstruktoren oder -methoden aufzurufen. Erbende Schnittstellen bündeln einfach alle Methoden und Eigenschaften der Schnittstellen, von denen sie erben.

Zeichnen wir in einem Klassendiagramm eine Schnittstelle ein, zeigen wir Vererbung durch gestrichelte Linien.

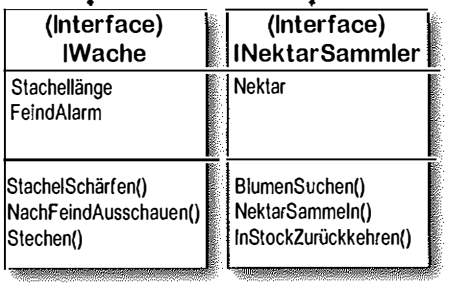
```
public interface IArbeiterin
{
  string MöglicheArbeiten { get; }
  int OffenSchichten { get; }
  void AufgabeErledigen(string arbeit, int schichten)
  void SchichtArbeiten()
}
```

Wir haben eine neue Schnittstelle IArbeiterin erstellt, von der die anderen Schnittstellen erben.



Klassen, die Schnittstellen implementieren, die von IArbeiterin erben, müssen die Methoden und Eigenschaften von IArbeiterin implementieren

Wenn eine Klasse eine Schnittstelle implementiert, muss sie alle Eigenschaften und Methoden in dieser Schnittstelle umfassen. Erbt diese Schnittstelle von einer anderen Schnittstelle, müssen auch alle *ihre* Eigenschaften und Methoden implementiert werden.

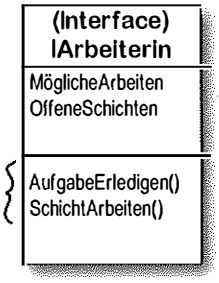



```
public interface IWache : IArbeiterin
{
    int AlarmStufe { get;)
    int Stachellänge { get; set;)
    bool NachFeindAusschauen();
    int StachelSchärfen(int länge);
}
```

Hier ist wieder die Schnittstelle IWache, aber jetzt erbt sie von der Schnittstelle IArbeiterin. Das sieht nach einer kleinen Änderung aus, bedeutet aber einen großen Unterschied bei allen Klassen, die IWache implementieren.

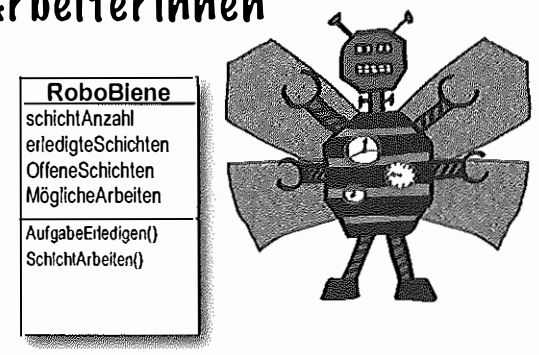
Eine Klasse, die IWache implementiert, muss jetzt nicht mehr nur diese Methoden implementieren ...

... sondern auch die der Schnittstelle IArbeiterin, die diese Schnittstelle erbt.



Eine RoboBiene kann die Arbeit von Arbeiterinnen erledigen und braucht keinen Honig

Erstellen wir eine neue Biene, eine RoboBiene, die mit Rapsöl läuft. Wir können sie dennoch die Schnittstelle IArbeiterin implementieren lassen, damit sie alles machen kann, was eine normale Arbeitsbiene auch kann.



Das ist unsere Basisklasse Roboter, damit Roboter mit Rapsöl laufen können.

```
public class Roboter
{
     public void RapsölVerbrennen() {...}
}

public class RoboBiene : Roboter, IArbeiterin
{
     private int schichtAnzahl;
     private int erledigteSchichten;
     public int OffeneSchichten
       {get {return schichtAnzahl - erledigteSchichten;}}
     private string job;
     public string möglicheArbeiten;{get{return job;}}
     public bool AufgabeErledigen(string arbeit, int schichten){...}
     public void SchichtArbeiten()
}
```

RoboBiene erbt von Roboter und implementiert IArbeiterin. Das bedeutet, dass es ein Roboter ist und die Arbeit einer Arbeiterin erledigen kann. Perfekt!

Die Klasse RoboBiene implementiert alle Methoden von IArbeiterin.

Wenn RoboBiene nicht alles in der Schnittstelle IArbeiterin implementiert, lässt sich der Code nicht kompilieren.

Denken Sie daran, dass es für andere Klassen in der Anwendung funktionell keinen Unterschied zwischen einer RoboBiene und einer normalen Arbeitsbiene gibt. Beide implementieren die Schnittstelle IArbeiterin, verhalten sich, sofern es das restliche Programm betrifft, also wie Arbeitsbienen.

Aber mit `is` könnten Sie die beiden Typen unterscheiden:

```
if (arbeitsbiene is Roboter) {
  // arbeitsbiene ist also
  // ein Roboter-Objekt.
}
```

Mit »is« können wir herausfinden, von welcher Klasse eine Klasse abgeleitet ist bzw. welche Schnittstellen sie implementiert.

Klassen können BELIEBIGE Schnittstellen implementieren, sie müssen nur die Verpflichtungen einhalten, indem sie alle Eigenschaften und Methoden implementieren.

is sagt welchen Typ Ihr Objekt hat, as sagt dem Compiler, wie er es behandeln soll

Manchmal müssen Sie auf einem Objekt eine Methode aus einer Schnittstelle aufrufen, die es implementiert. Aber was ist, wenn Sie nicht wissen, ob das Objekt den entsprechenden Typ hat? Dann nutzen Sie **is**, um das herauszufinden. Ist das der Fall, können Sie **as** nutzen, um das Objekt – von dem Sie jetzt wissen, dass es den richtigen Typ hat – zu behandeln, als hätte es die Methode, die Sie aufrufen müssen.

```
IArbeiterin[] bienen = new IArbeiterin[3];
    bienen[0] = new NektarWache();
    bienen[1] = new RoboBiene();
    bienen[2] = new Arbeiterin();

for (int i = 0; i < bienen.Length; i++) {
    if (bienen[i] is INektarSammler) {
        INektarSammler sammler;
        sammler = bienen[i] as INektarSammler;
        sammler.NektarSammeln();
```

Diese Bienen implementieren alle IArbeiterin, aber wir wissen nicht, welche die anderen Schnittstellen wie INektarSammler implementieren.

Wir durchlaufen die Bienen ...

... und prüfen, ob sie INektarSammler implementieren.

INektarSammler-Methoden können wir nicht auf Biene-Referenzen aufrufen. Sie haben nur den Typ IArbeiterin und wissen nichts über INektarSammler-Methoden.

Wir nutzen >>as<<, um zu sagen, dass dieses Objekt WIE eine INektarSammler-Implementierung behandelt werden soll.

JETZT können wir INektarSammler-Methoden aufrufen.

Spitzen Sie Ihren Bleistift

Werfen Sie einen Blick auf das Array auf der linken Seite. Schreiben Sie für all diese Anweisungen auf, welcher Wert für `i` dazu führte, dass der `is`-Ausdruck mit `true` ausgewertet werden würde. Streichen Sie außerdem die Anweisungen durch, die sich nicht kompilieren lassen.

```
IArbeiterin[] bienen = new IArbeiterin[8];
bienen[0] = new NektarWache();
bienen[1] = new RoboBiene();
bienen[2] = new Arbeiterin();
bienen[3] = bienen[0] as IArbeiterin;
bienen[4] = IWache;
bienen[5] = null;
bienen[6] = bienen[0];
bienen[7] = new NektarSammler();
```

1. `(bienen[i] is INektarSammler)`

   .....

2. `(bienen[i] is IWache)`

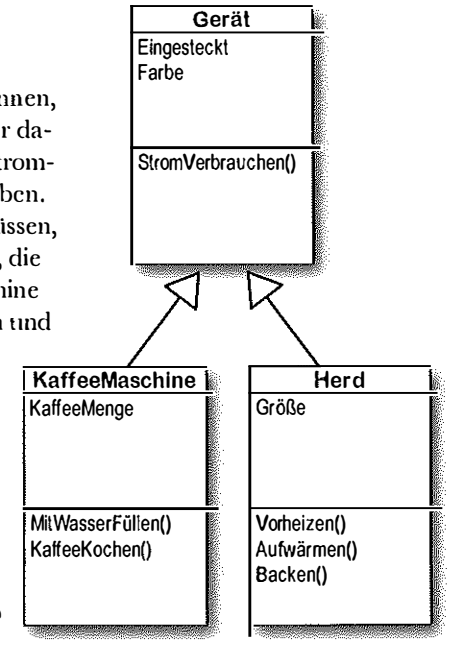
   .....

3. `(bienen[i] is IArbeiterin)`

   .....

Eine KaffeeMaschine ist auch ein Gerät

Wenn Sie ermitteln möchten, wie Sie Ihre monatlichen Stromkosten reduzieren können, interessieren Sie sich nicht dafür, was Ihre Geräte machen. Sie interessieren sich nur dafür, dass sie Strom verbrauchen. Würden Sie ein Programm schreiben, das Ihren Stromverbrauch überwacht, würden Sie also wahrscheinlich nur eine Klasse Gerät schreiben. Aber wenn Sie eine Kaffeemaschine und einen Herd auseinanderhalten können müssen, erstellen Sie eine Klassenhierarchie. Sie würden also Methoden und Eigenschaften, die für eine Kaffeemaschine oder einen Herd spezifisch sind, Klassen wie KaffeeMaschine und Herd hinzufügen und diese von einer Klasse Gerät mit allgemeinen Methoden und Eigenschaften ableiten.




```
public void VerbrauchMessen(Gerät gerät) {
    // Code, der die Daten einer Datenbank
    // für den Stromverbrauch hinzufügt.
}
```

Diese Methode überwacht den Stromverbrauch in einem Haus.

Später im Programm erscheint Code zur Überwachung des Stromverbrauchs der Kaffeemaschine.

```
KaffeeMaschine kalterKaffee = new KaffeeMaschine();
VerbrauchMessen(kalterKaffee);
```

Obwohl VerbrauchMessen() eine Gerät-Referenz erwartet, können Ssie die kalterKaffee-Referenz übergeben, weil KaffeeMaschine eine Unterklasse von Gerät ist.

Das haben Sie schon im letzten Kapitel gesehen, als wir eine BLT-Referenz an eine Methode übergeben haben, die ein Sandwich erwartet.

Spitzen Sie Ihren Bleistift — Lösung

Werfen Sie einen Blick auf das Array auf der linken Seite. Schreiben Sie für all diese Anweisungen auf, welcherWert für `i` dazu führte, dass der `is`-Ausdruck mit `true` ausgewertet werden würde. Streichen Sie außerdem die Anweisungen durch, die sich nicht kompilieren lassen.

```
IArbeiterin[] bienen = new IArbeiterin[8];
bienen[0] = new NektarWache();
bienen[1] = new RoboBiene();
bienen[2] = new Arbeiterin();
bienen[3] = bienen[0] as IArbeiterin;
bienen[4] = IWache;
bienen[5] = null;
bienen[6] = bienen[0];
bienen[7] = new INektarSammler();
```

NektarWache() implementiert IWache.

1. (bienen[i] is INektarSammler)

   0 und 6

2. (bienen[i] is IWache)

   0 und 6

3. (bienen[i] is IArbeiterin)

   0, 1, 2, 3 und 6

Umwandeln funktioniert mit Objekten und Schnittstellen

Setzen Sie eine Unterklasse anstelle der Basisklasse ein – beispielsweise eine Kaffeemaschine für ein Gerät oder ein BLT für ein Sandwich –, bezeichnet man das als **Upcasting** (hochwandeln). Das ist ein sehr mächtiges Werkzeug beim Aufbau von Klassenhierarchien. Der einzige Nachteil beim Upcasting ist, dass Sie nur die Eigenschaften und Methoden der Basisklasse verwenden können. Anders gesagt: Behandeln Sie eine Kaffeemaschine wie ein Gerät, können Sie ihm nicht sagen, dass es sich mit Wasser füllen oder Kaffee kochen soll. Aber Sie *können* sagen, ob es eingesteckt ist oder nicht, da das etwas ist, was Sie bei jedem Gerät machen können (deswegen ist die Eigenschaft Eingesteckt Teil der Klasse Gerät).

1 Erzeugen wir ein paar Objekte.

Wir können eine KaffeeMaschine und einen Herd erzeugen:

```
KaffeeMaschine kalterKaffee = new KaffeeMaschine();

Herd alteRöhre = new Herd();
```

Wir beginnen damit, dass wir ganz normal ein Herd- und ein KaffeeMaschine-Objekt erzeugen.

2 Was ist, wenn wir ein Array mit Geräten erstellen möchten?

Eine KaffeeMaschine können Sie nicht in ein Herd[]-Array stecken, einen Herd nicht in KaffeeMaschine[]-Array. Aber beide können Sie in ein Gerät[]-Array stecken:

```
Gerät[] küchenGeräte = new Gerät[2];

küchenGeräte[0] = kalterKaffee;

küchenGeräte[1] = alteRöhre;
```

Mit Upcasting können Sie ein Gerät-Array erzeugen, das Kaffeemaschinen und Herde aufnehmen kann.

3 Aber Sie können ein Gerät nicht wie einen Herd behandeln.

Über eine Gerät-Referenz können Sie **nur** auf die Methoden und Eigenschaften zugreifen, die mit Geräten zu tun haben. Sie **können keine** Kaffeemaschinenmethoden und -eigenschaften verwenden, ***selbst wenn Sie wissen, dass es eigentlich eine Kaffeemaschine ist.*** Diese Anweisungen funktionieren also, weil sie ein KaffeeMasehine-Objekt wie ein Gerät behandeln:

```
Gerät stromverbraucher = new KaffeeMaschine();

stromverbraucher.StromVerbrauchen();
```

stromverbraucher ist eine Gerät-Referenz auf ein KaffeeMaschine-Objekt.

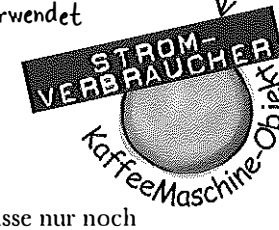
Diese Zeile lässt sich nicht kompilieren, weil stromverbraucher eine Gerät-Referenz ist und nur für Gerät-Aufgaben verwendet werden kann.

Aber versuchen Sie, sie wie eine Kaffeemaschine zu verwenden.

```
stromverbraucher.KaffeeKochen();
```

lässt sich der Code nicht mehr kompilieren, und die IDE meldet einen Fehler:

```
'Gerät' enthält keine Definition
für 'KaffeeKochen'
```



Das liegt daran, dass Sie nach einem Upcast von einer Unterklasse auf eine Basisklasse nur noch auf Member zugreifen können, **die vom Typ der Referenz definiert werden**, über die Sie das Objekt verwenden.

Mit Downcasting können Sie Ihr Gerät wieder zu einer Kaffeemaschine machen

Upcasting ist ein wertvolles Werkzeug, weil es Ihnen ermöglicht, eine Kaffemaschine oder einen Herd dort zu verwenden, wo Sie nur ein Gerät brauchen. Aber es hat einen großen Nachteil – nutzen Sie eine Gerät-Referenz, die auf ein KaffeeMaschine-Objekt zeigt, können Sie nur die Methoden und Eigenschaften nutzen, die zu Gerät gehören. Und das ist der Punkt, an dem **Downcasting** die Bühne betritt: Mit ihm nehmen Sie Ihre **zuvor per Upcasting** verallgemeinerte Referenz und ändern diese wieder zurück. Ob Ihr Gerät wirklich eine Kaffeemaschine ist, können Sie mit dem Schlüsselwort **`is`** ermitteln. Haben Sie sich dessen versichert, können Sie das Gerät mit dem Schlüsselwort **`as`** wieder in eine Kaffeemaschine zurückverwandeln.



1. **Wir beginnen mit der bereits umgewandelten Kaffeemaschine.**
   Hier ist der Code, den wir verwendet hatten:

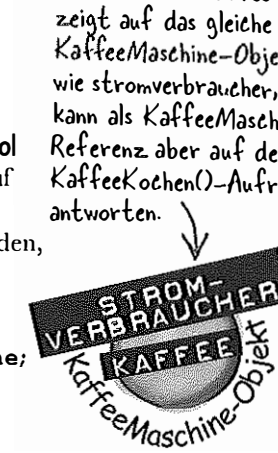
```
Gerät stromverbraucher = new KaffeeMaschine();
stromverbraucher.StromVerbrauchen();
```

2. **Aber was ist, wenn wir unsere KaffeeMaschine zurückwollen?**
   Der erste Schritt beim Downcasting ist, dass man mit `is` prüft, ob das wirklich möglich ist.

```
if (stromverbraucher is KaffeeMaschine)
   // Downcasten ist also möglich!
```

3. **Wir wissen, es ist eine KaffeeMaschine, nutzen wir sie also auch so!**
   Das Schlüsselwort `is` ist der erste Schritt. Wissen Sie, dass Ihre Gerät-Referenz auf ein KaffeeMaschine-Objekt zeigt, können Sie sie mit `as` downcasten. Und dann können Sie die Methoden und Eigenschaften von KaffeeMaschine wieder verwenden, aber weiterhin auch die von Gerät, da KaffeeMaschine von Gerät erbt.

```
if (stromverbraucher is KaffeeMaschine) {
   KaffeeMaschine kaffee = stromverbraucher as KaffeeMaschine;
   kaffee.KaffeeKochen();
}
```



Scheitert das Downcasting, liefert `as` null

Aber was passiert, wenn Sie versuchen, ein Herd-Objekt in eine KaffeeMaschine umzuwandeln? Sie erhalten null – und wenn Sie versuchen, die Referenz zu verwenden, bricht Ihr Programm mit einer Fehlermeldung ab.

```
if (stromverbraucher is KaffeeMaschine) {
   Herd bratenröhre = stromverbraucher as Herd;
   bratenröhre.Vorheizen();
}
```

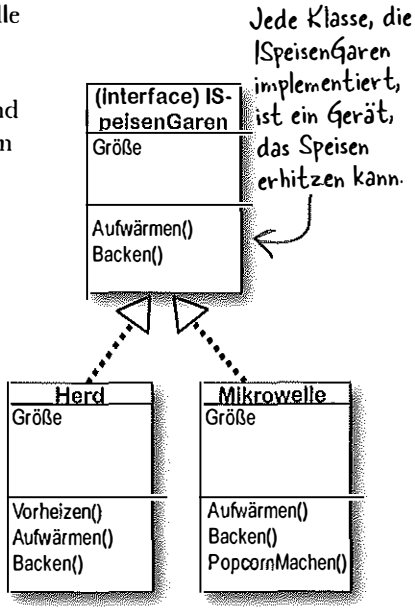
Die beiden passen nicht zusammen.

stromverbraucher ist KEIN Herd-Objekt. Beim Versuch, mit >>as<< ein Downcasting durchzuführen, wird die Referenz bratenröhre auf null gesetzt. Versuchen Sie dann, sie zu verwenden, erhalten Sie einen Fehler.



Upcasting und Downcasting funktionieren auch mit Schnittstellen

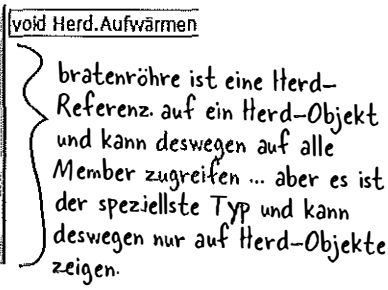
Sie wissen bereits, dass `is` und `as` auch mit Schnittstellen funktionieren – ebenso alle Upcasting- und Downcasting-Geschichten. Fügen wir eine ISpeisenGaren-Schnittstelle für Klassen hinzu, die Speisen erwärmen können. Und ergänzen wir auch eine Mikrowelle-Klasse – Mikrowelle und Herd implementieren beide die Schnittstelle ISpeisenGaren. Jetzt gibt es drei verschiedene Wege, auf ein Herd-Objekt zuzugreifen. Und die IntelliSense-Funktionalität der IDE hilft Ihnen, genau herauszufinden, was Sie tun können und was nicht.



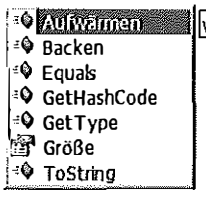

```
Herd bratenröhre = new Herd();
bratenröhre.
```

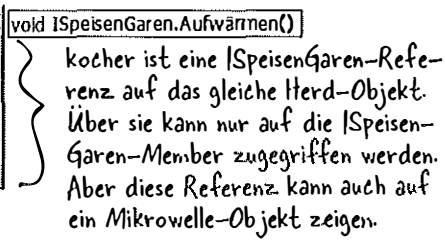
Sobald Sie den Punkt eingeben, öffnet sich ein IntelliSense-Fenster mit einer Liste aller Member, die Sie verwenden können.





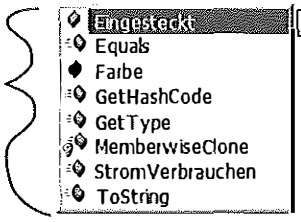

```
ISpeisenGaren kocher;
if (bratenröhre is ISpeisenGaren)
    kocher = bratenröhre as ISpeisenGaren;
    kocher.
```

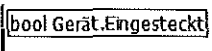





```
Gerät stromverbraucher;
if (bratenröhre is Gerät)
    stromverbraucher = bratenröhre;
    stromverbraucher.
```

stromverbraucher ist eine Gerät-Referenz. Sie können darüber nur auf die öffentlichen Member von Gerät zugreifen. Aber Sie können sie auch auf ein KaffeeMaschine-Objekt zeigen lassen.





Drei verschiedene Referenzen auf das gleiche Objekt können je nach Typ der Referenz auf unterschiedliche Methoden und Eigenschaften zugreifen.

Es gibt keine Dummen Fragen

F: Etwas langsamer, bitte – Sie sagten, dass ich immer upcasten, aber nicht immer downcasten kann. Warum?

A: Weil der Compiler Sie warnen kann, wenn Ihr Upcasting falsch ist. Upcasting schlägt nur fehl, wenn Sie ein Objekt über eine Referenz eines Typs behandeln wollen, den es nicht erbt oder implementiert. Das kann der Compiler sofort herausfinden und Ihnen einen entsprechenden Fehler melden.

Der Compiler weiß aber nicht, wie er prüfen soll, dass Sie *von* einer gültigen Referenz *auf* eine Referenz downcasten, die nicht gültig ist. Das liegt daran, dass Sie auf der rechten Seite des Schlüsselworts `as` jeden beliebigen Klassen- oder Schnittstellennamen angeben können. Ist das Downcasting nicht zulässig, liefert der `as`-Operator einfach `null`. Und es ist gut, dass der Compiler Sie daran nicht hindert, weil es viele Gelegenheiten gibt, bei denen man das machen möchte.

F: Jemand hat mir gesagt, dass eine Schnittstelle mit einem Vertrag vergleichbar ist. Das verstehe ich nicht. Was bedeutet das?

A: Ja, das haben wir auch schon gehört – viele Leute sagen gern, dass eine Schnittstelle wie ein Vertrag ist. (Und es ist auch eine beliebte Frage in Vorstellungsgesprächen.) in gewisser Hinsicht ist es auch wahr. Lassen Sie eine Klasse eine Schnittstelle implementieren, sagen Sie damit dem Compiler, dass Sie versprechen, die Klasse mit bestimmten Methoden auszustatten. Der Compiler nimmt Sie damit beim Wort.

Wie Schnittstellen funktionieren, kann man sich unserer Meinung nach besser merken, wenn man sie als eine Art Checkliste betrachtet. Der Compiler geht die Checkliste durch, um sicherzustellen, dass Sie Ihre Klasse mit allen Methoden aus der Schnittstelle ausgestattet haben. Tun Sie das nicht, schreit er auf und kompiliert Ihre Klasse nicht.

F: Was ist, wenn ich in meiner Schnittstelle einen Methodenrumpf angeben möchte? Geht das?

A: Nein. Das erlaubt Ihnen der Compiler nicht. Eine Schnittstelle darf keinerlei Anweisungen enthalten. Obgleich Sie Schnittstellen über den Doppelpunkt-Operator implementieren, ist das etwas anderes als das Erben von einer Klasse. Schnittstellen fügen Ihren Klassen keinerlei Verhalten hinzu. Sie sagen dem Compiler, dass er sicherstellen soll, dass Ihre Klasse alle Member besitzt, die die Schnittstelle vorgibt.

F: Warum soll man Schnittstellen dann verwenden? Mir scheint das einfach nur Beschränkungen mit sich zu bringen, ohne dabei eine Änderung an meiner Klasse zu bewirken.

A: Weil Sie Schnittstellenreferenzen auf Instanzen beliebiger Klassen zeigen lassen können, die die Schnittstelle implementieren. Und das ist wirklich nützlich – Sie können über eine Referenz eines Typs mit Instanzen der unterschiedlichsten Klassen arbeiten.

Hier ist ein kurzes Beispiel. Pferde, Ochsen, Esel und Stiere können alle Wagen ziehen. Aber in unserem Zoo-Simulator würden sie jeweils durch eine andere Klasse repräsentiert. Nehmen wir an, Sie würden in Ihrem Zoo Kutschfahrten anbieten und möchten ein Array mit allen Tieren erstellen, die Wagen ziehen können. Aber Sie können nicht einfach ein Array erzeugen, das all diese unterschiedlichen Typen speichert. Würden sie alle von der gleichen Basisklasse erben, könnten Sie ein Array mit ihnen erstellen. Aber das ist nicht so. Was also sollen Sie tun?

Das ist der Punkt, an dem Schnittstellen praktisch sind. Sie können eine Schnittstelle `IZieher` mit Methoden für das Herumziehen von Wagen erstellen. Dann können Sie Ihr Array so deklarieren:

```
IZieher[] zieherArray;
```

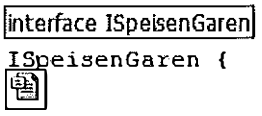
Jetzt können Sie in das Array eine Referenz auf jedes Tier einschließen, das die Schnittstelle `IZieher` implementiert.

F: Gibt es einen einfacheren Weg, Schnittstellen zu implementieren? Das ist eine Menge Tipparbeit!

A: Klar doch. Die gibt es! Die IDE bietet Ihnen eine mächtige Abkürzung, die Schnittstellen automatisch für Sie implementiert. Beginnen Sie einfach, Ihre Klasse einzugeben:

```
public class
   Mikrowelle :
         ISpeisenGaren
   { }
```

Klicken Sie auf ISpeisenGaren – unter dem »I« erscheint dann ein kleiner Balken. Lassen Sie die Maus über ihm schweben, sehen Sie darunter ein kleines Symbol:

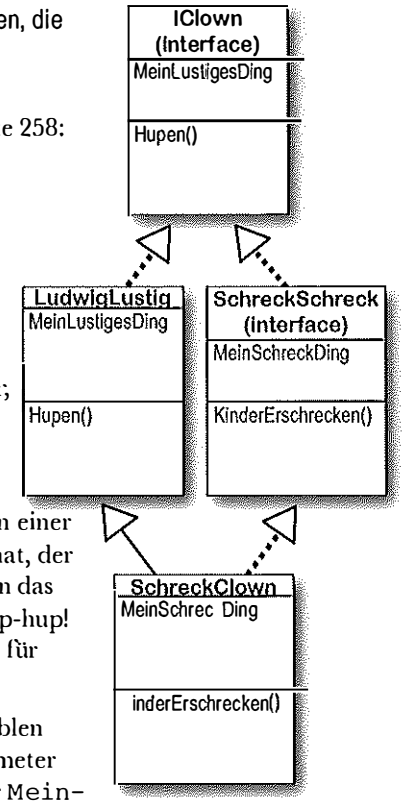


Klicken Sie auf das Symbol und wählen Sie im Menü »'ISpeisenGaren'-Schnittstelle implementieren«. Sie fügt dann automatisch alle Member hinzu, die Sie noch nicht implementiert haben. Jede enthält nur eine `throws`-Anweisung – diese bewirken, dass Ihr Programm angehalten wird, um Sie daran zu erinnern, dass Sie vergessen haben, eine von ihnen zu implementieren (mehr zu `throws` lernen Sie in Kapitel 10).

Eine Schnittstelle ist wie eine Checkliste für den Compiler.



Erweitern Sie die Schnittstelle IClown und nutzen Sie Klassen, die sie implementieren.



1. Nehmen Sie die Schnittstelle `IClown` aus dem »Tun Sie das!« von Seite 258:

```
public interface IClown {
    string MeinLustigesDing { get; }
    void Hupen();
}
```

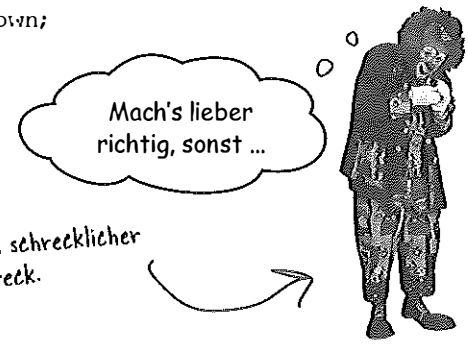
2. Erweitern Sie `IClown`, indem Sie die Schnittstelle `ISchreckClown` erstellen, die von `IClown` erbt. Sie sollten eine weitere `string`-Eigenschaft namens `MeinSchreckDing` mit Getter, aber ohne Setter, und die `void`-Methode `KinderErschrecken()` erstellen.

3. Erstellen Sie diese Klassen:

   - Ein Klasse namens `LustigLustig`, die ein lustiges Ding in einer privaten String-Variablen speichert und einen Konstruktor hat, der einen Parameter namens `lustigesDing` erwartet, auf den das private Feld gesetzt wird. Die Methode `Hupen()` sollte »Hup-hup! Ich habe « gefolgt von dem lustigen Ding sagen. Der Getter für `MeinLustigesDing` sollte das gleiche Ding liefern.
   - Ein Klasse `SchreckSchreck`, die in einer privaten Variablen einen int speichert, der an ihren Konstruktor über den Parameter `anzahlSchreckDinge` übergeben wurde. Der Getter für `MeinSchreckDing` sollte einen String liefern, der aus der an den Konstruktor übergebenen Zahl und dem String » Spinnen« besteht. Die Methode `Hupen()` zeigt einen Dialog, der »Buh! Hab dich!« sagt.

4. Hier ist der Code für einen Button – der allerdings nicht funktioniert. Haben Sie eine Idee, wie man ihn reparieren könnte?

```
private void button1_Click(object sender, EventArgs e) {
    SchreckSchreck maxeDerClown = new SchreckSchreck("Große Schuhe", 14);
    LustigLustig lustigerClown = maxeDerClown;
    ISchreckClown schreckClown = lustigerClown;
    schreckClown.KinderErschrecken();
```





Lösungen zu den Übungen

Erweitern Sie die Schnittstelle IClown und nutzen Sie Klassen, die sie implementieren.

```
public interface IClown {
    string MeinLustigesDing { get; }
    void Hupen();
}

public interface ISchreckClown : IClown {
    string MeinSchreckDing { get; }
    void KinderErschrecken();
}

public class LustigLustig : IClown {
    public LustigLustig(string lustigesDing) {
        this.lustigesDing = lustigesDing;
    }
    private string lustigesDing;
    public string MeinLustigesDing {
        get { return "Hup-hup! Ich habe " + lustigesDing; }
    }

    public void Hupen() {
        MessageBox.Show(this.MeinLustigesDing);
    }
}

public class SchreckSchreck : LustigLustig, ISchreckClown {
    public SchreckSchreck(string lustigesDing, int anzahlSchreckDinge)
        : base(lustigesDing) {
        this.anzahlSchreckDinge = anzahlSchreckDinge;
    }

    private int anzahlSchreckDinge;
    public string MeinSchreckDing {
        get { return "Ich habe " + anzahlSchreckDinge + " Spinnen"; }
    }

    public void KinderErschrecken() {
        MessageBox.Show("Buh! Hab dich!");
    }
}

private void button1_Click(object sender, EventArgs e) {
    SchreckSchreck maxeDerClown = new SchreckSchreck("Große Schuhe", 14);
    LustigLustig lustigerClown = maxeDerClown;
    ISchreckClown schreckClown = lustigerClown as SchreckSchreck;
    schreckClown.Hupen();
}
```

Die Methode Hupen() nutzt diesen Getter, um die Meldung anzuzeigen – sie muss im Code also nicht zweimal auftauchen.

Sie hätten die IClown-Member erneut implementieren können, aber warum sollen wir diese nicht einfach von LustigLustig erben?

Da SchreckSchreck eine Unterklasse von LustigLustig ist und LustigLustig IClown implementiert, tut SchreckSchreck das auch.

Sie können die LustigLustig-Referenz auf das SchreckSchreck-Objekt zeigen lassen, da SchreckSchreck von LustigLustig erbt. Aber eine ISchreckClown-Referenz können Sie nicht auf beliebige Clowns setzen, weil Sie nicht wissen, ob diese ein Schreck-Clown ist. Deswegen müssen Sie dieses Schlüsselwort nutzen.

Es gibt nicht nur public und private

Die Methoden, Felder und Eigenschaften einer Klasse bezeichnen wir als ihre Member (Mitglieder). Jedes Member kann mit den Zugriffsmodifizierern public oder private markiert werden.

Sie wissen bereits, wie wichtig das Schlüsselwort `private` ist, wie Sie es verwenden und wie es sich von `public` unterscheidet. C# hat einen Namen für sie: Sie werden als **Zugriffsmodifizierer bezeichnet**. Der Name ist sinnvoll, weil man mit einer Änderung des Zugriffsmodifizierers einer Eigenschaft, eines Felds oder einer Methode einer Klasse (ihren **Membern**) oder des Zugriffsmodifizierers der ganzen Klasse ändert, wie andere Klassen darauf zugreifen können. Es gibt ein paar weitere, die Sie verwenden werden, aber wir werden mit denen beginnen, die Sie kennen.

`public` bedeutet, dass jeder darauf zugreifen kann.

Markieren Sie eine Klasse oder ein Klassenmember als `public`, sagen Sie C#, dass jede Instanz anderer Klassen darauf zugreifen kann. Das ist der am wenigsten einschränkende Zugriffsmodifizierer. Und Sie haben bereits gesehen, wie er Sie in Bedrängnis bringen kann – markieren Sie Klassenmember nur dann als public, wenn Sie einen guten Grund dafür haben. So stellen Sie sicher, dass Ihre Klassen gut gekapselt sind.

`private` bedeutet, dass nur andere Member darauf zugreifen können.

Markieren Sie ein Klassenmember als `private`, können darauf nur andere Member dieser Klasse oder **andere Instanzen dieser Klasse zugreifen**. Eine Klasse können Sie nur als `private` markieren, wenn sie in eine andere Klasse eingebettet ist. Sie ist dann standardmäßig privat und nur für Instanzen der Containerklasse zugreifbar. Soll sie öffentlich sein, müssen Sie sie explizit als `public` markieren.

`protected` bedeutet public für Unterklassen und private für alle anderen.

Sie haben bereits gesehen, dass eine Unterklasse auf die privaten Felder ihrer Basisklasse zugreifen kann – sie muss das Schlüsselwort `base` verwenden, um an die öffentlichen Member des Basisobjekts heranzukommen. Wäre es nicht praktischer, wenn die Unterklasse auch auf die privaten Felder zugreifen könnte? Zu diesem Zweck gibt es den Zugriffsmodifizierer `protected`. Auf ein Klassenmember, das mit `protected` markiert ist, können alle Member dieser Klasse und alle Member ihrer Unterklassen zugreifen.

`internal` bedeutet public nur für die Klassen in einer Assembly.

Die eingebauten Klassen des .NET Framework stecken in **Assemblies** – Klassenbibliotheken, die Sie in der Verweisliste Ihres Projekts angeben können. Eine Liste von Assemblies können Sie sich anzeigen lassen, indem Sie mit der rechten Maustaste auf »Verweise« klicken und »Verweis hinzufügen ...« wählen – erstellen Sie eine neue Windows-Form-Anwendung, schließt die IDE automatisch die Referenzen ein, die erforderlich sind, um eine Windows-Form-Anwendung zu erstellen. Wenn Sie eine Assembly erstellen, erreichen Sie mit dem Schlüsselwort `internal`, dass die so markierten Klassen für die Assembly privat sind. Sie müssen also nur die Klassen veröffentlichen, die Sie möchten.

`sealed` sagt, dass von dieser Klasse keine Unterklassen erben können.

Es gibt einige Klassen, von denen man nicht erben kann. Das gilt für viele Klassen des .NET Framework – probieren Sie beispielsweise, eine Klasse zu erstellen, die von `String` erbt (das ist die Klasse, deren IsEmptyOrNull()-Methode Sie im letzten Kapitel verwendet haben). Was passiert? Der Compiler lässt Sie Ihren Code nicht erstellen – er meldet Ihnen den Fehler »Vom versiegelten Typ 'string' kann nicht abgeleitet werden«. Das können Sie auch mit Ihren eigenen Klassen machen – fügen Sie nach dem Zugriffsmodifizierer einfach `sealed` hinzu.

Zugriffsmodifizierer ändern Geltungsbereiche

Werfen wir einen genaueren Blick auf Zugriffsmodifizierer und die Auswirkungen, die sie auf den **Geltungsbereich** der verschiedenen Klassenmember haben. Wir haben zwei kleine Änderungen vorgenommen: Das Hintergrundfeld lustigesDing ist jetzt `protected`, und wir haben die Methode KinderErschrecken() so geändert, dass sie das Feld lustigesDing nutzt.

Nehmen Sie diese beiden Änderungen an Ihrer eigenen Übungslösung vor. Ändern Sie dann den Zugriffsmodifizierer protected wieder in private und schauen Sie, welche Fehler Sie erhalten.

Das Schlüsselwort »this« ändert ebenfalls den Geltungsbereich einer Variablen. Es sagt C#: »Betrachte die aktuelle Instanz der Klasse, um herauszufinden, womit ich verbunden bin – selbst wenn der Name dem eines Parameters oder einer lokalen Variablen entspricht«.

Das ist eine ziemlich häufige Verwendungsweise von »this«, da der Parameter und das Hintergrundfeld den gleichen Namen haben. lustigesDing verweist auf den Parameter, this.lustigesDing auf das Hintergrundfeld.

```
public interface IClown {
    string MeinLustigesDing { get; }
    void Hupen();
}

public interface ISchreckClown : IClown {
    string MeinSchreckDing { get; }
    void KinderErschrecken();
}

public class LustigLustig : IClown {
    public LustigLustig(string lustigesDing) {
        this.lustigesDing = lustigesDing;
    }
    protected string lustigesDing;
    public string MeinLustigesDing {
        get { return "Hup-hup! Ich habe " + lustigesDing; }
    }

    public void Hupen() {
        MessageBox.Show(this.MeinLustigesDing);
    }
}
```

Mit »this« sagen wir C#, dass wir das Hintergrundfeld meinen, nicht den Parameter gleichen Namens.

Das haben wir in protected geändert. Schauen Sie, welche Auswirkungen das auf die Methode SchreckSchreck.KinderErschrecken() hat.

Nutzen Sie »this« mit einer Eigenschaft, sagen Sie C#, dass es den Getter oder Setter ausführen soll.

Zugriffsmodifizierer unter der Lupe



```
public class SchreckSchreck : LustigLustig, ISchreckClown {
    public SchreckSchreck(string lustigesDing,
                          int anzahlSchreckDinge)
        : base(lustigesDing) {
        this.anzahlSchreckDinge = anzahlSchreckDinge;
    }

    private int anzahlSchreckDinge;
    public string MeinSchreckDing {
        get { return "Ich habe " + anzahlSchreckDinge + " Spinnen"; }
    }

    public void KinderErschrecken() {
        MessageBox.Show("Du kriegst mein "
                        + base.lustigesDing + " nicht");
    }
}

private void button1_Click(object sender, EventArgs e) {
    SchreckSchreck maxeDerClown = new SchreckSchreck("Große Schuhe", 14);
    LustigLustig lustigerClown = maxeDerClown;
    ISchreckClown schreckClown = lustigerClown as SchreckSchreck;
    schreckClown.Hupen();
}
```

anzahlSchreckDinge ist private. Das ist bei einem Unterstützungsfeld üblich. Dann können nur andere Instanzen von SchreckSchreck darauf zugreifen.

Das Schlüsselwort protected sagt C#, dass es etwas für alle privat machen soll außer für Instanzen von Unterklassen.

Das Schlüsselwort »base« sagt C#, dass es den Wert aus der Basisklasse verwenden soll. Das ist ein anderer Weg, den Geltungsbereich zu ändern.

Hätten wir lustigesDing privat gelassen, hätte Ihnen der Compiler einen Fehler gemeldet. Aber durch die Änderung in protected haben wir das Feld für alle Unterklassen von LustigLustig sichtbar gemacht.

Da dieser Click-Handler für einen Button nicht Teil von LustigLustig und SchreckSchreck, ist, kann er auf das geschützte Feld lustigesDing nicht zugreifen.

Er befindet sich außerhalb der beiden Klassen. Die Anweisungen darin haben also nur Zugriff auf die öffentlichen Member von LustigLustig- oder SchreckSchreck-Objekten.

Es gibt keine Dummen Fragen

F: Warum sollte ich eine Schnittstelle verwenden und nicht einfach alle benötigten Methoden gleich in meine Klassen schreiben?

A: Das könnte Ihnen eine Menge unterschiedlicher Klassen eintragen, wenn Ihre Programme langsam komplexer werden. Mit Schnittstellen können Sie diese Klassen anhand der Arbeit gruppieren, die diese erledigen. Sie helfen Ihnen sicherzustellen, dass alle Klassen, die eine bestimmte Art von Aufgabe erfüllen, dazu die gleichen Methoden verwenden. Die Klasse kann die Arbeit machen, die sie erledigen muss, und Sie müssen sich wegen der Schnittstelle keine Gedanken darüber machen, wie diese ihre Arbeit macht.

Hier ist ein Beispiel: Sie können eine Bus- und eine Segelboot-Klasse haben, die `IPassagiereTragen` implementieren. Nehmen wir an, die Schnittstelle `IPassagiereTragen` fordert, dass jede Klasse, die sie implementiert, die Methode `EnergieVerbrauchen()` hat. Ihr Programm könnte beide einsetzen, um Passagiere zu transportieren, obwohl die `EnergieVerbrauchen()`-Methode von Segelboot Windkraft nutzt und die von Bus Dieseltreibstoff.

Stellen Sie sich vor, Sie hätten die Schnittstelle `IPassagiereTragen` nicht. Dann wäre es schwerer, Ihrem Programm zu sagen, welche Fahrzeuge Personen transportieren können und welche nicht. Sie müssten in jede Klasse blicken, die Ihr Programm verwenden könnte, und herausfinden, ob diese eine Methode zum Transport von Passagieren bietet oder nicht. Dann müssten Sie alle Fahrzeuge, die Ihr Programm nutzt, mit der jeweils definierten Klasse für den Personentransport verwenden. Und da es keine Standardschnittstelle gibt, könnten diese alle möglichen Namen haben oder in anderen Methoden verborgen sein. Das kann schnell verwirrend werden.

F: Warum muss Ich eine Eigenschaft verwenden? Kann ich nicht einfach ein Feld einschließen?

A: Gute Frage. Eine Schnittstelle definiert nur einen Weg, auf dem eine Klasse eine bestimmte Aufgabe erledigen soll. Es ist kein Klasse, die Sie initialisieren können, und kann deswegen keine Informationen speichern. Würden Sie ein Feld hinzufügen, wäre das wie eine Variablendeklaration, und C# müsste diese Daten dann irgendwo speichern – Schnittstellen können keine Daten speichern.

F: Was ist der Unterschied zwischen einer richtigen Objektreferenz und einer Schnittstellenreferenz?

A: Sie wissen bereits, wie eine gewöhnliche Objektreferenz funktioniert. Erstellen Sie eine Instanz von `Skateboard` namens `vertBoard` und dann eine neue Referenz darauf, die `halfPipeBoard` heißt, dann zeigen diese auf das gleiche Ding. Aberwenn `Skateboard` die Schnittstelle `IStreetTricks` implementiert, können Sie eine Schnittstellenreferenz auf ein `Skateboard`-Objekt zeigen lassen, die nur die Methoden der Klasse `Skateboard` kennt, die sich auch in der Schnittstelle `IStreetTricks` befinden.

Alle drei Referenzen zeigen eigentlich auf das gleiche Objekt. Rufen Sie das Objekt über die Referenzen `halfPipeBoard` oder `vertBoard` auf, können Sie auf alle Klassenmember zugreifen. Rufen Sie es über die `IStreetBoard`-Referenz auf, haben Sie nur Zugriff auf die Schnittstellenmember.

F: Warum sollte ich denn Schnittstellenreferenzen verwenden, wenn diese einschränken, was ich mit dem Objekt machen kann?

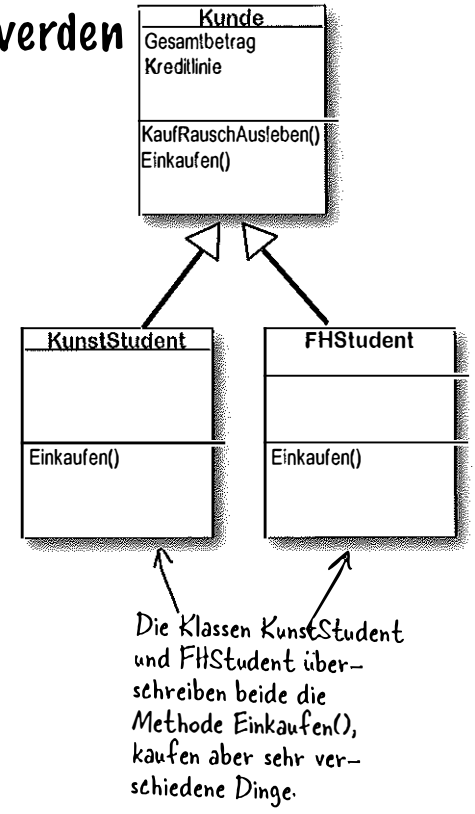
A: Schnittstellenreferenzen geben Ihnen eine Möglichkeit, mit Objekten unterschiedlicher Klassen zu arbeiten, die das Gleiche tun. Sie können ein Array mit dem Typ der Schnittstelle erstellen, über das Sie Informationen mit `IPassagiereTragen`-Objekten austauschen können, ganz gleich, ob das jeweilige Objekt ein Bus-, Pferd-, Einrad- oder Auto-Objekt ist. Diese Objekte erledigen diese Aufgabe wahrscheinlich alle etwas anders. Aber die Schnittstellenreferenz sichert, dass sie alle die gleichen Methoden mit den gleichen Parametern und Rückgabewerten besitzen. Sie können sie also alle auf die gleiche Weise aufrufen, um Informationen mit ihnen auszutauschen.

F: Warum sollte Ich etwas protected statt private oder public machen?

A: Weil Ihnen das hilft, Ihre Klassen besser zu kapseln. Häufig muss eine Unterklasse auf einen internen Teil seiner Basisklasse zugreifen. Beim Überschreiben von Eigenschaften kommt es beispielsweise recht häufig vor, dass man im Getter das Unterstützungsfeld in der Basisklasse verwendet und eine verarbeitete Version davon liefert. Aber wenn Sie Klassen erstellen, sollten Sie Dinge nur dann öffentlich machen, wenn Sie einen guten Grund dazu haben. Mit dem Zugriffsmodifizierer protected können Sie etwas nur für die Unterklassen veröffentlichen, die es brauchen, und für alle anderen privat halten.

Schnittstellenreferenzen kennen nur die in der Schnittstelle definierten Member.

Manche Klassen sollten nicht instantiiert werden



Erinnern Sie sich an die Klassenhierarchie unseres Zoosimulators? Ganz sicher instantiieren Sie irgendwann Nilpferde, Hunde oder Löwen. Aber was ist mit den Klassen Hundeartig und Katzenartig? Und was ist mit der Klasse Tier? Es scheint, als gäbe es Klassen, die nicht instantiiert werden müssen ... deren Instantiierung eigentlich sogar überhaupt keinen Sinn ergibt. Hier ist ein Beispiel.

Beginnen wir mit einer Klasse für einen Studenten, der in der Universitätsbuchhandlung einkauft.

```
public class Kunde {
    public void KaufRauschAusleben()
        while (Gesamtbetrag < Kreditlinie)
            Einkaufen();
    }
    public virtual void Einkaufen () {
        // Keine Implementierung - wir wissen nicht,
        // was der Student kaufen will!
    }
}
```

Hier ist die Klasse KunstStudent – eine Unterklasse von Kunde:

```
public class KunstStudent : Kunde {
    public override void Einkaufen () {
        FarbenKaufen();
        SchwarzerRolliKaufen();
        DepressiveMusikKaufen();
    }
}
```

Auch die Klasse FHStudent erbt von Kunde:

```
public class FHStudent : Kunde {
    public override void Einkaufen () {
        BleistifteKaufen();
        TaschenrechnerKaufen();
        ComputerspielKaufen();
    }
}
```

Was passiert, wenn Sie Kunde instantiieren? Kann das sinnvoll sein?

Eine abstrakte Klasse ist vergleichbar mit einer Kreuzung zwischen normaler Klasse und Schnittstelle

Angenommen, Sie brauchen so etwas wie eine Schnittstelle, das von Klassen verlangt, dass sie bestimmte Methoden und Eigenschaften implementieren. Aber Sie müssen in diese Schnittstelle etwas Code einschließen, damit bestimmte Methoden nicht in jeder Unterklasse separat implementiert werden müssen. Dann brauchen Sie eine **abstrakte Klasse**. Sie erhalten die Funktionalität einer Schnittstelle, können aber Code schreiben wie in einer gewöhnlichen Klasse.

Eine Methode, die eine Deklaration enthält, aber keinen Methodenrumpf, wird als abstrakte Methode bezeichnet. Unterklassen müssen alle abstrakten Methoden implementieren, genau als würden sie eine Schnittstelle implementieren.

Nur abstrakte Klassen können abstrakte Methoden haben. Stecken Sie in eine Klasse eine abstrakte Methode, dann müssen Sie die Klasse als abstract markieren, sonst lässt sie sich nicht kompilieren. Wie das geht, erklären wir Ihnen in Kürze.

Ein abstrakte Klasse ist wie eine gewöhnliche Klasse.
Abstrakte Klassen definieren Sie wie normale Klassen. Sie haben Felder und Methoden und können, auch genau wie normale Klassen, von anderen Klassen erben. Sie müssen fast nichts Neues lernen, weil Sie bereits alles wissen, was eine abstrakte Klasse ausmacht.

Eine abstrakte Klasse ist wie eine Schnittstelle.
Wenn Sie eine Klasse implementieren, sagen Sie zu, dass Sie alle Member implementieren, die die Schnittstelle definiert. Eine abstrakte Klasse funktioniert genau so – sie kann Deklarationen für Eigenschaften und Methoden enthalten, die von Unterklassen implementiert werden müssen.

Das Gegenteil von abstrakt ist konkret. Eine konkrete Methode ist eine Methode mit Rumpf und alle Klassen, mit denen Sie bisher gearbeitet haben, waren konkrete Klassen.

Aber eine abstrakte Klasse kann nicht instantiiert werden.
Der größte Unterschied zwischen einer **abstrakten** Klasse und einer **konkreten** Klasse ist, dass Sie eine abstrakte Klasse nicht mit `new` instantiieren können. Tun Sie das, meldet Ihnen C# einen Fehler, wenn Sie versuchen, Ihren Code zu kompilieren.

```
Es konnte keine Instanz der abstrakten Klasse
oder Schnittstelle 'EineKlasse' erstellt werden.
```

Diesen Fehler erhalten Sie, weil Sie abstrakte Methoden ohne Code haben! Klassen, denen Code fehlt, oder Schnittstellen lässt der Compiler Sie nicht instantiieren.



Moment ... eine Klasse, die ich nicht instantiieren kann? Wozu brauche ich so etwas?

Weil Sie etwas Code anbieten möchten, die Unterklassen den restlichen Code aber immer noch selbst füllen sollen.

Gelegentlich ***passieren schlimme Dinge***, wenn Sie Objekte erstellen, die nicht erstellt werden sollten. Die Klasse an der Spitze Ihres Klassendiagramms hat in der Regel Felder, die von Unterklassen gesetzt werden sollten. Ein Klasse Tier könnte eine Berechnung verwenden, die von einem bool-Feld namens HatSchwanz oder Wirbeltier abhängt, das sie selbst nicht setzen kann.

Diese Klasse nutzen die Sternfahrer Objekthausen, um Raketen zu verschiedenen Planeten zu schicken.

Hier ist ein Beispiel ...

Es ist sinnlos, diese Felder in der Basisklasse zu setzen, weil wir nicht wissen, welcher Planet bzw. welche Rakete verwendet wird.

```
public class PlanetMission {
  public long TreibstoffProKM;
  public long Geschwindigkeit;
  public long Entfernung;

  public long TreibstoffBedarf() {
    return Entfernung * TreibstoffProKM;
  }

  public long ZeitBedarf() {
    return Entfernung / Geschwindigkeit;
  }

  public string MissionBerechnen() {
    return "Sie benötigen "
      + Entfernung * TreibstoffProKM
      + " Treibstoffeinheiten.\r\nDie Reise wird "
      + ZeitBedarf() + " Stunden dauern.";
  }
}
```

Die Sternfahrer haben zwei Missionen – eine zum Mars, eine zur Venus.

```
public class Venus : PlanetMission {
    public Venus() {
        Entfernung = 40000000;
        TreibstoffProKM = 100000;
        Geschwindigkeit = 25000;
    }
}

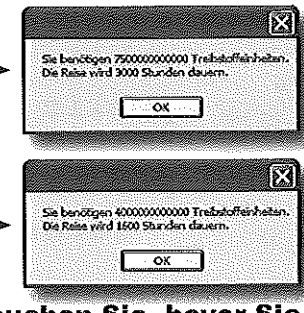
public class Mars : PlanetMission {
    public Mars() {
        Entfernung = 75000000;
        TreibstoffProKM = 100000;
        Geschwindigkeit = 25000;
    }
}
```

Die Konstruktoren für die Unterklassen Mars und Venus setzen die drei von PlanetMission geerbten Felder. Aber diese Felder würden nicht gesetzt, würden Sie PlanetMission direkt instantiieren. Was also passiert, wenn MissionBerechnen() versucht, sie zu verwenden?

```
private void button1_Click(object s, EventArgs e) {
  Mars mars = new Mars();
  MessageBox.Show(mars.MissionBerechnen());
}

private void button2_Click(object s, EventArgs e) {
  Venus venus = new Venus();
  MessageBox.Show(venus.MissionBerechnen());
}

private void button3_Click(object s, EventArgs e) {
  PlanetMission planet = new PlanetMission();
  MessageBox.Show(planet.MissionBerechnen());
}
```



Versuchen Sie, bevor Sie umblättern, herauszufinden, was passiert, wenn der Benutzer auf den Button klickt.

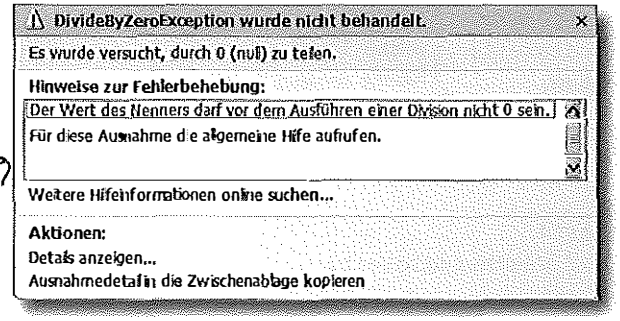
Manche Klassen sollten nicht instantiiert werden

Das Problem beginnt, wenn Sie eine Instanz der Klasse PlanetMission erstellen. Ihre `MissionBerechnen()`-Methode erwartet, dass die Felder von der Unterklasse gesetzt werden. Erfolgt das nicht, erhalten sie ihren Standardwert – null. Und wenn C# versucht, durch null zu teilen …

```
private void button3_Click(object s, EventArgs e) {
  PlanetMission planet = new PlanetMission();
  MessageBox.Show(planet.MissionBerechnen());
}
```

Es war nicht geplant, dass die Klasse PlanetMission instantiiert wird. Stattdessen sollte sie nur geerbt werden. Aber wir haben es gemacht, und damit beginnen die Probleme.

Als die Methode MissionBerechnen() versuchte, durch die Geschwindigkeit zu teilen, war diese null. Und das passiert, wenn Sie durch null teilen.



Lösung: Nutzen Sie eine abstrakte Klasse!

Markieren Sie eine Klasse als `abstract`, lässt C# Sie keinen Code schreiben, der diese instantiiert. Sie hat dann große Ähnlichkeit mit einer Schnittstelle – sie dient als Schablone für die Klassen, die von ihr erben.

Das Schlüsselwort `abstract` in der Klassendeklaration sagt C#, dass diese Klasse abstrakt ist und nicht instantiiert werden darf.

```
public abstract class PlanetMission {
  public long TreibstoffProKM;
  public long Geschwindigkeit;
  public int Entfernung;

  public long TreibstoffBedarf() {
    return Entfernung * TreibstoffProKM;
  }

  // Hier folgt der Rest der Klasse ...
}
```

Jetzt weigert sich C#, unser Programm zu kompilieren, bis wir die Zeile entfernt haben, die eine Instanz von PlanetMission erstellt.

Eine abstrakte Methode hat keinen Rumpf

Alle Methoden in Schnittstellen sind automatisch abstrakt. Sie müssen das Schlüsselwort `abstract` in Schnittstellen also nicht angeben. In abstrakten Klassen schon, da diese abstrakte und konkrete Methoden enthalten können.

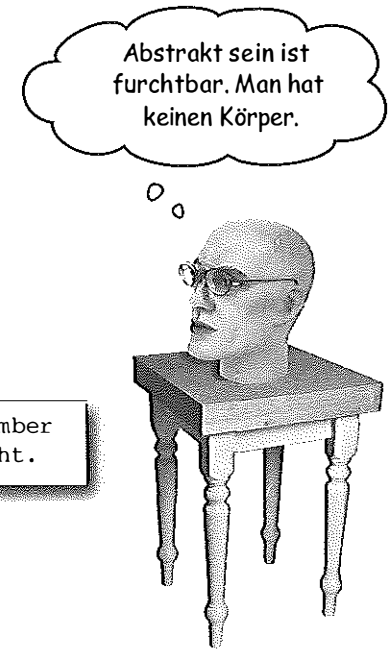
Sie wissen, dass eine Schnittstelle Deklarationen für Methoden und Eigenschaften, aber keine Implementierungen für diese enthält. Das liegt daran, dass jede Methode in einer Schnittstelle eine **abstrakte Methode** ist. Auch in abstrakten Klassen können Sie abstrakte Methoden haben – müssen diese aber explizit mit dem Schlüsselwort **`abstract`** markieren, weil abstrakte Klassen auch ganz gewöhnliche Methoden enthalten können.

```
public abstract class PlanetMission {

    public abstract void MissionEinrichten(
        int Entfernung, int TreibstoffProKM,
        long Geschwindigkeit);

    // Der Rest der Klasse ...
```

Diese abstrakte Methode ist wie die in einer Schnittstelle – sie hat keinen Rumpf und muss von jeder Klasse implementiert werden, die von PlanetMission erbt. Sonst lässt das Programm sich nicht kompilieren.



Fügen wir diese Methode ein und versuchen wir, das Programm zu kompilieren, liefert uns die IDE einen Fehler.

```
'VenusMission' implementiert das geerbte abstrakte Member
'PlanetMission.MissionEinrichten(long, int, int)' nicht.
```

Also implementieren wir sie doch! Dann verschwindet auch der Fehler!

```
public class Venus : PlanetMission {
    public Venus() {
        MissionEinrichten(40000000, 100000, 25000);
    }
    public override MissionEinrichten(int entfernung,
                                      long treibstoffProKM, int geschwindigkeit) {
        this.Entfernung = entfernung;
        this.TreibstoffProKM = treibstoffProKM;
        this.Geschwindigkeit = geschwindigkeit;
    }
}
```

Spitzen Sie Ihren Bleistift

Hier können Sie Ihre künstlerischen Fähigkeiten demonstrieren. Auf der linken Seite finden Sie Sätze von Klassen- und Schnittstellendeklarationen. Ihre Aufgabe ist es, auf der rechten Seite die entsprechenden Klassendiagramme zu zeichnen. Das erste haben wir für Sie erstellt. Vergessen Sie nicht, für Schnittstellenimplementierungen gestrichelte Linien zu verwenden und durchgezogene für das Erben von Klassen.

Code:

1)
```
public interface Foo { }
public class Bar : Foo { }
```

2)
```
public interface Vinn { }
public abstract class Vout : Vinn { }
```

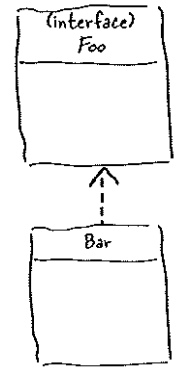
3)
```
public abstract class Muffie : Whuffie { }
public class Fluffie : Muffie { }
public interface Whuffie { }
```

4)
```
public class Zoop { }
public class Boop : Zoop { }
public class Goop : Boop { }
```

5)
```
public class Gamma : Delta, Epsilon { }
public interface Epsilon { }
public interface Beta { }
public class Alpha : Gamma,Beta { }
public class Delta { }
```

Wie sieht das Bild aus?

1)



2)

3)

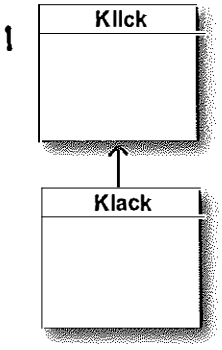
4)

5)

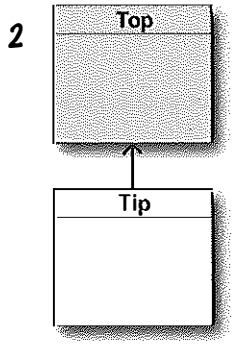
Auf der linken Seite sehen Sie Klassendiagramme. Sie haben die Aufgabe, diese in gültige C#-Deklarationen zu verwandeln. Nummer 1 haben wir bereits erledigt.

Diagramme:

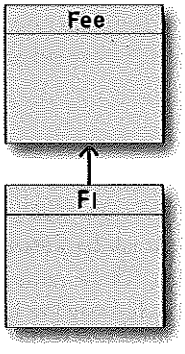
1



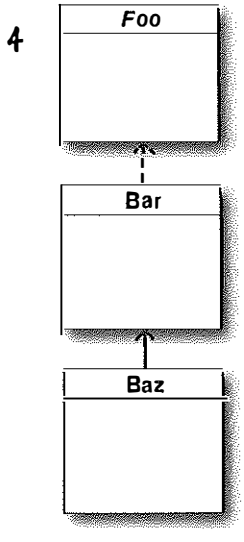
2



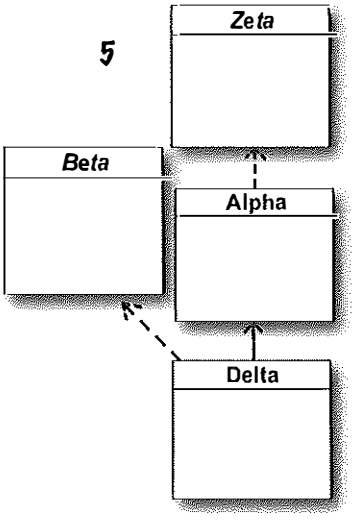
3



4



5



Wie sieht die Deklaration aus?

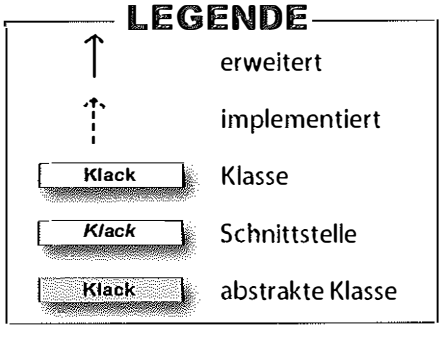
1) `public class Klick { }`
`public class Klack : Klick { }`

2)

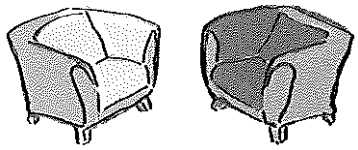
3)

4)

5)



Kamingespräche



Heute Abend: **Eine abstrakte Klasse und eine Schnittstelle kriegen sich über die drängende Frage »Wer ist wichtiger?« in die Haare.**

Abstrakte Klasse:

Meiner Meinung nach ist klar, wer von uns beiden wichtiger ist. Programmierer müssen ihre Arbeit erledigen. Machen wir uns doch nichts vor: In der Hinsicht bringst du doch fast nichts.

Du kannst doch nicht wirklich denken, dass du wichtiger bist als ich. Was du machst, ist ja nicht mal richtige Vererbung. Was wirst du noch mal – implementiert?

Besser? Du spinnst doch. Ich bin viel flexibler als du. Ich kann konkrete Methoden haben oder auch abstrakte. Und wenn ich möchte, kann ich sogar virtuelle Methoden haben. Gut, ich kann nicht istantiiert werden ... aber du ja schließlich auch nicht. Und ich kann so gut wie alles machen, was eine gewöhnliche Klasse kann.

Schnittstelle:

Nett. Scheint ja fast, als könnte das interessant werden.

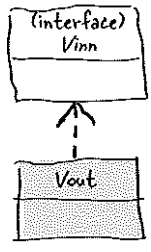
Das schon wieder: Schnittstellen sind nicht einmal richtige Vererbung. Schnittstellen werden nur implementiert. Das ist doch vollkommen ignorant. Implementierung ist genauso gut wie Vererbung, eigentlich ist es sogar besser!

Ja? Und was ist, wenn eine Klasse gleichzeitig von dir ***und*** einem deiner Kumpel erben will? **Von zwei Klassen gleichzeitig kann man nicht erben.** Man muss sich entschließen, von welcher Klasse man erben will. Das ist absolut rücksichtslos! Schnittstellen kann eine Klasse so viele implementieren, wie sie will. Wie war das noch mal – flexibel? Mit mir kann ein Programmierer seine Klassen dazu bringen, alles zu tun.

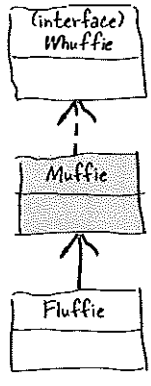
Spitzen Sie Ihren Bleistift

Lösung

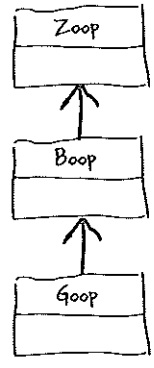
2)



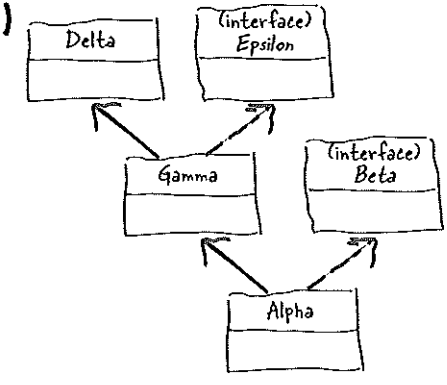
3)



4)



5)



Wie sieht das Bild aus?

Abstrakte Klasse:

Übertreibst du da nicht etwas?

Das ist genau das Geschwafel, das ich von einer Schnittstelle erwartet habe. Code ist ungemein wichtig! Das ist das, was Programme zum Laufen bringt.

Meinst du? Das würde ich bezweifeln – Programmierer wollen immer wissen, was in ihren Eigenschaften und Methoden steckt.

Klar doch. Als ob Programmierer programmieren könnten.

Schnittstelle:

Du meinst wohl auch, nur weil du Code enthalten kannst, wärst du die größte Erfindung, seit Brot geschnitten werden kann. Aber du kannst es nicht ändern, Freund – eine Klasse kann immer nur von einer einzigen Klasse erben. Ich würde das beschränkt nennen. Gut, ich kann keinen Code einschließen. Aber Mensch, Code wird einfach etwas überschätzt.

In neun von zehn Fällen geht es Programmierern darum sicherzugehen, dass ein Objekt bestimmte Eigenschaften und Methoden hat. Wie die implementiert sind, interessiert ihn überhaupt nicht.

Klar. Irgendwann. Aber überleg mal, wie oft dir Programmierer untergekommen sind, die Methoden schreiben, die ein Objekt erwarten, die nur eine bestimmte Methode haben müssen, und denen es in dem Augenblick vollkommen egal ist, wie die Methode aufgebaut ist. Sie ist einfach nur da. Bumm! Der Programmierer schreibt eine Schnittstelle und ... das Problem ist gelöst!

Oh nee ... ***ich geb's auf***!

---

2)
```
public abstract class Top { }
public class Tip : Top { }
```

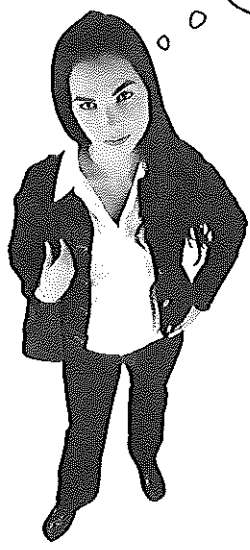
3)
```
public abstract class Fee { }
public abstract class Fi : Fee { }
```

4)
```
public interface Foo { }
public class Bar : Foo { }
public class Baz : Bar { }
```

5)
```
public interface Zeta { }
public class Alpha : Zeta { }
public interface Beta { }
public class Delta : Alpha, Beta { }
```

Delta erbt von Alpha und implementiert Beta.

Wie sieht die Deklaration aus?



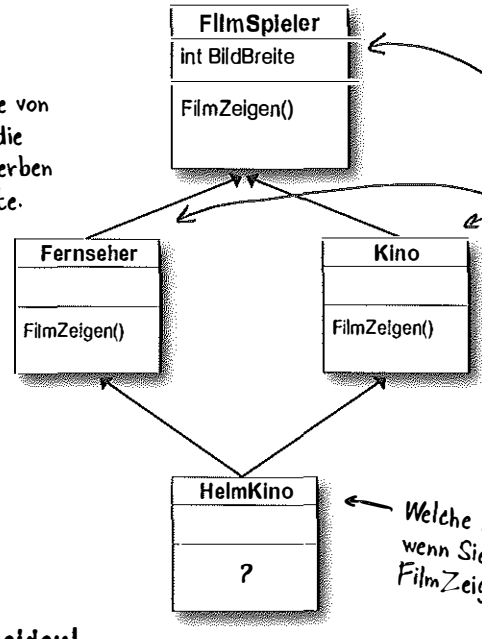
Ich verstehe nicht so ganz, warum ich nicht von zwei Klassen erben kann. Wenn ich von mehreren Klassen erben muss, kann ich also nur Schnittstellen verwenden. Das ist doch eine ziemlich große Beschränkung von C#, oder?

Es ist keine Beschränkung, es ist ein Schutz.

Würde C# Ihnen gestatten, von mehreren Klassen zu erben, würde das eine Menge Probleme mit sich bringen. Erlaubt eine Sprache Klassen, von mehreren Basisklassen zu erben, nennt man das **Mehrfachvererbung**. Und indem C# Ihnen stattdessen Schnittstellen gibt, schützt es Sie vor einem riesigen Unglück, das wir gerne so bezeichnen ...

Der Deadly Diamond of Death!

Fernseher und Kino erben beide von FilmSpieler und überschreiben die Methode FilmZeigen(). Beide erben auch die Eigenschaft BildBreite.

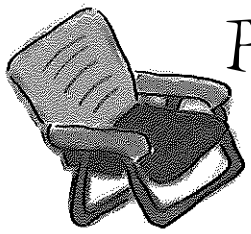


Stellen Sie sich vor, dass die Eigenschaft BildBreite von Fernseher und Kino jeweils mit unterschiedlichen Werten verwendet wird. Was passiert, wenn HeimKino beide Werte für BildBreite verwenden muss – beispielsweise, um Fernsehspiele und Kinofilme zu zeigen?

Welche FilmZeigen()-Methode läuft, wenn Sie auf einem HeimKino-Objekt FilmZeigen() aufrufen?

Mehrdeutigkeiten vermeiden!

Eine Sprache, die den Deadly Diamond of Death zulässt, kann zu einigen ziemlich hässlichen Situationen führen, weil man besondere Regeln braucht, um mit derartig mehrdeutigen Situationen klarzukommen ... und das bedeutet zusätzliche Arbeit für Sie, während Sie Ihr Programm schreiben! C# schützt Sie davor, sich damit herumschlagen zu müssen, indem es Ihnen Schnittstellen gibt. Sind `Fernseher` und `Kino` Schnittstellen statt Klassen, kann die gleiche `FilmZeigen()`-Methode beide zufriedenstellen. Die Schnittstelle kümmert sich nur darum, ob es eine Methode namens `FilmZeigen()` gibt.



Pool-Puzzle

Sie haben die ***Aufgabe***, die leeren Zeilen im Code mit den Codeschnipseln aus dem Pool zu füllen. Einzelne Schnipsel **dürfen** mehrfach verwendet werden, und Sie werden nicht alle Schnipsel benötigen. Das ***Ziel*** ist es, eine Klasse zu erstellen, die sich kompilieren lässt und die gezeigte Ausgabe erzeugt.

```
public ..... Nase {
  ..... ;
    string Gesicht { get; }
}

public abstract class ..... :.....{
    public virtual int Ohr()
    {
        return 7;
    }
    public Picasso(string gesicht)
    {
     ..... = gesicht;
    }
    public virtual string Gesicht {
     .....{ ..... ; }
    }
    string gesicht;
}

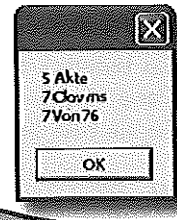
public class ..... :.....{
    public Clowns() : base("Clowns") { }
}

public class ..... :.....{
    public Akte() : base("Akte") { }
    public override ..... {
        return 5;
    }
}

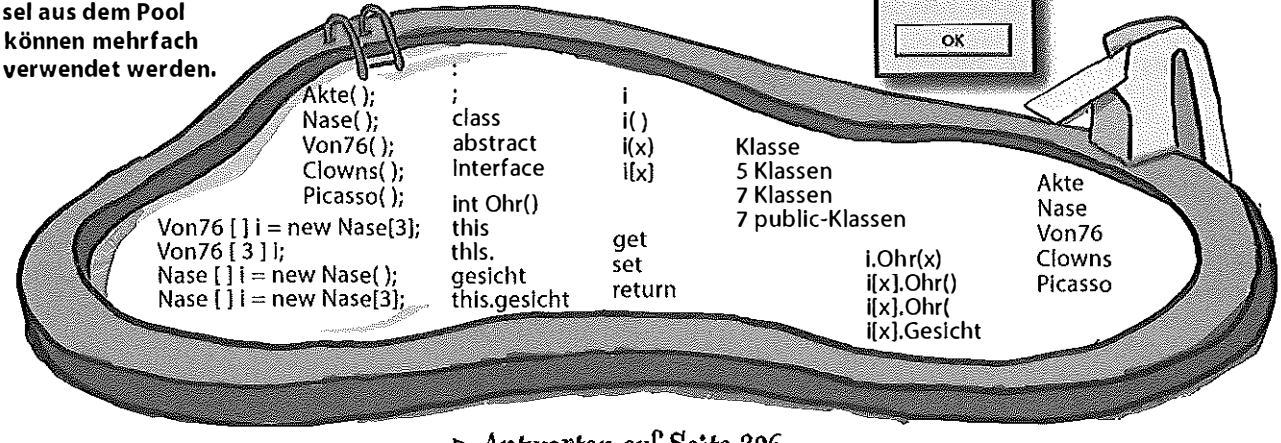
public class ..... :.....{
    public override string Gesicht {
      get { return "Von76"; }
    }
    public static void Main(string[] args) {
        string ergebnis = "";
        Nase[] i = new Nase[3];
        i[0] = new Akte();
        i[1] = new Clowns();
        i[2] = new Von76();
        for (int x = 0; x < 3; x++) {
            ergebnis += (..... + " "
                       + ..... ) + "\n";
        }
        MessageBox.Show(ergebnis);
    }
}
```

Hier ist der Einstiegspunkt – das ist ein vollständiges C#-Programm.

Ausgabe



Hinweis: Die Schnipsel aus dem Pool können mehrfach verwendet werden.



Antworten auf Seite 306.



Der Gedanke, Daten und Code zu Klassen zu kombinieren, war revolutionär, als er eingeführt wurde – aber Sie haben Ihre C#-Programme bis hierhin bereits so aufgebaut und können das für sich als normales Programmieren betrachten.

Sie sind ein objektorientierter Programmierer.

Es gibt einen Namen für das, was Sie die ganze Zeit gemacht haben. Man nennt es **objektorientierte Programmierung** oder OOP. Bevor Programmiersprachen wie C# entworfen wurden, verwendete man beim Programmieren keine Objekte und Methoden. Man verwendete einfach Funktionen (so nennt man Methoden in einem Nicht-OOP-Programm), die sich alle an einem Ort befanden – als wäre jedes Programm eine einzige riesige statische Klasse, die nur statische Methoden besitzt. Damals war es viel schwerer, Programme zu erstellen, die die Probleme modellierten, die wir bislang gelöst haben. Glücklicherweise werden Sie nie Programme ohne OOP schreiben müssen, da es ein Kernkonzept von C# ist.

Die vier Prinzipien der objektorientierten Programmierung

Wenn Programmierer von OOP reden, meinen sie vier wichtige Prinzipien. Diese sollten Ihnen mittlerweile vertraut sein, weil Sie mit jedem von ihnen bereits gearbeitet haben. Die ersten drei werden Sie schon aufgrund ihrer Namen wiedererkennen: **Vererbung**, **Abstraktion** und **Kapselung**. Das letzte wird als **Polymorphie** bezeichnet. Das klingt etwas seltsam, aber es wird sich herausstellen, dass Sie auch dieses bereits kennen.

Das bedeutet einfach, dass Sie eine Klasse oder eine Schnittstelle haben, die von einer anderen erbt.

Vererbung

Kapselung bedeutet, dass man ein Objekt erstellt, das seine internen Zustände in privaten Feldern nachhält und öffentliche Eigenschaften und Methoden nutzt, damit andere Klassen mit den Teilen dieser internen Daten arbeiten können, die sie benötigen.

Kapselung

Abstraktion

Abstraktion nutzen Sie, wenn Sie ein Klassenmodell erstellen, das mit einer allgemeineren – oder abstrakteren – Klasse beginnt, von der dann spezifischere Klassen erben.

Polymorphie

Das Wort »Polymorphie« bedeutet »viele Formen«. Fällt Ihnen eine Gelegenheit ein, bei der ein Objekt in unserem Code viele Formen angenommen hat?

Polymorphie bedeutet, dass ein Objekt viele verschiedene Formen annehmen kann

Jedes Mal, wenn Sie eine Spottdrossel anstelle eines Tieres nutzen oder in einem Rezept, das nur irgendeinen Käse verlangt, einen Parmesan verwenden, nutzen Sie **Polymorphie**, beispielsweise jedes Mal beim Up- oder Downcasting. Es bedeutet, dass Sie ein Objekt in einer Methode oder Anweisung nutzen, die eigentlich etwas anderes erwartet.

Polymorphie nutzen Sie, wenn Sie eine Instanz einer Klasse in einer Anweisung oder Methode verwenden, die einen anderen Typ erwartet, beispielsweise eine Elternklasse oder eine Schnittstelle, die sie implementiert.

Achten Sie in der nächsten Übung auf Polymorphie!

Gleich werden Sie sich an eine richtig große Übung machen – die größte, an der Sie bislang gearbeitet haben – und in ihr werden Sie Polymorphie extensiv nutzen. Halten Sie also die Augen offen. Hier ist eine Liste mit vier typischen Arten, auf die Sie Polymorphie einsetzen werden. Zu jedem Punkt haben wir ein Beispiel gegeben (in der Übung werden Sie allerdings nicht genau diese Zeilen sehen). Sobald Sie ähnlichen Code in dem Code sehen, den Sie für die Übung schreiben, **haken Sie den Punkt auf der Liste ab**.

- ☐ Eine Referenzvariable eines bestimmten Typs nehmen und ihr ein Objekt eines anderen Typs zuweisen.

```
NektarWache berta = new NektarWache();
INektarSammler sammler = berta;
```

- ☐ Upcasting – in einer Anweisung oder Methode, die eine Instanz einer bestimmten Klasse erwartet, eine Instanz einer Unterklasse verwenden.

```
rex = new Hund();
zoowärter.TierFüttern(rex);
```

Erwartet TierFüttern() ein Tier-Objekt und erbt Hund von Tier, können Sie TierFüttern() eine Hund-Referenz übergeben.

- ☐ Eine Referenzvariable erstellen, deren Typ eine Schnittstelle ist, und ihr ein Objekt eines Typs zuweisen, der diese Schnittstelle implementiert.

```
IWache kasimir = new Wache();
```

Auch das ist Upcasting!

- ☐ Downcasting mit dem Schlüsselwort `as`.

```
void StockPflegen(IArbeiterin arbeiter) {
    if (arbeiter is Stockpflege) {
        Stockpflege pfleger = arbeiter as Stockpflege;
        ...
```

Die Methode `StockPflegen()` erwartet einen `IArbeiterin`-Parameter. Sie nutzt `as`, um die Arbeiterin einer `Stockpflege`-Referenz zuzuweisen.

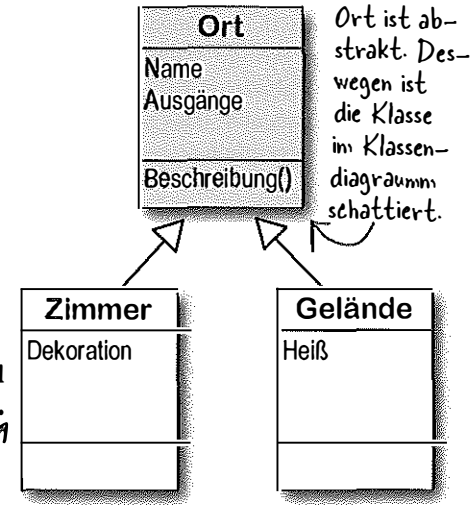


Lange Übung

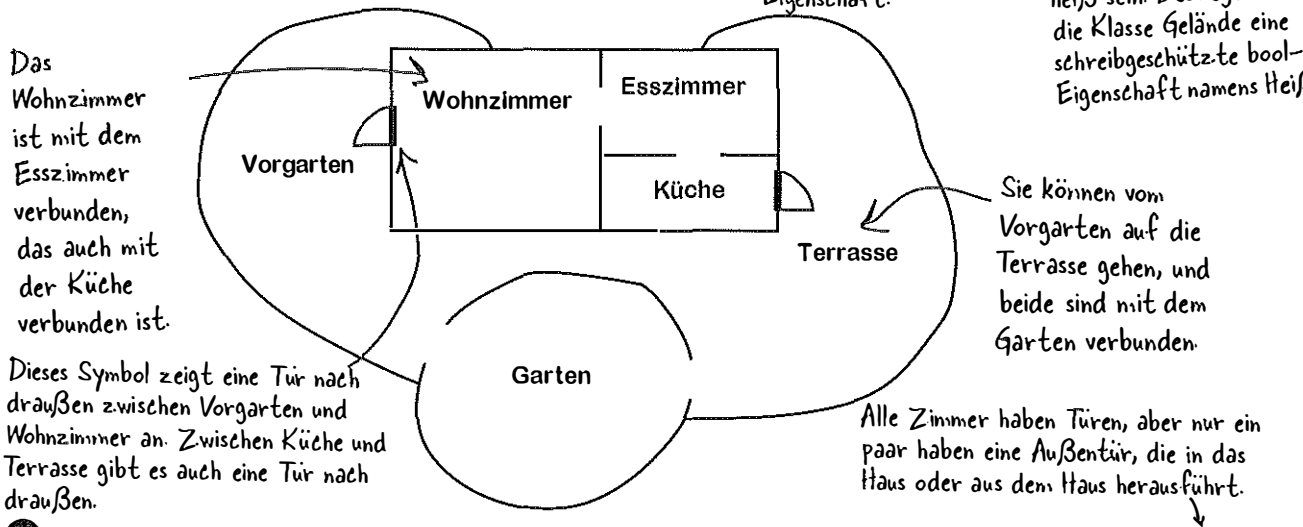
Bauen wir ein Haus! Erstellen Sie das Modell eines Hauses, in dem Ort und Zimmer über Klassen repräsentiert werden und jeder Ort, der eine Tür hat, über eine Schnittstelle.

1 **Beginnen Sie mit diesem Klassenmodell.**
Jedes Zimmer bzw. jeder Ort in Ihrem Haus wird durch ein eigenes Objekt repräsentiert. Die Innenräume erben alle von `Zimmer`, und Bereiche des Außengeländes erben von `Gelände`, einer Unterklasse der gleichen Basisklasse `Ort`. Diese hat zwei Felder: `Name` ist der Name des Orts (»Küche«) und `Ausgänge` ist ein Array mit Ort-Objekten, mit denen der aktuelle Ort verbunden ist. `wohnzimmer.Name` entspräche also `"Esszimmer"` und `wohnzimmer.Ausgänge` dem Array `{ wohnzimmer, küche }`.

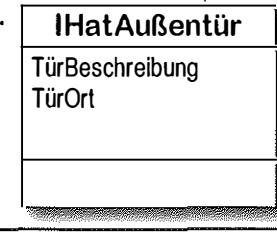
➔ Erzeugen Sie ein Windows-Form-Anwendung-Projekt und fügen Sie ihm die Klassen Ort, Zimmer und Gelände hinzu.



2 **Sie brauchen den Bauplan für das Haus.**
Dieses Haus hat drei Zimmer, einen Vorgarten und einen Garten. Es gibt zwei Türen: die Vordertür, die Wohnzimmer und Vorgarten verbindet, und die Hintertür, die Küche und Terrasse verbindet.



3 **Nutzen Sie die Schnittstelle IHatAußentür für Zimmer mit Außentür.**
Das Haus hat zwei Außentüren, die Vorder- und die Hintertür. Jeder Ort, der eine hat (Vorgarten, Terrasse, Wohnzimmer, Küche), sollte `IHatAußentür` implementieren. Die schreibgeschützte Eigenschaft `TürBeschreibung` enthält eine Beschreibung der Tür (die Vordertür ist eine »Eichentür mit Messinggriff«, die Hintertür eine »Glastür«). Die Eigenschaft `TürOrt` enthält eine Referenz auf den Ort, zu dem die Tür führt (`küche`).



4 Hier ist die Klasse Ort.

Als Ausgangspunkt können Sie die Klasse `Ort` verwenden:

```
public abstract class Ort {
    public Ort(string name) {
        this.name = name;
    }
    public Ort[] Ausgänge;
    private string name;
    public string Name {
        get { return name; }
    }
    public virtual string Beschreibung {
        get {
            string beschreibung = "Sie stehen im " + name
                + ". Es gibt Türen zu folgenden Orten: ";
            for (int i = 0; i < Ausgänge.Length; i++) {
                beschreibung += " " + Ausgänge[i].Name;
                if (i != Ausgänge.Length - 1)
                    beschreibung += ",";
            }
            beschreibung += ".";
            return beschreibung;
        }
    }
}
```

Der Konstruktor setzt das Feld name, das als Unterstützungsfeld für die schreibgeschützte Eigenschaft Name dient.

Beschreibung ist eine virtuelle Methode, die Sie überschreiben müssen

Das öffentliche Feld Ausgänge ist ein Array mit Ort-Referenzen, das alle anderen Orte nachhält, mit denen dieser Ort verbunden ist.

Die Klasse Zimmer überschreibt und erweitert Beschreibung, um die Dekoration hinzuzufügen, und Gelände fügt die Temperatur hinzu.

Die Eigenschaft Beschreibung liefert einen String mit einer Beschreibung des Zimmers, die den Namen und die verbundenen Orte einschließt (die aus dem Feld Ausgänge[] kommen). Unterklassen müssen diese überschreiben, um sie zu ändern.

Denken Sie daran, dass Ort eine abstrakte Klasse ist – Sie können von ihr erben oder Referenzen des Typs deklarieren, aber sie können sie nicht instantiieren.

5 Erstellen Sie die Klassen.

Erzeugen Sie die Klassen `Zimmer` und `Gelände` auf Basis des Klassenmodells. Erzeugen Sie dann zwei weitere Klassen: `AußenMitTür`, die von `Gelände` erbt und `IHatAußentür` implementiert, und `ZimmerMitTür`, die von `Zimmer` erbt und `IHatAußentür` implementiert.

Die folgenden Klassendeklarationen können Ihnen auf die Sprünge helfen:

Beginnen Sie jetzt mit den Klassen – auf der nächsten Seite geben wir Ihnen mehr Informationen zu ihnen.

```
public class AußenMitTür : Gelände, IHatAußentür
{
    // Hier kommt die Eigenschaft TürOrt hin.
    // Hier kommt die Eigenschaft TürBeschreibung hin.
}

public class ZimmerMitTür : Zimmer, IHatAußentür
{
    // Hier kommt die Eigenschaft TürOrt hin.
    // Hier kommt die Eigenschaft TürBeschreibung hin.
}
```

Das wird eine ziemlich umfangreiche Übung werden ... aber wir versprechen, dass sie Spaß machen wird! Und wenn wir damit durch sind, beherrschen Sie diesen Kram ganz sicher.

Wir sind noch nicht fertig - blättern Sie um!

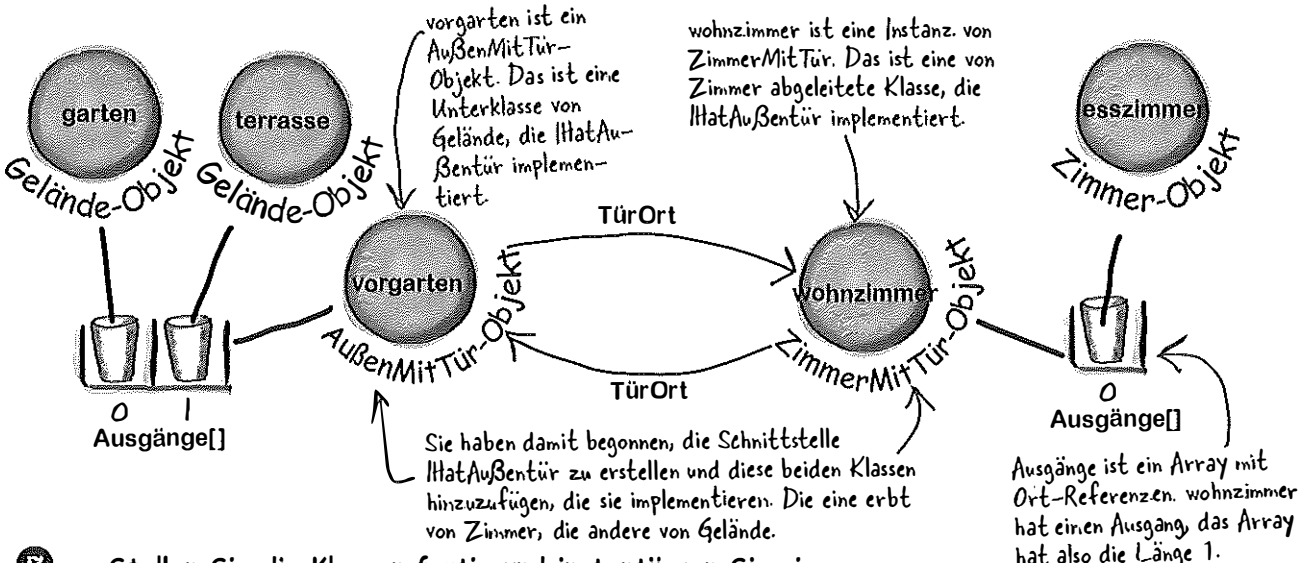


Lange Übung (Fortsetzung)

Jetzt haben Sie das Klassenmodell und können die Objekte für alle Teile des Hauses sowie ein Formular erstellen, um es zu untersuchen.

6 Wie Ihre Hausobjekte funktionieren.

Hier ist die Architektur für zwei unserer Objekte, `vorgarten` und `wohnzimmer`. Da beide eine Tür haben, müssen es Instanzen einer Klasse sein, die `IHatAußentür` implementiert. Die Eigenschaft `TürOrt` hält eine Referenz auf den Ort auf der anderen Seite der Tür.



7 Stellen Sie die Klassen fertig und instantiieren Sie sie.

Jetzt haben Sie alle Klassen, können sie fertigstellen und von ihnen Objekte instantiieren.

- ★ Sie müssen sicherstellen, dass der Konstruktor für die Klasse Gelände die schreibgeschützte Eigenschaft Heiß setzt und die Eigenschaft Beschreibung mit dem Text »Hier ist es heiß.« überschreibt, wenn Heiß true ist. Auf der Terrasse ist es heiß, in Garten und Vorgarten nicht.
- ★ Der Konstruktor für Zimmer muss die Dekoration setzen und sollte die Eigenschaft Beschreibung mit »Hier sehen Sie *(die Dekoration)*.« überschreiben. Im Wohnzimmer liegt ein Perserteppich, im Esszimmer hängt ein Kristallleuchter, und in der Küche gibt es Edelstahlgeräte und eine Glastür, die auf die Terrasse führt.
- ★ Ihr Formular muss diese Objekte erstellen und Referenzen darauf speichern. Fügen Sie ihm dazu eine Methode namens `ObjekteErstellen()` hinzu, die Sie aus dem Formularkonstruktor aufrufen.
- ★ Instantiieren Sie die Objekte für die fünf Orte im Haus. So geht das für das Wohnzimmer:

Das Formular hat ein Feld für jeden Ort.

```
ZimmerMitTür wohnzimmer = new ZimmerMitTür("Wohnzimmer",
      "einen Perserteppich" , "eine Eichentür mit Messinggriff");
```

Ausgänge ist ein Array mit Ort-Referenzen.

- ★ Die Methode `ObjekteErstellen()` muss die `Ausgänge[]`-Felder aller Objekte füllen:

```
vorgarten.Ausgänge = new Ort[] { terrasse, garten };
```

Diese Zeile erzeugt eins und fügt ihm zwei Referenzen hinzu.

Das sind geschweifte Klammern, alles andere führt zu einem Fehler.

8 Erstellen Sie ein Formular, um das Haus zu erforschen.

Erstellen Sie ein einfaches Formular, über das Sie das Haus erforschen können. Es hat ein mehrzeiliges Textfeld namens beschreibung, das die Beschreibung des aktuellen Zimmers anzeigt. Eine ComboBox namens ausgänge führt alle Ausgänge des aktuellen Zimmers auf. Es hat zwei Buttons: geheZu geht in den in der ComboBox ausgewählten Raum, und geheRaus ist nur sichtbar, wenn es eine Außentür gibt.



9 Jetzt müssen Sie das Formular noch funktionsfähig machen!

Nun haben Sie alle Teile und müssen diese nur noch zusammenfügen.

- ★ Ihr Formular braucht ein Feld namens aktuellerOrt, das den aktuellen Ort nachhält.
- ★ Fügen Sie dann eine ZuOrtGehen()-Methode hinzu, die als Parameter einen Ort hat. Als Erstes sollte diese Methode aktuellerOrt auf den neuen Ort setzen. Dann leert sie die ComboBox mit der Methode Items.Clear() und fügt ihr die Namen aller Orte im Array Ausgänge[] hinzu, indem sie die Items.Add()-Methode der ComboBox verwendet. Setzen Sie zum Schluss die ComboBox-Eigenschaft SelectedIndex auf null, damit das erste Element in der Liste angezeigt wird.
- ★ Setzen Sie das Textfeld so, dass es die Beschreibung des aktuellen Orts enthält.
- ★ Prüfen Sie mit dem Schlüsselwort **`is`**, ob der aktuelle Ort eine Tür hat. Hat er eine, machen Sie den »Durch die Tür gehen«-Button über seine Visible-Eigenschaft sichtbar, andernfalls unsichtbar.
- ★ Gehen Sie zum in der ComboBox gewählten Ort, wenn auf den Button »Hierhin gehen:« geklickt wird.
- ★ Gehen Sie an den Ort, mit dem der aktuelle Ort verbunden ist, wenn auf den Button »Durch die Tür gehen« geklickt wird.

Tipp: Wenn Sie in der ComboBox ein Element wählen, entspricht der Index des gewählten Elements dem Index des entsprechenden Elements im Array Ausgänge[].

Noch ein Tipp: Das aktuellerOrt-Feld des Formulars ist eine Ort-Referenz. Auch wenn sie auf ein Objekt zeigt, das IHatAußentür implementiert, können Sie nicht einfach »aktuellerOrt.TürOrt« verwenden, weil TürOrt kein Feld in Ort ist. Sie müssen downcasten, wenn Sie den Türort aus dem Objekt abfragen möchten.



Hier ist der Code für die Modellierung des Hauses. Für die Darstellung der Zimmer und Orte haben wir Klassen verwendet, für die Orte mit Tür eine Schnittstelle.

```
public interface IHatAußentür {
    string TürBeschreibung { get; }
    Ort TürOrt { get; set; }
}

public class Zimmer : Ort {
    private string dekoration;

    public Zimmer(string name, string dekoration)
        : base(name) {
        this.dekoration = dekoration;
    }

    public override string Beschreibung {
        get {
            return base.Beschreibung + " Sie sehen " + dekoration + ".";
        }
    }
}

public class ZimmerMitTür : Zimmer, IHatAußentür {
    public ZimmerMitTür(string name, string dekoration, string türBeschreibung)
        : base(name, dekoration)
    {
        this.türBeschreibung = türBeschreibung;
    }

    private string türBeschreibung;
    public string TürBeschreibung {
        get { return türBeschreibung; }
    }

    private Ort türOrt;
    public Ort TürOrt {
        get { return türOrt; }
        set { türOrt = value; }
    }
}
```

Hier ist die Schnittstelle IHatAußentür.

Die Klasse Zimmer erbt von Ort und fügt ein Unterstützungsfeld für die schreibgeschützte Eigenschaft Dekoration ein. Ihr Konstruktor setzt das Feld.

Die Klasse ZimmerMitTür erbt von Zimmer und implementiert IHatAußentür. Sie macht alles, was die Klasse Zimmer macht, fügt dem Konstruktor aber noch eine Beschreibung der Außentür hinzu. Außerdem enthält sie zusätzlich TürOrt, eine Referenz auf den Ort, zu dem die Tür führt. TürBeschreibung und TürOrt werden von IHatAußentür verlangt.

```
public class Gelände : Ort {
    private bool heiß;
    public bool Heiß { get { return heiß; } }

    public Gelände(string name, bool heiß)
        : base(name)
    {
        this.heiß = heiß;
    }

    public override string Beschreibung {
        get {
            string neueBeschreibung = base.Beschreibung;
            if (heiß)
                neueBeschreibung += " Es ist sehr heiß.";
            return neueBeschreibung;
        }
    }
}

public class AußenMitTür : Gelände, IHatAußentür {
    public AußenMitTür(string name, bool heiß, string türBeschreibung)
        : base(name, heiß)
    {
        this.türBeschreibung = türBeschreibung;
    }

    private string türBeschreibung;
    public string TürBeschreibung {
        get { return türBeschreibung; }
    }

    private Ort türOrt;
    public Ort TürOrt {
        get { return türOrt; }
        set { türOrt = value; }
    }

    public override string Beschreibung {
        get {
            return base.Beschreibung + " Sie sehen " + türBeschreibung + ".";
        }
    }
}
```

Gelände hat große Ähnlichkeit mit Zimmer – die Klasse erbt von Ort und fügt ein Unterstützungsfeld für die Eigenschaft Heiß hinzu, die in der Methode Beschreibung() verwendet wird, die sie überschreibt.

AußenMitTür erbt von Gelände, implementiert IHatAußentür und hat große Ähnlichkeit mit ZimmerMitTür.

Die Beschreibung-Eigenschaft der Basisklasse wird unabhängig davon gefüllt, ob ein Ort heiß ist oder nicht. Sie lässt sich die Beschreibung von der Elternklasse liefern und hängt daran die Beschreibung der Tür an.

Wir sind noch nicht fertig – blättern Sie um!



Lange Übung Lösung (Fortsetzung)

Hier ist der Code für das Formular. Das steckt alles in der Form1-Deklaration in Form1.cs.

```
public partial class Form1 : Form
{
    Ort aktuellerOrt;

    ZimmerMitTür wohnzimmer;
    Zimmer esszimmer;
    ZimmerMitTür küche;

    AußenMitTür vorgarten;
    AußenMitTür terrasse;
    Gelände garten;

    public Form1() {
       InitializeComponent();
       ObjekteErstellen();
       ZuOrtGehen(wohnzimmer);
    }

    private void ObjekteErstellen() {
       wohnzimmer = new ZimmerMitTür("Wohnzimmer", "einen Perserteppich",
              "eine Eichentür mit Messinggriff");
       esszimmer = new Zimmer("Esszimmer", "einen Kristallleuchter");
       küche = new ZimmerMitTür("Küche", "Edelstahlgeräte", "eine Glastür");

       vorgarten = new AußenMitTür("Vorgarten", false, "eine Eichentür mit Messinggriff");
       terrasse = new AußenMitTür("Terrasse", true, "eine Glastür");
       garten = new Gelände("Garten", false);

       esszimmer.Ausgänge = new Ort[] { wohnzimmer, küche };
       wohnzimmer.Ausgänge = new Ort[] { esszimmer };
       küche.Ausgänge = new Ort[] { esszimmer };
       vorgarten.Ausgänge = new Ort[] { terrasse, garten };
       terrasse.Ausgänge = new Ort[] { vorgarten, garten };
       garten.Ausgänge = new Ort[] { terrasse, vorgarten };

       wohnzimmer.TürOrt = vorgarten;
       vorgarten.TürOrt = wohnzimmer;

       küche.TürOrt = terrasse;
       terrasse.TürOrt = küche;
    }
```

So hält das Formular nach, welcher Raum gerade angezeigt wird.

Mit diesen Referenzvariablen hält das Formular die einzelnen Räume des Hauses nach.

Der Konstruktor des Formulars erzeugt die Objekte und ruft dann die Methode ZuOrtGehen() auf.

Bei der Erstellung der Objekte müssen zuerst die Klassen instantiiert und dabei die richtigen Parameter übergeben werden.

Hier übergeben wir dem AußenMitTür-Konstruktor die Türbeschreibung.

Hier werden die Ausgänge[]-Arrays für die einzelnen Instanzen gefüllt. Wir mussten damit warten, bis alle Instanzen erstellt wurden, da wir sonst nichts hätten, was wir in die Arrays hätten stecken können.

Bei den IHatAußentür-Objekten müssen wir den Türort setzen.

```
    private void ZuOrtGehen(Ort neuerOrt) {
        aktuellerOrt = neuerOrt;

        ausgänge.Items.Clear();
        for (int i = 0; i < aktuellerOrt.Ausgänge.Length; i++)
            ausgänge.Items.Add(aktuellerOrt.Ausgänge[i].Name);
        ausgänge.SelectedIndex = 0;

        beschreibung.Text = aktuellerOrt.Beschreibung;

        if (aktuellerOrt is IHatAußentür)
            durchTürGehen.Visible = true;
        else
            durchTürGehen.Visible = false;
    }

    private void hierhinGehen_Click(object sender, EventArgs e) {
        ZuOrtGehen(aktuellerOrt.Ausgänge[ausgänge.SelectedIndex]);
    }

    private void durchTürGehen_Click(object sender, EventArgs e) {
        IHatAußentür mitTür = aktuellerOrt as IHatAußentür;
        ZuOrtGehen(mitTür.TürOrt);
    }
}
```

Die Methode ZuOrtGehen() zeigt im Formular einen neuen Ort an.

Erst müssen wir die ComboBox leeren, dann können wir ihr die Ausgänge des neuen Orts hinzufügen. Abschließend setzen wir den ausgewählten Index (die markierte Zeile) auf null, damit das erste Element in der Liste angezeigt wird. Vergessen Sie nicht die DropDownStyle-Eigenschaft des DropDown auf >>DropDownList<< zu setzen – dann kann der Benutzer nichts in die ComboBox eintippen.

Das macht den >>Durch die Tür gehen<<-Button unsichtbar, wenn der aktuelle Ort IHatAußentür nicht implementiert.

Klickt der Benutzer auf den Button >>Hierin gehen:<<, wird zu dem Ort gewechselt, der in der ComboBox ausgewählt ist.

Wir mussten das Schlüsselwort as verwenden, um aktuellerOrt in IHatAußentür umzuwandeln, damit wir Zugriff auf das Feld TürOrt erhalten.

Aber wir sind noch nicht fertig!

Ein Hausmodell zu erstellen ist ja ganz nett, aber wäre es nicht noch besser, wenn wir daraus ein Spiel machen könnten? Na los! Sie spielen Verstecken mit dem Computer. Wir müssen eine `Gegner`-Klasse erstellen, die jemanden repräsentiert, der sich in einem der Zimmer versteckt hat. Und wir müssen das Haus viel größer machen. Und natürlich brauchen wir noch Orte, an denen man sich verbergen kann! Wir ergänzen die Schnittstelle so, dass einige Zimmer Verstecke enthalten können. Schließlich müssen wir das Formular aktualisieren, damit Sie in den Verstecken suchen können und festgehalten wird, wie viele Züge Sie benötigten, um Ihren Gegner zu finden. Klingt interessant? Klar!

Fangen wir an!



Zeit, sich zu verbergen! Erweitern Sie Ihr ursprüngliches Haus-Programm, indem Sie ihm weitere Räume, Verstecke und einen Gegner hinzufügen, der sich vor Ihnen versteckt.

Erstellen Sie ein neues Projekt und nutzen Sie die Funktion »Vorhandenes Element hinzufügen«, um die Klassen aus dem ersten Teil der Übung darin einzufügen.

1 Fügen Sie die Schnittstelle IVersteck hinzu.

Hier brauchen wir nichts Ausgefallenes. Jede Ort-Unterklasse, die `IVersteck` implementiert, bietet dem Gegner einen Ort, an dem er sich verbergen kann. Sie braucht nur ein String-Feld, um den Namen des Verstecks zu speichern (»im Wandschrank«, »unter dem Bett« usw.).

- Geben Sie ihm einen Getter, aber keinen Setter – wir werden es im Konstruktor setzen, da wir das Versteck nie ändern müssen, nachdem es einem Raum einmal gegeben wurde.

2 Fügen Sie Klassen hinzu, die IVersteck implementieren.

Sie benötigen zwei weitere Klassen: `GeländeMitVersteck` (die von `Gelände` erbt) und `Zimmer-MitVersteck` (die von Zimmer erbt). Lassen wir außerdem alle Räume, die eine Tür haben, auch ein Versteck haben. Sie erben also von `ZimmerMitVersteck` statt von `Zimmer`.

Jeder Raum mit einer Außentür hat also auch ein Versteck.

3 Fügen Sie eine Klasse für Ihren Gegner hinzu.

Das Gegner-Objekt sucht sich zufällig ein Versteck im Haus, und Sie sollen ihn finden.

- Er braucht ein privates Ort-Feld (`ort`), damit er festhalten kann, wo er sich befindet, und ein privates Random-Feld (`zufall`), das er verwendet, um zufällig ein Versteck auszuwählen.
- Der Konstruktor erwartet einen Ausgangspunkt und setzt `ort` darauf sowie `zufall` auf eine neue `Random`-Instanz. Er beginnt im Vorgarten (das wird vom Formular übergeben) und geht zufällig von Versteck zu Versteck. Zu Spielbeginn macht er zehn Züge. Trifft er auf eine Außentür, wirft er eine Münze, um zu entscheiden, ob er durch diese geht oder nicht.
- Fügen Sie eine `Bewegen()`-Methode hinzu, die Ihren Gegner von seinem aktuellen Ort zu einem neuen gehen lässt. Ist er in einem Raum mit einer Tür, wirft er eine Münze, um zu entscheiden, ob er durch diese geht. Ist `zufall.Next(2)` gleich 1, geht er durch die Tür. Dann wählt er zufällig einen der Ausgänge seines aktuellen Orts und benutzt ihn. Bietet dieser Ort kein Versteck, wird dieser Vorgang wiederholt – er wählt zufällig Ausgänge, bis er ein Versteck gefunden hat.
- Fügen Sie eine `Prüfen()`-Methode hinzu, die als Parameter einen Ort erwartet und true liefert, wenn dieser dem Versteck entspricht, andernfalls false.

6 Fügen Sie dem Haus weitere Zimmer hinzu.

Erweitern Sie die Methode `ObjekteErstellen()`, um dem Haus weitere Räume hinzuzufügen:

- Fügen Sie eine **Treppe** mit einem Handlauf aus Holz hinzu, die das Wohnzimmer mit dem **Korridor im 1. Stock** verbindet, der ein Bild mit einem Hund und einen Wandschrank zum Verstecken enthält.
- Der obere Korridor verbindet drei Räume: ein **Schlafzimmer** mit einem großen Bett, ein **Kinderzimmer** mit einem kleinen Bett und ein **Bad** mit einem Waschbecken und einer Dusche. Unter dem Bett und in der Dusche könnte man sich verstecken.
- Vorgarten und Terrasse sind mit der **Auffahrt** verbunden, in der man sich in der Garage verstecken könnte. Die Laube im **Garten** bietet ein weiteres Versteck.

5 Aktualisieren Sie dann das Formular.

Sie müssen dem Formular ein paar Buttons hinzufügen. Das werden wir etwas ausgefeilter machen, indem wir diese in Abhängigkeit vom Spielzustand sichtbar oder unsichtbar machen.

Den oberen Button und die ComboBox nutzen Sie genau so wie zuvor. Sie sind allerdings nur sichtbar, wenn das Spiel läuft.

Beginnt das Spiel, wird nur der Verstecken-Button angezeigt. Klicken Sie darauf, zählt das Formular im Textfeld bis zehn und ruft zehnmal die Bewegen()-Methode des Gegners auf. Dann macht es den Button unsichtbar.



Der mittlere Button heißt suchen.

Das ist der Button, über den Sie das Versteck in einem Ort überprüfen. Er ist nur sichtbar, wenn Sie sich in einem Raum mit Versteck befinden. Wird er angezeigt, wird die Text-Eigenschaft von »button3« in »[Name des Verstecks] suchen« geändert – bei einem Raum mit einem Versteck unter dem Bett steht darauf also »unter dem Bett suchen«.

6 Machen Sie die Buttons funktionsfähig.

Dem Formular müssen zwei neue Buttons hinzugefügt werden.

Gehen Sie zu Kapitel 2 zurück, wenn Sie eine Auffrischung zu DoEvents() und Sleep() brauchen.

- ★ Die mittlere Button prüft das Versteck im aktuellen Zimmer und ist nur in Räumen mit Versteck sichtbar. Dazu nutzt er die `Prüfen()`-Methode des Gegners. Haben Sie den Gegner gefunden, wird das Spiel zurückgesetzt.
- ★ Über den unteren Button starten Sie das Spiel. Er zählt bis zehn, indem er im Textfeld »1...« anzeigt, 200 Millisckunden wartet, »2...« anzeigt, wieder wartet und so weiter. Nach jeder Zahl ruft er die `Bewegen()`-Methode des Gegners auf. Dann zeigt er eine halbe Sekunde »Ich komme!« an, und das Spiel beginnt.

7 Fügen Sie Methoden zum Neuzeichnen des Formulars und Rücksetzen des Spiels hinzu.

Fügen Sie eine `FormularNeuZeichen()`-Methode hinzu, die im Textfeld beschreibung den erforderlichen Text anzeigt, die Buttons sichtbar und unsichtbar macht und auf dem mittleren Button den richtigen Text anzeigt. Ergänzen Sie dann eine `SpielZurücksetzen()`-Methode, die ausgeführt wird, wenn Sie den Gegner finden. Sie setzt das Gegner-Objekt zurück, damit es wieder im Vorgarten beginnt und sich neu versteckt, wenn der Benutzer auf den Button »Verstecken!« klickt. Im Formular sollte nur noch das Textfeld und der Button »Verstecken!« sichtbar sein. Das Textfeld sollte anzeigen, wo Sie den Gegner fanden und wie viele Züge Sie benötigten.

8 Halten Sie fest, wie viele Versuche der Spieler macht.

Achten Sie darauf, dass das Textfeld anzeigt, wie oft der Spieler Verstecke geprüft oder den Raum gewechselt hat. Wird der Gegner gefunden, sollte ein Fenster mit der Meldung »Sie benötigten X Züge!« angezeigt werden.



9 Das Programm sollte gleich beim Start richtig aussehen.

Wird das Programm gestartet, sollten Sie nur ein leeres Textfeld und den »Verstecken!«-Button sehen. Ein Klick auf den Button startet das Spiel.



Lösungen zu den Übungen (Fortsetzung)

Hier ist der restliche Code für das Formular. Die Event-Handler für die Buttons hierhinGehen und durchTürGehen sind mit denen aus dem ersten Teil der Übung identisch. Blättern Sie also ein paar Seiten zurück, um sie zu sehen.

```
private void SpielZurücksetzen(bool nachrichtAnzeigen) {
    if (nachrichtAnzeigen) {
        MessageBox.Show("Sie benötigten " + Züge + " Züge!");
        IVersteck fundort = aktuellerOrt as IVersteck;
        beschreibung.Text = "Sie haben Ihren Gegner in " + Züge
              + " Zügen gefunden! Er verbarg sich " + fundort.VersteckName + ".";
    }
    Züge = 0;
    verstecken.Visible = true;
    hierhinGehen.Visible = false;
    suchen.Visible = false;
    durchTürGehen.Visible = false;
    ausgänge.Visible = false;
}

private void suchen_Click(object sender, EventArgs e) {
    Züge++;
    if (gegner.Prüfen(aktuellerOrt))
        SpielZurücksetzen(true);
    else
        FormularNeuZeichen();
}

private void verstecken_Click(object sender, EventArgs e) {
    verstecken.Visible = false;

    for (int i = 1; i <= 10; i++) {
        gegner.Bewegen();
        beschreibung.Text = i + "... ";
        Application.DoEvents();
        System.Threading.Thread.Sleep(200);
    }

    beschreibung.Text = "Ich komme!";
    Application.DoEvents();
    System.Threading.Thread.Sleep(500);

    hierhinGehen.Visible = true;
    ausgänge.Visible = true;
    ZuOrtGehen(wohnzimmer);
}
```

Die Methode SpielZurücksetzen() setzt das Spiel zurück. Sie zeigt die abschließende Meldung an und verbirgt dann alle Buttons außer dem >>Verstecken!<<-Button.

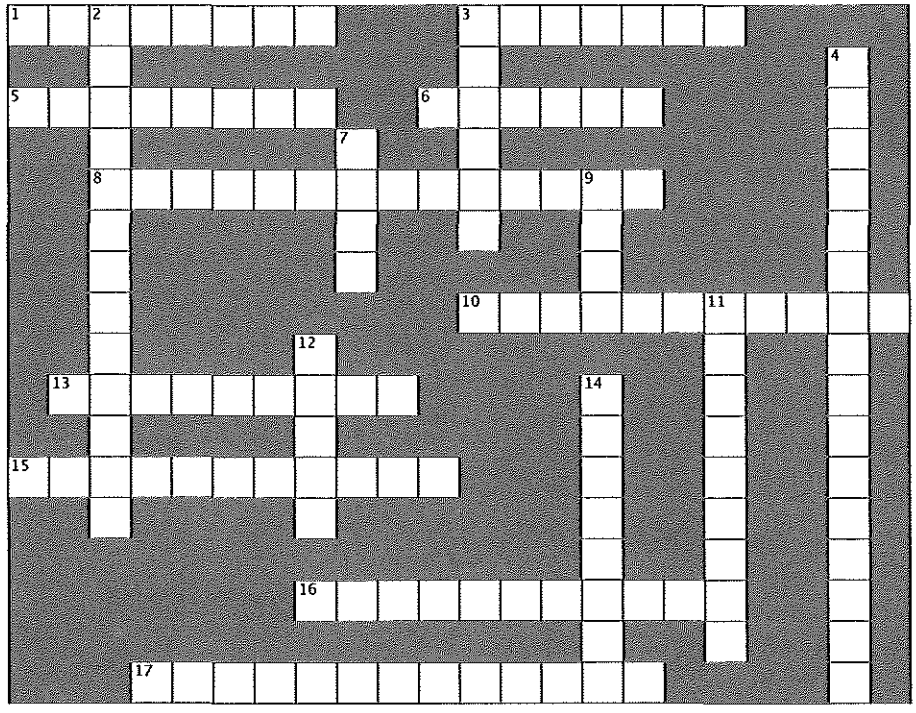
Wir möchten den Namen des Verstecks anzeigen, aber aktuellerOrt ist eine Ort-Referenz, die uns keinen Zugriff auf das Feld VersteckName bietet. Glücklicherweise können wir das Schlüsselwort as verwenden, um es auf eine IVersteck-Referenz downzucasten, die auf das gleiche Objekt zeigt.

Mit einem Klick auf den Button suchen wird geprüft, ob sich der Gegner im aktuellen Raum befindet. Ist das der Fall, wird das Spiel zurückgesetzt. Wenn nicht, wird das Formular neu gezeichnet (um die Anzahl an Zügen zu aktualisieren).

Über den Button verstecken wird das Spiel gestartet. Als Erstes macht er sich selbst unsichtbar. Dann zählt er bis 10 und sagt dem Gegner, dass er sich bewegen soll. Schließlich macht er den ersten Button und die ComboBox sichtbar und lässt den Spieler im Wohnzimmer mit der Suche beginnen. Die Methode ZuOrtGehen() ruft FormularNeuZeichen() auf.



Objekt-Kreuzworträtsel



Waagerecht

1. Alle Methoden in einer Schnittstelle sind automatisch _____.

3. Ein Zugriffsmodifizierer, der für nichts innerhalb einer Schnittstelle zulässig ist.

5. C# erlaubt keine _____-Vererbung.

6. Das Schlüsselwort is liefert true, wenn ein _____ die angegebene Schnittstelle implementiert.

8. Das können Sie mit einer Schnittstelle machen.

10. Verschieben Sie gemeinsame Methoden aus spezifischeren Klassen in eine allgemeinere Klasse, von der die anderen erben, nutzen Sie dieses OOP-Prinzip.

13. Eins der vier OOP-Prinzipien, das Sie mit dem Doppelpunkt-Operator implementieren.

15. Übergeben Sie eine Unterklasse an eine Methode, die eigentlich die Basisklasse erwartet, nutzen Sie dieses OOP-Prinzip.

16. Implementiert eine Klasse, die eine Schnittstelle implementiert, nicht alle Methoden, Getter und Setter, die darin definiert werden, lässt sich das Projekt nicht _____.

17. Eine abstrakte Klasse können Sie nicht _____.

Senkrecht

2. Eine Klasse, die sie implementiert, muss alle Methoden, Getter und Setter einschließen, die sie definiert.

3. Alles in einer Schnittstelle ist automatisch _____.

4. Eine Programmierung, die darauf basiert, dass Verhalten und Zustände zu Klassen und Objekten zusammengefasst werden, nennt man _____.

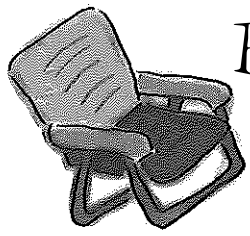
7. Eigentlich können Schnittstellen kein _____ einschließen, aber sie können Getter und Setter definieren, die von außen wie eins aussehen.

9. Implementiert Ihre Klasse eine Schnittstelle, die von einer anderen Schnittstelle _____, muss sie auch alle Member dieser zweiten Schnittstelle implementieren.

11. Das OOP-Prinzip, bei dem es darum geht, private Daten zu verbergen und nur Methoden und Felder zu veröffentlichen, auf die andere Klassen zugreifen müssen.

12. Das, was einer abstrakten Methode fehlt.

14. Eine abstrakte Klasse kann abstrakte und _____ Methoden einschließen.



Pool-Puzzle, Lösung zu Seite 287

Sie haben die Aufgabe, die leeren Zeilen im Code mit den Codeschnipseln aus dem Pool zu füllen. Einzeine Schnipsel dürfen mehrfach verwendet werden, und Sie werden nicht alle Schnipsel benötigen. Das Ziel ist es, eine Klasse zu erstellen, die sich kompilieren lässt und die gezeigte Ausgabe erzeugt.

```
public interface Nase {
    int Ohr() ;
    string Gesicht { get; }
}

public abstract class Picasso : Nase {
    public virtual int Ohr()
    {
        return 7;
    }
    public Picasso(string gesicht)
    {
        this.gesicht = gesicht;
    }
    public virtual string Gesicht {
        get { return gesicht ; }
    }
    string gesicht;
}

public class Clowns : Picasso {
    public Clowns() : base("Clowns") { }
}
```

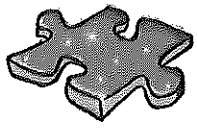
Felddeklarationen können in der Klasse an beliebiger Stelle stehen! Der Code ist aber leichter lesbar, wenn sie am Anfang stehen. Es ist jedoch trotzdem zulässig, das Feld gesicht am Ende der Klasse Picasso zu deklarieren.

Hier ruft die Klasse Akte den Konstruktor von Picasso auf, da das die Klasse ist, von der sie erbt. Sie übergibt ihm >>Akte<<, und das wird im Feld gesicht gespeichert.

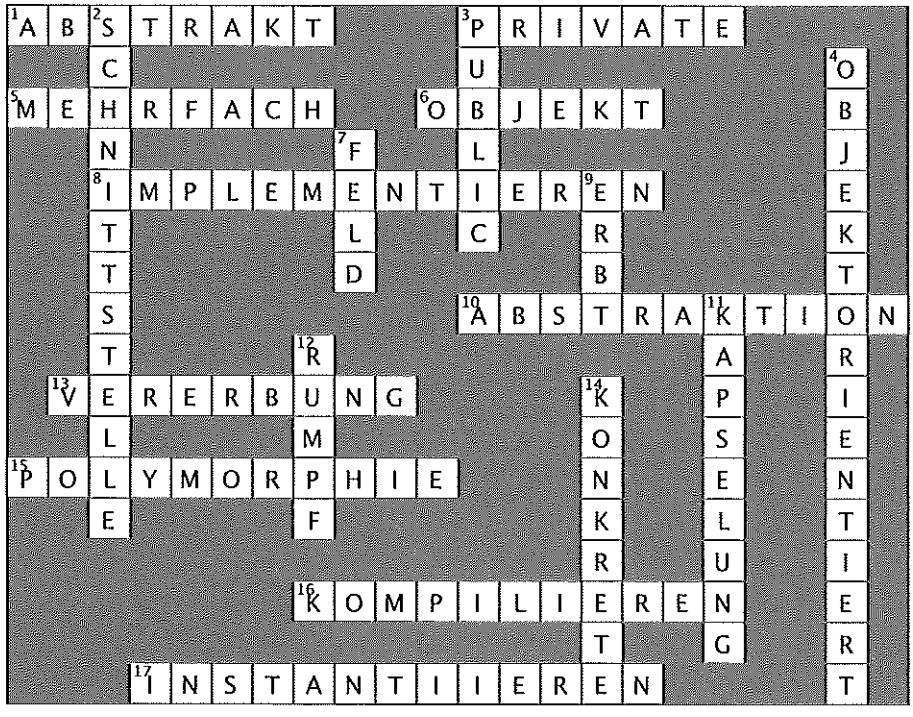
```
public class Akte : Picasso {
    public Akte() : base("Akte") { }
    public override int Ohr() {
        return 5;
    }
}

public class Von76 : Clowns {
    public override string Gesicht {
        get { return "Von76"; }
    }
    public static void Main(string[] args) {
        string ergebnis = "";
        Nase[] i = new Nase[3];
        i[0] = new Akte();
        i[1] = new Clowns();
        i[2] = new Von76();
        for (int x = 0; x < 3; x++) {
            ergebnis += ( i[x].Ohr() + " "
                + i[x].Gesicht ) +
"\n";
        }
        MessageBox.Show(ergebnis);
    }
}
```

`Gesicht` ist ein Getter, der den Wert des Felds `gesicht` liefert. Beide werden in Picasso definiert, aber auf das Feld können Unterklassen nur über den öffentlichen Getter zugreifen.



Objekt-Kreuzworträtsel, Lösung



8 Enums und Auflistungen

Daten in Massen speichern



Eigentlich tritt alles immer in Massen auf.

Im wahren Leben wird man mit Daten nie in kleinen Fragmenten konfrontiert. Nein, sie begegnen Ihnen immer in **Massen, Bergen und Haufen**. Und Sie brauchen ziemlich leistungsfähige Werkzeuge, um sie zu organisieren. Genau das sind **Auflistungen**. Mit diesen können Sie alle Daten **speichern, sortieren und verwalten**, die Ihr Programm durchforsten muss. Sie können sich darauf konzentrieren, wie Ihr Programm mit den Daten arbeitet, und es den Auflistungen überlassen, sich für Sie um die Daten selbst zu kümmern.

Gelegentlich sind Strings nicht gut genug, wenn es darum geht, Kategorien von Daten zu speichern

Angenommen, Sie haben mehrere Arbeitsbienen, die alle durch `Arbeiterin` repräsentiert werden. Wie würden Sie dann einen Konstruktor schreiben, der Ihre Aufgabe als Parameter übernimmt? Wenn Sie für die Aufgabenbeschreibung einen String verwenden, könnte das zu Code wie diesem führen:

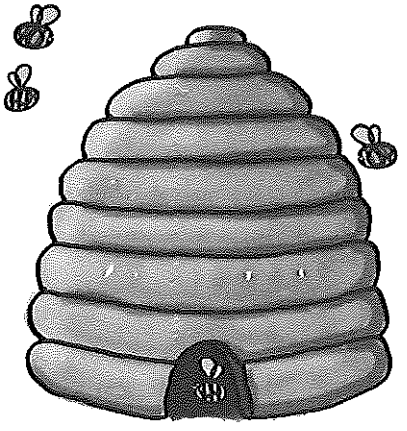
Unsere Bienenstockverwaltung hält die Aufgaben der einzelnen Arbeiterinnen mit einem String wie »Wachdienst« oder »Nektarsammler« fest.

Bei unserem Code wäre es möglich, dem Konstruktor diese Werte zu übergeben, obwohl das Programm nur Wachdienst, Nektarsammler und ähnliche Bienenaufgaben unterstützt.

```
Arbeiterin willi = new Arbeiterin("Staatanwalt");
Arbeiterin lilli = new Arbeiterin("Hundesitter");
Arbeiterin livia = new Arbeiterin("TV-Moderator");
```

Dieser Code lässt sich problemlos kompilieren. Aber für eine Biene sind diese Aufgaben wenig sinnvoll. Die Klasse Arbeiterin sollte Werte dieser Art nicht als Daten zulassen.

Sicher, Sie könnten dem `Arbeiterin`-Konstruktor Code hinzufügen, der jeden String darauf prüft, ob er eine zulässige Bienenaufgabe beschreibt. Fügen Sie neue Aufgaben hinzu, die Bienen erledigen können, müssen Sie diesen Code allerdings ändern und die Klasse Arbeiterin neu kompilieren. Und diese Lösung ist auch aus weiteren Gründen recht kurzsichtig. Was ist, wenn Sie andere Klassen haben, die mit den Aufgaben arbeiten müssen, die Arbeitsbienen erledigen können? Doppelter Code lässt sich dann nicht mehr vermeiden ... und wohin dieser Weg führt, wissen Sie bereits.



Wir benötigen also ein Mittel, das uns zu sagen erlaubt: »Hey, hier sind nur ganz bestimmte Werte erlaubt.« Wir müssen die Werte **aufzählen** können, die an bestimmten Stellen verwendet werden dürfen.

Mit Enumerationen können Sie erlaubte Werte aufzählen

Eine Enumeration oder kurz ein Enum ist ein Datentyp, der für bestimmte Datenteile nur bestimmte Werte erlaubt. Wir könnten beispielsweise eine Enumeration namens `Aufgaben` definieren und darin die erlaubten Aufgaben angeben:

Das Zeug in den geschweiften Klammern wird als die Aufzählungsliste bezeichnet, die einzelnen Begriffe als Aufzählungselemente und das Ganze als Aufzählung.

Aber die meisten Leute sagen einfach Enum.

Das ist der Name des Enums.

Auf das letzte Element der Aufzählungsliste muss kein Komma folgen. Aber wird eins angegeben, erleichtert das das Umordnen per Cut-and-Paste.

```
public enum Aufgabe {
    NektarSammler,
    Wachdienst,
    StockPflege,
    Bienenschule,
    Brutpflege,
    HonigProduktion,
}
```

All das sind gültige Aufgaben. Jeder Eintrag kann als Aufgabenwert verwendet werden.

Trennen Sie die einzelnen Werte mit Kommata und beenden Sie die Aufzählung mit einer geschweiften Klammer.

Diese Elemente können Sie jetzt so mit Typen referenzieren:

Das ist der Name des Enums.

Schließlich der Wert aus dem Enum, den Sie verwenden möchten.

```
Arbeiterin nanny = new Arbeiterin(Aufgabe.Brutpflege);
```

Wir haben den Arbeiterin-Konstruktor so geändert, dass er Aufgaben als Parametertyp erwartet.

Andere Werte sind nicht erlaubt. Sie können sich nicht einfach einen neuen Wert für das Emm ausdenken. Tun Sie das, lässt sich das Programm nicht mehr kompilieren.

```
private void button1_Click(object sender EventArgs e)
{
      Arbeiterin willi = new Arbeiterin(Aufgaben.Staatsanwalt);
}
```

Hier ist der Fehler, den der Compiler meldet.

```
'Aufgaben' enthält keine Definition für
'Staatsanwalt'.
```

Mit Enums können Zahlen über Namen repräsentiert werden

Manchmal kann man mit Zahlen leichter arbeiten, wenn man Namen für sie hat. In einem Enum können Sie den Werten Zahlen zuweisen und die Namen dann verwenden, um auf die Zahlen zu verweisen. So können Sie vermeiden, dass in Ihrem Code ein Haufen unerklärlicher Zahlen herumgeistert. Hier ist ein Enum, das die Punktwerte für die Kunststücke bei einem Hundedressur-Wettbewerb festhält. Es befindet sich **innerhalb der Klasse Hundedressur**. Möchten Sie es außerhalb dieser Klasse verwenden, müssen Sie es also über `Hundedressur.Bewertungen` aufrufen.

Das ist das Enum Hundedressur.Bewertungen.

```
class Hundedressur{
    public enum Bewertungen {
        Sitz = 7,
        Männchen = 25,
        Rolle = 50,
        Apportieren = 10,
        BeiFuß = 5,
        Aus = 20,
    }

...
    // Weiter unten in der Klasse ...
    int wert = (int)Bewertungen.Apportieren * 2;
    MessageBox.Show(wert.ToString());
    Hundedressur wertung = (Hundedressur)wert;
    MessageBox.Show(wertung.ToString());
```

Die Werte müssen nicht in einer bestimmten Reihenfolge stehen, und einem Wert können mehrere Namen gegeben werden.

Geben Sie nach einem Namen >>=<< und dann die Zahl an, die dieser Name repräsentieren soll.

Der (int-)Cast sagt dem Compiler, dass er das Aufzählungselement in die Zahl umwandeln soll, die es repräsentiert. Da Bewertungen.Aus den Wert 20 hat, liefert (int)Bewertungen.Aus also den int-Wert 20.

Das liefert >>20<<, 10 * 2.

ToString() liefert den Namen ... hier >>Aus<<.

Sie können das Enum in eine Zahl umwandeln und damit Berechnungen durchführen, oder Sie können die Methode `ToString()` verwenden, um den Namen als String zu behandeln. Weisen Sie einem Namen keine Zahl zu, werden die Elemente in der Liste automatisch nummeriert. Das erste Element erhält den Wert 0, das zweite 1 usw.

Aber was ist, wenn Sie bei einem Aufzählungselement eine richtig große Zahl verwenden möchten? Standardmäßig haben die Zahlen in einem Enum den Typ `int`. Aber der verwendete Typ kann über den `:`-Operator angegeben werden:

Ein Enum kann wie hier in eine Klasse gesteckt oder aber eigenständig gespeichert werden.

```
public enum Bewertungen : long {
        Sitz = 7,
        Männchen = 2500000000025
}
```

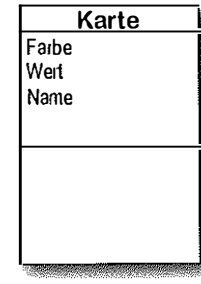
Das sagt dem Compiler, dass die Werte im Enum Bewertungen long-Werte sind und keine int-Werte.

Versuchen Sie, diesen Code zu kompilieren, ohne dabei long als Typ anzugeben, erhalten Sie folgende Fehlermeldung:

```
Der Typ 'long' kann nicht implizit in 'int'
konvertiert werden.
```



Setzen Sie Ihre neuen Enum-Kenntnisse ein, um eine Klasse zu erstellen, die eine Spielkarte festhält.



1 Erstellen Sie ein Projekt und fügen Sie ihm die Klasse Karte hinzu.
Sie benötigen zwei öffentliche Felder: Farbe (Karo, Herz, Pik oder Kreuz) und Wert (Ass, Zwei, Drei ... Zehn, Bauer, Dame, König). Und Sie brauchen eine schreibgeschützte Eigenschaft für den Namen (»Pik-Ass«, »Herz-Dame«).

2 Nehmen Sie zwei Enums, um die Farben und Werte zu definieren.
Sorgen Sie dafür, dass `(int)Karte.Farben.Karo` gleich 0, `Herz` gleich 1, `Pik` gleich 2 und `Kreuz` gleich 3 ist. Verwenden Sie für Werte diese Kartenwerte: `(int)Karte.Werte.Ass` sollte 1 sein, `Zwei` 2, `Drei` 3 usw. `Bauer` sollte gleich 11, `Dame` gleich 12 und `König` gleich 13 sein.

3 Fügen Sie eine Eigenschaft für den Namen der Karte hinzu.
`Name` sollte eine schreibgeschützte Eigenschaft sein. Der Getter sollte einen String liefern, der die Karte beschreibt. Dieser Code läuft in einem Formular, das die `Name`-Eigenschaft der Karte abruft und anzeigt:

```
Karte karte = new Karte(Karte.Farben.Karo, Karte.Werte.Ass);
string kartenname = karte.Name;
```

Damit das funktioniert, muss die Klasse Karte einen Konstruktor haben, der zwei Werte annimmt.

Der Wert von Kartenname sollte »`Karo-Ass`« sein.

4 Lassen Sie von einem Button den Namen einer zufälligen Karte anzeigen.
Sie können Ihr Programm eine Karte mit einem zufälligen Wert und einer zufälligen Farbe erstellen lassen, indem Sie eine Zufallszahl zwischen 0 und 3 auf `Karte.Farben` und eine andere Zufallszahl zwischen 1 und 13 auf `Karte.Werte` casten. Dazu können Sie eine Funktionalität der eingebauten Klasse `Random` nutzen, die Ihnen drei Möglichkeiten bietet, `Next()` aufzurufen:

Mehrere Möglichkeiten, eine Methode aufzurufen, bezeichnet man als Überladungen. Dazu später mehr ...

```
Random zufall = new Random();
int zahlZwischen0und3 = zufall.Next(4);
int zahlZwischen1und13 = zufall.Next(1, 14);
int beliebigeZufallszahl = zufall.Next();
```

Das fordert Random auf, einen Wert größer gleich 1 und kleiner als 14 zu liefern.



Es gibt keine Dummen Fragen

F: Moment. Als Ich den Code eingab, öffnete sich ein Inteiii-Sense-Fenster, das irgendetwas von »3 von 3« laberte, als ich die Random.Next()-Methode verwendete. Was soll das?

A: Sie haben gesehen, dass die Methode **überladen** ist. Hat eine Klasse eine Methode, die Sie auf mehrere Weisen aufrufen können, bezeichnet man das als Überladen. Verwenden Sie eine Klasse mit einer überladenen Methode, zeigt Ihnen die IDE alle zur Verfügung stehenden Optionen. in diesem Fall sind das die drei Next()-Methoden, die Random bietet. Sobald Sie »zufaii.Next(« eingeben, öffnet die IDE ein IntelliSense-Fenster, das die Parameter für die verschiedenen Überladungen anzeigt. Über die Hoch- und Runter-Pfeile neben »3 von 3« können Sie zwischen diesen wechseln. Das ist nützlich, wenn Sie es mit Methoden zu tun haben, die Dutzende Überladungen bieten. Achten Sie also darauf, dass Sie die richtige Überladung der Next()-Methode auswählen! Aber zerbrechen Sie sich noch nicht den Kopf darüber – später in diesem Kapitel werden wir ausführlich über das Überladen sprechen.



Listen sind flexibler als Arrays

Über die in das .NET Framework eingebaute Klasse List können Sie mit Objekten eine ganze Menge Dinge anstellen, die mit einem braven Array nicht möglich sind. Schauen Sie sich einige der Dinge an, die mit einer List möglich sind.

1 **Sie können eine Liste erstellen.**

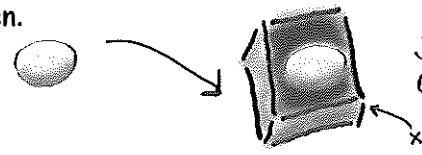
```
List<Ei> eierkarton = new List<Ei>();
```



Auf dem Heap wird ein neues List-Objekt erstellt, das zu Anfang leer ist.

2 **Sie können ihr etwas hinzufügen.**

```
Ei x = new Ei();
eierkarton.Add(x);
```

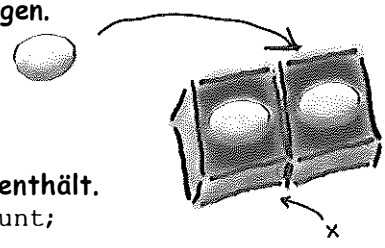


Jetzt vergrößert sich das List-Objekt, um ein Ei aufzunehmen ....

3 **Sie können ihr noch etwas hinzufügen.**

```
Ei y = new Ei();

eierkarton.Add(y);
```



... und vergrößert sich weiter, um ein zweites Ei-Objekt aufzunehmen.

4 **Sie können ermitteln, wie viel sie enthält.**

```
int größe = eierkarton.Count;
```

5 **Sie können prüfen, ob sie etwas Bestimmtes enthält.**

```
bool stecktDrin = eierkarton.Contains(x);
```

Jetzt können Sie darin nach einem der Eier suchen. Hier erhalten Sie auf alle Fälle true.

6 **Sie können herausfinden, wo etwas steckt.**

```
int idx = eierkarton.IndexOf(y);
```

Der Index für x wäre 0, der für y 1.

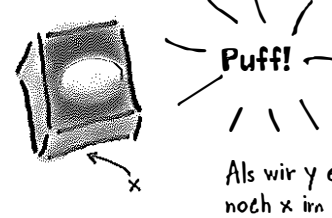
7 **Sie können abfragen, wie viel sie speichern kann.**

```
int limit = eierkarton.Capacity;
```

Das sagt Ihnen, wie viele Objekte das List-Objekt speichern kann, bevor es sich vergrößern muss.

8 **Sie können etwas herausnehmen.**

```
eierkarton.Remove(y);
```



Als wir y entfernten, blieb nur noch x im List-Objekt zurück, also schrumpfte es!



Füllen Sie den Rest der Tabelle aus, indem Sie sich den List-Code auf der linken Seite ansehen und auf der rechten Seite eintragen, wie der Code aussehen müsste, würden wir stattdessen ein gewöhnliches Array verwenden. Wir erwarten nicht, dass Sie das alles gleich richtig machen ... versuchen Sie sich einfach mal daran.

Gehen Sie davon aus, dass diese Anweisungen nacheinander ausgeführt werden.

Ein paar haben wir für Sie gemacht ...

List	Array
`List<String> dieListe = new List <String>();`	String [] dieListe = new String[2];
`String a = "Ja!";`	String a = "Ja!";
`dieListe.Add(a);`	
`String b = "Brummer";`	String b = "Brummer";
`dieListe.Add(b);`	
`int größe = dieListe.Count;`	
`object o = dieListe[1];`	
`bool stecktDrin = dieListe.Contains(b);`	

Tipp: Hier benötigen Sie mehrere Codezeilen.



Spitzen Sie Ihren Bleistift
Lösung

Sie sollten die Lücken in der Tabelle mit dem Code füllen, den wir schreiben müssten, wenn wir mit einem gewöhnlichen Array das Gleiche erreichen wollen wie mit dem List-Code auf der linken Seite.

List	Array
`List<String> dieListe = new List <String>();`	`String[] dieListe = new String[2];`
`String a = "Ja!"`	`String a = "Ja!";`
`dieListe.Add(a);`	`dieListe[0] = a;`
`String b = "Brummer";`	`String b = "Brummer";`
`dieListe.Add(b);`	`dieListe[1] = b;`
`int größe = dieListe.Count;`	`int größe = dieListe.Length;`
`object o = dieListe[1];`	`object o = dieListe[1];`

`bool stecktDrin = dieListe.Contains(b);`	`bool stecktDrin = false;`
	`for (int i = 0; i < dieListe.Length; i++) {`
	`  if (b == dieListe[i]) {`
	`  stecktDrin = true;`
	`  }`
	`}`

Listen sind Objekte und nutzen Methoden wie alle Objekte, mit denen Sie bisher gearbeitet haben. Die Methoden, die für List-Objekte verfügbar sind, können Sie sehen, wenn Sie hinter dem Namen der List-Referenz einen . eingeben. Diesen Methoden übergeben Sie Parameter genau wie denen, die Sie in Ihren eigenen Klassen erstellt haben.

Bei Arrays sind Sie viel eingeschränkter: Die Größe des Arrays muss festgelegt werden, wenn Sie es erstellen. Die Logik für alle Aufgaben, die Sie ausführen wollen, müssen Sie selbst schreiben.

Das .NET Framework bietet eine Array-Klasse, die einige dieser Dinge etwas vereinfacht ... aber wir befassen uns hier mit List-Objekten, weil diese viel einfacher zu verwenden sind.

Listen schrumpfen und wachsen dynamisch

Das Wunderbare an einer `List` ist, dass Sie bei ihrer Erstellung nicht wissen müssen, wie groß sie sein muss. Eine `List` wächst und schrumpft automatisch, um sich an ihren Inhalt anzupassen. Hier ist ein Beispiel mit ein paar Methoden, die die Arbeit mit Listen viel leichter machen, als sie es mit Arrays wäre:

Wir deklarieren eine List mit Schuh-Objekten namens Schuhschrank.

Die new-Anweisung kann auch in der Methode List.Add() verwendet werden.

```
List<Schuh> schuhschrank = new List<Schuh>();

schuhschrank.Add(new Schuh()
    { Art = Art.Sneakers, Farbe = "Schwarz" });
schuhschrank.Add(new Schuh()
    { Art = Art.Clogs, Farbe = "Braun" });
schuhschrank.Add(new Schuh()
    { Art = Art.Lackschuhe, Farbe = "Schwarz" });
schuhschrank.Add(new Schuh()
    { Art = Art.Slipper, Farbe = "Weiß" });
schuhschrank.Add(new Schuh()
    { Art = Art.Slipper, Farbe = "Rot" });
schuhschrank.Add(new Schuh()
    { Art = Art.Sneakers, Farbe = "Grün" });
```

`foreach` ist eine besondere Art von Schleife für Listen. Sie führt ihren Anweisungsblock für jedes Objekt in der Liste aus. Diese Schleife erzeugt einen Bezeichner namens `schuh`. Während die Schleife die Liste durchläuft, setzt sie `schuh` der Reihe nach auf die einzelnen Listenelemente.

Das ist die Anzahl von Schuh-Objekten in der Liste.

foreach funktionieren auch mit Arrays! Sie funktionieren mit jeder Art Auflistung.

```
int anzahlSchuhe = schuhschrank.Count;
foreach (Schuh schuh in schuhschrank) {
    schuh.Art = Art.Flipflops;
    schuh.Farbe = "Orange";
}
```

Diese foreach-Schleife durchläuft alle Schuhe im Schrank.

Die Methode Remove() entfernt Objekte über die Referenz, die Methode RemoveAt() über den Index.

```
schuhschrank.RemoveAt(4);
```

Die Methode Clear() löscht alle Objekte aus der Liste.

```
Schuh schuhDrei = schuhschrank[3];
Schuh schuhFünf = schuhschrank[5];
schuhschrank.Clear();

schuhschrank.Add(schuhDrei);
if (schuhschrank.Contains(schuhFünf))
    Console.WriteLine("Überraschung.");
```

Bevor wir die Liste geleert haben, haben wir Referenzen auf zwei Schuhe gespeichert. Einen haben wir wieder eingefügt, der andere fehlt noch.

Diese Zeile wird nie ausgeführt, da Contains() false liefert. Wir haben nur schuhDrei in die leere Liste eingefügt, nicht schuhFünf.

Das ist die Schuh-Klasse, die wir einsetzen ...

```
public class Schuh {
    public Art Art;
    public string Farbe;
}

public enum Art {
    Sneakers,
    Slipper,
    Sandalen,
    Flipflops,
    Lackschuhe,
    Clogs,
}
```

List-Objekte können beliebige Typen speichern

Sie haben bereits gesehen, dass ein List-Objekt Strings oder Schuhe speichern kann. Sie könnten auch eine Liste mit int-Werten oder beliebigen anderen Objekten erstellen. Das macht List zu einer **generischen Auflistung**. Erstellen Sie ein neues List-Objekt und binden Sie es an einen bestimmten Typ: Sie können Listen mit int-Werten, mit Strings oder mit Schuh-Objekten haben. Das vereinfacht die Arbeit mit Listen – haben Sie eine Liste erstellt, kennen Sie immer den Typ der Daten, die sich darin befinden.

Das heißt nicht, dass Sie hier den Buchstaben T angeben. Das ist nur eine Notation, die Sie sehen, wenn eine Klasse oder Schnittstelle mit allen Typen arbeitet. Der <T>-Teil heißt, dass Sie hier einen beliebigen Typ wie List<Schuh> angeben und damit den Typ der Elemente einschränken können.

```
List<T> name = new List<T>();
```

Listen können entweder sehr flexibel sein (jeden Typ zulassen) oder sehr restriktiv. Sie tun also, was Arrays tun, und noch einiges mehr.

Das .NET Framework bietet einige generische Schnittstellen, die es den von Ihnen erstellten Auflistungen ermöglichen, mit jedem Typ zu arbeiten. List implementiert diese Schnittstellen. Deswegen kann man eine Liste mit int-Werten erstellen und mit ihr auf ziemlich die gleiche Weise arbeiten wie mit einer Liste mit Schuh-Objekten.

Testen Sie das selbst. Geben Sie in der IDE das Wort `List` ein, klicken Sie mit der rechten Maustaste darauf und wählen Sie »Gehe zu Definition«. Das führt Sie zur Deklaration für die Klasse List. Sie implementiert ein paar Schnittstellen:

Hier kommen RemoveAt(), IndexOf() und Insert() her.

```
public class List<T> : IList<T>,
ICollection<T>, IEnumerable<T>, IList,
ICollection, IEnumerable
```

Hier kommen Add(), Clear(), CopyTo() und Remove() her. Das ist die Basis aller generischen Auflistungen.

Das ermöglicht Ihnen unter anderem die Verwendung von foreach.

Punkt für Punkt

- **List** ist eine Klasse des .NET Framework.
- Eine Liste **passt ihre Größe dynamisch** an die Anforderungen an. Sie hat eine bestimmte Kapazität – und diese wächst, während Sie der Liste weitere Elemente hinzufügen.
- Mit **Add()** stecken Sie etwas in eine Liste, mit **Remove()** entfernen Sie etwas aus ihr.
- Mit **RemoveAt()** können Sie Objekte über Ihren Index entfernen.
- Den Typ der Liste deklarieren Sie über ein **Typargument**. Das ist der Typname in spitzen Klammern. Beispiel: List<Frosch> heißt, dass die Liste nur Frosch-Objekte speichern kann.
- Mit **IndexOf()** können Sie ermitteln, an welcher Stelle (und ob) etwas in der Liste steckt.
- Die Anzahl der Elemente der Liste erhalten Sie mit der Eigenschaft **Count**.
- Mit der Methode **Contains()** können Sie prüfen, ob die Liste ein bestimmtes Objekt enthält.
- **foreach** ist ein spezieller Schleifentyp, der alle Elemente in einer Liste durchläuft und Code dafür ausführt. Die Syntax für eine foreach-Schleife ist **foreach (string s in StringList)**. Der foreach-Schleife müssen Sie nicht sagen, dass sie inkrementieren muss. Sie durchläuft automatisch alle Elemente in der Liste.



Code-Magneten

Können Sie die Codeschnipsel so zusammenpuzzeln, dass sie ein funktionierendes Formular ergeben, das die MessageBox unten anzeigt, wenn auf einen Button geklickt wird?

```
private void button1_Click(object sender,
EventArgs e){
```

```
a.RemoveAt(2);
```

```
List<string> a = new List<string>();
```

```
public void drucken (List<string> a){
```

```
if (a.Contains("zwei")) {
    a.Add(zweiMalZwei);
}
```

```
a.Add(zilch);
a.Add(erstes);
a.Add(zweites);
a.Add(drittes);
```

```
}
```

```
string erg = "";
```

```
if (a.Contains("drei")){
      a.Add("vier");
    }
```

```
foreach (string element in a)
{
    erg += "\n" + element;
}
```

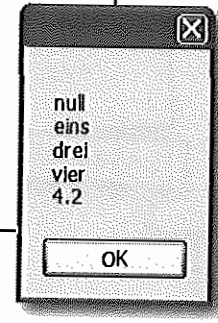
```
MessageBox.Show(erg);
```

```
}
```

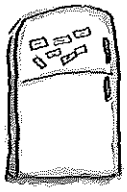
```
if (a.IndexOf("vier") != 4) {
      a.Add(viertes);
        }
```

```
}
```

```
drucken(a);
```




```
string zilch = "null";
string erstes =  "eins";
string zweites = "zwei";
string drittes = "drei";
string viertes = "4.2";
string zweiMalZwei = "2.2";
```



Code-Magneten, Lösung




```
private void button1_Click(object sender,
EventArgs e){
    List<string> a = new List<string>();
    string zilch = "null";
    string erstes =  "eins";
    string zweites = "zwei";
    string drittes = "drei";
    string viertes = "4.2";
    string zweiMalZwei = "2.2";
    a.Add(zilch);
    a.Add(erstes);
    a.Add(zweites);
    a.Add(drittes);
    if (a.Contains("drei")){
        a.Add("vier");
    }
    a.RemoveAt(2);
    if (a.IndexOf("vier") != 4) {
        a.Add(viertes);
    }
    if (a.Contains("zwei")) {
        a.Add(zweiMalZwei);
    }
    drucken(a);
}
public void drucken (List<string> a){
    string erg = "";
    foreach (string element in a)
    {
        erg += "\n" + element;
    }
    MessageBox.Show(erg);
}
}
```

Sehen Sie, warum >>2.2<< der Liste nicht hinzugefügt wird, obwohl es hier deklariert wird?

RemoveAt() entfernt das Element am Index 2 – das ist das dritte Element in der Liste.

Die Methode drucken() nutzt eine foreach-Schleife dazu, eine String-Liste zu durchlaufen, um erst alle Elemente an den String erg anzuhängen und diesen dann auszugeben.

Die foreach-Schleife durchläuft alle Elemente in der Liste und baut aus ihnen einen langen String auf.

Es gibt keine Dummen Fragen

F: Warum sollte ich Enums verwenden, wo es doch Listen gibt? Kann ich mit Listen nicht das gleiche Problem lösen?

A: Enums sind etwas anders als Listen. Enums sind ein praktisches Mittel zur Speicherung von ***Listen mit Konstanten***, das es Ihnen ermöglicht, über einen Namen auf sie zuzugreifen. Sie machen Ihren Code besser lesbar und stellen sicher, dass Sie immer die richtigen Namen verwenden, um auf häufig verwendete Werte zuzugreifen.

Eine Liste kann so ziemlich alles speichern. Da es eine Liste von ***Objekten*** ist, kann jedes Element in einer Liste eigene Methoden und Eigenschaften haben. Enums hingegen muss einer der **Werttypen** von C# zugewiesen werden (wie die, die Sie auf den ersten Seiten von Kapitel 4 verwendet haben). Sie können in ihnen also keine Referenztypen speichern.

Enums können ihre Größe auch nicht dynamisch ändern. Sie können keine Schnittstellen implementieren oder Methoden haben, und Sie müssen sie auf einen anderen Typ casten, um einen Wert aus einem Enum in einer anderen Variablen zu speichern. Zählen Sie das zusammen, gibt es ziemlich große Unterschiede zwischen diesen beiden Möglichkeiten, Daten zu speichern. Beide sind auf ihre jeweilige Weise sehr nützlich.

F: Gut, das klingt, als wären Listen ziemlich mächtig. Warum sollte ich dann noch Arrays verwenden?

A: Arrays benötigen weniger Speicherplatz und Rechenzeit. Führen Sie berechnungsintensive Operationen aus – beispielsweise wenn Sie eine Operation Tausende von Malen durchführen –, können Listen dazu führen, dass Ihr Programm merklich langsamer läuft. Glücklicherweise können Sie eine Liste mit der Methode `ToArray()` leicht in ein Array umwandeln ... und mit einem der Überladungen des `List`-Konstruktors können Sie Arrays ebenso leicht in Listen umwandeln.

Na gut, wir sprechen hier von wirklich minimalen Leistungssteigerungen, die Sie mit Arrays erzielen können, wenn die gleiche Sache unendlich oft gemacht wird.

F: Dieses »generisch« verstehe ich nicht. Wieso sind Listen generische Auflistungen? Warum sind Arrays keine?

A: Eine generische Auflistung ist ein Auflistungsobjekt (oder ein eingebautes Objekt, mit dem Sie Massen von Objekten speichern und verwalten können), das so eingerichtet wurde, dass es nur einen Datentyp speichern kann.

F: Gut. Das erklärt den »Auflistung«-Teil. Aber was macht diese »generisch«?

A: Früher führten Supermärkte allgemeine Waren, die in großen weißen Paketen mit schwarzer Schrift verpackt waren, die nur sagte, was darin ist (»Chips«, »Cola«, »Seife« usw.). Die allgemeine Marke bezog sich nur auf das, was im Paket ist, nicht darauf, wie es angezeigt wird.

Das Gleiche passiert mit generischen Datentypen. Ihre List<> funktioniert immer gleich, egal was darin steckt. Eine Liste mit Schuh-Objekten, Karte-Objekten, int-Werten, long-Werten oder selbst mit anderen Listen verhält sich auf Container-Ebene immer gleich. Egal was in der Liste selbst drinsteckt, Sie können immer hinzufügen, entfernen, löschen usw.

Der Begriff »generisch« bezieht sich darauf, dass die Klasse List selbst mit beliebigen Typen arbeiten kann, obwohl spezifische List-Instanzen immer nur mit einem Typ arbeiten.

Genau das macht das <T>-Zeug. Es bindet eine List-Instanz an einen bestimmten Typ. Aber die Klasse List selbst ist allgemein und kann mit JEDEM Typ arbeiten. Das unterscheidet generische Auflistungen von allem, was Sie bisher gesehen haben.

F: Kann man auch Listen ohne Typ haben?

A: Nein. Jede Liste – genauer, jede generische Auflistung (mehr zu den anderen generischen Auflistungen werden Sie gleich erfahren) – muss mit einem Typ verknüpft sein. C# bietet einen nicht-generischen Listentyp namens ArrayList, der jede Art von Objekt speichern kann. Wenn Sie eine ArrayList verwenden möchten, müssen Sie in Ihren Code eine »`using System.Collections;`«-Zeile einschließen.

Generische Auflistungen sind eine neuere Erweiterung von C# – in frühen Versionen der Sprache gab es sie noch nicht. Aber sie sind so nützlich, dass nicht-generische Auflistungen kaum noch verwendet werden ... und deswegen werden wir uns nicht ausführlicher mit ihnen befassen.

Beim Erstellen eines List-Objekts geben Sie immer den Typ der Daten an, den darin gespeicherte Daten haben werden. Listen können Werttypen und Referenztypen speichern.

Auflistungsinitialisierer funktionieren wie Objektinitialisierer

C# bietet Ihnen eine nette Kurznotation, die die Tipparbeit beim Erstellen und Füllen von Listen verringert. Bei der Erstellung eines neuen List-Objekts können Sie einen **Auflistungsinitialisierer** einsetzen, um ihm eine Ausgangselementliste zuzuweisen. Diese werden hinzugefügt, sobald die Liste erzeugt wird.

Diesen Code haben Sie vor ein paar Seiten gesehen – er erzeugt eine neue List<Schuh> und füllt sie mit neuen Schuh-Objekten.

```
List<Schuh> schuhschrank = new List<Schuh>();
schuhschrank.Add(new Schuh() { Art = Art.Sneakers, Farbe = "Schwarz" });
schuhschrank.Add(new Schuh() { Art = Art.Clogs, Farbe = "Braun" });
schuhschrank.Add(new Schuh() { Art = Art.Lackschuhe, Farbe = "Schwarz" });
schuhschrank.Add(new Schuh() { Art = Art.Slipper, Farbe = "Weiß" });
schuhschrank.Add(new Schuh() { Art = Art.Slipper, Farbe = "Rot" });
schuhschrank.Add(new Schuh() { Art = Art.Sneakers, Farbe = "Grün" });
```

Der gleiche Code mit einem Auflistungsinitialisierer

Auflistungsinitialisierer sind ein C# 3.0-Feature. Wenn Sie immer noch Visual Studio 2005 verwenden, sollten Sie Visual Studio 2008 Express herunterladen, da dieser Code sonst nicht funktioniert.

Einen Auflistungsinitialisierer erstellen Sie, indem Sie im Initialisiererblock einfach nur die Add()-Parameter angeben.

```
List<Schuh> schuhschrank = new List<Schuh>() {
    new Schuh() { Art = Art.Sneakers, Farbe = "Schwarz" },
    new Schuh() { Art = Art.Clogs, Farbe = "Braun" },
    new Schuh() { Art = Art.Lackschuhe, Farbe = "Schwarz" },
    new Schuh() { Art = Art.Slipper, Farbe = "Weiß" },
    new Schuh() { Art = Art.Slipper, Farbe = "Rot" },
    new Schuh() { Art = Art.Sneakers, Farbe = "Grün" },
};
```

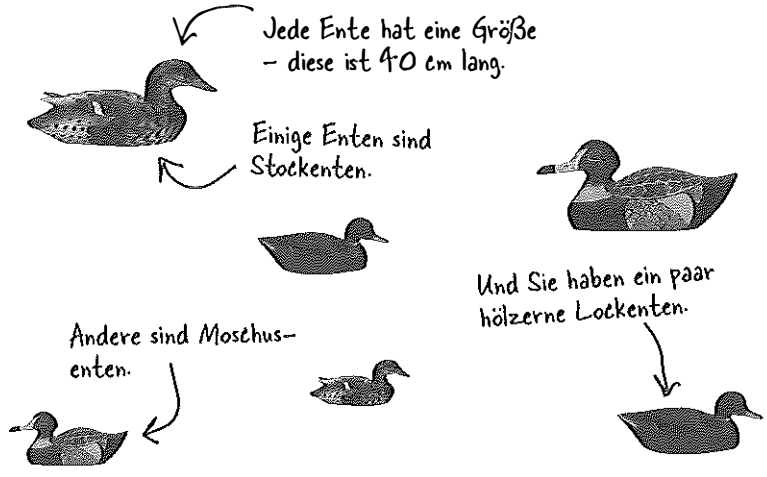
Auf die Anweisung, die die Liste erstellt, folgen geschweifte Klammern, in denen die Elemente durch Kommata getrennt angegeben werden.

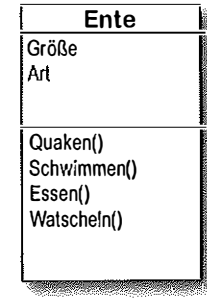
Hier sind die Elemente new-Anweisungen, aber es können beliebige Ausdrücke sein.

Auflistungsinitialisierer machen Ihren Code kompakter, da sie Listenerstellung und -initialisierung kombinieren.

Erstellen wir eine Liste mit Enten

Hier ist eine Ente-Klasse, die Ihre umfangreiche Entensammlung festhält. (Sie sammeln *doch* Enten, oder?)






```
public class Ente {
    public int Größe;
    public EntenArt Art;
}
```

Die Klasse hat zwei öffentliche Felder. Außerdem hat sie einige Methoden, die wir hier nicht zeigen.

```
public enum EntenArt {
    Stockente,
    Moschusente,
    Lockente
}
```

Wir werden ein Enum namens EntenArt verwenden, um die Arten der Enten in Ihrer Sammlung nachzuhalten.

Hier ist der Initialisierer

Wir haben sechs Enten und werden eine List<Ente> mit einem Initialisierer mit sechs Anweisungen erstellen. Die Anweisungen im Initialisierer erstellen eine neue Ente und nutzen einen Objektinitialisierer, um die Größe- und Art-Felder der Enten zu setzen.

```
List<Ente> enten = new List<Ente>() {
    new Ente() { Art = EntenArt.Stockente, Größe = 40 },
    new Ente() { Art = EntenArt.Moschusente, Größe = 42 },
    new Ente() { Art = EntenArt.Lockente, Größe = 35 },
    new Ente() { Art = EntenArt.Moschusente, Größe = 28 },
    new Ente() { Art = EntenArt.Stockente, Größe = 37 },
    new Ente() { Art = EntenArt.Lockente, Größe = 36 },
};
```

Listen sind leicht, aber SORTIEREN kann verzwickt sein

Möglichkeiten, Zahlen oder Buchstaben zu sortieren, fallen einem schnell ein. Aber was ist, wenn man zwei Objekte sortieren soll, insbesondere wenn diese mehrere Felder haben? In einigen Fällen möchte man Objekte über den Wert im Namensfeld sortieren, in anderen könnte es sinnvoller sein, Objekte über ihre Größe oder ihr Geburtsdatum zu sortieren. Es gibt viele Möglichkeiten, sie zu ordnen, und Listen unterstützen alle.

Sie könnten Ente über die Größe sortieren ...

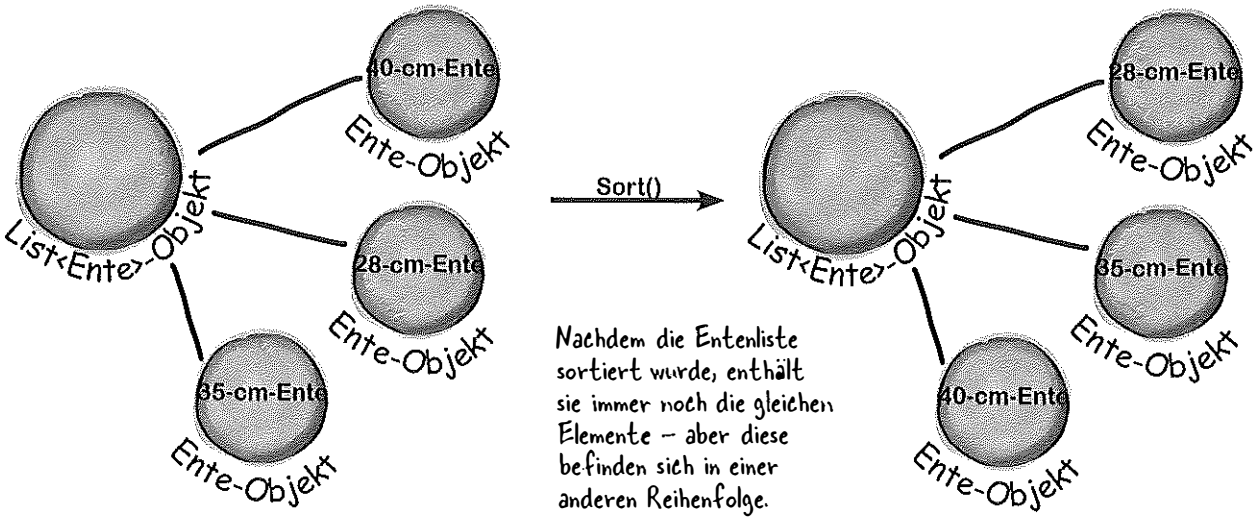


... oder über den Typ:



Listen wissen, wie sie sich sortieren müssen

Jede Liste besitzt eine Sort()-Methode, die all ihre Elemente umordnet und in eine Ordnung bringt. Listen wissen bereits, wie die meisten eingebauten Typen zu sortieren sind, und man kann ihnen leicht beibringen, wie andere Klassen sortiert werden sollen.



Zwei Wege, Ihre Enten zu sortieren

Die Methode List.Sort() weiß, wie ein Typ oder eine Klasse sortiert werden muss, wenn dieser bzw. diese die Schnittstelle **IComparable** implementiert. Diese Schnittstelle hat nur ein Member – eine Methode namens CompareTo(). Sort() nutzt die CompareTo()-Methode eines Objekts, um es mit anderen Objekten zu vergleichen, und nutzt ihren Rückgabewert (einen int), um zu bestimmen, welches Objekt zuerst kommt.

Aber manchmal muss man eine Liste von Objekten sortieren, die `IComparable` nicht implementieren. .NET bietet eine weitere Schnittstelle, die in diesem Fall helfen kann. Sie können Sort() eine Instanz einer Klasse übergeben, die `IComparer` implementiert. Auch diese Schnittstelle hat nur eine Methode. Über die Compare()-Methode des Comparer-Objekts vergleicht die Sort()-Methode der Liste Paare von Objekten, um herauszufinden, welches in der sortierten Liste zuerst kommen muss.

Jede Klasse, die IComparable implementiert und eine CompareTo()-Methode anbietet, arbeitet mit der in Listen eingebauten Sort()-Methode.

Die CompareTo()-Methode eines Objekts vergleicht es mit einem anderen Objekt

Damit wir unsere Liste sortieren können, könnten wir die Klasse Ente also zum einen so modifizieren, dass sie IComparable implementiert. Dazu fügen wir ihr eine CompareTo()-Methode hinzu, die eine Ente-Referenz als Parameter nimmt. Sollte die zu vergleichende Ente in der sortierten Liste nach der aktuellen Ente kommen, liefert CompareTo() eine positive Zahl.

Hier ist eine Ente-Klasse, die sich selbst auf Basis der Entengröße sortiert:

```
public class Ente : IComparable<Ente> {
    public int Größe;
    public EntenArt Art;
```

Wenn Sie IComparable implementieren, geben Sie den zu vergleichenden Typ an, wenn Sie die Implementierung deklarieren.

```
    public int CompareTo(Ente vergleichsEnte) {
        if (this.Größe > vergleichsEnte.Größe)
            return 1;
        else if (this.Größe < vergleichsEnte.Größe)
            return -1;
        else
            return 0;
    }
}
```

Die meisten CompareTo()-Methoden sehen ganz ähnlich aus wie diese hier. Erst vergleicht die Methode das Feld Größe mit dem Größe-Feld der anderen Ente. Ist die aktuelle Ente größer, liefert die Methode 1, andernfalls –1. Sind sie gleich, wird 0 geliefert.

Möchten Sie Ihre Liste von groß nach klein sortieren lassen, müssen Sie CompareTo() eine positive Zahl liefern lassen, wenn die zu vergleichende Ente kleiner als die aktuelle Ente ist, und eine negative, wenn die zu vergleichende Ente größer ist.



Mit IComparer sagen Sie Ihren Listen, wie sie sortieren sollen

Für Listen ist in das .NET Framework eine besondere Schnittstelle eingebaut, mit der Sie eine eigene Sortierlogik aufbauen können. Indem Sie die Schnittstelle **`IComparer`** implementieren, können Sie Ihrer `List` genau sagen, wie Ihre Objekte sortiert werden sollen. Das tun Sie, indem Sie die `Compare()`-Methode der `IComparer`-Schnittstelle implementieren. Sie erwartet zwei Objektparameter, x und y, und liefert einen `int`. Ist x kleiner als y, sollte ein negativer Wert geliefert werden. Sind beide gleich, sollte null, und ist x größer als y, sollte ein positiver Wert geliefert werden.

Hier ist ein Beispiel dafür, wie Sie eine Comparer-Klasse implementieren, die Ente-Objekte anhand der Größe vergleicht:

Diese Klasse implementiert IComparer und gibt den Objekttyp an, den Sie sortieren kann: Ente-Objekte.

```
public class EnteComparer_NachGröße : IComparer<Ente>
{
        public int Compare(Ente x, Ente y)
        {
            if (x.Größe < y.Größe)
                return -1;
            if (x.Größe > y.Größe)
                return 1;
            return 0;
        }
}
```

Die beiden Parameter haben immer den gleichen Typ.

Compare() liefert einen int und hat zwei Parameter des Typs, den Sie sortieren.

In dieser Methode können Sie Vergleiche beliebigen Typs einsetzen.

-1 bedeutet, dass Objekt x vor Objekt y kommen sollte. x ist »kleiner als« y.

Eine 1 bedeutet, dass Objekt x nach Objekt y kommen sollte. x ist »größer als« y.

0 bedeutet, dass diese beiden Objekte als gleich betrachtet werden sollten (gleich in Bezug auf diese Vergleichsberechnung).

Wie Ihre Liste sortiert wird, ist von Ihrer Implementierung von IComparer abhängig.

Erstellen Sie eine Instanz Ihrer Comparer-Klasse

Möchten Sie mithilfe von IComparer sortieren, müssen Sie eine Instanz der Klasse erstellen, die die Schnittstelle implementiert. Dieses Objekt hat nur einen einzigen Zweck – sie soll List.Sort() helfen, die Liste zu sortieren. Aber wie bei jeder (nicht-statischen) Klasse müssen Sie eine Instanz erzeugen, damit Sie die Logik nutzen können.

Den Initialisierungscode für die Liste haben wir weggelassen. Den haben Sie vor ein paar Seiten schon gesehen.

```
List<Ente> enten = new List<Ente>() { ... };

EnteComparer_NachGröße ec = new EnteComparer_NachGröße();

enten.Sort(ec);
```

Als Parameter übergeben Sie Sort() eine Referenz auf die EnteComparer_NachGröße-Instanz.

Von klein nach groß sortiert ...



Mehrere IComparer-Implementierungen bedeuten mehrere Wege, Ihre Objekte zu sortieren

Sie können mehrere IComparer-Klassen erstellen, die jeweils eine eigene Logik haben, die eine andere Aufgabe erfüllen. Dann können Sie den gewünschten Comparer verwenden, wenn Sie auf bestimmte Weise sortieren müssen. Hier ist eine weitere Ente-Comparer-Implementierung.

Dieser Comparer sortiert nach Entenart. Denken Sie daran, dass Sie beim Vergleich der Art die Index-Werte vergleichen.

```
class EnteComparer_NachArt : IComparer<Ente> {
   public int Compare(Ente x, Ente y) {
      if (x.Art < y.Art)
          return -1;
      if (x.Art > y.Art)
          return 1;
      else
          return 0;
        }
}
```

Wir vergleichen die Art-Eigenschaften der Enten, damit die Enten auf Basis der Indexwerte des Enums EntenArt sortiert werden.

Stockente kommt also vor Moschusente und diese vor Lockente.

Beachten Sie, dass »größer als« und »kleiner als« hier eine andere Bedeutung haben. Ein Typ ist »größer als« ein anderer, damit wir eine Ordnung haben, die wir verwenden können.

Hier ist ein Beispiel dafür, wie Enums und Listen zusammenarbeiten. Enums stehen für Zahlen und werden in der Sortierung der Liste verwendet.

```
EnteComparer_NachArt ecArt = new EnteComparer_NachArt();
enten.Sort(ecArt);
```

Nach Entenart sortiert ...



IComparer kann komplexe Vergleiche durchführen

Erstellen Sie eine separate Klasse für das Sortieren Ihrer Enten, hat das den Vorteil, dass Sie in diese Klasse eine komplexere Logik einbauen können – und Member ergänzen können, die dabei helfen, die Sortierkriterien für die Liste festzulegen.

Hier ist eine komplexere Klasse für den Vergleich von Enten. Ihre Compare()-Methode erwartet die gleichen Parameter, schaut aber auf das öffentliche Feld SortierenNach, um nachzusehen, wie die Enten sortiert werden sollen.

```
public class EnteComparer : IComparer<Ente> {
    public enum Sortierung {
        GrößeDannArt,
        ArtDannGröße,
    }

    public Sortierung SortierenNach = Sortierung.GrößeDannArt;

    public int Compare(Ente x, Ente y) {
        if (SortierenNach == Sortierung.GrößeDannArt)
            if (x.Größe > y.Größe)
                return 1;
            else if (x.Größe < y.Größe)
                return -1;
            else
                if (x.Art > y.Art)
                    return 1;
                else if (x.Art < y.Art)
                    return -1;
                else
                    return 0;
        else
            if (x.Art > y.Art)
                return 1;
            else if (x.Art < y.Art)
                return -1;
            else
                if (x.Größe > y.Größe)
                    return 1;
                else if (x.Größe < y.Größe)
                    return -1;
                else
                    return 0;
    }
}

EnteComparer ec = new EnteComparer();

ec.SortierenNach = EnteComparer.Sortierung.ArtDannGröße;
enten.Sort(ec);

ec.SortierenNach = EnteComparer.Sortierung.GrößeDannArt;
enten.Sort(ec);
```

Dieses Enum sagt dem Objekt, wie die Enten sortiert werden sollen.

Diese if-Anweisung prüft das Feld SortierenNach. Ist es auf GrößeDannArt gesetzt, wird zunächst über die Größe der Enten, dann innerhalb jeder Größe über die Art sortiert.

Statt einfach 0 zu liefern, wenn zwei Enten die gleiche Größe haben, prüft der Comparer ihre Art und liefert nur 0, wenn zwei Enten die gleiche Größe und die gleiche Art haben.

Ist SortierenNach nicht auf GrößeDannArt gesetzt, sortiert der Comparer die Enten erst über die Entenart. Haben zwei Enten die gleiche Art, wird ihre Größe verglichen.

So verwenden wir dieses Comparer-Objekt. Erst instantiieren wir es. Dann können wir das SortierenNach-Feld des Objekts setzen, bevor wir enten.Sort() aufrufen. Wie die Liste die Enten sortiert, können Sie jetzt ganz einfach anpassen, indem Sie das Feld ändern.



Erzeugen Sie fünf Zufallskarten und sortieren Sie diese dann.

Konsolenanwendungen, die Sie von der Kommandozeile ausführen, nutzen die Methode Console.WriteLine() sehr oft. Aber das geht auch in Windows-Anwendungen.

1 Erzeugen Sie Code, der einen gemischten Kartenstapel erzeugt.

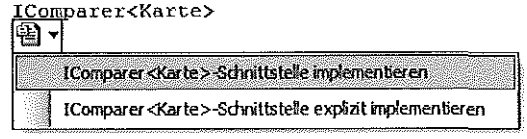
Fügen Sie einem Formular einen Button hinzu, der zufällig fünf Karte-Objekte erzeugt. Nutzen Sie nach der Erstellung der einzelnen Objekte die eingebaute Methode `Console.WriteLine()`, um ihren Namen in die **Ausgabe** zu schreiben. Das, was in die Ausgabe geschrieben wird, können Sie einsehen, indem Sie, während das Programm läuft, im Ansicht-Menü »Ausgabe« wählen.

2 Erzeugen Sie zur Sortierung eine Klasse, die IComparer<Karte> implementiert.

Das ist ein guter Punkt, um auf IDE-Hilfe zur Implementierung von Schnittstellen zurückzugreifen:

```
public class KarteComparer_NachWert : IComparer<Karte>
```

Klicken Sie dann auf IComparer und lassen Sie den Mauszeiger über dem I schweben. Darunter erscheint dann ein Kästchen. Klicken Sie darauf, öffnet die IDE ein Fenster:



Klicken Sie auf »Implement interface IComparer<Karte>«, fügt die IDE automatisch die Eigenschaften und Methoden ein, die Sie implementieren müssen. In diesem Fall wird eine leere Compare()-Methode für den Vergleich zweier Karten, `x` und `y`, erzeugt. Schreiben Sie die Methode so, dass sie 1 liefert, wenn `x` größer als `y` ist, −1, wenn es kleiner ist, und 0, wenn beide gleich sind. Sorgen Sie hier dafür, dass Könige nach Bauern, Bauern nach Vieren und Vieren nach Assen kommen.

3 Stellen Sie sicher, dass die Ausgabe richtig aussieht.

So sollte das Ausgabefenster aussehen, wenn Sie auf den Button klicken.

Nutzen Sie die eingebaute Methode Console.WriteLine(), wird diesem Ausgabefenster eine Zeile hinzugefügt.



Ihr IComparer-Objekt muss die Karten nach dem Wert sortieren. Die Karte mit dem kleinsten Wert steht also am Anfang der Liste.



Lösungen zu den Übungen

Erzeugen Sie fünf Zufallskarten und sortieren Sie diese dann.

Das ist das Herz der Kartensortierung, das von der eingebauten Methode List.Sort() verwendet wird. Sort() nimmt ein IComparer-Objekt, das eine Methode hat: Compare(). Diese Implementierung vergleicht zwei Karten erst anhand des Werts, dann anhand der Farbe.

```
public class KarteComparer_NachWert : IComparer<Karte> {
    public int Compare(Karte x, Karte y) {
        if (x.Wert < y.Wert) {
            return -1;
        }
        if (x.Wert > y.Wert) {
            return 1;
        }
        if (x.Farbe < y.Farbe) {
            return -1;
        }
        if (x.Farbe > y.Farbe) {
            return 1;
        }
        return 0;
    }
}
```

Hat x einen höheren Wert, wird 1 geliefert. Hat x einen kleineren Wert, wird -1 geliefert. Denken Sie daran, dass die Return-Anweisungen die Methode sofort beenden.

Diese Anweisungen werden nur ausgeführt, wenn x und y den gleichen Wert haben – d.h. keine der vorangehenden return-Anweisungen ausgeführt wurde.

Wurde keine der anderen vier return-Anweisungen ausgeführt, müssen die Karten gleich sein – also wird 0 geliefert.

Hier ist eine `List` mit `Karte`-Objekten, in der die Karten gespeichert werden. Befinden diese sich einmal in der Liste, ist das Sortieren ein Klacks.

```
private void button1_Click(object sender, EventArgs e)
{
    Console.WriteLine("Fünf zufällige Karten:");
    List<Karte> karten = new List<Karte>();
    for (int i = 0; i < 5; i++)
    {
        karten.Add(new Karte((Karte.Farben)zufall.Next(4),
                             (Karte.Werte)zufall.Next(1, 14)));
        Console.WriteLine(karten[i].Name);
    }

    Console.WriteLine("Die gleichen Karten sortiert:");
    karten.Sort(new KarteComparer_NachWert());
    foreach (Karte karte in karten)
    {
        Console.WriteLine(karte.Name);
    }
}
```

Schlüssel und Werte speichern Sie in Wörterbüchern

Eine Liste ist wie eine lange Seite voll mit Namen. Aber was ist, wenn Sie zu jedem Namen gern auch eine Adresse hätten? Oder für jeden Wagen in der Liste Garage ein Kfz-Kennzeichen? Dann brauchen Sie ein **Dictionary** oder Wörterbuch. In einem Dictionary können Sie einen besonderen Wert – den **Schlüssel** – mit anderen Daten – dem **Wert** – verknüpfen. Eine Sache noch: In einem Dictionary kann ein bestimmter Schlüssel **nur einmal erscheinen.**

Wör·ter·buch

Ein Buch, das die Wörter einer Sprache in alphabetischer Reihenfolge aufführt und ihre Bedeutung angibt.

Das sind die Schlüssel, über die Sie eine Definition in einem Wörterbuch nachschlagen.

Das ist der Wert, nämlich die Daten, die mit einem bestimmten Schlüssel verknüpft sind.

So deklarieren Sie in C# ein `Dictionary`:

```
Dictionary <TKey, TValue> kv = new Dictionary <TKey, TValue>();
```

Das ist wie bei List<T>. Das <T> bedeutet, dass hier ein Typ hinkommt. Sie können also einen Typ für den Schlüssel (TKey) und einen für den Wert (TValue) deklarieren.

Das erste Element in spitzen Klammern ist immer der Schlüssel, das zweite der Wert.

Und hier ist das `Dictionary` im Einsatz:

```
private void button1_Click(object sender, EventArgs e)
{
    Dictionary<string, string> wortDefinition =
        new Dictionary<string, string>();

    wortDefinition.Add ("Wörterbuch", "Ein Buch, das die Wörter einer Sprache in "
                        + "alphabetischer Reihenfolge aufführt und ihre Bedeutung angibt");
    wortDefinition.Add ("Schlüssel", "Ein Ding, das ein Mittel für den Zugriff auf "
                        + "unser Verständnis von etwas gibt.");
    wortDefinition.Add ("Wert", "Eine Größe, Menge oder Zahl.");

    if (wortDefinition.ContainsKey("Schlüssel")){
        MessageBox.Show(wortDefinition["Schlüssel"]);
    }

}
```

Dieses Wörterbuch hat Strings als Schlüssel und Werte. Es ist wie ein echtes Wörterbuch: Begriff und Definition.

Mit der Methode Add() fügen Sie Schlüssel und Werte hinzu.

Add() nimmt einen Schlüssel und einen Wert.

ContainsKey() sagt Ihnen, ob das Dictionary einen Schlüssel enthält. Praktisch, nicht wahr?

So erhalten Sie den Wert für einen Schlüssel. Das sieht wie ein Array-Index aus: Rufe den Wert für den Schlüssel an diesem Index ab.

Die Dictionary-Funktionalität im Überblick

Wörterbücher haben große Ähnlichkeit mit Listen. Beide Typen sind flexibel, lassen Sie mit einer Menge Datentypen arbeiten und besitzen zusätzlich viele eingebaute Funktionen. Hier sind die wichtigsten `Dictionary`-Methoden:

★ Ein Element hinzufügen.

Einem Wörterbuch fügen Sie ein Element hinzu, indem Sie seiner `Add()`-Methode einen Schlüssel und einen Wert übergeben.

```
Dictionary<string, string> dasWörterbuch = new Dictionary<string, string>();

    dasWörterbuch.Add("der Schlüssel", "der Wert");
```

★ Über den Schlüssel einen Wert nachschlagen.

Die wichtigste Verwendung von Wörterbüchern ist das Nachschlagen von Werten – das ist auch nur vernünftig, da Sie diese Werte in einem Wörterbuch gespeichert haben, damit Sie sie über einen eindeutigen Schlüssel nachschlagen können.

```
    string wertSuchen = dasWörterbuch["der Schlüssel"];
```

★ Ein Element entfernen.

Wie bei Listen entfernen Sie Elemente aus einem Wörterbuch mit der Methode `Remove()`. Remove() müssen Sie nur den Schlüssel übergeben, um Schlüssel und Wert zu entfernen.

```
    dasWörterbuch.Remove("der Schlüssel");
```

Die Schlüssel in einem Wörterbuch sind eindeutig. Jeder erscheint nur ein einziges Mal. Werte können beliebig oft erscheinen – zwei Schlüssel können den gleichen Wert haben. So weiß das Dictionary-Objekt, welches Element gemeint ist, wenn Sie einen Schlüssel nachschlagen.

★ Eine Liste der Schlüssel abrufen.

Eine Liste aller Schlüssel im Wörterbuch können Sie als `KeyCollection` abrufen, die Sie in einer foreach-Schleife durchlaufen können. Eine KeyCollection nutzen Sie so:

```
foreach (string schlüssel in dasWörterbuch.Keys) { ... };
```

Keys ist eine Eigenschaft eines Dictionary-Objekts. Die Schlüssel in diesem Dictionary sind Strings, Keys ist also eine Auflistung mit Strings.

★ Eine Liste der Werte abrufen.

Eine Liste der Werte im Wörterbuch können Sie als `ValueCollection` abrufen. Meist nutzen Sie auch ValueCollection-Objekte in foreach-Schleifen:

```
foreach (string wert in dasWörterbuch.Values) { ... };
```

Da dieses Wörterbuch String-Werte hat, muss auch die foreach-Variable ein String sein.

Schlüssel und Wert können auch unterschiedliche Typen haben

Wörterbücher sind sehr vielseitig und können fast alles aufnehmen: Strings, Zahlen und sogar Objekte. Hier ist ein Beispiel für ein Wörterbuch, das einen int als Schlüssel und ein Ente-Objekt als Wert speichert. Das `Ente`-Objekt hat `Größe`- und `Art`-Felder, die im Konstruktor gesetzt werden.

Hier wird das Wörterbuch deklariert. Es speichert Zahlen und Enten. Wir fügen dem Wörterbuch die Enten hinzu und geben ihnen eine eindeutige ID als Schlüssel.

```
Dictionary<int, Ente> entenBuch = new Dictionary<int, Ente>();

entenBuch.Add(5155, new Ente() { Art = EntenArt.Stockente, Größe = 15 });

entenBuch.Add(6256, new Ente() { Art = EntenArt.Stockente, Größe = 14 });

entenBuch.Add(2799, new Ente() { Art = EntenArt.Stockente, Größe = 13 });
```

Die Eigenschaft Count gibt an, wie viele Schlüssel/Wert-Paare das Wörterbuch enthält.

```
int wieViele = entenBuch.Count;

Console.WriteLine("Es gibt {0} Enten.", wieViele);
```

Müssen Sie Schlüssel/Wert-Paare abrufen, können Sie ein KeyValue-Pair<> verwenden.

Diese Schleife weist das jeweils aktuelle Schlüssel/Wert-Paar idEnte zu.

```
foreach (KeyValuePair<int, Ente> idEnte in entenBuch)

        Console.WriteLine("Schlüssel/Wert-Paar: {0}: {1}, {2}",

            idEnte.Key, idEnte.Value.Größe, idEnte.Value.Art.ToString() );
```

Jedes KeyValuePair hat einen Schlüssel und einen Wert. Da der Wert eine Ente ist, können Sie ihre Felder verwenden.

```
foreach (Ente ente in entenBuch.Values)

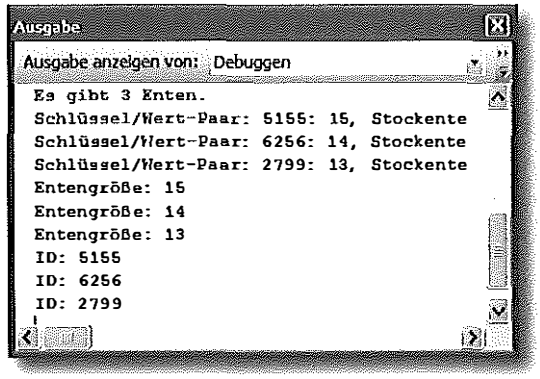
    Console.WriteLine("Entengröße: {0}", ente.Größe);
```

Diese foreach-Schleife durchläuft alle Werte im Wörterbuch.

```
foreach (int schlüssel in entenBuch.Keys)

    Console.WriteLine("ID: {0}", schlüssel);
```

Diese foreach-Schleife zieht alle Schlüssel aus dem Wörterbuch heraus.



Diese Ausgabe gibt der Code auf der Konsole aus.



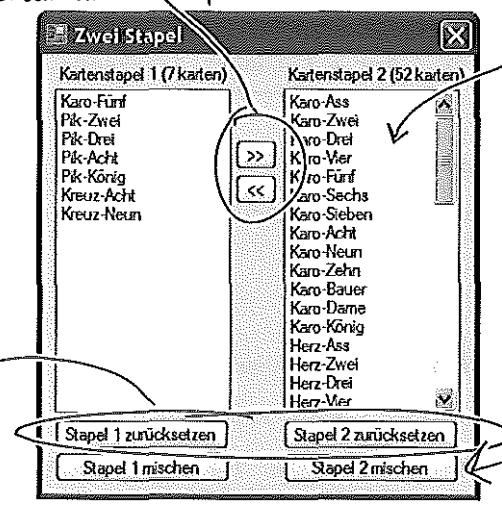
1 Erstellen Sie ein Formular, um Karten zwischen zwei Stapeln zu tauschen.

Die Klasse Karte haben wir bereits erstellt. Jetzt wollen wir eine Klasse erstellen, die beliebig viele Karten aufnehmen kann. Wir werden sie `Kartenstapel` nennen. Ein richtiges Kartenspiel hat 52 Karten, aber die Klasse `Kartenstapel` kann beliebig viele Karten speichern – auch keine.

Erstellen Sie dann ein Formular, das Ihnen den Inhalt von zwei `Kartenstapel`-Objekten zeigt. Beim Start des Programms hat Stapel 1 bis zu zehn zufällige Karten und Stapel 2 einen vollständigen Kartensatz mit 52 Karten. Beide Stapel sind nach Farbe und Wert sortiert – und beide können über Buttons zurück in ihren Ausgangszustand versetzt werden. Außerdem hat das Formular zwei Buttons (<< und >>), um Karten zwischen den Stapeln zu verschieben.

Diese Buttons heißen nachStapel2. (oben) und nachStapel1 (unten). Sie verschieben Karten von einem Stapel in den anderen.

Mit der Name-Eigenschaft können Sie Buttons einen Namen geben, der Ihren Code besser lesbar macht. Klicken Sie dann doppelt auf den Button, erhält sein Event-Handler einen entsprechenden Namen.



Verwenden Sie für die Stapel zwei ListBox-Steuerelemente. Ein Klick auf den Button nachStapel1 verschiebt die ausgewählte Karte von Stapel 1 nach Stapel 2.

Diese Buttons heißen mischen1 and mischen2.. Sie rufen die entsprechenden Kartenstapel.Mischen()-Methode auf und zeichnen dann den Stapel neu.

Die Buttons rücksetzen1 und rücksetzen2. rufen erst StapelRücksetzen() und dann StapelNeuZeichnen() auf.

Neben den Event-Handlern für die sechs Buttons braucht das Formular noch zwei Methoden. Erstellen Sie zunächst eine StapelRücksetzen()-Methode, die den Stapel wieder in seinen Ausgangszustand versetzt. Sie nimmt einen int als Parameter: Ist dessen Wert 1, wird der erste Kartenstapel zurückgesetzt, indem er neu auf einen Stapel mit bis zu zehn zufälligen Karten initialisiert wird, ist der Wert 2, erhält der zweite Kartenstapel wieder einen vollständigen Satz von 52 Karten. Fügen Sie dann diese Methode hinzu:

StapelNeuZeichnen() aktualisiert die beiden ListBox-Objekte, damit sie den aktuellen Zustand der Kartenstapel-Objekte spiegeln.

```
private void StapelNeuZeichnen(int DeckNumber) {
    if (DeckNumber == 1) {
        listBox1.Items.Clear();
        foreach (string kartenname in stapel1.KartenNamenAbrufen())
            listBox1.Items.Add(kartenname);
        label1.Text = "Kartenstapel 1 (" + stapel1.Anzahl + " Karten)";
    } else {
        listBox2.Items.Clear();
        foreach (string kartenname in stapel2.KartenNamenAbrufen())
            listBox2.Items.Add(kartenname);
        label2.Text = "Kartenstapel 2 (" + stapel2.Anzahl + " Karten)";
    }
}
```

Sehen Sie sich an, wie wir foreach verwenden, um die Karten im Stapel der jeweiligen ListBox hinzuzufügen.

2 Erstellen Sie die Klasse Kartenstapel.

Eine Klassendeklaration ohne Implementierung nennt man »Gerüst«.

Hier ist das Gerüst für die Klasse Kartenstapel. Ein paar Methoden haben wir für Sie gefüllt. Die Methoden Mischen() und KartenNamenAbrufen() müssen Sie füllen, die Methode Sortieren() funktionsfähig machen. Außerdem haben wir zwei nützliche **überladene Konstruktoren** eingefügt: Einer erstellt einen vollständigen Kartensatz, der andere lädt ein Array mit Karte-Objekten in den Stapel.



Der Kartenstapel speichert die Karten in einer privaten Liste, damit sie gut gekapselt sind.

Erhält der Konstruktor keinen Parameter, erstellt er einen vollständigen Kartensatz.

```
public class Kartenstapel {
    private List<Karte> karten;
    private Random zufall = new Random();

    public Kartenstapel() {
        karten = new List<Karte>();
        for (int farbe = 0; farbe <= 3; farbe++)
            for (int value = 1; value <= 13; value++)
                karten.Add(new Karte((Karte.Farben)farbe, (Karte.Werte)value));
    }

    public Kartenstapel(Karte[] anfangsKarten) {
        karten = new List<Karte>(anfangsKarten);
    }

    public int Anzahl { get { return karten.Count; } }

    public void Hinzufügen(Karte neueKarte) {
        karten.Add(neueKarte);
    }

    public Karte Ausgeben(int index) {
        Karte ausgabeKarte = karten[index];
        karten.RemoveAt(index);
        return ausgabeKarte;
    }

    public void Mischen() {
        // Mischt die Karten, indem sie zufällig sortiert werden.
    }

    public string[] KartenNamenAbrufen() {
        // Liefert ein String-Array mit den Namen aller Karten.
    }

    public void Sortieren() {
        karten.Sort(new KarteComparer_NachFarbe());
    }
}
```

Der überladene Konstruktor nimmt einen Parameter – ein Array mit Karten, das zum Anfangsstapel wird.

Ausgeben() gibt eine Karte aus – es wird eine Karte aus dem Stapel entfernt und eine Referenz darauf geliefert. Die Karte wird am übergebenen Index ausgegeben. Die oberste Karte erhalten Sie mit 0.

Tipp: Die SelectedIndex-Eigenschaft der ListBox entspricht immer dem Index der Karte in der Liste. Sie können sie direkt an Ausgeben() übergeben. Ist keine Karte ausgewählt, ist der Wert der Eigenschaft 0. Dann sollte Ausgeben() nichts tun.

Sie müssen die Methoden Mischen() und KartenNamenAbrufen() füllen und eine Klasse schreiben, die IComparer implementiert, damit Sortieren() funktioniert. Und Sie müssen die bereits geschriebene Klasse Karte hinzufügen. Machen Sie das über »Vorhandenes Element hinzufügen«. Vergessen Sie nicht, den Namensraum zu ändern.

Noch ein Tipp: Mischen() können Sie leicht über das Formular testen. Klicken Sie so lange auf »Stapel 1 zurücksetzen«, bis Sie einen Stapel mit drei Karten erhalten. So sehen Sie leicht, ob Ihr Mischen()-Code funktioniert.



Lösungen zu den Übungen

Erstellen Sie eine Klasse zur Speicherung eines Kartenstapels und ein Formular, das diese nutzt.

```
public class Kartenstapel {
    private List<Karte> karten;
    private Random zufall = new Random();

    public Kartenstapel() {
        karten = new List<Karte>();
        for (int farbe = 0; farbe <= 3; farbe++)
            for (int wert = 1; wert <= 13; wert++)
                karten.Add(new Karte((Karte.Farben)farbe, (Karte.Werte)wert));
    }

    public Kartenstapel(Karte[] anfangsKarten) {
        karten = new List<Karte>(anfangsKarten);
    }

    public int Anzahl { get { return karten.Count; } }

    public void Hinzufügen(Karte neueKarte) {
        karten.Add(neueKarte);
    }

    public Karte Ausgeben(int index) {
        Karte ausgabeKarte = karten[index];
        karten.RemoveAt(index);
        return ausgabeKarte;
    }

    public void Mischen() {
        List<Karte> neueKarten = new List<Karte>();
        while (karten.Count > 0) {
            int karte = zufall.Next(karten.Count);
            neueKarten.Add(karten[karte]);
            karten.RemoveAt(karte);
        }
        karten = neueKarten;
    }

    public string[] KartenNamenAbrufen() {
        string[] kartennamen = new string[karten.Count];
        for (int i = 0; i < karten.Count; i++)
            kartennamen[i] = karten[i].Name;
        return kartennamen;
    }

    public void Sortieren() {
        karten.Sort(new KarteComparer_NachFarbe());
    }
}
```

Hier ist der Konstruktor, der den vollständigen Kartensatz mit 52 Karten erstellt. Er nutzt eine geschachtelte Schleife. Die äußere durchläuft zuerst die vier Farben. Das bedeutet, dass die innere, die die 13 Werte durchläuft, vier Mal ausgeführt wird, einmal für jede Farbe.

Das ist der zweite Konstruktor – diese Klasse hat zwei überladene Konstruktoren, die jeweils unterschiedliche Parameter erwarten.

Die Methoden Hinzufügen und Ausgeben sind einfach – sie nutzen Methoden der Liste karten. Die Methode Ausgeben entfernt eine Karte aus der Liste, und die Methode Hinzufügen fügt der Liste eine Karte hinzu.

Mischen() erzeugt eine neue List<Karten> namens neueKarten. Dann zieht sie zufällig Karten aus dem Feld karten und steckt sie in neueKarten, bis karten leer ist. Ist sie damit durch, setzt sie karten auf neueKarten. Auf die alte Instanz zeigt dann keine Referenz mehr. Sie kann also vom Garbage Collector eingesammelt werden.

Die Methode KartenNamenAbrufen() muss ein Array erzeugen, das alle Kartennamen aufnehmen kann. Diese nutzt eine for-Schleife. Sie könnten aber auch eine foreach-Schleife nutzen.

```
class KarteComparer_NachFarbe : IComparer<Karte>
{
    public int Compare(Karte x, Karte y)
    {
        if (x.Farbe > y.Farbe)
            return 1;
        if (x.Farbe < y.Farbe)
            return -1;
        if (x.Wert > y.Wert)
            return 1;
        if (x.Wert < y.Wert)
            return -1;
        return 0;
    }
}
```

Nach Farbe zu sortieren geht ganz ähnlich wie das Sortieren nach Wert. Der einzige Unterschied ist, dass hier erst die Farben verglichen werden und die Werte nur, wenn die Farben gleich sind.

Statt if/else if haben wir eine Folge von if-Anweisungen verwendet. Das funktioniert, weil die einzelnen Anweisungen nur ausgeführt werden, wenn die vorangehenden nicht ausgeführt wurden – andernfalls hätten die vorangehenden bereits einen Wert zurückgeliefert.

```
    Kartenstapel stapel1;
    Kartenstapel stapel2;
    Random zufall = new Random();

    public Form1() {
        InitializeComponent();
        StapelRücksetzen(1);
        StapelRücksetzen(2);
        StapelNeuZeichnen(1);
        StapelNeuZeichnen(2);
    }
```

Der Konstruktor des Formulars muss die beiden Stapel zurücksetzen. Dann zeichnet er sie.

```
    private void StapelRücksetzen(int stapelNr) {
        if (stapelNr == 1) {
            int kartenanzahl = zufall.Next(1, 11);
            stapel1 = new Kartenstapel(new Karte[] { });
            for (int i = 0; i < kartenanzahl; i++)
                stapel1.Hinzufügen(new Karte((Karte.Farben)zufall.Next(4),
                                             (Karte.Werte)zufall.Next(1, 14)));
            stapel1.Sortieren();
        } else
            stapel2 = new Kartenstapel();
    }
```

Um den ersten Stapel zurückzusetzen, ruft diese Methode zunächst zufall.Next() auf, um zu bestimmen, wie viele Karten der Stapel haben soll, und erstellt dann einen neuen leeren Stapel. Danach nutzt sie eine for-Schleife, um diesem Stapel entsprechend viele zufällige Karten hinzuzufügen. Der zweite Stapel wird einfach zurückgesetzt, indem eine neue Instanz von Kartenstapel erzeugt wird.

Die Methode StapelNeuZeichnen() steht bereits in den Anleitungen zu dieser Übung.

Wir sind noch nicht fertig – blättern Sie um!



Lösungen zu den Übungen (Fortsetzung)

Geben Sie Ihren Steuerelementen Namen, macht das Ihren Code besser lesbar. Hießen diese Buttons button1_Click, button2_Click usw., wüssten Sie nicht, den Code welches Buttons Sie vor sich haben.

Hier ist der Rest des Codes für das Formular.

```
    private void rücksetzen1_Click(object sender, EventArgs e) {
        StapelRücksetzen(1);
        StapelNeuZeichnen(1);
    }

    private void rücksetzen2_Click(object sender, EventArgs e) {
        StapelRücksetzen(2);
        StapelNeuZeichnen(2);
    }

    private void mischen1_Click(object sender, EventArgs e) {
        stapel1.Mischen();
        StapelNeuZeichnen(1);
    }

    private void mischen2_Click(object sender, EventArgs e) {
        stapel2.Mischen();
        StapelNeuZeichnen(2);
    }

    private void nachStapel1_Click(object sender, EventArgs e) {
        if (listBox2.SelectedIndex >= 0)
            if (stapel2.Anzahl > 0) {
                stapel1.Hinzufügen(stapel2.Ausgeben(listBox2.SelectedIndex));
            }
        StapelNeuZeichnen(1);
        StapelNeuZeichnen(2);
    }

    private void nachStapel2_Click(object sender, EventArgs e) {
        if (listBox1.SelectedIndex >= 0)
            if (stapel1.Anzahl > 0)
                stapel2.Hinzufügen(stapel1.Ausgeben(listBox1.SelectedIndex));
        StapelNeuZeichnen(1);
        StapelNeuZeichnen(2);
    }
}
```

Diese Buttons sind ziemlich einfach – erst mischen sie den Stapel oder setzen ihn zurück, dann zeichnen sie ihn neu.

Über die SelectedIndex-Eigenschaft der ListBox können Sie herausfinden, welche Karte der Benutzer ausgewählt hat. Diese verschieben Sie dann vom einen Stapel in den anderen. (Ist der Wert der Eigenschaft kleiner 0, ist keine Karte ausgewählt. Der Button macht dann nichts.) Wurde die Karte verschoben, müssen beide Stapel neu gezeichnet werden.

Auch Sie können Methoden überladen

Bislang haben Sie **überladene Methoden** und sogar überladene Konstruktoren genutzt, die Teil der eingebauten .NET Framework-Klassen sind. Sie wissen also bereits, wie nützlich sie sein können. Wäre es nicht cool, wenn Sie in Ihre eigenen Klassen überladene Methoden einbauen könnten? Ja, auch Sie können das -- und es ist nicht einmal schwer! Schreiben Sie einfach eine oder zwei weitere Methoden mit dem gleichen Namen, aber einer anderen Parameterliste.



1 **Erzeugen Sie ein neues Projekt und fügen Sie ihm die Klasse Karte hinzu.**
Dazu klicken Sie im Projektmappen-Explorer wieder mit der rechten Maustaste auf das Projekt und wählen im Menü Hinzufügen »Vorhandenes Element«. Die IDE erstellt eine Kopie der Klasse und fügt sie dem Projekt hinzu. Da die Datei **noch den Namensraum aus dem alten Projekt hat**, müssen Sie oben in der Datei Karte.cs die `namespace`-Zeile so ändern, dass sie dem neuen Projekt entspricht.

Tun Sie das nicht, müssen Sie den alten Namensraum angeben, um auf Karte zuzugreifen (alterNamensraum.Karte).

2 **Fügen Sie Karte neue überladene Methoden hinzu.**
Erzeugen Sie zwei `static EntsprichtKarte()`-Methoden. Die erste soll die Farbe einer Karte prüfen, die zweite ihren Wert. Beide liefern nur true, wenn die Karte passt.

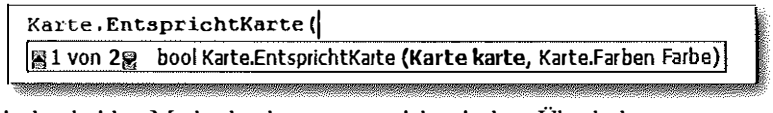
```
public static bool EntsprichtKarte(Karte karte, Karte.Farben Farbe) {
    if (karte.Farbe == Farbe) {
        return true;
    } else {
        return false;
    }
}
public static bool EntsprichtKarte(Karte karte, Karte.Werte Wert) {
    if (karte.Wert == Wert) {
        return true;
    } else {
        return false;
    }
}
```

Überladene Methoden müssen nicht statisch sein. Aber es ist gut, wenn Sie das Schreiben statischer Methoden etwas üben.

3 **Fügen Sie dem Formular einen Button hinzu, um die neuen Methoden zu verwenden.**
Fügen Sie dem Button diesen Code hinzu:

```
Karte karte = new Karte(Karte.Farben.Herz, Karte.Werte.Drei);
bool kartePasst = Karte.EntsprichtKarte(karte, Karte.Farben.Kreuz);
```

Tippen Sie »`EntsprichtKarte(`« ein, zeigt Ihnen die IDE, dass Sie tatsächlich eine Methode überladen haben:



Spielen Sie etwas mit den beiden Methoden herum, um sich mit dem Überladen vertraut zu machen.



Lange Übung

Erstellen Sie ein Go Fish!-Spiel, das Sie gegen den Computer spielen können.

Dies ist eine etwas andere Übung ...

Es ist recht wahrscheinlich, dass Sie C# lernen, weil Sie eine Anstellung als professioneller Entwickler suchen. Deswegen haben wir diese Übung wie eine übliche Arbeitsanweisung aufgebaut. Wenn Sie als Programmierer in einem Team arbeiten, bauen Sie meist kein vollständiges Programm auf. Stattdessen erstellen Sie nur einen *Teil* eines größeren Programms. Wir werden Ihnen also ein Puzzle geben, in das einige Teile bereits eingesetzt wurden. Der Code für das Formular steht vollständig auf der nächsten Seite unter Schritt 3. Sie müssen ihn nur eingeben – vielleicht scheint Ihnen das ja ein Vorteil zu sein, aber es bedeutet, dass Ihre Klassen ***mit diesem Code arbeiten müssen***. Und das kann eine Herausforderung sein!

1 Beginnen Sie mit der Spezifikation

Jedes professionelle Software-Projekt beginnt mit einer Spezifikation und dieses ist keine Ausnahme. Sie werden ein Spiel für die Quartettvariante ***Go Fish!*** erstellen. Es gibt viele Regelvarianten für dieses Spiel, deswegen finden gibt es hier zunächst eine Zusammenfassung der Regeln, die wir verwenden werden:

- Das Spiel beginnt mit einem Satz mit 52 Karten. Jeder Spieler erhält fünf Karten. Der verbleibende Kartenstapel wird als **Stock** bezeichnet. Die Spieler fragen einander abwechselnd nach einem Wert (»Hat einer Siebener?«). Hat ein anderer Spieler Karten mit diesem Wert, muss er Sie abtreten. Hat kein anderer Spieler eine Karte mit diesem Wert, muss der Spieler »Fischen gehen«, indem er eine Karte aus dem Stapel nimmt.
- Das Spiel des Ziel ist es Quartette zu sammeln. Ein Quartett ist ein vollständiger Satz aller vier Karten mit dem gleichen Wert. Der Spieler, der am Ende des Spiels die meisten die meisten Quartette hat, ist der Gewinner. Sobald ein Spieler ein Quartett zusammen hat, legt er es offen auf den Tisch, damit jeder sehen kann, wer welche Quartette hat.
- Deckt ein Spieler ein Quartett auf, kann das dazu führen, dass er keine Karten mehr hat. Passiert das, muss er fünf neue Karten aus dem Stock ziehen. Verbleiben im Stock weniger als sechs Karten, nimmt er alle. Das Spiel ist beendet, sobald der Stock leer ist. Der Sieger wird bestimmt, indem gezählt wird, wer die meisten Quartette hat.
- Bei dieser Computer-Version von Go Fish gibt es zwei Computerspieler und eine normalen Spieler. Jede Runde beginnt damit, dass der Spieler unter seinen angezeigten Handkarten ein aussucht. Das macht er, indem er eine der Karten auswählte und anzeigt, dass er danach fragen wird. Dann fragen die beiden Computerspieler nach ihren Karten. Das Ergebnis der einzelnen Runden wird angezeigt. Das wird wiederholt, bis es einen Sieger gibt.
- Das Spiel kümmert sich automatisch um den Austausch der Karten und das Ablegen der Quartette. Gibt es einen Sieger ist das Spiel vorüber. Dann zeigt das Spiel den Namen des Siegers an (oder der Sieger, falls es unentschieden ausgeht). Es kann nichts mehr gemacht werden – der Spieler muss das Programm neu starten, um ein neues Spiel zu beginnen.

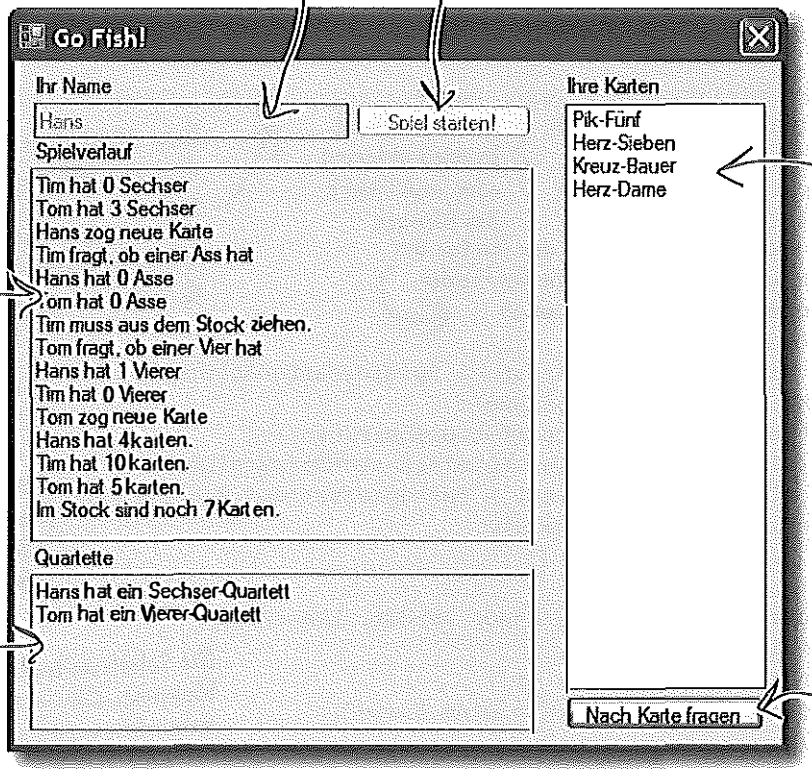
Wenn man nicht weiß, was man aufbaut, weiß man auch nicht, wie man anfangen soll. Deswegen beginnen richtige Software-Projekte meist mit einer Spezifikation, die Ihnen sagt, was Sie aufbauen müssen.

8 Erstellen Sie das Formular.

Erstellen Sie das Formular für das Spiel. Es sollte eine ListBox für die Karten des Spielers und zwei TextBox-Steuerelemente für den Fortschritt des Spiels haben sowie einen Button, über den der Spieler nach Karten fragen kann. Um einen Zug zu machen, wählt der Benutzer eine der Karten auf seiner Hand aus und klickt auf den Button, um die Computerspieler zu fragen, ob sie diese Karte haben.

Die Name-Eigenschaft des TextBox-Steuerelements sollte auf textName gesetzt sein. In diesem Screenshot ist es deaktiviert, sollte aber aktiviert werden, wenn das Programm gestartet wird.

Setzen Sie die Name-Eigenschaft dieses Buttons auf buttonStart. Er ist in diesem Screenshot deaktiviert, beginnt im Spiel aber aktiviert. Er wird deaktiviert, nachdem ein Spiel gestartet wurde.



Das sind die TextBox-Steuerelemente namens textSpielverlauf und textQuartette.

Die aktuellen Karten des Spielers werden in der ListBox namens listHand angezeigt. Diesen Namen können Sie über die Eigenschaft Name ändern.

Setzen Sie die ReadOnly-Eigenschaft der zwei TextBox-Steuerelemente auf True – das macht sie zu schreibgeschützten Textfeldern.

Setzen Sie die Name-Eigenschaft dieses Buttons auf buttonFragen und seine Enabled-Eigenschaft auf False. Das deaktiviert ihn, d.h., er kann nicht angeklickt werden. Das Formular aktiviert ihn, sobald das Spiel beginnt.

Wir sind noch nicht fertig - blättern Sie um!



Lange Übung (Fortsetzung)

Hier ist der Code für das Formular.
Geben Sie ihn genau so ein, wie Sie ihn hier sehen. Der Code, den Sie selbst schreiben müssen, muss mit diesem Code arbeiten.

```
public partial class Form1 : Form {
    public Form1() {
        InitializeComponent();
    }

    private Spiel spiel;

    private void buttonStart_Click(object sender, EventArgs e) {
        if (String.IsNullOrEmpty(textName.Text)){
            MessageBox.Show("Geben Sie Ihren Namen ein", "Kann Spiel noch nicht starten");
            return;
        }
        spiel = new Spiel(textName.Text, new string[] { "Tim", "Tom" }, textSpielverlauf);
        buttonStart.Enabled = false;
        textName.Enabled = false;
        buttonFragen.Enabled = true;
        FormularAktualisieren();
    }

    private void FormularAktualisieren() {
        listHand.Items.Clear();
        foreach (String kartenname in spiel.SpielerKartenAbrufen())
            listHand.Items.Add(kartenname);
        textQuartette.Text = spiel.QuartettBeschreiben();
        textSpielverlauf.Text += spiel.SpielerKartenBeschreiben();
        textSpielverlauf.SelectionStart = textSpielverlauf.Text.Length;
        textSpielverlauf.ScrollToCaret();
    }

    private void buttonFragen_Click(object sender, EventArgs e) {
        textSpielverlauf.Text = "";
        if (listHand.SelectedIndex < 0) {
            MessageBox.Show("Wählen Sie eine Karte");
            return;
        }
        if (spiel.RundeSpielen(listHand.SelectedIndex)) {
            textSpielverlauf.Text += "Der Sieger ist ... " + spiel.SiegerNameAbrufen();
            textQuartette.Text = spiel.QuartettBeschreiben();
            buttonFragen.Enabled = false;
        } else
            FormularAktualisieren();
    }
}
```

Das ist die einzige Klasse, mit der das Formular interagiert. Sie wickelt das gesamte Spiel ab.

Die Eigenschaft Enabled aktiviert oder deaktiviert das Element.

Beim Start eines neuen Spiels wird eine Instanz der Klasse Spiel erstellt, der Button >>Fragen<< aktiviert, der Button >>Spiel starten<< deaktiviert und das Formular neu gezeichnet.

Diese Methode löscht die ListBox mit den Karten des Spielers, füllt sie neu und aktualisiert die Textfelder.

SelectionStart und ScrollToCaret() scrollen ans Ende der TextBox. Muss mehr Text angezeigt werden, als in das Textfeld passt, wird also an ihr Ende gescrollt.

Die SelectionStart-Zeile schiebt den blinkenden Cursor ans Ende. Nachdem er verschoben wurde, scrollt ScrollToCaret() die Textbox zum Cursor.

Der Spieler wählt eine der Karten und klickt auf den Button >>Fragen<<, um zu schauen, ob die anderen Spieler eine Karte mit dem gleichen Wert haben. Eine Spielrunde wickelt die Klasse Spiel über die Methode RundeSpielen() ab.

4 Auch diesen Code brauchen Sie.

Sie benötigen den Code, den Sie zuvor für die Klassen `Karte`, `Kartenstapel` und `KarteComparer_NachWert` geschrieben haben. Der Klasse Kartenstapel müssen Sie allerdings noch einige Methoden hinzufügen ... und Sie müssen verstehen, wie diese funktionieren, damit Sie sie verwenden können.

Mit der Methode Gucken() können Sie eine der Karten einsehen, ohne sie herauszunehmen.

```
public Karte Gucken(int kartenNr) {
    return karten[kartenNr];
}
```

Eine Überladung von Ausgeben(), die den Code übersichtlicher macht. Wird kein Parameter übergeben, wird die oberste Karte vom Stapel genommen.

```
public Karte Ausgeben() {
    return Ausgeben(0);
}
```

Die Methode EnthältWert() sucht im gesamten Stapel nach Karten eines bestimmten Werts und liefert true, wenn welche gefunden werden. Haben Sie eine Idee, wie Sie diese im Go Fish!-Spiel einsetzen werden?

```
public bool EnthältWert(Karte.Werte wert) {
    foreach (Karte karte in karten)
        if (karte.Wert == wert)
            return true;
    return false;
}
```

Die Methode WerteEntnehmen() nutzen Sie, wenn Sie den Code erstellen, der ein Quartett aus dem Stapel entnimmt. Er sucht nach Karten mit einem bestimmten Wert, zieht sie aus dem Stapel heraus und liefert einen neuen Stapel mit diesen Karten.

```
public Kartenstapel WerteEntnehmen(Karte.Werte wert) {
    Kartenstapel rückgabeStapel = new Kartenstapel(new Karte[] { });
    for (int i = karten.Count - 1; i >= 0; i--)
        if (karten[i].Wert == wert)
            rückgabeStapel.Hinzufügen(Ausgeben(i));
    return rückgabeStapel;
}
```

Die Methode QuartettGesammelt() prüft einen Stapel darauf, ob er ein Quartett mit vier Karten des Werts enthält, der als Parameter übergeben wurde. Sie liefert true, wenn der Stapel ein Quartett enthält, andernfalls false.

```
public bool QuartettGesammelt(Karte.Werte wert) {
    int anzahlKarten = 0;
    foreach (Karte karte in karten)
        if (karte.Wert == wert)
            anzahlKarten++;
    if (anzahlKarten == 4)
        return true;
    else
        return false;
}
```

Die Methode NachWertSortieren() sortiert den Stapel mithilfe der Klasse KarteComparer_NachWert.

```
public void NachWertSortieren() {
    karten.Sort(new KarteComparer_NachWert());
}
```

→ Immer noch nicht fertig - blättern Sie um!



Lange Übung (Fortsetzung)

8 **Jetzt kommt der SCHWERE Teil: Erstellen Sie die Klasse Spieler.**
Für jeden der drei Spieler im Spiel gibt es eine Instanz der Klasse Spieler. Sie werden vom `buttonStart`-Event-Handler erzeugt.

Lesen Sie die Kommentare genau – sie sagen Ihnen, was diese Methoden machen sollen. Sie haben die Aufgabe, diese Methoden zu füllen.

```
public class Spieler
{
    private string name;
    public string Name { get { return name; } }
    private Random zufall;
    private Kartenstapel karten;
    private TextBox formularTextBox;

    public Spieler(String name, Random zufall, TextBox formularTextBox) {
       // Der Spieler-Konstruktor initialisiert vier private Felder und fügt dann in das
       // Textfeld des Formulars eine Zeile ein, die sagt: "Tim ist dem Spiel beigetreten.«
       // Verwenden Sie den Namen in dem privaten Feld und hängen Sie an jede Zeile,
       // die Sie in das Textfeld einfügen, einen Zeilenumbruch("\r\n") an.
    }

    public List<Karte.Werte> QuartetteEntnehmen() {  } // Siehe die folgende Seite.

    public Karte.Werte ZufallswertAbrufen() {
       // Diese Methode ruft einen zufälligen Wert ab, der sich im Stapel befinden muss.
    }

    public Kartenstapel HatEinerKarten(Karte.Werte wert) {
       // So fragt ein Gegner, ob der aktuelle Spieler Karten eines bestimmten Werts hat.
       // Diesen ermitteln Sie per Kartenstapel.WerteEntnehmen(). Fügen Sie dem Textfeld die
       // Zeile "Tim hat 3 Sechser" hinzu - nutzen Sie dazu Karte.Plural().
    }

    public void NachKarteFragen(List<Spieler> spieler, int index, Kartenstapel stapel) {
       // Eine überladene Version von NachKarteFragen() - sie wählt mit ZufallswertAbrufen()
       // einen Wert aus dem Stapel und fragt mit NachKarteFragen() danach.
    }

    public void NachKarteFragen(List<Spieler> spieler, int index,
                                Kartenstapel stapel, Karte.Werte wert) {
       // Fragt nach einem Wert. Fügt dem Textfeld eine Zeile hinzu: "Tim fragt, ob jemand
       // eine Dame hat." Durchläuft dann die übergebende Spielerliste und fragt die Spieler
       // mit HatEinerKarten(), ob sie eine Karte dieses Werts haben. Er übergibt einen Stapel
       // Karten, der zu den Handkarten hinzugefügt wird. Halten Sie fest, wie viele Karten
       // hinzugefügt wurden. Wurden keine geliefert, wird eine aus dem übergebenen Stock
       // entnommen und dem Textfeld die Zeile "Tim musste aus Stock ziehen" hinzugefügt".
    }
    // Hier sind Eigenschaften und Methoden, die bereits für Sie geschrieben wurden.
    public int Kartenanzahl { get { return karten.Anzahl; } }
    public void KarteNehmen(Karte karte) { karten.Hinzufügen(karte); }
    public string[] KartenNamenAbrufen() { return karten.KartenNamenAbrufen(); }
    public Karte Gucken(int kartenNr) { return karten.Gucken(kartenNr); }
    public void HandkartenSortieren() { karten.NachWertSortieren(); }
}
```

Die Gucken()-Methode, die wir der Klasse Kartenstapel hinzugefügt haben, wird sich als praktisch erweisen. Mit ihr kann das Programm über den Index nach einer der Karten im Stapel suchen. Im Unterschied zu Ausgeben() entfernt sie diese Karte aber nicht.

```
public List<Karte.Werte> QuartetteEntnehmen() {
    List<Karte.Werte> Quartette = new List<Karte.Werte>();
    for (int i = 1; i <= 13; i++) {
        Karte.Werte wert = (Karte.Werte)i;
        int wieViele = 0;
        for (int karte = 0; karte < karten.Anzahl; karte++)
            if (karten.Gucken(karte).Wert == wert)
                wieViele++;
        if (wieViele == 4) {
            Quartette.Add(wert);
            karten.WerteEntnehmen(wert);
        }
    }
    return Quartette;
}
```

Sie müssen ZWEI überladene Versionen von NachKarteFragen() erstellen. Die erste wird vom Gegner verwendet, wenn er nach Karten fragt – sie sucht auf seiner Hand nach einer Karte, nach der er fragen kann. Die zweite wird verwendet, wenn der Spieler nach der Karte fragt. Beide fragen ALLE anderen Spieler (beide Computerspieler und den menschlichen Spieler) nach Karten mit passendem Wert.

6 **Der Klasse Karte müssen Sie diese Methode hinzufügen.**

Es ist eine statische Methode, die einen Wert nimmt und ihren Plural liefert – für die Zehn »Zehner« und für Ausnahmen wie Ass »Asse« (mit »e« am Ende). Da die Methode statisch ist, rufen Sie sie über die Klasse – `Karte.Plural()` – und nicht über eine Instanz.

```
public partial class Karte {
    public static string Plural(Karte.Werte wert) {
        if (wert == Werte.Ass || wert == Werte.König)
            return wert.ToString() + "e";
        else if (wert == Werte.Dame || wert == Werte.Bauer)
            return wert.ToString() + "n";
        else
            return wert.ToString() + "er";
    }
}
```

Fast fertig - noch einmal blättern!



Der Rest der Aufgabe: Erstellen Sie die Klasse Spiel.
Das Formular hält eine Instanz von Spiel. Diese verwaltet den Spielablauf. Achten Sie genau darauf, wie sie im Formular verwendet wird

```
public class Spiel {
    private List<Spieler> spieler;
    private Dictionary<Karte.Werte, Spieler> quartette;
    private Kartenstapel stapel;
    private TextBox formularTextBox;

    public Spiel(string spielerName, string[] gegnerNamen, TextBox formularTextBox) {
        Random zufall = new Random();
        this.formularTextBox = formularTextBox;
        spieler = new List<Spieler>();
        spieler.Add(new Spieler(spielerName, zufall, formularTextBox));
        foreach (string einSpieler in gegnerNamen)
            spieler.Add(new Spieler(einSpieler, zufall, formularTextBox));
        quartette = new Dictionary<Karte.Werte, Spieler>();
        stapel = new Kartenstapel();
        Ausgeben();
        spieler[0].HandkartenSortieren();
    }

    private void Ausgeben() {
       // Hier beginnt das Spiel - diese Methode wird nur zu Anfang des Spiels aufgerufen.
       // Der Stapel wird gemischt, und an jeden Spieler werden fünf Karten ausgegeben.
       // Rufen Sie dann die QuartetteEntnehmen()-Methode aller Spieler auf.
    }

    public bool RundeSpielen(int gewählteKarte) {
       // Spielt eine Runde. Der Parameter ist die vom Spieler gewählte Karte - fragen Sie
       // ihren Wert ab. Gehen Sie alle Spieler durch und rufen Sie ihre NachKarteFragen()-
       // Methode auf - achten Sie darauf, dass nach dem Wert der gewählten Karte gefragt
       // wird. Rufen Sie dann QuartetteEntnehmen() auf - liefert die Methode true, hat
       // der Spieler keine Karten mehr und muss neue ziehen. Sortieren Sie anschließend
       // die Karten des Spielers (damit das im Formular gut aussieht). Prüfen Sie dann,
       // ob der Stock noch Karten enthält. Ist er leer, setzen Sie die TextBox im
       // Formular auf "Der Stock ist leer. Das Spiel ist beendet!" und liefern true.
       // Andernfalls ist das Spiel noch nicht vorüber. Liefern Sie dann false.
    }

    public bool QuartetteEntnehmen(Spieler einSpieler) {
       // Entnimmt die Quartette der Spieler. Liefert true, wenn der Spieler dann keine
       // Karte mehr hat, andernfalls false. Die Quartette werden dem Wörterbuch Quartette
       // hinzugefügt. Hat ein Spieler all seine Karten für Quartette genutzt, gewinnt er.
    }

    public string QuartettBeschreiben() {
       // Liefert eine Beschreibung der Quartette im Wörterbuch Quartette: "Tim hat
       // ein Quartett mit Sechsern. (Zeilenumbruch) Max hat ein Quartett mit Assen."
    }
```

```
public string SiegerNameAbrufen() {
       // Diese Methode wird zu Spielende aufgerufen. Sie nutzt ein eigenes Spieler-
       // Wörterbuch (Dictionary<string, int> sieger), in dem sie die Anzahl an
       // Quartetten der Spieler im Wörterbuch quartette speichert. Sie nutzt eine
       // Schleife auf quartette.Keys -- foreach (Karte.Werte wert in quartette.Keys) --,
       // um das Wörterbuch sieger mit der entsprechenden Zahl an Quartetten zu füllen.
       // Dann durchläuft sie das Wörterbuch, um den Spieler mit den meisten Quartetten
       // zu finden. Danach geht sie es ein letztes Mal durch, um einen String mit einer
       // Siegerliste aufzubauen ("Tim und Max"). Gibt es einen Sieger, hat der die Form
       // "Max mit 3 Quartetten", andernfalls lautet er "Unentschieden zwischen Tim und
       // Tom mit 2 Quartetten".
}

// Hier sind ein paar kurze Methoden, die wir bereits für Sie geschrieben haben:

public string[] SpielerKartenAbrufen() {
    return spieler[0].KartenNamenAbrufen();
}

public string SpielerKartenBeschreiben() {
    string beschreibung = "";
    for (int i = 0; i < spieler.Count; i++) {
        beschreibung += spieler[i].Name + " hat " + spieler[i].Kartenanzahl;
        if (spieler[i].Kartenanzahl == 1)
            beschreibung += " karte.\r\n";
        else
            beschreibung += " karten.\r\n";
    }
    beschreibung += "Im Stock sind noch " + stapel.Anzahl+ " Karten.";
    return beschreibung;
}
```



Lange Übung Lösung

Hier sind die zusätzlichen Methoden für die Klasse Spiel.

```
public class Spiel {
    private void Ausgeben() {
        stapel.Mischen();
        for (int i = 0; i < 5; i++)
            foreach (Spieler einSpieler in spieler)
                einSpieler.KarteNehmen(stapel.Ausgeben());
        foreach (Spieler einSpieler in spieler)
            QuartetteEntnehmen(einSpieler);
    }

    public bool RundeSpielen(int gewählteKarte) {
        Karte.Werte frageKarte = spieler[0].Gucken(gewählteKarte).Wert;
        for (int i = 0; i < spieler.Count; i++) {
            if (i == 0)
                spieler[0].NachKarteFragen(spieler, 0, stapel, frageKarte);
            else
                spieler[i].NachKarteFragen(spieler, i, stapel);
            if (QuartetteEntnehmen(spieler[i])) {
                formularTextBox.Text += spieler[i].Name + " zog neue Karte\r\n";
                int karte = 1;
                while (karte <= 5 && stapel.Anzahl > 0) {
                    spieler[i].KarteNehmen(stapel.Ausgeben());
                    karte++;
                }
            }
            spieler[0].HandkartenSortieren();
            if (stapel.Anzahl == 0) {
                formularTextBox.Text = "Der Stock ist leer. Das Spiel ist beendet!\r\n";
                return true;
            }
        }
        return false;
    }

    public bool QuartetteEntnehmen(Spieler einSpieler)
    {
        List<Karte.Werte> entnommeQuartette = einSpieler.QuartetteEntnehmen();
        foreach (Karte.Werte wert in entnommeQuartette)
            quartette.Add(wert, einSpieler);
        if (einSpieler.Kartenanzahl == 0)
            return true;
        return false;
    }
```

Ausgeben() wird aufgerufen, wenn das Spiel beginnt – der Stapel wird gemischt, und dann erhält jeder Spieler fünf Karten. Anschließend werden alle Quartette entnommen, die einer der Spieler erhalten hat.

Hat der Spieler oder einer der Gegner nach einer Karte gefragt, entnimmt das Spiel alle erzielten Quartette. Hat ein Spieler keine Karten mehr, zieht er eine neue Hand, indem er bis zu fünf Karten aus dem Stock nimmt.

Sobald ein Spieler auf den Button »Nach Karte fragen« klickt, ruft das Spiel NachKarteFragen() mit dieser Karte auf. Dann ruft es NachKarteFragen() für jeden Gegner auf.

Nachdem die Runde gespielt wurde, sortiert das Spiel die Hand des Spielers, damit die Karten im Formular geordnet angezeigt werden. Dann prüft es, ob das Spiel vorüber ist. Ist das der Fall, liefert RundeSpielen() true.

QuartetteEntnehmen() prüft in den Karten des Spielers, ob er vier Karten mit dem gleichen Wert hat. Wenn ja, wird das Quartett dem Wörterbuch quartette hinzugefügt. Hat er anschließend keine Karten mehr, liefert die Methode true.

Das Formular muss eine Liste der Quartette anzeigen und nutzt QuartetteBeschreiben(), um das quartette-Wörterbuch des Spielers in Wörter zu überführen.

```
    public string QuartettBeschreiben() {
        string spielerQuartette = "";
        foreach (Karte.Werte wert in quartette.Keys)
            spielerQuartette += quartette[wert].Name + " hat ein "
                + Karte.Plural(wert) + "-Quartett\r\n";
        return spielerQuartette;
    }

    public string SiegerNameAbrufen() {
        Dictionary<string, int> sieger = new Dictionary<string, int>();
        foreach (Karte.Werte wert in quartette.Keys) {
            string name = quartette[wert].Name;
            if (sieger.ContainsKey(name))
                sieger[name]++;
            else
                sieger.Add(name, 1);
        }
        int höchsteQuartettAnzahl = 0;
        foreach (string name in sieger.Keys)
            if (sieger[name] > höchsteQuartettAnzahl)
                höchsteQuartettAnzahl = sieger[name];
        bool remis = false;
        string siegerListe = "";
        foreach (string name in sieger.Keys)
            if (sieger[name] == höchsteQuartettAnzahl)
            {
                if (!String.IsNullOrEmpty(siegerListe))
                {
                    siegerListe += " und ";
                    remis = true;
                }
                siegerListe += name;
            }
        siegerListe += " mit " + höchsteQuartettAnzahl
                               + " quartette";
        if (remis)
            return "Unentschieden zwischen " + siegerListe;
        else
            return siegerListe;
    }
}
```

Wurde die letzte Karte aufgenommen, wird der Sieger ermittelt. Das macht SiegerNameAbrufen(). Dazu wird ein Wörterbuch namens sieger eingesetzt. Die Namen der Spieler sind die Schlüssel, die Werte die jeweilige Anzahl an Quartetten.

Dann wird über das Wörterbuch ermittelt, welche die höchste Anzahl an Quartetten ist, die ein Spieler erzielt hat. Dieser Wert kommt in die Variable höchsteQuartettAnzahl.

Nachdem wir wissen, welche Spieler die meisten Quartette haben, kann die Methode einen String mit der Siegerliste aufbauen.

→ Wir sind noch nicht fertig – blättern Sie um!



Lange Übung Lösung (Fortsetzung)

Hier sind die zusätzlichen Methoden für die Klasse Spieler.

```
public Spieler(String name, Random zufall, TextBox formularTextBox) {
    this.name = name;
    this.zufall = zufall;
    this.formularTextBox = formularTextBox;
    this.karten = new Kartenstapel( new Karte[] {} );
    formularTextBox.Text += name + " ist dem Spiel beigetreten\r\n";
}

public Karte.Werte ZufallswertAbrufen() {
    Karte  zufallskarte = karten.Gucken(zufall.Next(karten.Anzahl));
    return  zufallskarte.Wert;
}

public Kartenstapel HatEinerKarten(Karte.Werte wert) {
    Kartenstapel handkarten = karten.WerteEntnehmen(wert);
    formularTextBox.Text += Name + " hat " + handkarten.Anzahl + " "
           + Karte.Plural(wert) + "\r\n";
    return handkarten;
}

public void NachKarteFragen(List<Spieler> spieler, int index, Kartenstapel stapel) {
    Karte.Werte zufallswert = ZufallswertAbrufen();
    NachKarteFragen(spieler, index, stapel, zufallswert);
}

public void NachKarteFragen(List<Spieler> spieler, int index,
      Kartenstapel stapel, Karte.Werte wert) {
    formularTextBox.Text += Name + " fragt, ob einer " + wert + " hat\r\n";
    int anzahlGegebeneKarten = 0;
    for (int i = 0; i < spieler.Count; i++) {
        if (i != index) {
            Spieler einSpieler = spieler[i];
            Kartenstapel gegebeneKarten = einSpieler.HatEinerKarten(wert);
            anzahlGegebeneKarten += gegebeneKarten.Anzahl;
            while (gegebeneKarten.Anzahl > 0)
                karten.Hinzufügen(gegebeneKarten.Ausgeben());
        }
    }
    if (anzahlGegebeneKarten == 0) {
        formularTextBox.Text += Name + " muss aus dem Stock ziehen.\r\n";
        karten.Hinzufügen(stapel.Ausgeben());
    }
}
```

Dies ist der Konstruktor für die Klasse Spieler. Er setzt die privaten Felder und fügt dem Fortschrittstextfeld eine Zeile hinzu, die sagt, wer der neue Mitspieler ist.

Die Methode ZufallswertAbrufen() nutzt Gucken(), um zufällig eine Karte auf der Hand eines Spielers zu suchen.

HatEinerKarten() nutzt WerteEntnehmen(), um alle Karten mit diesem Wert zu entnehmen und zurückzuliefern.

Es gibt zwei Überladungen von NachKarteFragen(). Die erste verwenden die Gegner – sie wählt zufällig eine Karte aus und ruft die andere NachKarteFragen()-Überladung auf.

Diese NachKarteFragen()-Überladung durchsucht die Karten der Spieler (den fragenden ausgenommen), ruft HatEinerKarten() auf und fügt übergebene Karten den eigenen Karten hinzu.

Wurden keine Karten übergeben, muss der Spieler über die Methode Ausgeben() eine Karte aus dem Stock ziehen.

Noch MEHR Auflistungstypen ...

`List` und `Dictionary` sind zwei der **eingebauten generischen Auflistungsklassen**, die Teil des .NET Framework sind. Listen und Wörterbücher sind sehr flexibel – auf die Daten darin können Sie in beliebiger Reihenfolge zugreifen. Aber gelegentlich müssen Sie einschränken, wie Ihr Programm mit Daten arbeitet, weil das *Ding*, das diese in Ihrem Programm repräsentieren, wie im wahren Leben funktioniert. In solchen Situationen nutzen Sie eine **`Queue`** oder einen **`Stack`**. Das sind zwei weitere generische Auflistungen, die Listen ähneln, aber besonders gut sind, wenn man sicherstellen will, in welcher Reihenfolge Daten verarbeitet werden.

Es gibt noch weitere Arten von Auflistungen, aber das sind die, mit denen Sie am ehesten in Berührung kommen werden.

Eine Queue nutzen Sie, wenn das Objekt, das Sie als Erstes speichern, auch das erste ist, das Sie verwenden werden:

- ★ Wagen, die eine Einbahnstraße hinunterfahren
- ★ Menschen, die in einer Schlange stehen
- ★ Kunden in der Warteschlange eines Kundendiensts
- ★ Alles, was auf der Basis »Wer zuerst kommt, kriegt zuerst« verarbeitet wird

Eine Queue (Schlange) ist eine First-in-First-out-Struktur (Als-Erstes-drin-Als-Erstes-raus). Das bedeutet, dass das erste Objekt, das Sie in die Queue stecken, auch das erste ist, das Sie zur Verwendung herausziehen.

Einen Stack nutzen Sie, wenn Sie immer das Objekt verwenden möchten, das als Letztes gespeichert wurde:

- ★ Möbel in einem Umzugswagen
- ★ Ein Stapel Quartette, aus dem Sie das zuletzt hinzugefügte als erstes herausziehen möchten
- ★ Eine Cheerleader-Pyramide, von der die obersten zuerst absteigen müssen ... stellen Sie sich das Chaos vor, wenn die unten zuerst gingen!

Ein Stack (Stapel) ist eine First-in-Last-out-Struktur (Als-Erstes-drin-Als-Letztes-raus). Das zuerst in einen Stapel gesteckte Objekt ist das, was als letztes wieder aus ihm herauskommt.

Generische Auflistungen sind ein wichtiger Teil des .NET Framework

Sie sind wirklich sehr nützlich – so nützlich, dass die IDE jeder Klasse in Ihrem Projekt automatisch folgende Zeile voranstellt:

```
using System.Collections.Generic;
```

Fast jedes größere Projekt, an dem Sie arbeiten, schließt irgendwelche generische Auflistungen ein, weil Programme ohne gespeicherte Daten nicht sein können. Und wenn Sie es mit Gruppen ähnlicher Dinge aus dem wirklichen Leben zu tun haben, fallen diese beinahe natürlich in eine Kategorie, die recht gut einer dieser Art von Auflistungen entspricht.

Sie können allerdings `foreach` verwenden, um einen Stack oder eine Queue zu durchlaufen, weil beide IEnumerable implementieren!

Eine Queue ist wie eine Liste, bei der Sie am einen Ende Dinge hineinstecken und am anderen wieder hinausnehmen. Aus einem Stack nehmen Sie die Dinge auf der Seite heraus, auf der Sie diese auch hineingesteckt haben.

Eine Queue ist FIFO - First-in-First-out

Eine **Queue** ist einer Liste recht ähnlich. Allerdings können Sie Elemente nicht einfach an einem beliebigen Index hinzufügen oder dort entnehmen. Einer Queue fügen Sie ein Objekt hinzu, indem Sie es mit **Enqueue()** einreihen. Das hängt das Objekt ans Ende der Schlange an. Mit **Dequeue()** nehmen Sie das erste Element am Anfang der Liste heraus. Wenn Sie das tun, wird das Objekt aus der Schlange entfernt, und die restlichen Objekte rücken in ihr eine Position auf.

Eine neue Queue mit Strings erstellen.

```
Queue<string> dieQueue = new Queue<string>();
dieQueue.Enqueue("Erster");
dieQueue.Enqueue("Zweiter");
dieQueue.Enqueue("Dritter");
dieQueue.Enqueue("Letzter");
```

Hier werden der Queue vier Elemente hinzugefügt. Ziehen wir sie aus der Queue heraus, erhalten wir sie genau in dieser Reihenfolge.

Mit Peek() können Sie sich das erste Element in der Queue ansehen, ohne es zu entnehmen.

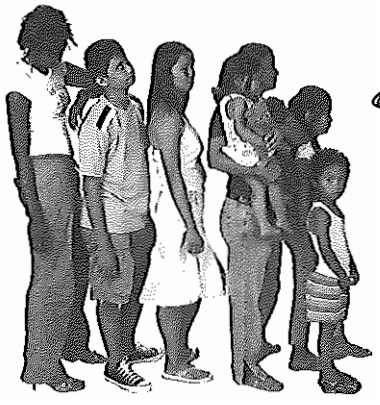
```
string einblicken = dieQueue.Peek(); ①
string erster = dieQueue.Dequeue(); ②
string nächster = dieQueue.Dequeue(); ②
int wieViele = dieQueue.Count; ④
dieQueue.Clear();
```

Das erste Dequeue() zieht das erste Element aus der Queue. Das zweite Element rückt an die erste Stelle und wird vom nächsten Dequeue() abgerufen.

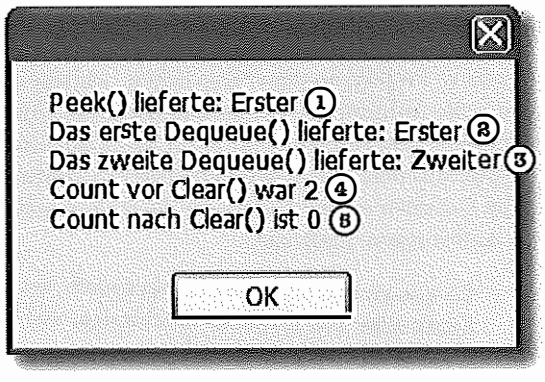
Clear() entfernt alle Objekte aus der Queue.

```
MessageBox.Show("Peek() lieferte: " + einblicken + "\n"
    + "Das erste Dequeue() lieferte: " + erster + "\n"
    + "Das zweite Dequeue() lieferte: " + nächster + "\n"
    + "Count vor Clear() war " + wieViele + "\n"
    + "Count nach Clear() ist " + dieQueue.Count); ③
```

Die Count-Eigenschaft der Queue liefert die Anzahl an Elementen in der Queue.



Objekte in einer Queue müssen warten, bis sie an der Reihe sind. Der Erste in der Schlange kommt auch als Erster raus.



Ein Stack ist LIFO – Last-in-First-out

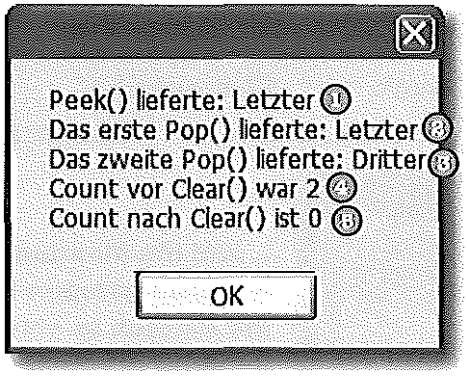
Ein **Stack** ist einer Queue sehr ähnlich – es gibt aber einen großen Unterschied. Sie schieben Elemente mit **Push()** auf einen Stack, und wenn Sie eins davon herunternehmen möchten, ziehen Sie es mit **Pop()** herunter. Ziehen Sie ein Element von einem Stapel herunter, erhalten Sie das Element, das Sie als Letztes auf ihn geschoben haben. Das ist vergleichbar mit einem Stapel mit Tellern, Zeitschriften oder irgendetwas anderem – Sie legen etwas auf den Stapel und müssen es wieder herunternehmen, damit Sie an die Dinge darunter herankommen.

Einen Stack erzeugen Sie genau wie alle anderen generischen Auflistungen.

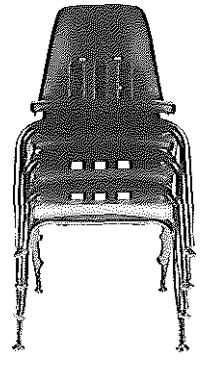
Schieben Sie ein Element auf einen Stack, werden die anderen Elemente nach unten geschoben.

```
Stack<string> derStack = new Stack<string>();
derStack.Push("Erster");
derStack.Push("Zweiter");
derStack.Push("Dritter");
derStack.Push("Letzter");
string einblicken = derStack.Peek();
string erster = derStack.Pop();
string nächster = derStack.Pop();
int wieViele = derStack.Count;
derStack.Clear();
MessageBox.Show("Peek() lieferte: " + einblicken + "\n"
    + "Das erste Pop() lieferte: " + erster + "\n"
    + "Das zweite Pop() lieferte: " + nächster + "\n"
    + "Count vor Clear() war " + wieViele + "\n"
    + "Count nach Clear() ist " + derStack.Count);
```

Ziehen Sie ein Element vom Stack herunter, erhalten Sie das Element, das Sie als Letztes daraufgepackt haben.



Das letzte Objekt, das Sie auf den Stapel geschoben haben, ist das erste, das Sie wieder herunterziehen.





Moment. Ich bin noch nicht zufrieden. Sie haben mir nichts davon erzählt, was ich mit einem Stack oder einer Queue machen kann, mit einer List dagegen nicht - die sparen mir doch bloß ein paar Zeilen Code. Und dann komme ich nicht einmal an die Elemente in der Mitte eines Stacks oder einer Queue heran. Bei einer Liste ist das kein Problem! Warum sollte ich das für ein paar Annehmlichkeiten aufgeben?

Sie geben nichts auf, wenn Sie einen Stack oder eine Queue verwenden.

Ein Queue-Objekt kann man problemlos in ein List-Objekt umwandeln. Und genauso leicht macht man aus einer List eine Queue, einer Queue einen Stack ... in der Tat können Sie eine List, eine Queue oder einen Stack aus jedem Objekt erstellen, das die Schnittstelle **`IEnumerable`** implementiert. Dazu müssen Sie nur einen überladenen Konstruktor verwenden, über den Sie als Parameter ein Auflistungsobjekt übergeben können, das Sie kopieren möchten. Das heißt, dass Sie die Flexibilität und Annehmlichkeit haben, Ihre Daten mit dem Auflistungstyp darzustellen, der für Ihre Verwendungszwecke am angemessensten ist.

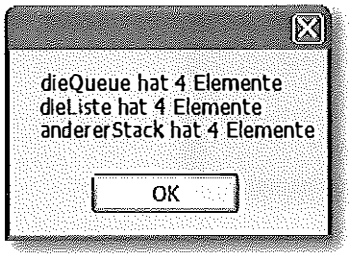
Richten wir einen Stack mit vier Elementen ein – hier ist das ein Stack mit Strings.

```
Stack<string> derStack = new Stack<string>();
derStack.Push("Erster");
derStack.Push("Zweiter");
derStack.Push("Dritter");
derStack.Push("Letzter");

Queue<string> dieQueue = new Queue<string>(derStack);
List<string> dieListe = new List<string>(dieQueue);
Stack<string> andererStack = new Stack<string>(dieListe);
MessageBox.Show("dieQueue hat " + dieQueue.Count + " Elemente\n"
    + "dieListe hat " + dieListe.Count + " Elemente\n"
    + "andererStack hat " + andererStack.Count + " Elemente\n");
```

Einen Stack kann man leicht in eine Queue verwandeln, aus der man leicht eine List machen kann, die man leicht in einen weiteren Stack verwandeln kann.

Alle vier Elemente wurden in die neuen Auflistungen kopiert.



... und die Member aller Auflistungstypen können Sie einfach in einer foreach-Schleife durchlaufen.

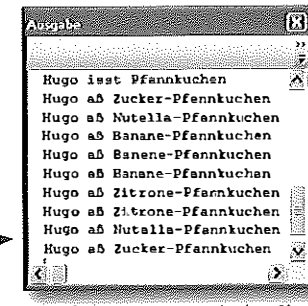


Schreiben Sie ein Programm, das in einem Restaurant Holzfällern zu Pfannkuchen verhilft. Beginnen Sie mit der Klasse Holzfäller. Entwerfen Sie dann das Formular und fügen Sie den Buttons Event-Handler hinzu.

```
public enum
{       Pfannkuchen
    Zucker,
    Nutella,
    Zitrone,
    Banane
}
```

1. Hier ist die Klasse Holzfäller. Ergänzen Sie den `PfannkuchenZahl`-Getter und die Methoden `PfannkuchenNehmen` und `PfannkuchenEssen`.

```
public class Holzfäller {
    private string name;
    public string Name { get { return name; } }
    private Stack<Pfannkuchen> mahlzeit;
    public Holzfäller(string name) {
        this.name = name;
        mahlzeit = new Stack<Pfannkuchen>();
    }
    public int PfannkuchenAnzahl { get { // Anzahl liefern } }
    public void PfannkuchenNehmen(Pfannkuchen gang, int anzahl) {
       // Dem Stapel Pfannkuchen ein paar hinzufügen.
    }
    public void PfannkuchenEssen() {
       // Diese Ausgabe in die Konsole schreiben.
    }
}
```



2. Erstellen Sie dieses Formular. Darin geben Sie in einem Textfeld die Namen der Holzfäller an, um sie in die Frühstücksschlange zu stellen. Mit dem Button »Nächster Holzfäller« geben Sie dem ersten Holzfäller einen Teller Pfannkuchen und sagen ihm, dass er diese essen soll. Den Event-Handler für diesen Button sehen Sie unten. Halten Sie die Holzfäller in einer Queue namens **essensschlange** fest.



Klickt der Benutzer auf »Holzfäller hinzufügen«, wird der Name im Textfeld name in die Queue essensschlange übernommen.

Ziehen Sie diese RadioButtons in die GroupBox, verknüpft die IDE sie automatisch. Der Benutzer darf dann immer nur einen von ihnen auswählen. In der Methode pfannkuchenHinzufügen können Sie sehen, welche Namen sie haben sollten.

Dieser Button sollte den ersten Holzfäller aus der Schlange nehmen, seine PfannkuchenEssen()-Methode aufrufen und dann die ListBox neu zeichnen.

Die hinzuzufügende ListeNeuZeichnen()-Methode muss die ListBox mit dem Inhalt der Queue neu zeichnen. Alle Buttons müssen sie aufrufen.

```
private void pfannkuchenHinzufügen_Click(...) {
    Pfannkuchen essen;
    if (zucker.Checked == true)
        essen = Pfannkuchen.Zucker;
    else if (nutella.Checked == true)
        essen = Pfannkuchen.Nutella;
    else if (zitrone.Checked == true)
        essen = Pfannkuchen.Zitrone;
    else
        essen = Pfannkuchen.Banane;

    Holzfäller aktuellerHolzfäller = essensschlange.Peek();
    aktuellerHolzfäller.PfannkuchenNehmen(essen,
                        (int)wieViele.Value);
    ListeNeuZeichnen();
}
```

Beachten Sie die »else if«-Syntax.

Peek() liefert eine Referenz auf den ersten Holzfäller in der Schlange.

Das NumericUpDown heißt wieViele und das Label nächsterKunde.



LÖSUNGEN ZU DEN ÜBUNGEN

```
private Queue<Holzfäller> essensschlange = new Queue<Holzfäller>();
private void holzfällerHinzufügen_Click(object sender, EventArgs e) {
    essensschlange.Enqueue(new Holzfäller(name.Text));
    name.Text = "";
    ListeNeuZeichnen();
}
private void ListeNeuZeichnen() {
    int anzahl = 1;
    schlange.Items.Clear();
    foreach (Holzfäller holzfäller in essensschlange) {
        schlange.Items.Add(anzahl + ". " + holzfäller.Name);
        anzahl++;
    }
    if (essensschlange.Count == 0) {
        groupBox1.Enabled = false;
        nächsterKunde.Text = "";
    } else {
        groupBox1.Enabled = true;
        Holzfäller aktuellerHolzfäller = essensschlange.Peek();
        nächsterKunde.Text = aktuellerHolzfäller.Name + " hat "
            + aktuellerHolzfäller.PfannkuchenAnzahl + " Pfannkuchen";
    }
}
private void nächsterHolzfäller_Click(object sender, EventArgs e) {
    Holzfäller nächsterInSchlange = essensschlange.Dequeue();
    nächsterInSchlange.PfannkuchenEssen();
    nächsterKunde.Text = "";
    ListeNeuZeichnen();
}

public class Holzfäller {
    private string name;
    public string Name { get { return name; } }
    private Stack<Pfannkuchen> mahlzeit;

    public Holzfäller(string name) {
        this.name = name;
        mahlzeit = new Stack<Pfannkuchen>();
    }

    public int PfannkuchenAnzahl { get { return mahlzeit.Count; } }

    public void PfannkuchenNehmen(Pfannkuchen essen, int wieViele) {
        for (int i = 0; i < wieViele; i++) {
            mahlzeit.Push(essen);
        }
    }

    public void PfannkuchenEssen() {
        Console.WriteLine(name + " ist Pfannkuchen");
        while (mahlzeit.Count > 0) {
            Console.WriteLine(name + " aß "
                + mahlzeit.Pop().ToString() + "-Pfannkuchen");
        }
    }
}
```

Die ListBox heißt »Schlange«, das Label zwischen den Buttons »nächsterKunde«.

Die Methode ListeNeuZeichnen() nutzt eine foreach-Schleife, um die Holzfäller aus der Queue zu ziehen und der ListBox hinzuzufügen.

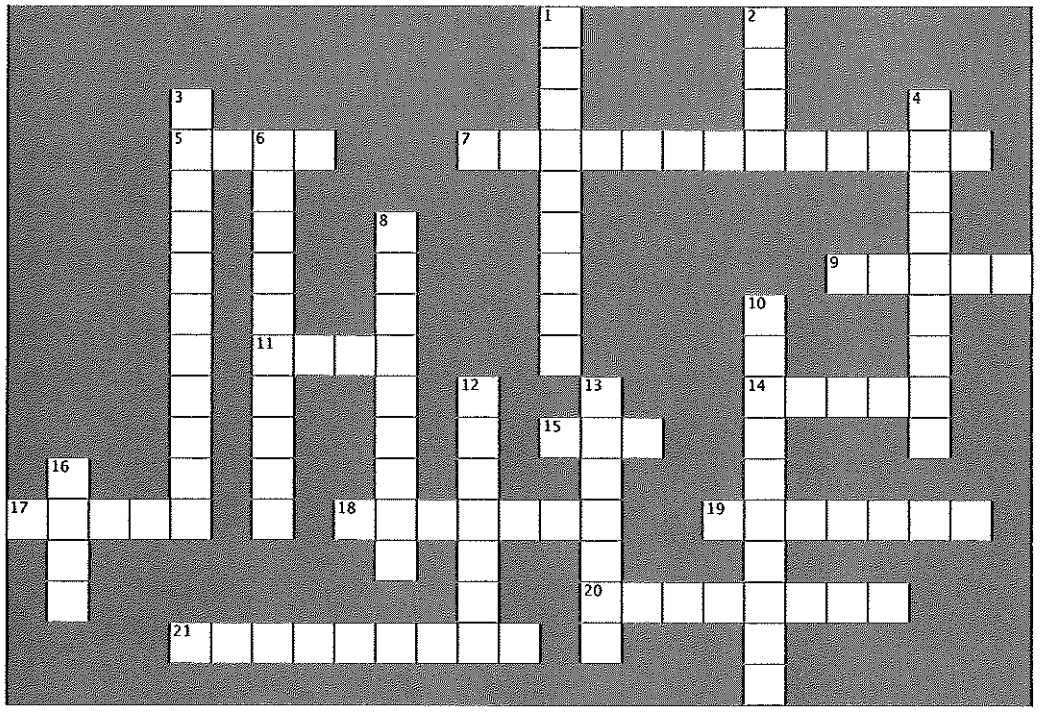
Diese if-Anweisung aktualisiert das Label mit den Informationen zum ersten Holzfäller in der Schlange.

Die Methode PfannkuchenNehmen aktualisiert den Stack mahlzeit.

Die PfannkuchenEssen-Methode nutzt eine Schleife, um die Mahlzeit des Holzfällers auszugeben.



Auflistungskreuzworträtsel



Waagerecht

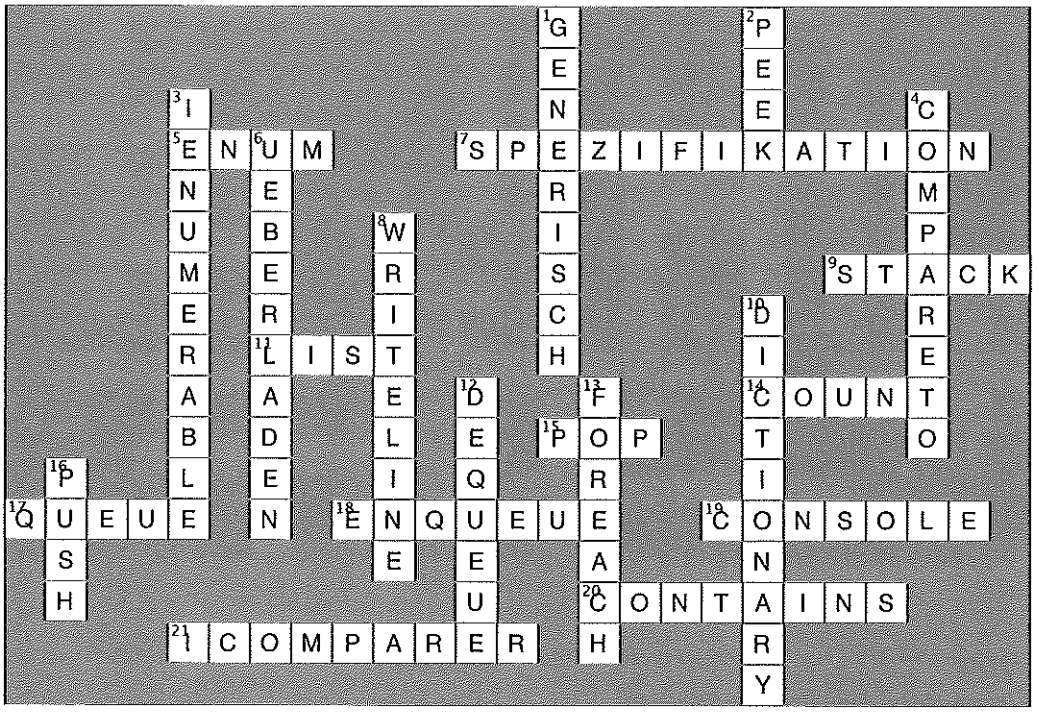
5. Ein einfaches Mittel, Kategorien festzuhalten.
7. Damit beginnen die meisten professionellen Projekte.
9. Bei dieser Auflistung gilt, wer zuerst kommt, geht als Letzter.
11. Ein Objekt, das wie ein Array ist, nur flexibler.
14. Eine Methode, mit der Sie ermitteln, wie viele Dinge sich in einer Auflistung befinden.
15. So entfernen Sie etwas von einem Stack.
17. Bei dieser Auflistung heißt es, wer zuerst kommt, geht zuerst.
18. Damit fügen Sie einer Queue etwas hinzu.
19. Über diese eingebaute Klasse kann Ihr Programm Text in die Ausgabe schreiben.
20. Eine Methode, mit der Sie ermitteln können, ob sich ein bestimmtes Objekt in einer Auflistung befindet.
21. Eine Klasse, die diese Schnittstelle implementiert, hilft Ihrer Liste, ihren Inhalt zu sortieren.

Senkrecht

1. Eine Instanz einer _____en Auflistung funktioniert nur mit einem bestimmten Typ.
2. Diese Methode liefert das nächste Objekt, das aus einem Stack oder einer Queue geliefert wird.
3. Alle generischen Auflistungen implementieren diese Schnittstelle.
4. Die einzige Methode in der Schnittstelle IComparable.
6. Zwei Methoden in einer Klasse mit dem gleichen Namen, aber anderen Parametern sind _____.
8. Der Name der Methode, die Sie verwenden, um einen String an die Ausgabe zu senden.
10. Mit dieser generischen Auflistung können Sie Schlüsseln Werte zuordnen.
12. Damit entfernen Sie etwas aus einer Queue.
13. Eine besondere Art von Schleife, die nur mit Auflistungen funktioniert.
16. Damit fügen Sie einem Stack etwas hinzu.



Auflistungskreuzworträtsel, Lösung



Name: Datum:

C#-Workshop

Die Suche

Dieser Workshop gibt Ihnen eine Spezifikation für ein Programm, das Sie erstellen sollen, um die Fertigkeiten zum Einsatz zu bringen, die Sie in den letzten Kapiteln erworben haben.

Dieses Projekt ist umfangreicher als alle, die Ihnen bisher begegnet sind. Lesen Sie sich also alles erst einmal durch, bevor Sie beginnen, und gönnen Sie sich etwas Zeit. Es macht nichts, wenn Sie hängen bleiben – es passiert hier nichts Neues. Sie können also im Buch weiterlesen und später zu diesem Workshop zurückkehren.

Ein paar Entwurfsdetails haben wir bereits für Sie eingesetzt. Und wir haben sichergestellt, dass Sie alle Teile haben, die Sie benötigen ... und sonst nichts.

Ob Sie die Arbeit zu Ende führen, liegt bei Ihnen.

Die Grafiken für diesen Workshop können Sie unter *http://www.headfirstlabs.com/books/hfcsharp/#code* herunterladen. Lösungsvorschläge finden Sie dort unter der Überschrift »Check out the labs«.

Die Spezifikation: Erstellen Sie ein Adventure-Spiel

Sie haben die Aufgabe, ein Adventure zu erstellen, in dem ein mächtiger Abenteurer von Level zu Level die tödlichsten Feinde besiegt. Sie werden ein **rundenbasiertes Spiel** erstellen. Das bedeutet, dass erst der Spieler und dann die Gegner ihren Zug machen. Der Spieler kann sich bewegen ***oder*** angreifen, und dann erhält jeder Gegner die Möglichkeit, sich zu bewegen ***und*** anzugreifen. Das Spiel geht so lange, bis der Spieler auf allen sieben Leveln seine Gegner besiegt hat oder stirbt.

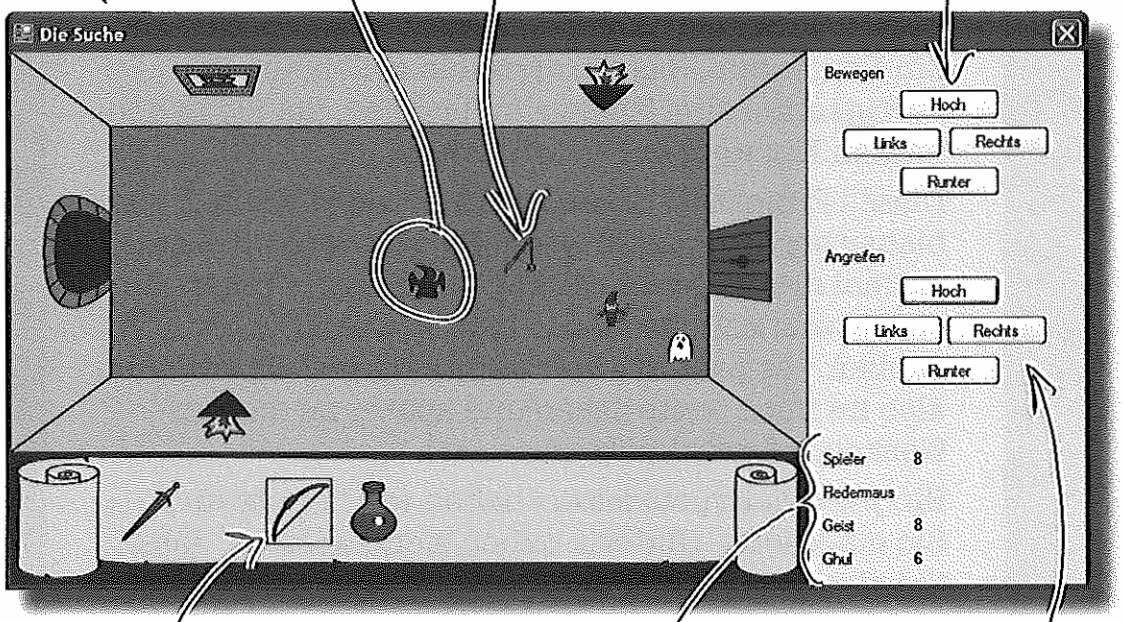
Die Gegner haben einen kleinen Vorteil – sie können in jeder Runde gehen, und nach der Bewegung greifen sie den Spieler an, wenn er in Reichweite ist.

Das Spielfenster präsentiert Ihnen aus der Vogelperspektive das Verlies, in dem der Spieler seine Gegner bekämpft.

Der Spieler und die Gegner bewegen sich durch das Verlies.

Unterwegs kann der Spieler Waffen und Zaubertränke einsammeln.

Der Spieler wird mit den vier Bewegungs-Buttons bewegt.



Dies sind die Gegenstände des Spielers. Hier werden die Dinge angezeigt, die der Spieler aufgehoben hat. Der Gegenstand, den er aktuell verwendet, wird eingerahmt. Ein Gegenstand wird durch einen Klick darauf ausgewählt und mit einem der Angreifen-Buttons eingesetzt.

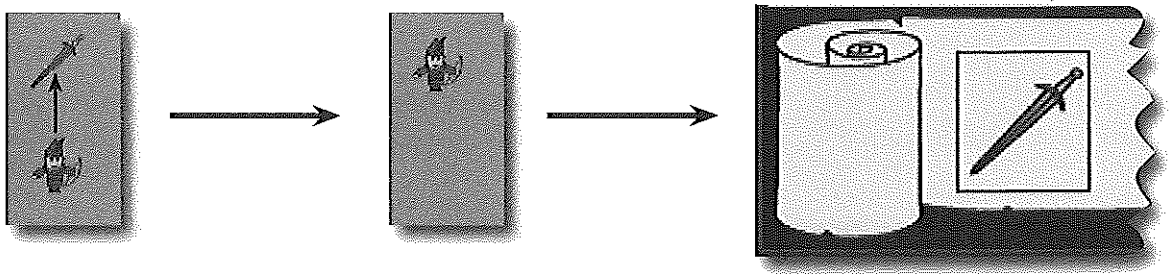
Das Spiel zeigt Ihnen die Anzahl an Trefferpunkten für den Spieler und die Gegner. Greift der Spieler einen Gegner an, verringern sich die Trefferpunkte des Gegners. Sind die Trefferpunkte null, stirbt der Spieler oder der Gegner.

Diese vier Buttons werden eingesetzt, um anzugreifen oder Zaubertränke zu sich zu nehmen.

Der Spieler nimmt Waffen auf ...

Im Verlies sind Waffen und Tränke verstreut, die der Spieler aufheben und einsetzen kann, um seine Gegner zu besiegen. Dazu muss er sich nur auf eine Waffe bewegen. Sie verschwindet dann vom Boden und erscheint im Bestand.

Ein schwarzer Rand um eine Waffe bedeutet, dass er diese gerade einsetzt. Waffen funktionieren unterschiedlich – sie haben unterschiedliche Angriffsbereiche, einige können nur in einer Richtung angreifen, andere haben eine größere Reichweite, und sie bewirken unterschiedliche Schaden, wenn ein Gegner mit ihnen getroffen wird.

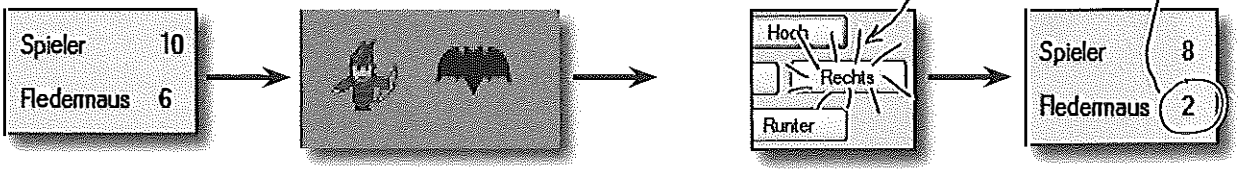


... und greift damit Gegner an

In jedem Level gibt es eine Waffe, die der Spieler aufheben und einsetzen kann, um seine Gegner zu besiegen. Wurde eine Waffe aufgesammelt, sollte sie vom Boden verschwunden sein.

Die Fledermaus befindet sich rechts vom Spieler, er muss also auf den Angreifen-Button Rechts klicken.

Der Angriff bewirkt hier, dass die Trefferpunkte der Fledermaus von 6 auf 2 sinken.



Auf höheren Leveln gibt es mehr Feinde

Es gibt drei verschiedene Arten von Feinden: eine Fledermaus, einen Geist und einen Ghul. Im ersten Level gibt es nur eine Fledermaus. Im siebten und letzten Level tauchen alle drei Feinde auf.

Ein Ghul bewegt sich schnell auf den Gegner zu und bewirkt großen Schaden, wenn er angreift.



Die Fledermaus fliegt etwas zufällig herum. Kommt sie in die Nähe des Spielers bewirkt sie kleinen Schaden.





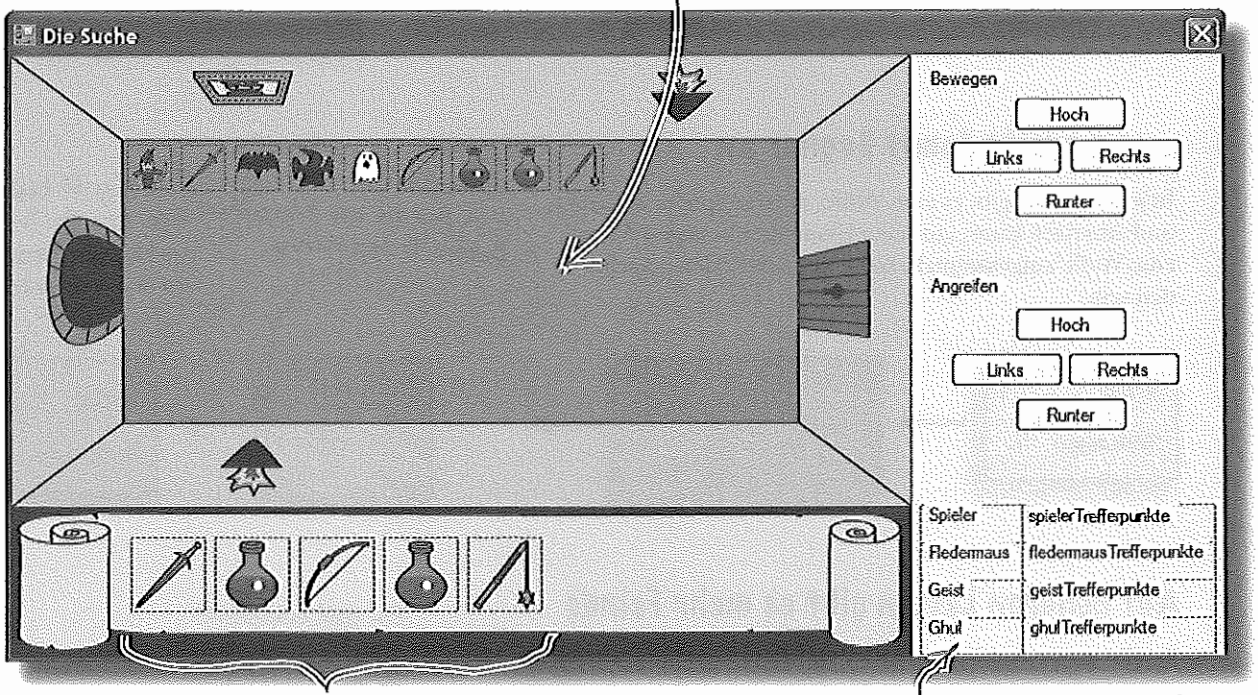
Der Geist bewegt sich langsam auf den Gegner zu. Sobald er in der Nähe des Spielers ist, greift er an und bewirkt mittleren Schaden.

Das Design: Erstellen Sie das Formular

Das Formular verleiht dem Spiel seine unverkennbare Gestalt. Nutzen Sie die `BackgroundImage`-Eigenschaft, um das Verlies und das Inventar anzuzeigen, und eine Reihe von `PictureBox`-Steuerelementen, um im Verlies den Spieler, die Waffen und die Feinde anzuzeigen. Um die Trefferpunkte für den Spieler, die Fledermaus, den Geist und den Ghul sowie die Buttons zum Bewegen und Angreifen anzuzeigen, nutzen Sie das `TableLayoutPanel`-Steuerelement.

Das Verlies selbst ist ein statisches Bild, das über die BackgroundImage-Eigenschaft des Formulars angezeigt wird.

Achten Sie darauf, dass die Eigenschaft BackgroundImage-Layout auf None gesetzt ist.



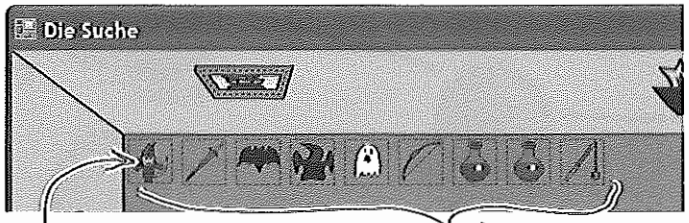
Jedes dieser Symbole steckt in einer PictureBox.

Die Trefferpunkte sowie die Angreifen- und Bewegen-Buttons werden in einem TableLayoutPanel angezeigt.

Das Hintergrundbild sowie die Grafiken für die Waffen, die Gegner und den Spieler können Sie von der O'Reilly-Website herunterladen: http://examples.oreilly.de/german-examples/hfcsharpger

Alles im Verlies steckt in einer PictureBox

Der Spieler sowie die Waffen und Feinde werden durch Symbole repräsentiert. Fügen Sie neun PictureBox-Elemente hinzu und setzen Sie ihre Visible-Eigenschaften auf False. Das Spiel kann diese dann bewegen und ihre Visible-Eigenschaften nach Bedarf umschalten.



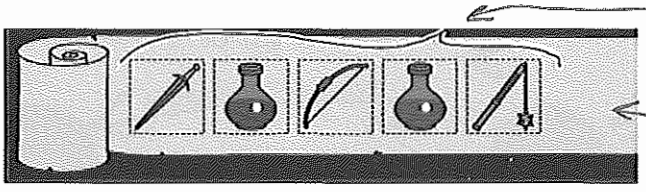
Fügen Sie dem Verlies neun PictureBox-Elemente hinzu. Nutzen Sie die Eigenschaft Size, um jedes 30 x 30 groß zu machen. Wohin Sie sie stecken, ist egal – sie werden vom Formular bewegt. Nutzen Sie den kleinen schwarzen Pfeil, der per Klick auf einer PictureBox erscheint, um jede auf eine der Grafiken von der Von Kopf bis Fuß-Website zu setzen.

Nachdem Sie die PictureBox-Elemente hinzugefügt haben, klicken Sie mit der rechten Maustaste auf das Symbol für den Spieler und wählen >>In den Vordergrund<<. Senden Sie dann auf gleiche Weise die drei Waffen in den Hintergrund. Das gewährleistet, dass das Spieler-Symbol >>über<< den Gegenständen bleibt, die aufgehoben werden.

Steuerelemente können einander in der IDE überschneiden. Das Formular muss deswegen wissen, welche oben und welche unten sind. Das machen die Designer-Befehle >>In den Vordergrund<< und >>In den Hintergrund<<.

Auch das Inventar enthält PictureBox-Elemente

Das Inventar des Spielers stellen Sie über fünf 50-x-50-PictureBox-Elemente dar. Setzen Sie ihre BackColor-Eigenschaft jeweils auf `Color.Transparent`. Da die Grafikdateien einen transparenten Hintergrund haben, sehen Sie dahinter die Schriftrolle und das Verlies.



Sie benötigen fünf weitere 50-x-50-PictureBoxen für den Bestand.

Stattet sich der Spieler mit einer der Waffen aus, sollte das Formular die BorderStyle-Eigenschaft für das Symbol dieser Waffe auf FixedSingle und die der restlichen Symbole auf None setzen.

Erstellen Sie das Fenster für die Spieldaten

Wie die Buttons für das Bewegen und Angreifen werden die Trefferpunkte in einem TableLayoutPanel dargestellt. Erzeugen Sie im Panel für die Trefferpunkte zwei Spalten und ziehen sie die Mittellinie dazwischen etwas nach links. Fügen Sie vier Zeilen hinzu, die jeweils 25% der Höhe einnehmen, und fügen Sie in jede der Zellen ein Label ein.

2 Spalten, 4 Zeilen ... 8 Zellen für Ihre Trefferpunkt-Daten.

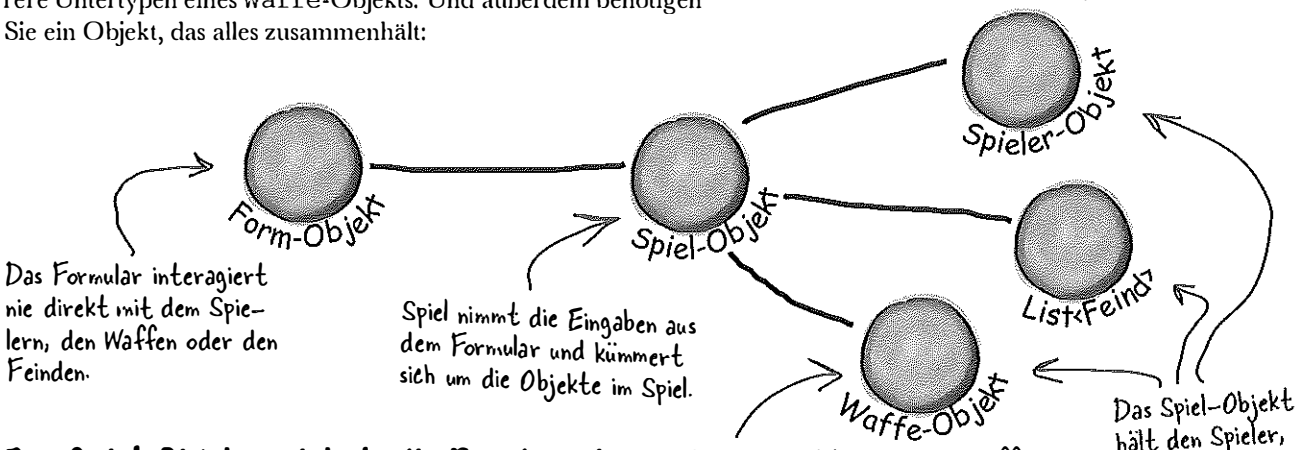
Spieler	spielerTrefferpunkte
Fledermaus	fledermausTrefferpunkte
Geist	geistTrefferpunkte
Ghul	ghulTrefferpunkte

Jede Zelle enthält ein Label. Die Werte darin können Sie während des Spiels aktualisieren.

Die Architektur: die Objekte verwenden

Im Spiel brauchen Sie mehrere Arten von Objekten: ein `Spieler`-Objekt, mehrere Untertypen eines `Feind`-Objekts und mehrere Untertypen eines `Waffe`-Objekts. Und außerdem benötigen Sie ein Objekt, das alles zusammenhält:

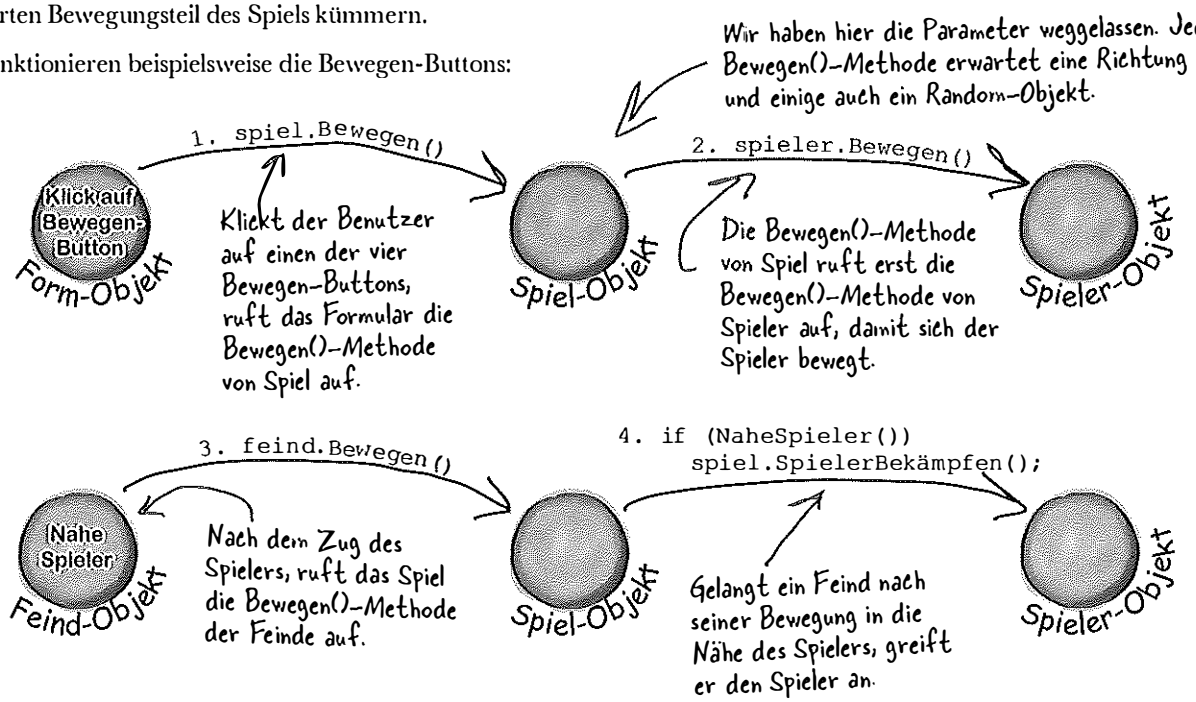
Das ist nur der allgemeine Überblick. Wir werden Ihnen eine Menge mehr Details dazu geben, wie sich der Spieler und die Gegner bewegen und wie die Feinde herausfinden, dass sie sich in der Nähe des Spielers befinden usw.



Das Spiel-Objekt wickelt die Runden ab

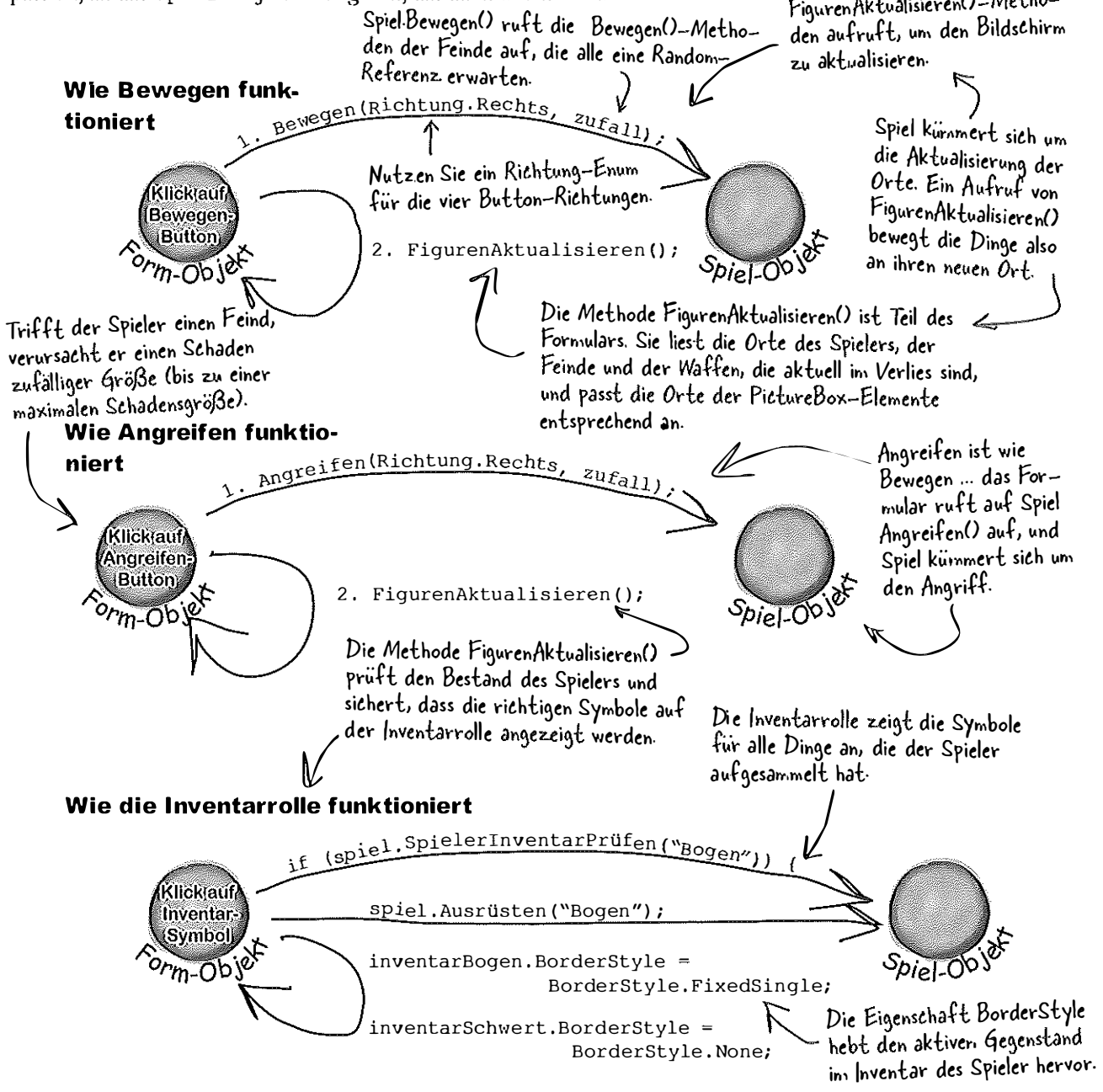
Wird im Formular auf einen der Bewegen-Buttons geklickt, ruft das Formular die `Bewegen()`-Methode des `Spiel`-Objekts auf. Diese Methode lässt erst den Spieler, dann die Feinde ihre Züge machen. `Spiel` muss sich um den rundenbasierten Bewegungsteil des Spiels kümmern.

So funktionieren beispielsweise die Bewegen-Buttons:



Das Formular delegiert Aktivitäten an das Spiel-Objekt

Bewegung, Angriff und Inventar gehen jeweils vom Formular aus. Ein Klick auf einen Bewegen- oder Angreifen-Button löst also Code in Ihrem Formular aus. Aber das `Spiel`-Objekt steuert die Objekte im Spiel. Das Formular muss also alles, was passiert, an das `Spiel`-Objekt weitergeben, das dann übernimmt:



Die Klasse Spiel erstellen

Den Code für die Klasse `Spiel` geben wir Ihnen unten vor. Sie müssen eine Menge tun – lesen Sie den Code also sorgfältig, bringen Sie ihn in die IDE und machen Sie sich an die Arbeit.

```
using System.Drawing;
```

Sie brauchen die Klassen Rectangle und Point aus System.Drawing. Fügen Sie Ihrer Klasse also diese Zeile hinzu.

```
public class Spiel {
  public List<Feind> Feinde;
  public Waffe WaffeInRaum;
```

Sind Feind und Waffe ordentlich gekapselt, können diese Felder problemlos öffentlich sein ... achten Sie also darauf, dass das Formular damit nichts Falsches anstellen kann.

Spiel hält ein privates Spieler-Objekt, mit dem das Formular nur über die Methoden in Spiel interagiert, nicht direkt.

```
  private Spieler spieler;
  public Point SpielerOrt { get { return spieler.Ort; } }
  public int SpielerTrefferpunkt { get { return spieler.Trefferpunkte; } }
  public List<string> SpielerWaffen { get { return spieler.Waffen; } }

  private int level = 0;
  public int Level { get { return level; } }

  private Rectangle grenzen;
  public Rectangle Grenzen { get { return grenzen; } }
```

Das Rectangle-Objekt hat die Felder Top, Bottom, Left und Right und ist als Spielfeld gut geeignet.

```
  public Spiel(Rectangle grenzen) {
    this.grenzen = grenzen;
    spieler = new Spieler(this,
      new Point(grenzen.Left + 10, grenzen.Top + 70);

  }
```

Spiel beginnt mit einer Grenze für das Verlies und erzeugt darin ein neues Spieler-Objekt.

```
  public void Bewegen(Richtung richtung, Random zufall) {
    spieler.Bewegen(richtung);
    foreach (Feind feind in Feinde)
      feind.Bewegen(zufall);
  }
```

Bewegung ist einfach: Der Spieler wird in die Richtung bewegt, die das Formular übergibt, und die Feinde werden in eine zufällige Richtung bewegt.

```
  public void Ausrüsten(string waffenname) {
    spieler.Ausrüsten(waffenname);
  }
  public bool SpielerInventarPrüfen(string waffenname) {
    return spieler.Waffen.Contains(waffenname);
  }
  public void SpielerBekämpfen(int maxSchaden, Random zufall) {
    spieler.Bekämpfen(maxSchaden, zufall);
  }
```

All das sind gute Beispiele für Kapselung ... Spiel weiß nicht, wie Spieler diese Aktionen verarbeitet. Spiel übergibt einfach die erforderlichen Informationen und überlässt Spieler den Rest.

```
  public void SpielerGesundheitErhöhen(int gesundheit, Random zufall) {
    spieler.GesundheitErhöhen(gesundheit, zufall);
  }

  public void Angreifen(Richtung richtung, Random zufall) {
    spieler.Angreifen(richtung, zufall);
    foreach (Feind feind in Feinde)
      feind.Bewegen(zufall);
  }

  private Point ZufallsortGenerieren(Random zufall) {
    return new Point(grenzen.Left +
      zufall.Next(grenzen.Right / 10 - grenzen.Left / 10) * 10,
          grenzen.Top +
      zufall.Next(grenzen.Bottom / 10 - grenzen.Top / 10) * 10);
  }

  public void NeuesLevel(Random zufall) {
    level++;
    switch (level) {
      case 1:
        Feinde = new List<Feind>();
        Feinde.Add(new Fledermaus(this, ZufallsortGenerieren(zufall)));
        WaffeInRaum = new Schwert(this, ZufallsortGenerieren(zufall));
        break;
    }
  }
}
```

Angreifen() ist fast genau wie Bewegen(). Der Spieler greift an, und dann sind die Gegner am Zug.

ZufallsortGenerieren() wird in der Methode NeuesLevel() hilfreich sein. Dort wird sie verwendet, um einen Ort für Feinde und Waffen zu bestimmen.

Das ist etwas mathematische Spielerei zur Ermittlung eines zufälligen Orts innerhalb des Rechtecks, das das Verlies repräsentiert.

Wir haben nur Level 1 hinzugefügt, die anderen Level sind Ihre Aufgabe.

Im Bestand haben wir nur Platz für jeweils einen blauen und einen roten Trank. Hat der Spieler bereits einen roten Trank, sollte das Spiel diesem Level keinen roten Trank mehr hinzufügen. (Gleiches gilt für den blauen Trank.)

Den Rest der Level fertigstellen

Sie sollten die Methode `NeuesLevel()` fertigstellen. Hier ist der Überblick über die einzelnen Level:

Level	Feinde	Waffen
2	Geist	Blauer Trank
3	Ghul	Bogen
4	Fledermaus, Geist	Bogen, falls in 3 nicht gesammelt, sonst blauer Trank
5	Fledermaus, Ghul	Roter Trank
6	Geist, Ghul	Keule
7	Fledermaus, Geist, Ghul	Keule, falls in 6 nicht gesammelt, sonst roter Trank
8	N/A	N/A – Spiel endet mit Application.Exit()

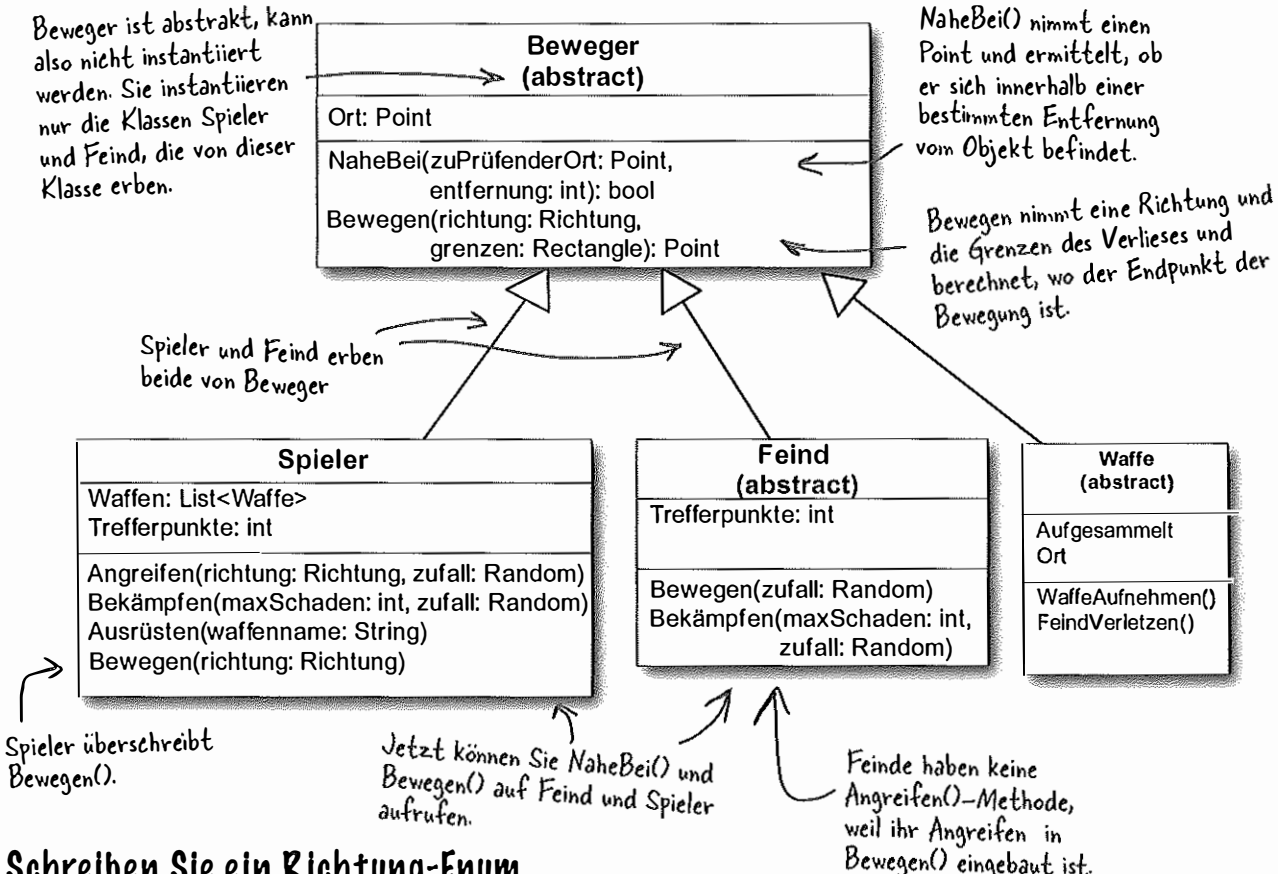
Ist der blaue Trank aus Level 2 noch im Bestand des Spielers, erscheint in diesem Level nichts.

Der Trank erscheint nur, wenn der rote aus Level 5 bereits aufgebraucht wurde.

Gemeinsame Verhalten finden: Bewegung

Sie wissen bereits, dass doppelter Code übel ist und dass er üblicherweise auftaucht, wenn zwei oder mehr Objekte das gleiche Verhalten teilen. Das ist auch bei unserem Verlies-Spiel der Fall ... sowohl Spieler als auch Feinde bewegen sich.

Erzeugen wir eine Klasse `Beweger`, um dieses Verhalten an einen Ort zu ziehen. `Spieler` und `Feind` werden beide von `Beweger` erben. Und obgleich sich Waffen nicht bewegen, erben auch sie von Beweger, weil sie einige der Eigenschaften und Methoden davon benötigen. `Beweger` hat eine `Bewegen()`-Methode zum Bewegen und die schreibgeschützte Eigenschaft `Ort`, die das Formular nutzen kann, um eine `Beweger`-Unterklasse zu positionieren.



Schreiben Sie ein Richtung-Enum

Die Klasse `Beweger` sowie einige andere Klassen brauchen ein `Richtung`-Enum. Erzeugen Sie dieses Enum und geben Sie ihm vier enumerierte Werte: `Hoch`, `Runter`, `Links` und `Rechts`.

Der Code für Beweger

Hier ist der Code für die Klasse Beweger:

```
public abstract class Beweger {
  private const int SchrittGröße = 10;
  protected Point ort;
  public Point Ort { get { return ort; } }
  protected Spiel spiel;

  public Beweger(Spiel spiel, Point ort) {
    this.spiel = spiel;
    this.ort = ort;
  }

  public bool NaheBei(Point zuPrüfenderOrt, int entfernung) {
    if (Math.Abs(ort.X - zuPrüfenderOrt.X) < entfernung &&
       (Math.Abs(ort.Y - zuPrüfenderOrt.Y) < entfernung)) {
      return true;
    } else {
      return false;
    }
  }
  public Point Bewegen(Richtung richtung, Rectangle grenzen) {
    Point neuerOrt = ort;
    switch (richtung) {
      case Richtung.Hoch:
        if (neuerOrt.Y - SchrittGröße >= grenzen.Top)
          neuerOrt.Y -= SchrittGröße;
        break;
      case Richtung.Runter:
        if (neuerOrt.Y + SchrittGröße <= grenzen.Bottom)
          neuerOrt.Y += SchrittGröße;
        break;
      case Richtung.Links:
        if (neuerOrt.X - SchrittGröße >= grenzen.Left)
          neuerOrt.X -= SchrittGröße;
        break;
      case Richtung.Rechts:
        if (neuerOrt.X + SchrittGröße <= grenzen.Rechts)
          neuerOrt.X += SchrittGröße;
        break;
      default: break;
    }
    return neuerOrt;
  }
}
```

Da geschützte Eigenschaften nur für Unterklassen verfügbar sind, kann das Formular ort nicht setzen ... es kann ihn nur über den öffentlichen Getter lesen.

Instanzen von Beweger nehmen das Spiel-Objekt und einen aktuellen Ort an.

Die Methode NaheBei vergleicht einen Point mit dem aktuellen Ort des Objekts. Sie liefert true, wenn sie weniger als entfernung weit voneinander entfernt sind, andernfalls false.

Die Methode Bewegen() versucht, einen Schritt in eine Richtung zu gehen. Kann sie das, liefert sie ein neues Point-Objekt. Trifft sie auf eine Grenze, liefert sie das ursprüngliche Point-Objekt.

Liegt der Endort außerhalb der Grenzen, bleiben Endort und Anfangsort gleich.

Schließlich wird der neue Ort geliefert (der aber immer noch der gleiche Ort sein kann wie der Startort!).

Die Klasse Spieler hält den Spieler nach

Hier ist ein Anfang für die Klasse Spieler. Beginnen Sie in der IDE mit diesem Code und ergänzen Sie ihn dann.

```
public class Spieler : Beweger {
  private Waffe verwendeteWaffe;
  private int trefferpunkte;
  public int Trefferpunkte { get { return trefferpunkte; } }

  private List<Waffe> inventar = new List<Waffe>();
  public List<string> Waffen {
    get {
      List<string> namen = new List<string>();
      foreach (Waffe waffe in inventar)
        namen.Add(waffe.Name);
      return namen;
    }
  }

  public Spieler(Spiel spiel, Point ort)
        : base(spiel, ort) {
    trefferpunkte = 10;
  }

  public void Bekämpfen(int maxSchaden, Random zufall) {
    trefferpunkte -= zufall.Next(1, maxSchaden);
  }

  public void GesundheitErhöhen(int gesundheit, Random zufall) {
    trefferpunkte += zufall.Next(1, gesundheit);
  }

  public void Ausrüsten(string waffenname) {
    foreach (Waffe waffe in inventar) {
      if (waffe.Name == waffenname)
        verwendeteWaffe = waffe;
    }
  }
}
```

Alle Eigenschaften von Spieler sind vor direktem Zugriff geschützt.

Ein Spieler kann im Inventar mehrere Waffen haben, aber immer nur eine verwenden.

Spieler erbt von Beweger. Der Konstruktor übergibt also spiel und ort an diese Basisklasse.

Der Spieler-Konstruktor setzt trefferpunkte auf 10 und ruft dann den Basisklassenkonstruktor auf.

Bekämpft ein Feind den Spieler, verursacht er einen Schaden zufälliger Stärke. Stärkt ein Trank die Gesundheit des Spielers, erhöht er sie um einen zufälligen Betrag.

Die Methode Ausrüsten() sagt dem Spieler, dass er eine seiner Waffen verwenden soll. Das Spiel-Objekt ruft diese Methode auf, wenn auf das Inventar-Symbol geklickt wird.

Ein Spieler-Objekt kann immer nur ein Waffe-Objekt verwenden.

Obwohl Tränke dem Spieler helfen, statt dem Feind zu schaden, werden sie vom Spiel als Waffen betrachtet. So kann das Inventar eine List<Waffe> sein, und das Spiel kann auf eine mit seiner WaffeInRaum-Referenz zeigen.

Schreiben Sie die Bewegen()-Methode für Spieler

`Spiel` ruft die `Bewegen()`-Methode von `Spieler` auf, um einem Spieler zu sagen, dass er sich in eine bestimmte Richtung bewegen soll. `Bewegen()` nimmt die Bewegungsrichtung (nutzen Sie das Enum `Richtung`, das Sie bereits schreiben sollten). Hier ist der Anfang dieser Methode:

Das passiert, wenn im Formular auf einen Bewegen-Button geklickt wird.

```
public void Bewegen(Richtung richtung) {
  base.ort = Bewegen(richtung, spiel.Grenzen);
  if (!spiel.WaffeInRaum.Aufgesammelt) {
    // Prüfen, ob eine Waffe in der Nähe ist, und diese
    // gegebenenfalls aufsammeln.
  }
}
```

Bewegen kommt aus der Basisklasse Beweger.

Sie müssen den Rest dieser Methode ergänzen. Prüfen Sie, ob sich die Waffe in der Nähe des Spielers befindet (nicht weiter als eine Einheit entfernt). Heben Sie die Waffe auf und fügen Sie sie dem Inventar des Spielers hinzu, wenn das der Fall ist.

Hebt der Spieler eine Waffe auf, muss diese aus dem Verlies verschwinden und im Inventar erscheinen.

Ist die Waffe die einzige Waffe, die der Spieler hat, sollte er sie sofort verwenden. Dann kann er sie im nächsten Zug sofort benutzen.

Waffe und Formular kümmern sich darum, dass die PictureBox der Waffe unsichtbar wird, wenn der Spieler sie aufsammelt ... das ist nicht Aufgabe der Klasse Spieler.

Auch Angreifen() brauchen Sie

Jetzt kommt die Methode `Angreifen()`. Diese wird per Klick auf einen der Angreifen-Buttons aufgerufen und erhält eine Richtung (wieder aus dem Enum `Richtung`). Hier ist die Signatur der Methode:

Die Waffen haben alle eine Angreifen()-Methode, die einen Richtung-Wert und ein Random-Objekt erwartet. Die Angreifen()-Methode des Spielers ermittelt, mit welcher Waffe er ausgerüstet ist, und ruft ihre Angreifen()-Methode auf.

```
public void Angreifen(Richtung richtung, Random zufall) {
  // Hier kommt Ihr Code hin.
}
```

Ist der Spieler noch nicht mit einer Waffe ausgestattet, macht diese Methode nichts. Trägt der Spieler eine Waffe, sollte sie die `Angreifen()`-Methode der Waffe aufrufen.

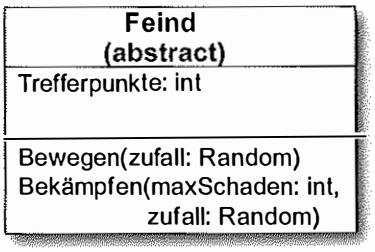
Ist die Waffe ein Trank, entfernt Angreifen() ihn aus dem Inventar, nachdem der Spieler ihn getrunken hat.

Tränke sind ein Sonderfall. Wird ein Trank verwendet, muss er aus dem Inventar des Spielers entfernt werden, da er nicht mehr verfügbar ist.

Tränke implementieren die Schnittstelle ITrank (mehr dazu in einer Minute). Sie können also das Schlüsselwort »is« verwenden, um zu prüfen, ob eine Waffe eine Implementierung von ITrank ist.

Fledermäuse, Geister und Ghule erben von Feind

Wir geben Ihnen eine weitere nützliche abstrakte Klasse: `Feind`. Für jede Art von Feind gibt es eine Klasse, die von `Feind` erbt. Die unterschiedlichen Feindarten bewegen sich auf unterschiedliche Weise. Die abstrakte Klasse `Feind` lässt die `Bewegen`-Methode also abstrakt – die drei Feind-Klassen müssen sie in Abhängigkeit von ihrem jeweiligen Bewegungsverhalten implementieren.




```
public abstract class Feind : Beweger {
  private const int SpielerNahdistanz = 25;
  private int trefferpunkte;
  public int Trefferpunkte { get { return trefferpunkte; } }
  public bool Tod { get {
       if (trefferpunkte <= 0) return true;
       else return false;
     }
  }
  public Feind(Spiel spiel, Point ort, int trefferpunkte)
       : base(spiel, ort) { this.trefferpunkte = trefferpunkte; }

  public abstract void Bewegen(Random zufall);

  public void Bekämpfen(int maxSchaden, Random zufall) {
     trefferpunkte -= zufall.Next(1, maxSchaden);
  }

  protected bool NaheSpieler() {
    return (NaheBei(spiel.SpielerOrt,
                    SpielerNahdistanz));
  }
  protected Richtung SpielerrichtungSuchen(Point spielerOrt) {
    Richtung bewegungsrichtung;
    if (spielerOrt.X > ort.X + 10)
          bewegungsrichtung = Richtung.Rechts;
    else if (spielerOrt.X < ort.X - 10)
          bewegungsrichtung = Richtung.Links;
    else if (spielerOrt.Y < ort.Y - 10)
          bewegungsrichtung = Richtung.Hoch;
    else
          bewegungsrichtung = Richtung.Runter;
     return bewegungsrichtung;
  }
}
```

Über diese schreibgeschützte Eigenschaft kann das Formular sehen, ob der Feind im Spielverlies sichtbar sein soll.

Diese Methode wird von den Feind-Unterklassen implementiert.

Greift der Spieler einen Feind an, ruft er die Bekämpfen()-Methode von Feind auf, die eine Zufallszahl von den Trefferpunkten abzieht.

Die Methode NaheSpieler() nutzt die statische Methode Beweger.NaheBei(), um zu ermitteln, ob der Feind in der Nähe des Spielers ist.

Übergeben Sie den Ort des Spielers an SpielerrichtungSuchen(), wird das Basisklassenfeld `ort` eingesetzt, um zu ermitteln, wo sich der Spieler in Bezug auf den Feind befindet. Es wird dann ein Richtung-Wert geliefert, der sagt, in welche Richtung der Feind gehen muss, um sich in Richtung des Spielers zu bewegen.

Schreiben Sie die Feind-Unterklassen

Die drei `Feind`-Unterklassen sind ziemlich einfach. Die Feinde haben jeweils eine andere anfängliche Anzahl von Trefferpunkten, bewegen sich anders und verursachen einen Schaden unterschiedlicher Größe. Für jede Unterklasse muss dem Konstruktor der Basisklasse `Feind` ein anderer Wert für den Parameter `anfänglicherTrefferpunkt` übergeben und eine andere `Bewegen()`-Methode geschrieben werden.

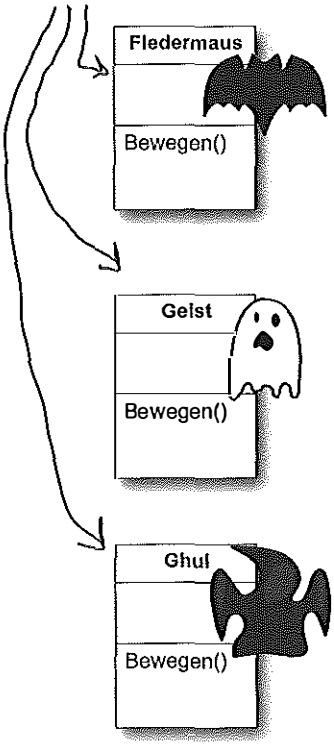
Hier ist ein Beispiel dafür, wie eine der Klassen aussehen könnte:

```
public class Fledermaus : Feind {
    public Fledermaus(Spiel spiel, Point ort)
        : base(spiel, ort, 6)
    { }
    public override void Bewegen(Random zufall) {
        // Hier kommt Ihr Code hin.
    }
}
```

Eine Fledermaus beginnt mit sechs Trefferpunkten, übergibt also diesen Wert an den Basisklassenkonstruktor.

Die Unterklassenkonstruktoren brauchen keine Implementierung, da sich die Basisklasse um alles kümmert.

Alles sind Unterklassen der Klasse Feind, die ihrerseits eine Unterklasse von Beweger ist.



Die Fledermaus fliegt recht wahllos herum, nutzt also einen Random-Wert, um die Hälfte der Zeit in eine zufällige Richtung zu fliegen.

Hat ein Feind keine Trefferpunkte mehr, wird er im Formular nicht mehr angezeigt. Aber er verbleibt in der Feind-Liste des Spiels, bis der Spieler das Level abschließt.

Die Fledermaus beginnt mit sechs Trefferpunkten. Sie bewegt sich auf den Spieler zu und greift an, **solange er noch Trefferpunkte hat.** Sie bewegt sich mit 50%iger Wahrscheinlichkeit auf den Spieler zu und mit 50%iger Wahrscheinlichkeit in eine zufällige Richtung. Nachdem sich die Fledermaus bewegt hat, prüft sie, ob sie in der Nähe des Spielers ist – ist sie es, greift sie den Spieler an und verursacht bis zu zwei Trefferpunkte Schaden.

Wir müssen sicherstellen, dass das Formular erkennt, ob ein Feind in einer Runde sichtbar sein soll.

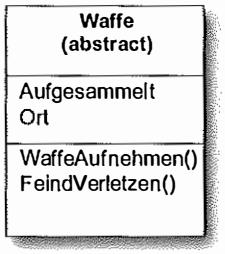
Der Geist ist schwerer zu besiegen als die Fledermaus. Bewegen und Angriff erfolgen nur, wenn er mehr als null Trefferpunkte hat. Er beginnt mit acht Trefferpunkten. In seiner Bewegungsphase bewegt er sich in einem von drei Fällen auf den Spieler zu. In zwei von drei Fällen verharrt er an dem Ort, an dem er sich befindet. Befindet er sich in der Nähe eines Spielers, greift er an und verursacht bis zu drei Trefferpunkte Schaden.

Der Ghul ist der härteste Gegner. Er beginnt mit zehn Trefferpunkten. Angriff und Bewegung erfolgen nur, wenn er mehr als null Trefferpunkte hat. Er bewegt sich in zwei von drei Fällen auf den Spieler zu und bleibt in einem von drei Fällen stehen. Ist er in der Nähe des Spielers, greift er mit bis zu vier Trefferpunkten an.

Der Geist und der Ghul nutzen Random, damit sie sich langsamer bewegen als der Spieler.

Waffe erbt von Beweger und jede einzelne Waffe von Waffe

Genau so wie wir eine `Feind`-Klasse haben, brauchen wir eine `Waffe`-Klasse. Jede Waffe hat einen Ort und eine Eigenschaft, die anzeigt, ob sie eingesammelt wurde oder nicht. Hier ist die Klasse `Waffe`:



Waffe erbt von Beweger, weil sie in FeindVerletzen() die Methoden NaheBei() und Bewegen() verwendet.

```
public abstract class Waffe : Beweger {

  protected Spiel spiel;
  private bool aufgesammelt;
  public bool Aufgesammelt { get { return aufgesammelt; } }
  private Point ort;
  public Point Ort { get { return ort; } }

  public Waffe(Spiel spiel, Point ort) {
    this.spiel = spiel;
    this.ort = ort;
    aufgesammelt = false;
  }

  public void WaffeAufnehmen() { aufgesammelt = true; }

  public abstract string Name { get; }

  public abstract void Angreifen(Richtung richtung, Random zufall);

  protected bool FeindVerletzen(Richtung richtung, int radius,
                                int schaden, Random zufall) {
    Point ziel = spiel.SpielerOrt;
    for (int entfernung = 0; entfernung < radius; entfernung++) {
      foreach (Feind feind in spiel.Feinde) {
        if (NaheBei(feind.Ort, ziel, radius)) {
          feind.Bekämpfen(schaden, zufall);
          return true;
        }
      }
      ziel = Bewegen(richtung, ziel, spiel.Grenzen);
    }
    return false;
  }
}
```

Eine aufgesammelte Waffe sollte nicht mehr angezeigt werden ... das Formular kann diesen Getter verwenden, um das herauszufinden.

Jede Waffe hat einen Ort im Spielverlies.

Der Konstruktor setzt die Felder spiel und ort und setzt das Feld aufgesammelt auf false (weil die Waffe noch nicht aufgehoben wurde).

Jede Waffe-Klasse muss die Eigenschaft Name und die Methode Angreifen() implementieren, die bestimmt, wie diese Waffe angreift.

Die Name-Eigenschaft der Waffe liefert ihren Namen (»Schwert«, »Keule«, »Bogen«).

Waffen haben unterschiedliche Reichweiten und bewirken Schäden unterschiedlicher Größe, deswegen implementieren die Waffen die Methode Angreifen() unterschiedlich.

Die Methode FeindVerletzen() wird von Angreifen() aufgerufen. Sie versucht, einen Feind in einer bestimmten Richtung und in einem bestimmten Umkreis zu finden. Tut sie das, ruft sie die Bekämpfen()-Methode des Feinds auf und liefert true. Ist kein Feind erreichbar, liefert sie false.

Waffen greifen unterschiedlich an

Jede `Waffe`-Unterklasse hat einen eigenen Namen und eine andere Angriffslogik. Sie sollen diese Klassen implementieren. Hier ist das Grundgerüst für eine Waffe-Unterklasse:

```
public class Schwert : Waffe {

  public Schwert(Spiel spiel, Point ort)
      : base(spiel, ort) { }

  public override string Name { get { return "Schwert"; } }

  public override void Angreifen(Richtung richtung, Random zufall) {
    // Hier kommt Ihr Code hin.
  }
}
```

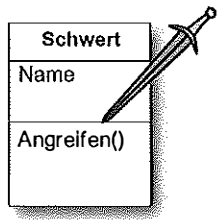
Jede Unterklasse repräsentiert eine dieser Waffen Schwert, Bogen oder Keule.

Die Unterklassen verlassen sich darauf, dass die Basisklasse die Initialisierung erledigt.

Den Namen der Waffe geben Sie einfach fest vor.

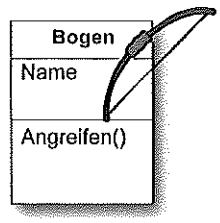
Das Spiel-Objekt übergibt die Richtung, in die angegriffen wird.

Der Spieler kann die Waffen immer wieder verwenden – sie gehen nicht verloren und nutzen sich auch nicht ab.

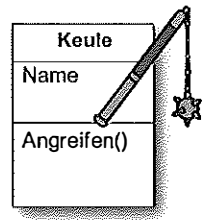


Das Schwert ist die erste Waffe, die der Spieler aufhebt. Es hat einen weiten Angriffsbereich: Greift es an, versucht es erst einen Feind in dieser Richtung zu treffen -- befindet sich dort kein Feind, geht es im Uhrzeigersinn eine Richtung weiter und sucht dort nach einem Feind. Findet es auch dort nichts, geht es auch entgegen dem Uhrzeigersinn eine Richtung über die ursprüngliche Richtung hinaus und sucht weiter nach einem Feind. Es hat eine Reichweite von 10 und verursacht 3 Schadenspunkte.

Denken Sie darüber genau nach ... was ist rechts der Richtung Links oder was links der Richtung Oben?



Der Bogen hat einen kleinen Angriffsbereich, aber eine große Reichweite – er hat eine Reichweite von 30, verursacht aber nur einen Schadenspunkt. Anders als das Schwert, das in drei Richtungen angreift (weil der Spieler es in einem weiten Bogen schwingt), schießt der Bogen nur in die Richtung, in die der Spieler angreift.



Die Keule ist die mächtigste Waffe im Verlies. Es spielt keine Rolle, in welche Richtung der Spieler angreift – da er sie immer einmal im Kreis schwingt. Sie greift jeden Feind in einem Umkreis von 20 Punkten an und verursacht sechs Schadenspunkte.

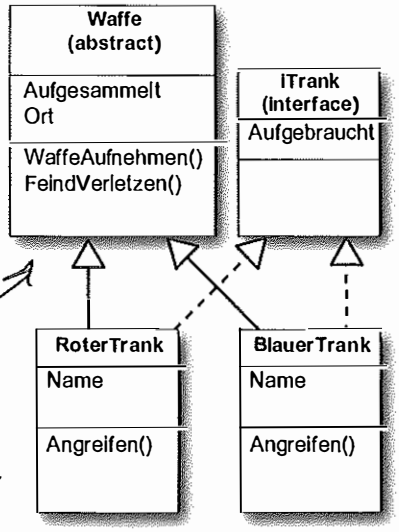
Die einzelnen Waffen rufen FeindVerletzen() auf unterschiedliche Weise auf. Die Keule greift in alle Richtungen an. Greift der Spieler nach rechts an, ruft sie also FeindVerletzen(Richtung.Rechts, 20, 6, zufall) auf. Trifft das keinen Feind, probiert sie es erst nach oben, dann nach links und unten -- so schwingt sie einen vollständigen Kreis.

Tränke implementieren die Schnittstelle ITrank

Es gibt zwei Tränke, einen blauen und einen roten, die die Gesundheit des Spielers verbessern. Sie funktionieren genau so wie Waffen – der Spieler sammelt sie im Verlies auf, rüstet sich mit ihnen aus, indem er im Inventar darauf klickt, und **nutzt sie, indem er auf einen der Angreifen-Buttons klickt**. Es ist also recht vernünftig, auch sie von der abstrakten Klasse `Waffe` abzuleiten.

Aber Tränke verhalten sich auch etwas anders. Sie benötigen also eine Schnittstelle `ITrank`, um ihnen ein weiteres Verhalten zu geben: die Gesundheit des Spielers zu erhöhen. Die Schnittstelle `ITrank` ist sehr einfach. Tränke müssen nur eine schreibgeschützte Eigenschaft namens `Aufgebraucht` hinzufügen, die `false` liefert, wenn der Spieler den Trank noch nicht verwendet hat, und `true`, wenn er ihn verwendet hat. Diese nutzt das Formular, um zu ermitteln, ob der Trank im Formular angezeigt werden soll.

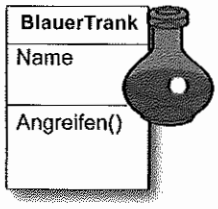
```
public interface ITrank {
  bool Aufgebraucht { get; }
}
```



ITrank sorgt dafür, dass Tränke nur einmal verwendet werden können. Außerdem kann auf Basis dieser Schnittstelle mit >>if (waffe is ITrank)<< herausgefunden werden, ob eine Waffe ein Trank ist.

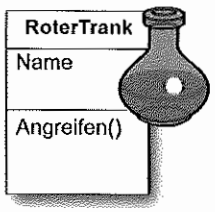
Tränke erben von Waffe, weil sie wie Waffen genutzt werden – der Spieler klickt im Inventar auf einen Trank, um sich mit ihm auszurüsten, und klickt dann auf einen der Angreifen-Buttons, um ihn zu verwenden.

Diese Klassen sollten mithilfe des Klassendiagramms und der Informationen hier unten geschrieben werden können.



Die `Name`-Eigenschaft der Klasse `BlauerTrank` sollte den String »Blauer Trank« liefern. Ihre `Angreifen()`-Methode wird aufgerufen, wenn der Spieler den Trank nutzt – er sollte die Gesundheit des Spielers um fünf Trefferpunkte erhöhen, indem die Methode `SpielerGesundheitErhöhen()` aufgerufen wird. Nachdem der Spieler den Trank verwendet hat, sollte die `Aufgebraucht`-Eigenschaft des Tranks `true` liefern.

Wenn der Spieler in Level 2 einen blauen Trank einsammelt, diesen nutzt und in Level 4 einen weiteren einsammelt, erstellt das Spiel zwei separate Instanzen von BlauerTrank.



Die Klasse `RoterTrank` ist `BlauerTrank` sehr ähnlich. Die einzigen Unterschiede sind, dass die Eigenschaft `Name` den String »Roter Trank« liefert und die Methode `Angreifen()` die Gesundheit des Spielers um bis zu zehn Trefferpunkte erhöht.

Das Formular bringt alles zusammen

Es gibt eine Instanz der Klasse `Spiel`, die in einem privaten Feld des Formulars festgehalten wird. Sie wird im `Load`-Event des Formulars erstellt, und die Event-Handler für das Formular nutzen die Felder und Methoden dieser `Spiel`-Instanz, um das Spiel zu steuern.

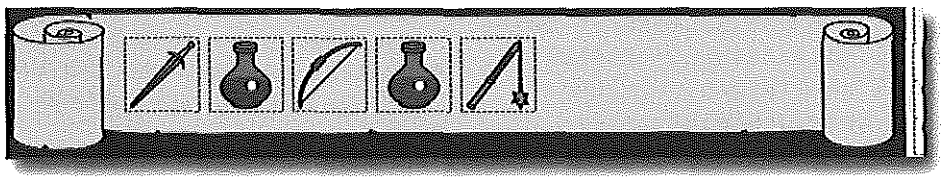
Alles beginnt im `Load`-Event-Handler des Formulars, der `Spiel` ein `Rectangle`-Objekt übergibt, das die Grenzen des Spielbereichs definiert. Hier ist etwas Formularcode für den Anfang:

```
private Spiel spiel;
private Random zufall = new Random();
private void Form1_Load(object sender,
                        EventArgs e) {
    spiel = new Spiel(new Rectangle(78, 57, 420, 155));
    spiel.NeuesLevel(zufall);
    FigurenAktualisieren();
}
```

Ein Rectangle nutzen

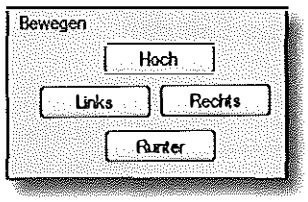
Bei der Arbeit mit Formularen werden Sie oft auf Rectangle-Objekte stoßen. Sie erstellen diese, indem Sie Werte für X, Y, Width und Height übergeben oder zwei Point-Objekte (für gegenüberliegende Ecken). Die Grenzen eines Rechtecks können Sie über Left, Right Top und Bottom abrufen.

Das sind die Grenzen des Verlieses in dem Hintergrundbild, das Sie herunterladen und dem Formular hinzufügen werden.



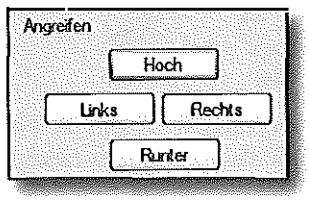
Klicken Sie auf jedes der PictureBox-Elemente, damit die IDE für jedes einen Event-Handler hinzufügt.

Das Formular hat separate Event-Handler für die Click-Events der PictureBox-Elemente. Klickt der Spieler auf ein Schwert, prüft dieser mit der `SpielerInventarPrüfen()`-Methode von `Spiel` zunächst, ob sich das Schwert im Inventar des Spielers befindet. Ist das der Fall, ruft das Formular `spiel.Ausrüsten()` auf, um den Spieler mit dem Schwert auszurüsten. Dann setzt es die `BorderStyle`-Eigenschaften, um das Schwert einzurahmen und einen eventuellen Rahmen von anderen Waffen zu entfernen.



Es gibt je einen Event-Handler für jeden der Bewegen-Buttons. Sie sind ziemlich einfach. Erst ruft der Button `spiel.Bewegen()` mit dem erforderlichen `Richtung`-Wert auf, dann die `FigurenAktualisieren()`-Methode des Formulars.

Achten Sie darauf, dass Sie die Buttons wieder zurücksetzen, wenn sich der Spieler mit Schwert, Bogen oder Keule ausrüstet.



Auch die Event-Handler für die vier Angreifen-Buttons sind leicht. Jeder Button ruft erst `spiel.Angreifen()` auf und dann die Methode `FigurenAktualisieren()`. Ist der Spieler mit einem Trank ausgerüstet, werden diese auf gleiche Weise verwendet – indem `spiel.Angreifen()` aufgerufen wird –, aber Tränke haben keine Richtung. Machen Sie also die Buttons Links, Rechts und Unten unsichtbar, wenn der Spieler mit einem Trank ausgerüstet ist, und ändern Sie den Text auf dem Button Oben in »Drink«.

Die FigurenAktualisieren()-Methode des Formulars bringt PictureBoxen in Position

Das letzte Puzzleteil ist die `FigurenAktualisieren()`-Methode des Formulars. Nachdem sich die Objekte bewegt und aufeinander eingewirkt haben, aktualisiert das Formular alles ... damit die Visible-Eigenschaften der PictureBox-Elemente für Waffen, die eingesammelt wurden, auf false gesetzt werden, Feinde und Spieler an ihrem neuen Ort gezeichnet (und Tote unsichtbar gemacht) werden und das Inventar aktualisiert wird.

Hier sehen Sie, was Sie tun müssen:

1 **Aktualisieren Sie die Position und die Daten des Spielers.**
Erst müssen Sie die PictureBox für den Spieler und das Label mit den Trefferpunkten aktualisieren. Dann benötigen Sie ein paar Variablen, um zu prüfen, ob Sie die verschiedenen Feinde angezeigt haben.

```
public void FigurenAktualisieren() {
  spieler.Location = spiel.SpielerOrt;
  spielerTrefferpunkte.Text =
    spiel.SpielerTrefferpunkt.ToString();

  bool fledermausZeigen = false;
  bool geistZeigen = false;
  bool ghulZeigen = false;
  int gezeigteFeinde = 0;
  // Hier muss noch mehr Code hin ...
```

Die Variable fledermausZeigen wird auf true gesetzt, wenn wir die PictureBox der Fledermaus sichtbar machen. Das Gleiche gilt für geistZeigen und ghulZeigen.

3 **Aktualisieren Sie den Ort und die Trefferpunkte der Feinde.**
Jeder Feind könnte sich an einem anderen Ort befinden oder eine andere Anzahl Trefferpunkte haben. Die Feinde müssen Sie aktualisieren, nachdem Sie den Spieler aktualisiert haben:

```
foreach (Feind feind in spiel.Feinde) {
  if (feind is Fledermaus) {
    fledermaus.Location = feind.Ort;
    fledermausTrefferpunkte.Text = feind.Trefferpunkte.ToString();
    if (feind.Trefferpunkte > 0) {
      fledermausZeigen = true;
      gezeigteFeinde++;
    }
  }
  // usw.
```

Das kommt unmittelbar nach dem Code oben.

Das hat keinen unerheblichen Einfluss auf die Sichtbarkeit der PictureBox für den Feind.

In Ihrer foreach-Schleife brauchen Sie zwei weitere if-Anweisungen wie diese – eine für den Geist, eine für den Ghul.

Nachdem Sie alle Feinde in diesem Level durchlaufen haben, prüfen Sie die Variable. Wurde die Fledermaus getötet, ist fledermausZeigen weiterhin false. Machen Sie dann die PictureBox unsichtbar und leeren Sie das Label für die Trefferpunkte. Tun Sie dann das Gleiche mit geistZeigen und ghulZeigen.

3 Aktualisieren Sie die PictureBox-Elemente für die Waffen.

Deklarieren Sie die Variable `waffenSteuerung` und nutzen Sie eine große `switch`-Anweisung, um sie auf die PictureBox zu setzen, die der Waffe im Raum entspricht.

```
schwert.Visible = false;
bogen.Visible = false;
roterTrank.Visible = false;
blauerTrank.Visible = false;
keule.Visible = false;
Control waffenSteuerung = null;
switch (spiel.WaffeInRaum.Name) {
   case "Schwert":
      waffenSteuerung = schwert; break;
```

Achten Sie darauf, dass die Namen Ihrer Steuerelemente diesen Namen entsprechen. Sind sie nicht gleich, kann das zu Fehlern führen, die schwer aufzuspüren sind.

Sie brauchen weitere Fälle für die anderen Waffen.

Die weiteren Fälle sollten die Variable `waffenSteuerung` auf das erforderliche Steuerelement für das Formular setzen. Nach der `switch`-Anweisung setzen Sie `waffenSteuerung.Visible` auf `true`, um die entsprechende Waffe anzuzeigen.

4 Setzen Sie die Visible-Eigenschaft der Inventar-PictureBoxen.

Prüfen Sie die `SpielerInventarPrüfen()`-Methode von `Spiel`, um herauszufinden, welche der verschiedenen Inventar-Symbole angezeigt werden müssen.

5 Hier ist der Rest der Methode.

Der Rest der Methode macht Folgendes: Erst wird geprüft, ob der Spieler bereits die Waffe aufgehoben hat, die sich im Raum befindet, um diese dann gegebenenfalls unsichtbar zu machen. Dann wird geprüft, ob der Spieler gestorben ist. Als Letztes wird geprüft, ob der Spieler alle Feinde besiegt hat, er also ins nächste Level gelangt.

```
waffenSteuerung.Location = spiel.WaffeInRaum.Ort;
if (spiel.WaffeInRaum.Aufgesammelt) {
  waffenSteuerung.Visible = false;
} else {
  waffenSteuerung.Visible = true;
}
if (spiel.SpielerTrefferpunkt <= 0) {
  MessageBox.Show("Sie sind tot");
  Application.Exit();
}
if (gezeigteFeinde < 1) {
  MessageBox.Show("Sie haben alle Feinde in diesem Level besiegt");
  spiel.NeuesLevel(zufall);
  FigurenAktualisieren();
}
```

In jedem Level kann eine Waffe aufgehoben werden. Wurde sie aufgehoben, müssen Sie das Symbol unsichtbar machen.

Application.Exit() beendet das Programm sofort. Es ist Teil von System.Windows.Forms. Sie müssen also die erforderliche using-Anweisung angeben, möchten Sie die Methoden außerhalb eines Formulars nutzen.

Gibt es auf diesem Level keine Feinde mehr, hat der Spieler alle besiegt und kann zum nächsten Level weitergehen.

Der Spaß fängt gerade erst an!

Sieben Level, drei Feinde ... noch ist das ein recht bescheidenes Spiel. Aber Sie können es verbessern. Hier sind ein paar Anregungen ...

Machen Sie die Feinde schlauer!
Haben Sie eine Idee, wie man die Bewegen()-Methoden der Feinde so ändern könnte, dass sie schwerer zu besiegen sind? Schauen Sie, ob Sie ihre Konstanten in Eigenschaften umwandeln und Mittel hinzufügen können, sie während des Spiels zu ändern.

Fügen Sie mehr Level hinzu!
Das Spiel muss nicht nach sieben Leveln enden. Versuchen Sie, weitere hinzuzufügen ... haben Sie eine Idee, wie man das Spiel endlos weitergehen lassen könnte? Schreiben Sie eine coole Animation mit tanzenden Geistern und Fledermäusen, die angezeigt wird, wenn der Spieler gewinnt! Das Spiel endet ziemlich abrupt, wenn der Spieler stirbt. Fällt Ihnen ein benutzerfreundlicheres Ende ein? Vielleicht könnte man den Benutzer das Spiel neu starten oder das letzte Level neu versuchen lassen.

Fügen Sie weitere Feindarten hinzu!
Sie müssen die Gefahren nicht auf Ghule, Geister und Fledermäuse beschränken. Schauen Sie, ob Sie dem Spiel weitere Feinde hinzufügen können.

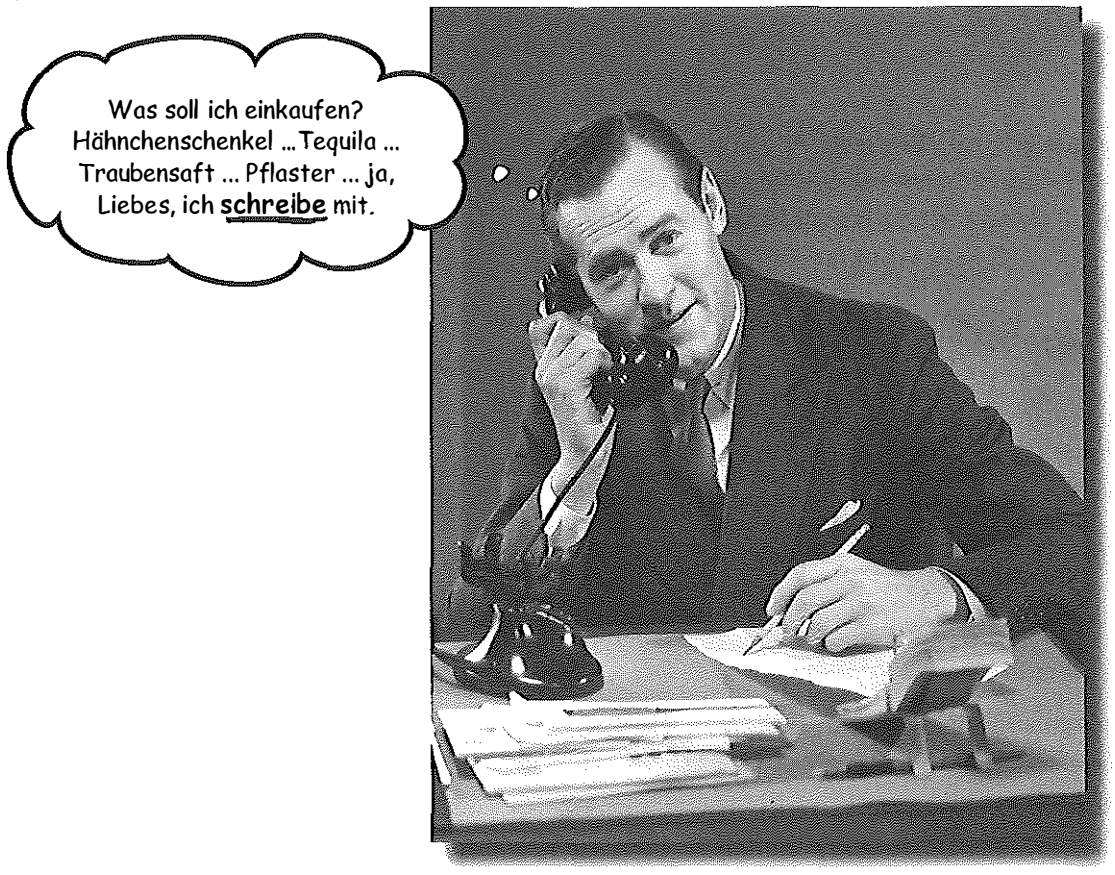
Fügen Sie weitere Waffen hinzu!
Der Spieler braucht unbedingt weitere Hilfe, um seine neuen Feinde zu besiegen. Denken Sie sich weitere Möglichkeiten aus, auf die Waffen angreifen oder Tränke wirken können. Nutzen Sie, dass Waffe eine Unterklasse von Beweger ist – schaffen Sie weitere Waffen, denen der Spieler hinterherjagen muss!

Fügen Sie weitere Grafiken hinzu!
Unter **www.headfirstlabs.com/books/hfcsharp/** finden Sie Grafiken für weitere Feinde, Waffen und andere Grafiken, die Ihrer Fantasie auf die Sprünge helfen sollen.

Hier können Sie sich beweisen! Haben Sie eine coole neue Version des Spiels erstellt? Kommen Sie ins Head First C#-Forum und stellen Sie sie vor: www.headfirstlabs.com/books/hfcsharp/

9 Dateien lesen und schreiben

Speichere das Array, rette die Welt



Gelegentlich zahlt es sich aus, wenn man Dinge festhält.

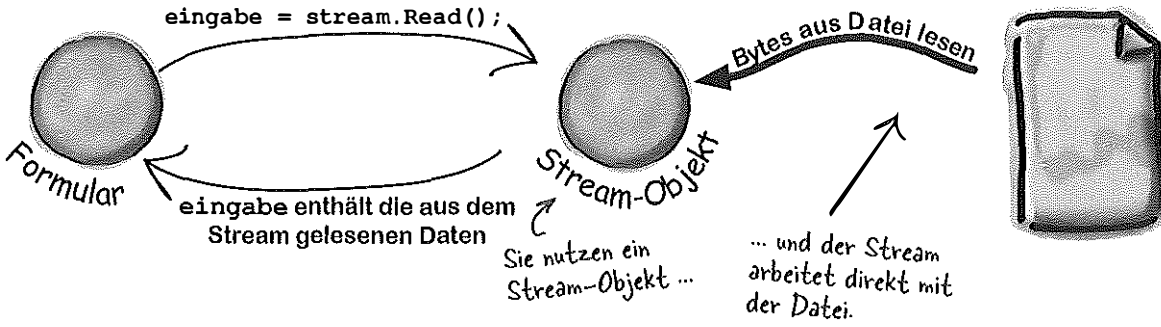
Bisher hatten alle Ihre Programme nur ein Kurzzeitgedächtnis. Sie starten, laufen ein Weile und enden dann. Manchmal reicht das nicht aus, insbesondere wenn Sie mit wichtigen Daten arbeiten. Sie müssen **Ihre Arbeit speichern** können. In diesem Kapitel werden wir uns ansehen, **wie man Daten in einer Datei speichert** und diese **Informationen dann wieder** aus einer Datei **einliest**. Sie werden etwas über die .NET-**Stream-Klassen** lernen und sich mit den Mysterien **Hexadezimal** und **Binär** auseinandersetzen.

C# nutzt Streams, um Daten zu lesen und zu schreiben

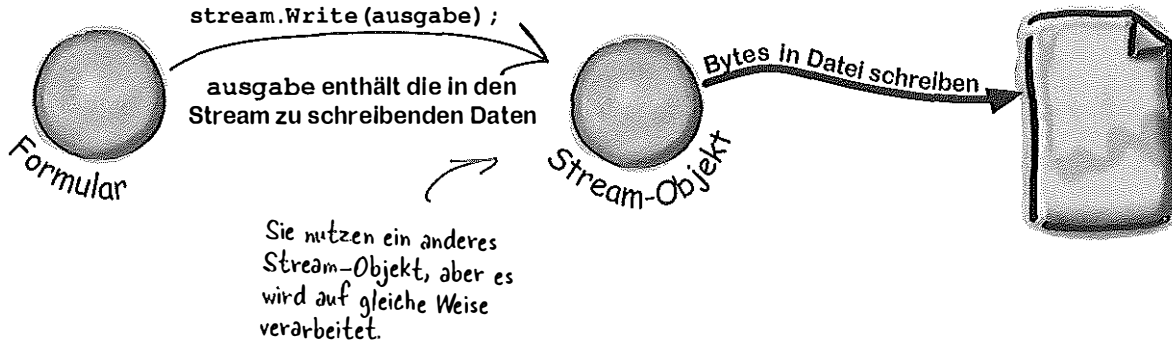
Über **Streams** (Ströme) bringen Sie im .NET Framework Daten in Ihr Programm hinein oder aus Ihrem Programm heraus. Jedes Mal, wenn Ihr C#-Programm eine Datei liest oder schreibt, sich über ein Netzwerk mit einem anderen Computer verbindet oder überhaupt irgendetwas macht, bei dem **Bytes empfangen oder versendet** werden, nutzen Sie Streams.

Wenn Sie Daten aus einer Datei lesen oder in eine Datei schreiben, nutzen Sie ein Stream-Objekt.

Nehmen wir an, Sie haben ein einfaches Programm – ein Formular mit einem Event-Handler, der Daten aus einer Datei lesen muss.

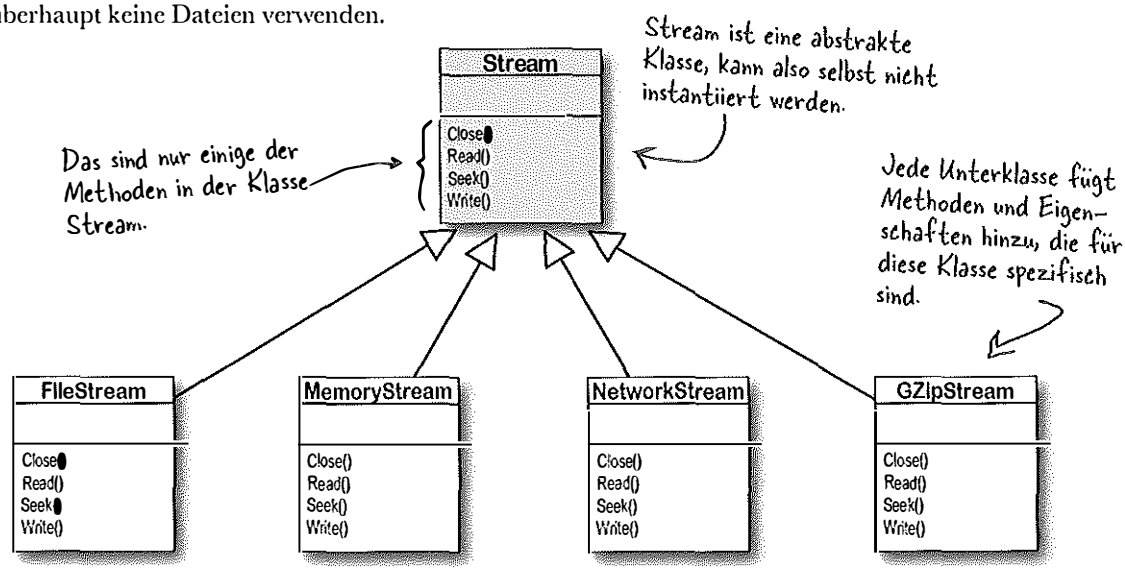


Und wenn Ihr Programm Daten in die Datei schreiben muss, kann es ein weiteres Stream-Objekt verwenden.



Unterschiedliche Streams lesen und schreiben andere Dinge

Jeder Stream ist eine Unterklasse der abstrakten Klasse **`Stream`** (daher der Name), und es gibt eine Menge eingebauter Stream-Klassen für die verschiedensten Aufgaben. Wir werden uns vorwiegend mit dem Lesen und Schreiben von gewöhnlichen Dateien beschäftigen, aber alles in diesem Kapitel lässt sich genauso gut auf komprimierte oder verschlüsselte Dateien anwenden oder Netzwerk-Streams, die überhaupt keine Dateien verwenden.



Mit `FileStream` können Sie aus Dateien lesen und in Dateien schreiben.

Mit `MemoryStream` können Sie Daten aus dem Speicher lesen und in ihn schreiben.

Mit `NetworkStream` können Sie Daten an andere Computer bzw. Geräte in einem Netzwerk senden und Daten von ihnen empfangen.

Mit `GZipStream` können Sie Daten komprimieren, damit sie weniger Platz benötigen und leichter gespeichert und heruntergeladen werden können.

Was Sie mit einem Stream machen können:

1. **In den Stream schreiben.**
   Sie können Ihre Daten in einen Stream schreiben und verwenden dazu die `Write()`-Methode des Streams.

2. **Aus dem Stream lesen.**
   Mit der Methode `Read()` können Sie über einen Stream Daten aus einer Datei, einem Netzwerk, dem Speicher oder irgendetwas anderem lesen.

3. **Die Position im Stream ändern.**
   Die meisten Streams unterstützen eine `Seek()`-Methode, mit der Sie eine Position im Stream finden können, damit Sie Daten an einem bestimmten Punkt einfügen können.

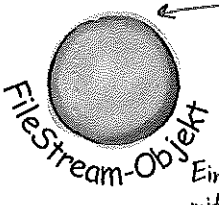
Mit Streams können Sie Daten lesen und schreiben. Nutzen Sie die richtige Art von Stream für Ihre Daten.

Ein FileStream schreibt Bytes in eine Datei

Muss Ihr Programm ein paar Zeilen Text in eine Datei schreiben, müssen diverse Dinge passieren:

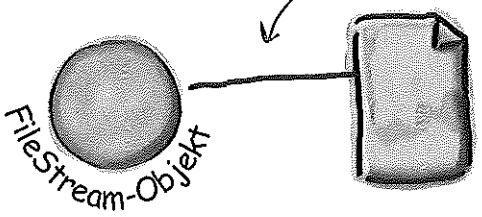
Achten Sie darauf, dass Sie jedem Programm, das Streams verwendet, `using System.IO;` hinzufügen.

1. Erstellen Sie ein FileStream-Objekt und sagen Sie ihm, dass es in die Datei schreiben soll.



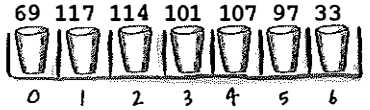
Ein FileStream kann immer nur mit einer Datei verbunden sein.

2. Der FileStream verknüpft sich mit einer Datei.

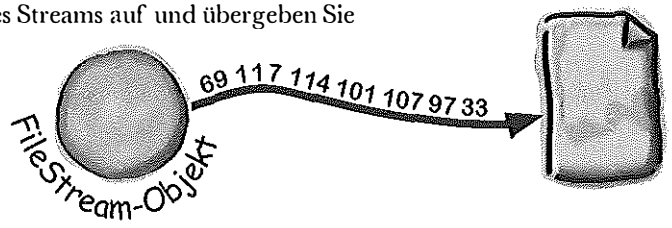


3. Streams schreiben Bytes in Dateien. Sie müssen den String, den Sie in die Datei schreiben möchten, also in ein `byte`-Array umwandeln.

Das bezeichnet man als kodieren. Damit werden wir uns später noch mehr befassen.

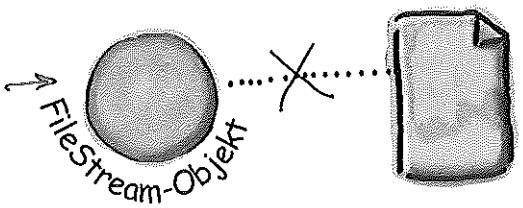
Eureka! → 

4. Rufen Sie die `Write()`-Methode des Streams auf und übergeben Sie ihr das `byte`-Array.



5. Schließen Sie die Datei, damit andere Programme darauf zugreifen können.

Es ist sehr wichtig, dass Sie den Stream schließen. Andernfalls bleibt die Datei gesperrt. Andere Programme können sie nicht verwenden, solange Sie den Stream nicht geschlossen haben.



Wie man Daten in eine Textdatei schreibt

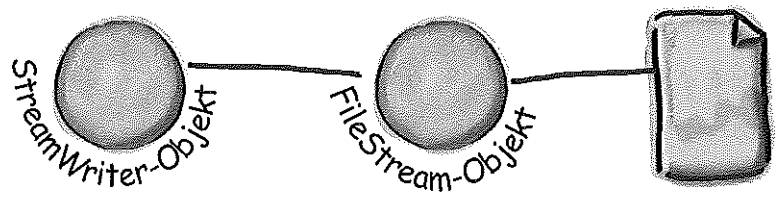
C# bietet die praktische Klasse **StreamWriter**, die all diese Dinge in einem einzelnen Schritt erledigt. Sie müssen nur ein `StreamWriter`-Objekt erstellen und ihm einen Dateinamen geben. Es erzeugt dann **automatisch** einen `FileStream` und verknüpft ihn mit der Datei. Dann können Sie die `Write()`- und `WriteLine()`-Methoden des StreamWriters verwenden, um die gewünschten Daten in die Datei zu schreiben.

StreamWriter erzeugt und verwaltet automatisch ein FileStream-Objekt für Sie.

1 Erstellen oder öffnen Sie eine Datei mit dem StreamWriter-Konstruktor.

Sie können dem `StreamWriter`-Konstruktor einen Dateinamen übergeben. Der StreamWriter öffnet dann automatisch die Datei. Eine Überladung von `StreamWriter` nimmt zusätzlich einen `bool`: Dieser sollte true sein, wenn Sie ans Ende einer bestehenden Datei schreiben wollen, oder false, wenn die bestehende Datei gelöscht und eine neue Datei gleichen Namens erstellt werden soll.

```
StreamWriter writer = new StreamWriter(@"C:\NeueDateien\Pizza backen.txt", true);
```

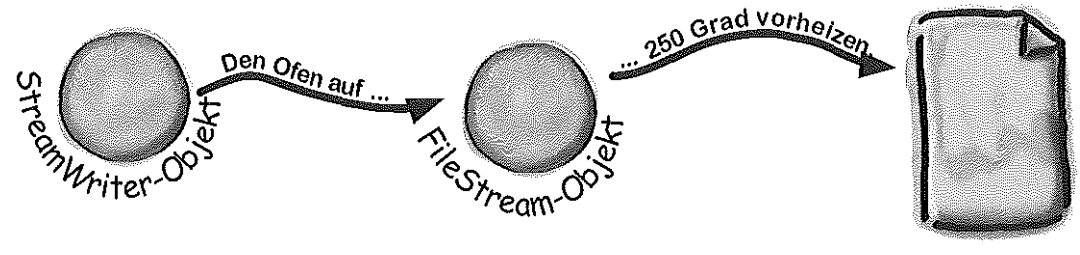


Stellen Sie einem String @ voran, sagt das C#, dass in ihm keine Escape-Zeichen wie \t für einen Tabulator oder \n für einen Zeilenumbruch berücksichtigt werden sollen.

2 Die Methoden Write() und WriteLine() schreiben in die Datei.

Diese Methoden funktionieren genau so wie die in `Console`: `Write()` schreibt Text, `WriteLine()` schreibt Text und hängt ans Ende einen Zeilenumbruch an. Schließen Sie in den String »{0}«, »{1}«, »{2}« usw. ein, schließt die Methode weitere übergebene Parameter in den String ein: »{0}« wird durch den ersten Parameter nach dem String, »{1}« durch den zweiten ersetzt usw.

```
writer.WriteLine("Den {0} auf {1} vorheizen.", gerät, temp.ToString());
```



3 Rufen Sie dann die Methode Close() auf, um die Datei freizugeben.

Lassen Sie den Stream offen und mit der Datei verbunden, hält er die Datei gesperrt. Kein anderes Programm kann sie dann verwenden. Achten Sie also darauf, dass Sie Ihre Dateien immer schließen!

```
writer.Close();
```

Der nächste teuflische Plan des Fieslings

Die Bürger von Objekthausen leben seit Langem in Angst vor dem Fiesling. Jetzt setzt er einen `StreamWriter` ein, um einen weiteren bösen Plan umzusetzen. Schauen wir uns an, was da abläuft:

Diese Zeile erzeugt ein StreamWriter-Objekt und sagt diesem, wo die Datei ist.

Der Pfad-String beginnt mit einem @-Zeichen, damit das >>\<< nicht als eine Escape-Sequenz interpretiert wird.

```
StreamWriter sw = new StreamWriter(@"c:\geheim_plan.txt");
sw.WriteLine("Wie ich Captain Amazing vernichten werde");
sw.WriteLine("Der nächste geniale Fiesling-Plan");
sw.Write("Ich werde eine Klonarmee aufbauen und auf ");
sw.WriteLine("die Bürger von Objekthausen loslassen.");
string ort = "das Einkaufszentrum";
for (int nummer = 0; nummer <= 6; nummer++){
    sw.WriteLine("Klon {0} greift {1} an", nummer, ort);
    if (ort == "das Einkaufszentrum") { ort = "die Innenstadt"; }
    else { ort = "das Einkaufszentrum"; }
}
sw.Close();
```

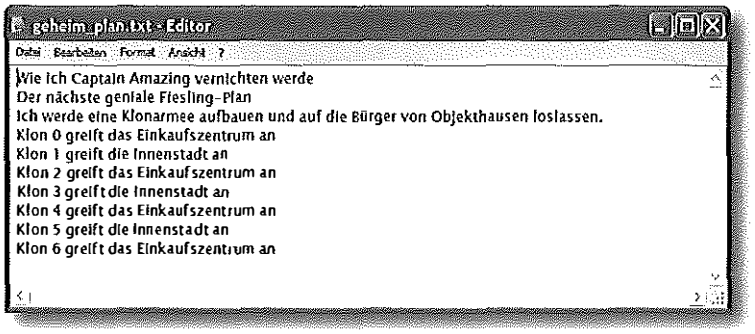
WriteLine() hängt an den Text einen Zeilenumbruch an. Write() sendet einfach den Text, ohne einen Zeilenumbruch anzuhängen.

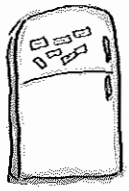
Sehen Sie, was in diesem Code mit der Variablen ort passiert?

Sie können im Text {} verwenden, um Variablen in den zu schreibenden Text zu interpolieren. {0} wird durch den ersten Parameter nach dem String ersetzt, {1} durch den zweiten und so weiter.

Close() gibt alle Verbindungen zu der Datei und Ressourcen für den StreamWriter frei. Der Text wird nicht geschrieben, wenn Sie den Stream nicht schließen.

Was der Code oben erzeugt.





StreamWriter-Magneten

Sie haben unten den button1_Click()-Code. Bauen Sie mit den Magneten den Code für die Klasse Flobbo auf, der bewirkt, dass ein Klick auf den Event-Handler die unten gezeigte Ausgabe erzeugt. Viel Glück!

```
private void button1_Click(object sender, EventArgs e) {
    Flobbo f = new Flobbo("blau gelb");
    StreamWriter sw = f.Snobbo();
    f.Blobbo(f.Blobbo(f.Blobbo(sw), sw), sw);
}
```

```
sw.WriteLine(Zap);
Zap = "rot orange";
return true;
```

```
}
```

```
sw.WriteLine(Zap);
sw.Close();
return false;
```

```
public bool Blobbo
    (bool fertig, StreamWriter sw) {
```

```
public bool Blobbo(StreamWriter sw) {
```

```
sw.WriteLine(Zap);
Zap = "grün pink";
return false;
```

```
}
```

```
return new
    StreamWriter("macaw.txt");
```

```
}
```

```
}
```

```
}
```

```
private string Zap;

public Flobbo(string Zap) {
    this.Zap = Zap;
}
```

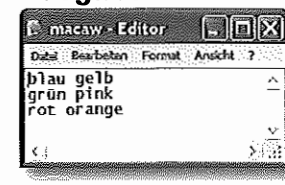
```
public class Flobbo {
```

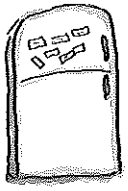
```
if (fertig) {
```

```
} else {
```

```
public StreamWriter Snobbo() {
```

Ausgabe:





StreamWriter-Magneten, Lösung

Sie sollten aus den Magneten die Klasse Flobbo erstellen, um die gewünschte Ausgabe zu erzeugen.

```
private void button1_Click(object sender, EventArgs e) {
    Flobbo f = new Flobbo("blau gelb");
    StreamWriter sw = f.Snobbo();
    f.Blobbo(f.Blobbo(f.Blobbo(sw), sw), sw);
}
```

```
public class Flobbo {
    private string Zap;

    public Flobbo(string Zap) {
        this.Zap = Zap;
    }

    public StreamWriter Snobbo() {
        return new
            StreamWriter("macaw.txt");
    }

    public bool Blobbo(StreamWriter sw) {
        sw.WriteLine(Zap);
        Zap = "grün pink";
        return false;
    }

    public bool Blobbo
        (bool fertig, StreamWriter sw) {
        if (fertig) {
            sw.WriteLine(Zap);
            sw.Close();
            return false;
        } else {
            sw.WriteLine(Zap);
            Zap = "rot orange";
            return true;
        }
    }
}
```

Die Methode Blobbo() ist überladen – sie hat zwei Deklarationen mit verschiedenen Parametern.

Achten Sie darauf, dass Sie die Datei schließen, wenn Sie fertig sind.

Ausgabe:



Zum Lesen und Schreiben braucht man zwei Objekte

Lesen wir den Plan des Fieslings mit einem anderen Stream, einem `StreamReader`. `StreamReader` funktioniert genau wie ein `StreamWriter`, aber statt in eine Datei zu schreiben, übergeben Sie dem Reader im Konstruktor den Namen einer zu lesenden Datei. Die Methode `ReadLine()` liefert einen String, der die nächste Zeile aus der Datei enthält. Sie können eine Schleife schreiben, die Zeilen aus der Datei liest, bis die Eigenschaft `EndOfStream` true ist – dann gibt es keine Zeilen mehr zu lesen.

```
StreamReader reader =
    new StreamReader(@"c:\geheim_plan.txt");
StreamWriter writer =
    new StreamWriter(@"e:\emailAnCaptainAmazing.txt");
writer.WriteLine("To: CaptainAmazing@objekthausen.net");
writer.WriteLine("From: Inspektor@objekthausen.net");
writer.WriteLine("Subject: Können Sie noch einmal die Welt retten?");
writer.WriteLine();
writer.WriteLine("Wir haben den neue Fiesling-Plan aufgedeckt:");
while (!reader.EndOfStream) {
  string zeileAusPlan = reader.ReadLine();
  writer.WriteLine("Der Plan -> " + zeileAusPlan);
}
writer.WriteLine();
writer.WriteLine("Können Sie uns helfen?");
writer.Close();
reader.Close();
```

Übergeben Sie dem StreamReader-Konstruktor den Namen der zu lesenden Datei.

Dieses Programm nutzt einen StreamReader, um den Plan des Fieslings zu lesen, und einen StreamWriter, um eine Datei zu schreiben, die an Captain Amazing gemailt wird.

Ein leerer WriteLine()-Aufruf schreibt eine leere Zeile.

EndOfStream ist die Eigenschaft, die Ihnen sagt, ob es noch ungelesene Daten gibt.

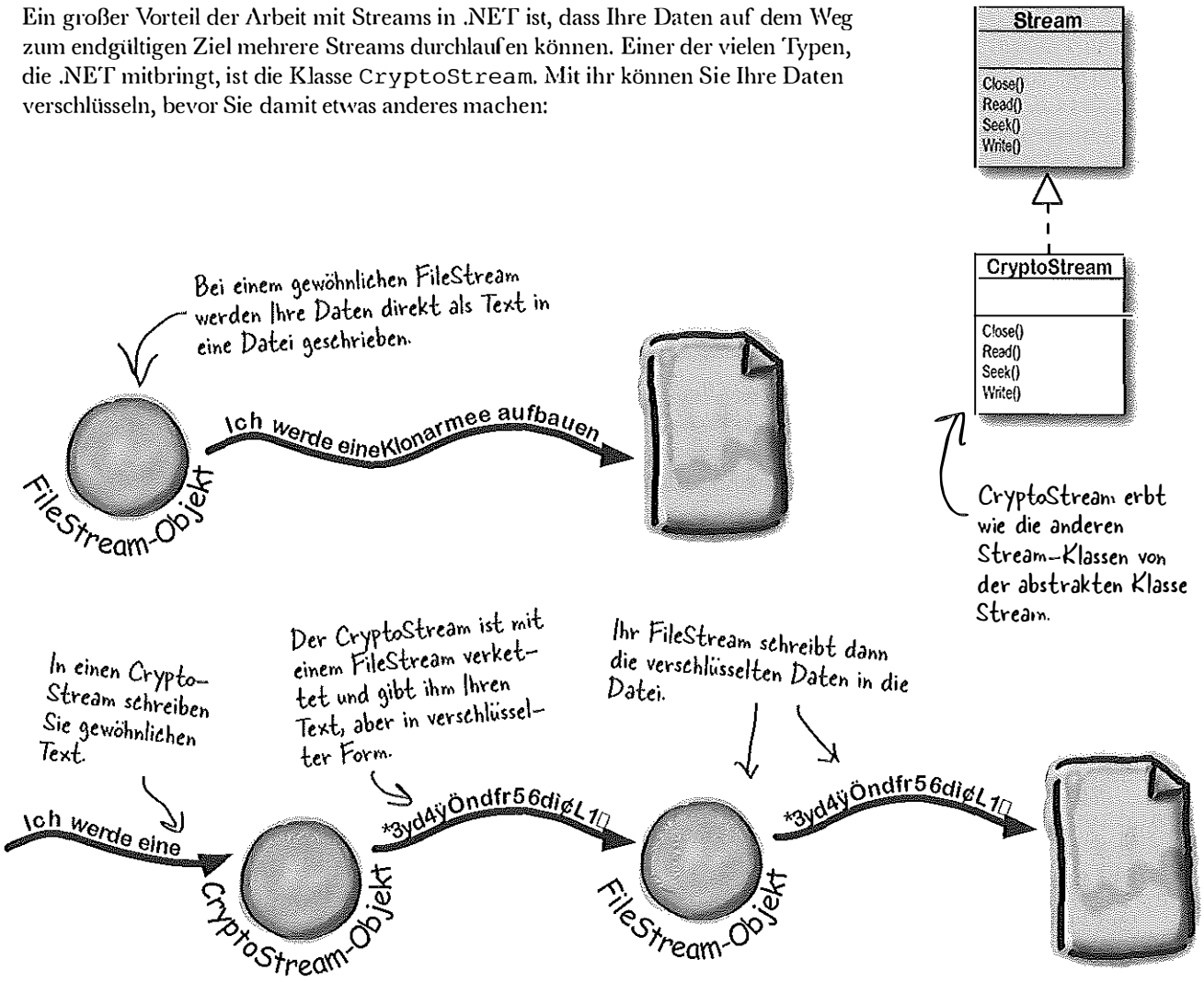
Diese Zeile liest jeweils eine Schleife aus dem Reader und schreibt sie in den Writer.

Achten Sie darauf, dass Sie jeden Stream schließen, den Sie öffnen.



Daten können durch mehrere Streams laufen

Ein großer Vorteil der Arbeit mit Streams in .NET ist, dass Ihre Daten auf dem Weg zum endgültigen Ziel mehrere Streams durchlaufen können. Einer der vielen Typen, die .NET mitbringt, ist die Klasse `CryptoStream`. Mit ihr können Sie Ihre Daten verschlüsseln, bevor Sie damit etwas anderes machen:



Streams können VERKETTET werden. Ein Stream kann in einen anderen Stream schreiben, der wieder in einen anderen Stream schreibt ... das endet irgendwann in einem Netzwerk oder einer Datei.

Pool-Puzzle



Sie haben die ***Aufgabe***, die leeren Zeilen im Code mit den Codeschnipseln aus dem Pool zu füllen. Einzelne Schnipsel können mehrfach verwendet werden, und Sie werden nicht alle Schnipsel benötigen. Das ***Ziel*** ist es, eine Klasse zu erstellen, die sich kompilieren lässt und die gezeigte Ausgabe erzeugt.

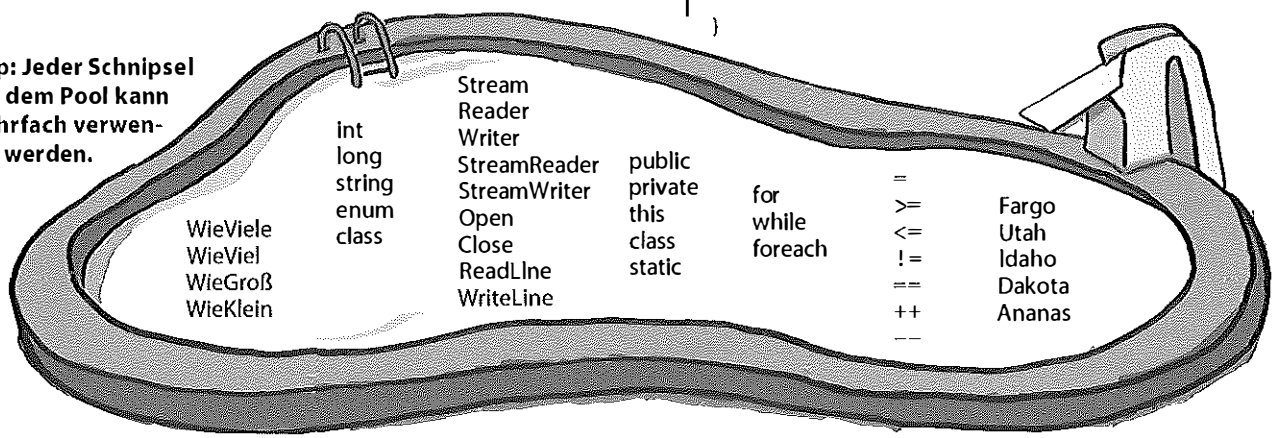


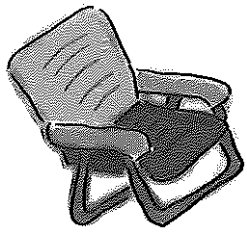

```
public class Ananas {
  const _______ l = "lieferung.txt";
  public ______  _______
        { Norden, Süden, Osten, Westen, Flamingo }
  public static void Main() {
    __________ a = new ____________("auftrag.txt");
    Pizza pz = new Pizza(new ____________(l, true));
    pz._________(Fargo.Flamingo);
    for (______ w = 3; w >= 0; w--) {
      Pizza i = new Pizza
            (new ____________(l, false));
      i.Idaho((Fargo)w);
      Party p = new Party(new ____________(l));
      p._____________(a);
    }
    a._____________("Das wars schon!!");
    a._____________();
  }
}
```

```
public class Pizza {
  private _____________ _______;
  public Pizza(____________ _______) {
    _______.Writer = Writer;
  }
  public void _______(_______.Fargo f) {
    Writer._____________(f.ToString());
    Writer._____________();
  }
}
```

```
public class Party {
  private _____________ Reader;
  public Party(_____________ Reader) {
    ____________.Reader = Reader;
  }
  public void WieViel(____________ m) {
    m._____________(Reader._____________());
    Reader._____________();
  }
}
```

Tipp: Jeder Schnipsel aus dem Pool kann mehrfach verwendet werden.





Pool-Puzzle, Lösung

Dieses Enum wird mit seiner ToString()-Methode verwendet, um einen Großteil der Ausgabe zu erzeugen.

```
public class Ananas {
    const string l = "lieferung.txt";
    public enum Fargo { Norden, Süden, Osten, Westen, Flamingo }
    public static void Main() {
        StreamWriter a = new StreamWriter("auftrag.txt");
        Pizza pz = new Pizza(new StreamWriter(l, true));
        pz.Idaho(Fargo.Flamingo);
        for (int w = 3; w >= 0; w--) {
            Pizza i = new Pizza(new StreamWriter(l, false));
            i.Idaho((Fargo)w);
            Party p = new Party(new StreamReader(l));
            p.WieViel(a);
        }
        a.WriteLine("Das wars schon!!");
        a.Close();
    }
}

public class Pizza {
    private StreamWriter Writer;
    public Pizza(StreamWriter Writer) {
        this.Writer = Writer;
    }
    public void Idaho(Ananas.Fargo f) {
        Writer.WriteLine(f.ToString());
        Writer.Close();
    }
}

public class Party {
    private StreamReader Reader;
    public Party(StreamReader Reader) {
        this.Reader = Reader;
    }
    public void WieViel(StreamWriter m) {
        m.WriteLine(Reader.ReadLine());
        Reader.Close();
    }
}
```

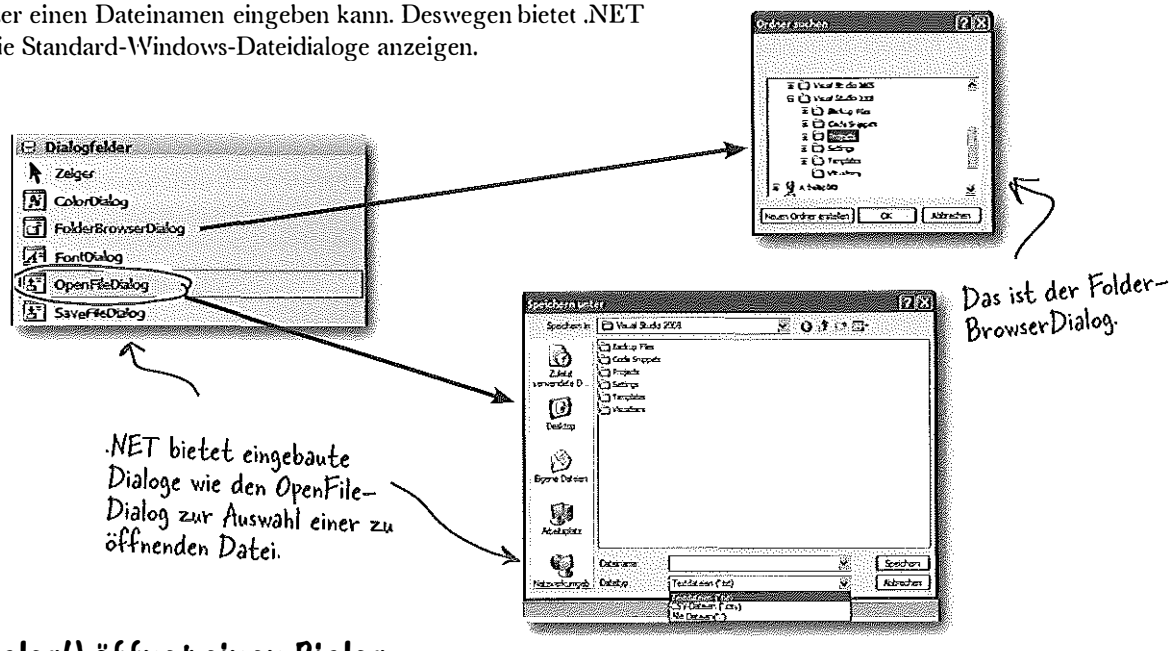
Hier ist der Einstiegspunkt für das gesamte Programm. Es wird StreamWriter erzeugt, der an die Klasse Party übergeben wird. Dann werden die Fargo-Member durchlaufen und zur Ausgabe an die Methode Pizza.Idaho() übergeben.

Die Klasse Pizza hält einen StreamWriter als ein privates Feld, und ihre Idaho()-Methode schreibt über die Methode ToString()Fargo-Werte in die Datei.

Die Klasse Party hat ein StreamReader-Feld, und ihre WieViel()-Methode liest eine Zeile aus dem StreamReader und schreibt sie in einen StreamWriter.

Öffnen Sie mit eingebauten Objekten Standarddialoge

Wenn Sie an einem Programm arbeiten, das Dateien liest und schreibt, müssen Sie eventuell irgendwann einen Dialog einblenden, über den der Benutzer einen Dateinamen eingeben kann. Deswegen bietet .NET Objekte, die Standard-Windows-Dateidialoge anzeigen.



ShowDialog() öffnet einen Dialog

Diese Dialoge kann man leicht anzeigen lassen. Das ist alles, was Sie tun müssen:

Diese Schritte gehen wir in einer Minute durch.

1. Erzeugen Sie eine Instanz der Dialog-Klasse. Das können Sie mit `new` machen oder indem Sie das Element aus der Toolbox ins Formular ziehen.

2. Setzen Sie die Eigenschaften des Dialog-Objekts. Zu den nützlichen Eigenschaften zählen `Title` (setzt den Text in der Titelleiste), `InitialDirectory` (gibt das Ausgangsverzeichnis an) und `FileName` (bei Dialogen zum Öffnen und Schließen von Dateien).

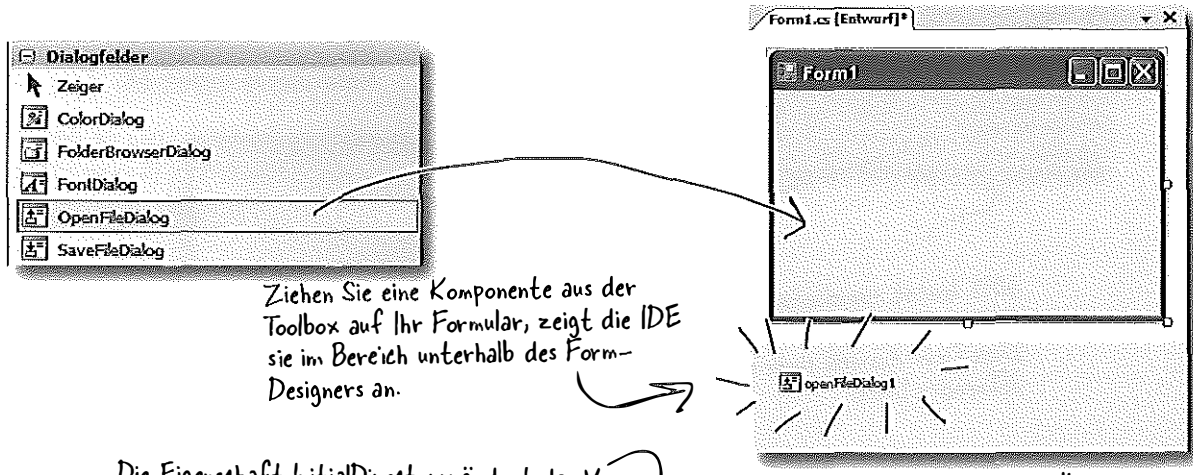
3. Rufen Sie die `ShowDialog()`-Methode des Objekts auf. Diese blendet den Dialog ein und kehrt erst zurück, nachdem der Benutzer auf den OK- oder den Abbrechen-Button geklickt hat oder das Fenster schließt.

4. Die Methode `ShowDialog()` liefert ein `DialogResult`. Das ist ein Enum. Einige ihrer Member sind `OK` (d.h., der Benutzer hat auf OK geklickt), `Cancel`, `Yes` und `No` (für Ja/Nein-Dialoge).

Dialoge sind einfach ein weiteres .NET-Steuerelement

Sie können die standardmäßigen Windows-Dateidialoge Ihrem Programm auch hinzufügen, indem Sie sie einfach auf Ihr Formular bewegen – ziehen Sie dazu ein `OpenFileDialog`-Steuerelement aus der Toolbox und lassen Sie es auf Ihr Formular fallen. Aber es wird nicht angezeigt wie ein sichtbares Steuerelement, sondern es erscheint in dem Raum unterhalb Ihres Formulars. Das liegt daran, dass der Dialog eine **Komponente** ist, das sind **nicht-visuelle Toolbox-Steuerelemente**, die im Formular nicht direkt erscheinen, darin aber genau so verwendet werden können wie gewöhnliche Steuerelemente.

>>Nicht-visuell<< bedeutet einfach, dass der Dialog nicht im Formular erscheint, wenn Sie ihn darauf ziehen.



Ziehen Sie eine Komponente aus der Toolbox auf Ihr Formular, zeigt die IDE sie im Bereich unterhalb des Form-Designers an.

Die Eigenschaft InitialDirectory ändert das Verzeichnis, das unmittelbar angezeigt wird, wenn der Dialog geöffnet wird.

Über die Eigenschaft Filter können Sie die Filter ändern, die unten im Dialog angezeigt werden, und die anzuzeigenden Dateitypen festlegen.

```
openFileDialog1.InitialDirectory = @"c:\MeinOrdner\Default\";
openFileDialog1.Filter = "Textdateien (*.txt)|*.txt|"
  + "CSV-Dateien (*.csv)|*.csv|Alle Dateien(*.*)|*.*";
openFileDialog1.FileName = "standardtext.txt";
openFileDialog1.CheckFileExists = true;
openFileDialog1.CheckPathExists = false;
DialogResult ergebnis = openFileDialog1.ShowDialog();
if (ergebnis == DialogResult.OK){
    OpenSomeFile(openFileDialog1.FileName);
}
```

Diese Eigenschaften sagen dem Dialog, dass eine Fehlermeldung angezeigt werden soll, wenn der Benutzer versucht, eine nicht vorhandene Datei oder einen nicht vorhandenen Pfad zu öffnen.

Sie zeigen den Dialog mit der Methode ShowDialog() an, die ein DialogResult-Objekt zurückliefert. Das ist ein Enum, über das Sie prüfen können, ob der Benutz r auf den OK-Button geklickt hat. Es wird auf DialogResult.OK gesetzt, wenn der Benutzer auf OK geklickt hat, und auf DialogResult.Cancel, wenn er auf Abbrechen geklickt hat.

Dialoge sind auch Objekte

Ein **`OpenFileDialog`** zeigt den üblichen Windows »Öffnen«-Dialog an, und ein **`SaveFileDialog`** zeigt den »Speichern«-Dialog an. Sie können sie anzeigen, indem Sie Instanzen erstellen, auf denen Sie Eigenschaften setzen und die Methode `ShowDialog()` aufrufen. Die Methode `ShowDialog()` liefert einen `DialogResult`-Wert (das ist ein Enum, das verwendet wird, weil einige Dialoge mehr als zwei Buttons oder Ergebnisse haben und ein einfacher `bool`-Wert nicht ausreichen würde).

Ziehen Sie das Objekt für einen Speichern-Dialog aus der Toolbox auf das Formular, fügt die IDE einfach eine solche Zeile in die InitializeComponent()-Methode Ihres Formulars ein.

```
saveFileDialog1 = new SaveFileDialog();
saveFileDialog1.InitialDirectory = @"c:\MeinOrdner\Default\";
saveFileDialog1.Filter = "Textdateien (*.txt)|*.txt|"
  + "CVS-Dateien (*.csv)|*.csv|Alle Dateien (*.*)|*.*";
DialogResult ergebnis = saveFileDialog1.ShowDialog();
if (ergebnise == DialogResult.OK){
    SaveTheFile(saveFileDialog1.FileName);
}
```

Die Eigenschaften ShowDialog() und FileName funktionieren genau so wie in einem OpenFileDialog.

Ein SaveFileDialog öffnet den üblichen Windows-»Speichern als...«-Dialog.



Mit der Eigenschaft Title können Sie diesen Text ändern.

Die Methode ShowDialog() blendet den Dialog ein und öffnet den Ordner, der mit der Eigenschaft InitialDirectory angegeben wurde.

Die »Dateityp«-Liste geben Sie mit der Eigenschaft Filter vor.

Wählt der Benutzer eine Datei, wird der vollständige Pfad in der Eigenschaft FileName gespeichert.

Über das DialogResult, das die Methode ShowDialog() zurückliefert, können Sie ermitteln, auf welchen Button der Benutzer geklickt hat.

Mit den eingebauten Klassen File und Directory können Sie mit Dateien und Verzeichnissen arbeiten

Wie `StreamWriter` erzeugt die Klasse `File` für Sie einen Stream, über den Sie mit Dateien arbeiten können. Mit Ihren Methoden können Sie die wichtigsten Operationen durchführen, ohne dass Sie erst einen `FileStream` erzeugen müssen. Mit `Directory`-Objekten können Sie mit ganzen Verzeichnissen inklusive Dateien arbeiten. Sie können sie einsetzen, um ohne Aufwand Änderungen an der Struktur Ihrer Dateien vorzunehmen.

Dinge, die Sie mit einem File-Objekt tun können:

1. **Prüfen, ob sie existiert**
   Mit der Methode `Exists()` können Sie prüfen, ob eine Datei existiert. Sie liefert true, wenn die Datei existiert, und andernfalls false.

2. **Aus einer Datei lesen und in sie schreiben**
   Mit der Methode `OpenRead()` können Sie Daten aus einer Datei abrufen, mit den Methoden `Create()` oder `OpenWrite()` können Sie in eine Datei schreiben.

3. **Text an die Datei anhängen**
   Mit der Methode `AppendAllText()` können Sie Text an eine bereits erstellte Datei anhängen. Die Datei wird erstellt, wenn sie noch nicht besteht, wenn die Methode aufgerufen wird.

4. **Informationen zur Datei abrufen**
   Die Methoden `GetLastAccessTime()` und `GetLastWriteTime()` liefern den letzten Zugriffs- oder Veränderungszeitpunkt der Datei.

FileInfo ist wie File

Müssen Sie mit einer Datei viele Sachen machen, sollten Sie eine Instanz der Klasse FileInfo verwenden, statt mit den statischen Methoden von File zu arbeiten.

Die Klasse FileInfo macht so ungefähr alles, was auch die Klasse File macht, Sie müssen sie aber instantiieren, damit Sie sie verwenden können. Sie können Instanzen von FileInfo erstellen und ihre `Exists()`-Methode oder ihre `OpenRead()`-Methode auf genau die gleiche Weise verwenden.

Der Hauptunterschied ist, dass die Klasse File bei einer kleineren Anzahl von Operationen schneller ist und die Klasse FileInfo für umfangreichere Aufgaben besser geeignet ist.

Dinge, die Sie mit einem Verzeichnis tun können:

1. **Ein neues Verzeichnis erstellen**
   Mit der Methode `CreateDirectory()` erstellen Sie ein neues Verzeichnis. Dazu müssen Sie nur den Pfad angeben. Die Methode macht für Sie den Rest.

2. **Eine Liste der Dateien in einem Verzeichnis abrufen**
   Mit der Methode `GetFiles()` können Sie ein Array mit den Dateien in einem Verzeichnis abrufen. Nennen Sie ihr einfach das Verzeichnis, zu dem Sie Informationen wünschen, dann macht sie sich für Sie an die Arbeit.

3. **Ein Verzeichnis löschen**
   Verzeichnisse zu löschen ist auch recht einfach. Nutzen Sie die Methode `Delete()`.

Es gibt keine Dummen Fragen

F: Das mit den {0}- und {1}-Klamotten bei dem StreamWriter verstehe ich noch nicht.

A: Geben Sie Strings in eine Datei aus, müssen Sie oft den Inhalt mehrerer Variablen ausgeben. Beispielsweise könnten Sie Folgendes schreiben müssen:

```
writer.Write("Mein Name ist" + name +
    " und ich bin " + alter + "Jahre");
```

Das Kombinieren von Strings mit + ist lästig und fehleranfällig. Leichter ist es, wenn man {0} und {1} nutzt:

```
writer.Write(
  "Mein Name ist {0} und ich bin {1} Jahre",
   name, alter);
```

Das ist besser lesbar, insbesondere wenn man in Code mehrere Variablen einschließen muss.

F: Warum haben Sie dem String mit dem Dateinamen das @ vorangestellt?

A: Bestimmte Zeichen werden in Strings über Escape-Sequenzen wie `\n` und `\t` dargestellt. Bei der Eingabe von Dateinamen führt das zu Schwierigkeiten, weil sie häufig viele Backslash-Zeichen enthalten. Stellen Sie einem String @ voran, sagt das C#, dass Escape-Sequenzen nicht interpretiert werden sollen. Sie können in Strings dann sogar Zeilenumbrüche eingeben, die in die Ausgabe übernommen werden:

```
string zweiZeilen = @"Das ist ein
zweizeiliger String.";
```

F: Und was heißen \n oder \t?

A: Das sind Escape-Sequenzen. `\n` ist ein Newline-Zeichen und `\t` ist ein Tabulator. `\r` ist ein Zeilenumbruchzeichen, die Hälfte eines Zeilenumbruchs unter Windows – in Windows-Textdateien enden Zeilen mit `\r\n`. Wenn Sie in Ihrem String *tatsächlich* einen Backslash verwenden wollen, der von C# nicht als Anfang einer Escape-Sequenz interpretiert wird, müssen Sie den Backslash ***verdoppeln***: `\\`.

F: Am Anfang des Kapitels haben Sie von der Umwandlung eines Strings in ein byte-Array gesprochen. Wie kann denn das funktionieren?

A: Wahrscheinlich haben Sie schon tausendmal gehört, dass Dateien auf der Festplatte durch Bits und Bytes dargestellt werden. Das bedeutet, dass das Betriebssystem eine Datei, die Sie auf die Festplatte schreiben, als eine lange Folge von Bytes betrachtet. Erinnern Sie sich noch, dass wir in Kapitel 4 gelernt haben, dass eine `byte`-Variable eine Zahl zwischen 0 und 255 speichern kann? Jede Datei auf Ihrer Festplatte ist eine lange Folge von Zahlen zwischen 0 und 255. Die Programme, die diese Dateien lesen und schreiben, müssen diese Bytes wieder zu Daten mit einer Bedeutung machen. Öffnen Sie eine Datei im Editor, wandelt er alle Bytes in Zeichen um – aus 69 wird beispielsweise `E`, aus 97 `a`. Geben Sie im Editor Text ein, wandelt er diese Zeichen beim Speichern in Bytes um. Wenn Sie einen String in einen Stream schreiben, müssen Sie das Gleiche tun.

F: Wenn ich einfach mit einem StreamWriter in eine Datei schreiben kann, was interessiert es mich dann, dass der für mich einen FileStream erstellt?

A: Wenn Sie nichts anderes machen, als sequenziell in eine Datei zu schreiben oder aus einer zu lesen, brauchen Sie nur einen `StreamReader` oder `StreamWriter`. Aber sobald Sie komplexere Dinge machen wollen, müssen Sie mit anderen Streams arbeiten. Wenn Sie irgendwann einmal Daten wie Zahlen, Arrays, Collections oder Objekte in eine Datei schreiben müssen, reicht ein `StreamWriter` nicht mehr. Was genau Sie dann tun müssen, werden wir uns in wenigen Augenblicken ansehen.

F: Was ist, wenn ich eigene Dialoge erstellen möchte?

A: Das können Sie natürlich machen. Sie können Ihrem Projekt ein neues Formular hinzufügen und ihm das gewünschte Aussehen verleihen. Dann instantiieren Sie es mit `new` (genau so, wie Sie das `OpenFileDialog`-Objekt erstellt haben). Rufen Sie seine `ShowDialog()`-Methode auf, funktioniert es wie ein gewöhnlicher Dialog. In Kapitel 13 werden wir uns ausführlicher ansehen, wie man einem Programm weitere Formulare hinzufügt.

F: Warum muss ich mich darum kümmern, dass Streams geschlossen werden, wenn ich sie nicht mehr brauche?

A: Hat Ihnen schon einmal ein Programm gesagt, dass es eine Datei nicht öffnen könne, weil diese »beschäftigt« sei? Windows sperrt Dateien, wenn sie von einem Programm verwendet werden, damit andere Programme sie nicht verwenden können. Rufen Sie `Close()` nicht auf, kann es passieren, dass Ihr Programm eine Datei offen hält, bis es selbst beendet wird.



.NET bietet zwei eingebaute Klassen mit einem Haufen statischer Methoden zur Arbeit mit Dateien und Ordnern. Die Klasse **`File`** bietet Methoden zur Arbeit mit Dateien. Die Klasse **`Directory`** lässt Sie mit Verzeichnissen arbeiten. Schreiben Sie auf, was die folgenden Codezeilen machen.

Code	Was der Code macht

`if (!Directory.Exists(@"c:\SYP")) {`
`  Directory.CreateDirectory(@"c:\SYP");`
`}`	

`if (Directory.Exists(@"c:\SYP\Bonk")) {`
`  Directory.Delete(@"c:\SYP\Bonk");`
`}`	

`Directory.CreateDirectory(@"c:\SYP\Bonk");`	
`Directory.SetCreationTime(@"c:\SYP\Bonk",` `  new DateTime(1976, 09, 25));`	

`string[] dateien =`
`  Directory.GetFiles(@"c:\Windows\",`
`  "*.log", SearchOption.AllDirectories);`	

`File.WriteAllText(@"c:\SYP\Bonk\komisch.txt",`
`  @"Das ist die erste Zeile`
`und das ist die zweite Zeile`
`und das ist die letzte Zeile");`	

`File.Encrypt(@"c:\SYP\Bonk\komisch.txt");` Versuchen Sie zu erraten, was diese Zeile macht – das ist Ihnen noch nicht begegnet.	
`File.Copy(@"c:\SYP\Bonk\komisch.txt",` `  @"c:\SYP\kopie.txt");`	
`DateTime dieZeit =` `  Directory.GetCreationTime(@"c:\SYP\Bonk");`	
`File.SetLastWriteTime(@"c:\SYP\kopie.txt",` `  dieZeit);`	
`File.Delete(@"c:\SYP\Bonk\komisch.txt");`	

Öffnen und speichern Sie Dateien mit Dateidialogen (und das mit nur ein paar Codezeilen)

Sie können ein Programm erstellen, das eine Textdatei öffnet. Mit nur wenigen Zeilen Code und standardmäßigen .NET-Steuerelementen ermöglicht es Ihnen, Änderungen an der Datei vorzunehmen. Das geht so:

Tun Sie das!



1 **Erstellen Sie ein einfaches Formular.**
Sie brauchen nur eine TextBox und zwei Buttons. Ziehen Sie auch jeweils ein OpenFileDialog- und SaveFileDialog-Steuerelement auf das Formular. Klicken Sie doppelt auf die Buttons, um ihre Event-Handler zu erzeugen, und **fügen Sie dem Formular ein privates String-Feld namens name hinzu**. Vergessen Sie nicht, am Anfang der Datei die `using`-Anweisung für `System.IO` hinzuzufügen.

2 **Verbinden Sie den Öffnen-Button mit dem OpenFileDialog.**
Der Öffnen-Button zeigt einen OpenFileDialog an und nutzt File.ReadAllText(), um den Inhalt der Datei in das Textfeld einzulesen:

```
private void öffnen_Click(object sender, EventArgs e) {
  if (openFileDialog1.ShowDialog() == DialogResult.OK) {
    name = openFileDialog1.FileName;
    textBox1.Clear();
    textBox1.Text = File.ReadAllText(name);
  }
}
```

Ein Klick auf Öffnen zeigt das OpenFileDialog-Steuerelement an.

3 **Richten Sie jetzt den Speichern-Button ein.**
Der Speichern-Button speichert die Datei mit der Methode File.WriteAllText():

```
private void speichern_Click(object sender, EventArgs e) {
  if (saveFileDialog1.ShowDialog() == DialogResult.OK) {
    name = saveFileDialog1.FileName;
    File.WriteAllText(name, textBox1.Text);
  }
}
```

Die Methoden ReadAllText() und WriteAllText() sind Teile der Klasse File. Diese ist Ihnen bereits begegnet und wird auf den folgenden Seiten noch ausführlicher betrachtet.

4 **Spielen Sie mit den anderen Eigenschaften der Dialoge.**

- ★ Nutzen Sie die Title-Eigenschaft des SaveFileDialog, um den Text in der Titelleiste zu ändern.
- ★ Setzen Sie die InitialFolder-Eigenschaft so, dass der OpenFileDialog in einem bestimmten Verzeichnis beginnt.
- ★ Filtern Sie den OpenFileDialog mit der Eigenschaft Filter so, dass er nur Textdateien anzeigt.

Fügen Sie keinen Filter hinzu, bleibt die Aufklappliste unten in den Öffnen- und Speichern-Dialogen leer. Probieren Sie es mit diesem Filter: >>Textdateien (*.txt)|*.txt<<.



Spitzen Sie Ihren Bleistift
Lösung

.NET bietet zwei eingebaute Klassen mit einem Haufen statischer Methoden zur Arbeit mit Dateien und Ordnern. Die Klasse **`File`** bietet Methoden zur Arbeit mit Dateien. Die Klasse **`Directory`** lässt Sie mit Verzeichnissen arbeiten. Sie sollten aufschreiben, was die einzelnen Codezeilen bewirken.

Code	Was der Code macht

`if (!Directory.Exists(@"c:\SYP")) {`
`  Directory.CreateDirectory(@"c:\SYP");`
`}`
	Prüft, ob der Ordner C:\SYP existiert, und erstellt ihn, falls nicht.

`if (Directory.Exists(@"c:\SYP\Bonk")) {`
`  Directory.Delete(@"c:\SYP\Bonk");`
`}`
	Prüft, ob der Ordner C:\SYP\Bonk existiert, und löscht ihn gegebenenfalls.

`Directory.CreateDirectory(@"c:\SYP\Bonk");`	Erzeugt das Verzeichnis C:\SYP\Bonk.
`Directory.SetCreationTime(@"c:\SYP\Bonk",` `  new DateTime(1976, 09, 25));`	Setzt die Erstellungszeit des Ordners C:\SYP\Bonk auf den 25. September 1976.

`string[] dateien =`
`  Directory.GetFiles(@"c:\Windows\",`
`  "*.log", SearchOption.AllDirectories);`
	Ruft eine Liste aller Dateien in C:\Windows einschließlich aller Unterverzeichnisse ab, die dem Muster *.log entsprechen.

`File.WriteAllText(@"c:\SYP\Bonk\komisch.txt",`
`  @"Das ist die erste Zeile`
`und das ist die zweite Zeile`
`und das ist die letzte Zeile");`
	Erzeugt die Datei »komisch.txt« (falls sie noch nicht existiert) im Ordner C:\SYP\Bonk und schreibt drei Zeilen Text hinein.

`File.Encrypt(@"c:\SYP\Bonk\komisch.txt");` Das ist eine Alternative zu einem CryptoStream.	Nutzt die eingebaute Windows-Verschlüsselung, um die Datei »komisch.txt« zu verschlüsseln.
`File.Copy(@"c:\SYP\Bonk\komisch.txt",` `  @"c:\SYP\kopie.txt");`	Kopiert die Datei C:\SYP\Bonk\komisch.txt in die Datei C:\SYP\kopie.txt.
`DateTime dieZeit =` `  Directory.GetCreationTime(@"c:\SYP\Bonk");`	Deklariert die Variable dieZeit und setzt sie auf die Erstellungszeit des Ordners C:\SYP\Bonk.
`File.SetLastWriteTime(@"c:\SYP\kopie.txt",` `  dieZeit);`	Ändert die letzte Schreibzeit von kopie.txt in C:\SYP\ so, dass sie der in der Variablen dieZeit gespeicherten Zeit entspricht.
`File.Delete(@"c:\SYP\Bonk\komisch.txt");`	Löscht die Datei C:\SYP\Bonk\komisch.txt.

IDisposable sorgt dafür, dass Ihre Objekte entsorgt werden

Viele .NET-Klassen implementieren eine besonders nützliche Schnittstelle namens `IDisposable`. Sie **hat nur ein Member**: eine Methode namens **`Dispose()`**. Implementiert eine Klasse `IDisposable`, sagt Ihnen dies, dass es wichtige Aufgaben gibt, die erledigt werden müssen, wenn ein Objekt dieser Klasse gelöscht wird. In der Regel bedeutet das, dass sie **Ressourcen reserviert hat**, die sie erst zurückgibt, wenn das ausdrücklich verlangt wird. Mit der Methode `Dispose()` sagen Sie dem Objekt, wie diese Ressourcen freigegeben werden sollen.

Deklarieren Sie ein Objekt in einem using-Block, wird die Dispose()-Methode dieses Objekts automatisch aufgerufen.

Mit »Gehe zu Definition« (über die Sie später mehr erfahren) können Sie sich in der IDE die offizielle C#-Definition von `IDisposable` ansehen. Gehen Sie in Ihr Projekt und geben Sie irgendwo IDisposable ein. Klicken Sie dann mit der rechten Maustaste darauf und wählen Sie im Menü »Gehe zu Definition« . Es wird eine neue Registerkarte mit Code geöffnet. Expandieren Sie diesen Code, sehen Sie Folgendes:

```
namespace System
{
    // Zusammenfassung:
    //     Definiert eine Methode zur Freigabe von
    //     reservierten, nicht verwalteten Ressourcen.
    public interface IDisposable
    {
        // Zusammenfassung:
        //     Führt anwendungsspezifische Aufgaben durch,
        //     die mit der Freigabe, der Zurückgabe
        //     oder dem Zurücksetzen von nicht verwalteten
        //     Ressourcen zusammenhängen.
        void Dispose();
    }
}
```

Viele Klassen reservieren wichtige Ressourcen wie Speicher, Dateien und andere Objekte. Sie übernehmen diese und geben sie erst wieder frei, wenn Sie ihnen sagen, dass diese Ressourcen nicht mehr benötigt werden.

Jede Klasse, die IDisposable implementiert, gibt sofort alle reservierten Ressourcen frei, wenn ihre Dispose()-Methode aufgerufen wird. Das ist fast immer das Letzte, was Sie tun, wenn Sie ein Objekt nicht mehr benötigen.

re-ser-vie-ren, Verb
Ressourcen für einen bestimmten Zweck anfordern.
*Die Programmierer waren genervt, weil ihr Projektleiter alle Konferenzräume für ein sinnloses Seminar **reserviert** hatte.*

Gehe zu Definition

Das ist ein praktisches Feature der IDE, über das Sie automatisch zur Definition einer Variablen, eines Objekts oder einer Methode springen können. Klicken Sie mit der rechten Maustaste auf das Element und wählen >>Gehe zu Definition<<, springt sie automatisch zu dem Ort, an dem es definiert wird.

Vermeiden Sie Dateisystemfehler mit using-Anweisungen

Wir bestehen bereits das ganze Kapitel darauf, dass Sie Ihre **Streams schließen müssen**. Das liegt daran, dass einige der häufigsten Fehler, über die Programmierer bei der Arbeit mit Dateien stolpern, durch Streams verursacht werden, die nicht ordentlich geschlossen wurden. Aber C# gibt Ihnen ein wundervolles Mittel, mit dem Sie sicherstellen, dass das überhaupt nicht passiert: `IDisposable` und die Methode `Dispose()`. Wenn Sie **Ihren Stream-Code in eine using-Anweisung einhüllen**, schließt er automatisch alle Streams für Sie. Dazu müssen Sie **Ihre Stream-Referenz** einfach in einer using-Anweisung deklarieren, auf die ein Block Code (in geschweiften Klammern) folgt, der diese Referenz nutzt. Tun Sie das, ruft die using-Anweisung **automatisch die `Dispose()`-Methode des Streams auf**, wenn Sie mit der Ausführung des Codeblocks fertig ist. Das funktioniert so:

Diese >>using<<-Anweisungen sind anders als die, die Sie am Anfang Ihrer Dateien angeben.

... auf die ein Codeblock in geschweiften Klammern folgt.

Eine using-Anweisung enthält immer eine Objektdeklaration ...

```
using (StreamWriter sw = new StreamWriter("geheim_plan.txt")) {
    sw.WriteLine("Wie ich Captain Amazing vernichten werde");
    sw.WriteLine("Der nächste geniale Fiesling-Plan");
    sw.WriteLine("Ich werde eine Klonarmee");
}
```

Diese Anweisungen können das mit der using-Anweisung erzeugte Objekt verwenden wie jedes andere Objekt.

Endet die using-Anweisung, wird die Dispose()-Methode des verwendeten Objekts aufgerufen.

Hier ist das verwendete Objekt, der von sw referenzierte Log-Stream, der in der using-Anweisung deklariert wurde. Es wird also die Dispose()-Methode der Klasse Stream ausgeführt ... diese schließt den Stream.

Jeder Stream hat eine Dispose()-Methode, die den Stream schließt. Deklarieren Sie einen Stream in einer using-Anweisung, schließt er sich immer selbst!

Mehrere Objekte brauchen mehrere using-Anweisungen

Sie können using-Anweisungen aufeinanderstapeln – Sie brauchen keine weiteren geschweiften Klammern oder Einrückungen.

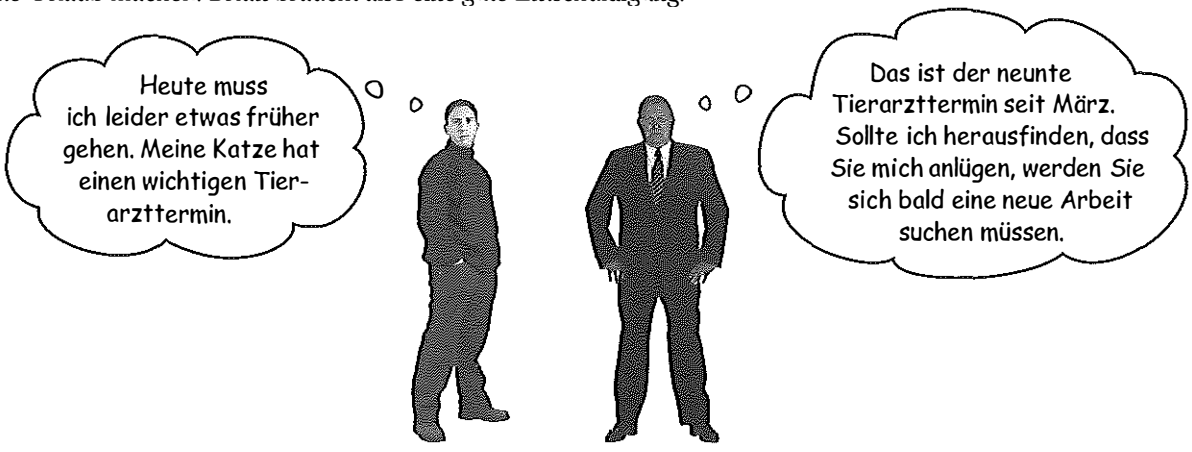
```
using (StreamReader reader = new StreamReader("geheim_plan.txt"));
using (StreamWriter writer = new StreamWriter("email.txt"));
{
    // Anweisungen, die reader und writer verwenden.
}
```

Jetzt müssen Sie auf den Streams nicht mehr Close() aufrufen, weil die using-Anweisung sie automatisch schließt.

Wenn Sie Streams nutzen, sollten Sie diese IMMER in einer using-Anweisung deklarieren. Das sichert, dass sie geschlossen werden.

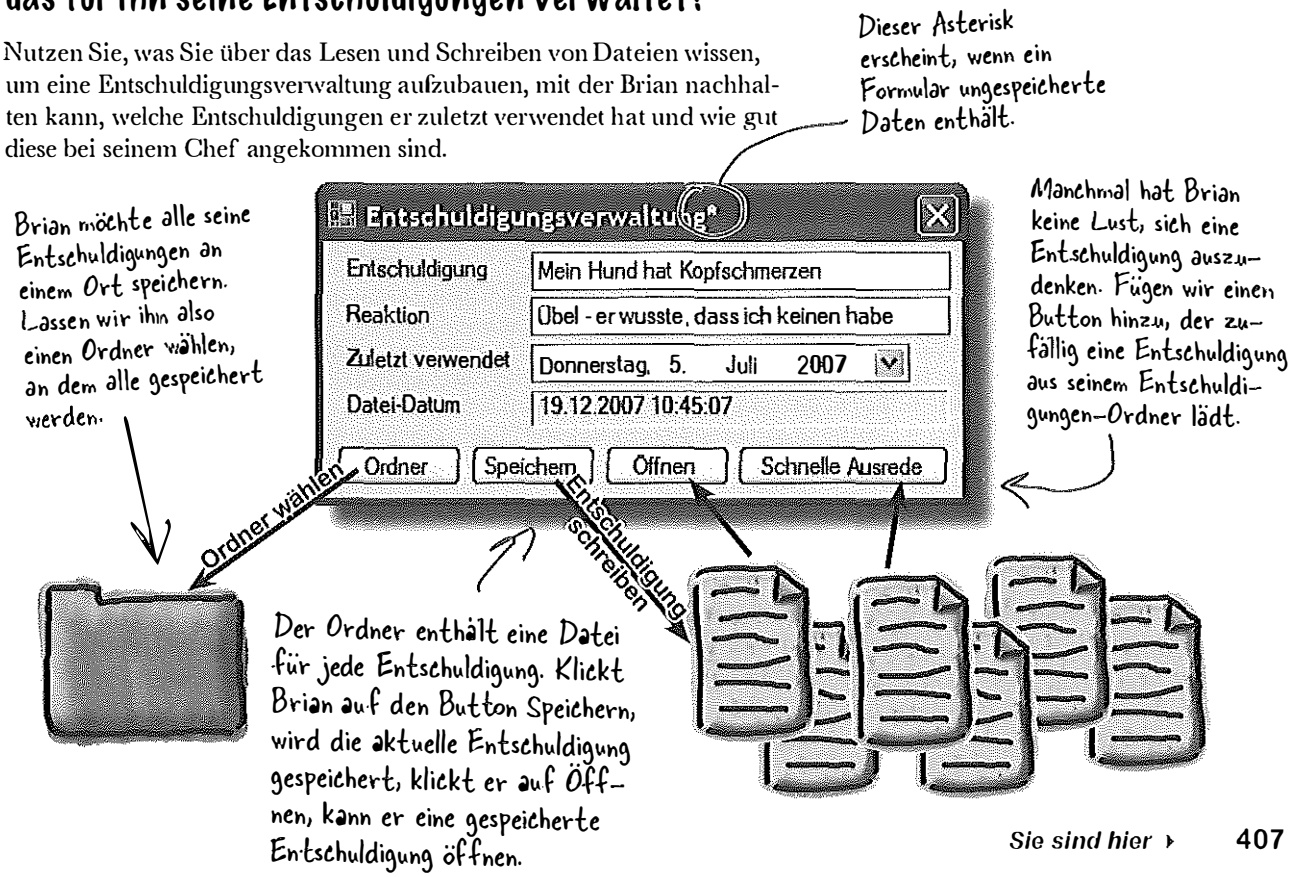
Ärger im Büro

Kennen Sie Brian schon? Er mag seinen Job als C#-Programmierer, noch *mehr* aber mag er es, sich gelegentlich einen Tag freizunehmen. Aber sein Chef **hasst** es, wenn seine Leute Urlaub machen. Brian braucht also eine gute Entschuldigung.



Können Sie Brian mit einem Programm helfen, das für ihn seine Entschuldigungen verwaltet?

Nutzen Sie, was Sie über das Lesen und Schreiben von Dateien wissen, um eine Entschuldigungsverwaltung aufzubauen, mit der Brian nachhalten kann, welche Entschuldigungen er zuletzt verwendet hat und wie gut diese bei seinem Chef angekommen sind.





Erstellen Sie die Entschuldigungsverwaltung, damit Brian die Entschuldigungen für die Arbeit verwalten kann.

Entschuldigung
Beschreibung: string
Reaktion: string
LetzteVerwendung: DateTime
Dateipfad: string
DateiÖffnen(string)
Speichern(string)

1 Erstellen Sie das Formular.

Dieses Formular hat ein paar spezielle Eigenschaften:

- Wenn das Formular geladen wird, **ist nur der Ordner-Button aktiviert** – deaktivieren Sie die anderen drei Buttons, bis der Benutzer einen Ordner wählt.
- Wird eine Entschuldigung geöffnet oder gespeichert, wird das Dateidatum in einem Label angezeigt, bei dem AutoSize auf False und BorderStyle auf Fixed3D gesetzt sind.
- Nach dem Speichern einer Entschuldigung wird ein Dialog mit der Meldung »Entschuldigung gespeichert« angezeigt.
- Der Ordner-Button öffnet einen Ordner-Dialog. Wählt der Benutzer einen Ordner, werden die anderen drei Buttons aktiviert.
- Das Formular weiß, ob es ungespeicherte Änderungen gibt. Gibt es keine ungespeicherten Änderungen, lautet der Text in der Titelleiste »Entschuldigungsverwaltung«. Hat der Benutzer eins der drei Felder geändert, wird an die Titelleiste ein Asterisk (*) angehangen, der verschwindet, wenn die Daten gespeichert werden oder eine neue Entschuldigung geöffnet wird.
- Das Formular muss den aktuellen Ordner festhalten und wissen, ob die aktuelle Entschuldigung gespeichert wurde oder nicht. Ob die Entschuldigung ungespeicherte Änderungen enthält, können Sie mit den **Changed-Event-Handlern** für die Eingabeelemente festhalten.

Klicken Sie doppelt auf ein Textfeld, um einen Changed-Event-Handler dafür zu erstellen.

2 Erzeugen Sie die Klasse Entschuldigung und speichern Sie eine Instanz im Formular.

Im Feld dieEntschuldigung wird die aktuelle Entschuldigung gespeichert. Sie brauchen **drei überladene Konstruktoren**: einen für das Laden des Formulars, einen für das Öffnen einer Datei und einen für eine zufällige Entschuldigung. Fügen Sie die Methoden `DateiÖffnen()`, mit der die Konstruktoren eine Entschuldigung öffnen können, und `Speichern()` hinzu. Fügen Sie dann diese `FormularAktualisieren()`-Methode hinzu, um die Steuerelemente zu aktualisieren (sie gibt Ihnen einige **Hinweise** zu der Klasse):

```
private void FormularAktualisieren(bool geändert) {
  if (!geändert)  {
     this.beschreibung.Text = dieEntschuldigung.Beschreibung;
     this.reaktion.Text = dieEntschuldigung.Reaktion;
     this.letzteVerwendung.Value = dieEntschuldigung.LetzteVerwendung;
     if (!String.IsNullOrEmpty(dieEntschuldigung.Dateipfad))
        datum.Text = File.GetLastWriteTime(dieEntschuldigung.Dateipfad).ToString();
        this.Text = "Entschuldigungsverwaltung";
     }
     else
        this.Text = "Entschuldigungsverwaltung*";
     this.formGeändert = geändert;
}
```

Dieser Parameter, den Sie über ein Feld im Formular nachhalten, zeigt an, ob sich das Formular geändert hat.

Das ! heißt NICHT und lässt prüfen, ob der Pfad null ist

Auf einen Doppelklick auf die Textfelder erzeugt die IDE die Changed-Event-Handler. Diese ändern zunächst die Entschuldigung-Instanz und rufen dann FormularAktualisieren(true) auf – Sie müssen noch für die Änderungen am Formular sorgen.

3 Lassen Sie den Button Ordner einen Ordner-Dialog öffnen.

Klickt der Benutzer auf den Button Ordner, sollte das Formular einen »Ordner suchen«-Dialog öffnen. Das Formular speichert den Ordner in einem Feld, das die anderen Dialoge verwenden können. Beim **ersten Laden** des Formulars sind die Buttons Speichern, Öffnen und Schnelle Ausrede **dealktiviert**. Wenn der Benutzer einen Ordner wählt, werden sie durch den Ordner-Button aktiviert.

4 Lassen Sie den Speichern-Button die Entschuldigung in einer Datei speichern.

Ein Klick auf den Speichern-Button sollte einen Speichern unter-Dialog öffnen.

- ★ Jede Entschuldigung wird in einer separaten Datei gespeichert. Die erste Zeile der Datei ist die Entschuldigung, die zweite das Ergebnis, die dritte die letzte Verwendungszeit (nutzen Sie die `ToString()`-Methode eines DateTimePicker-Objekts). Die Klasse `Entschuldigung` sollte eine `Speichern()`-Methode haben, die eine Entschuldigung in einer Datei speichert.
- ★ Wird der Speichern unter-Dialog geöffnet, sollte der Ordner geöffnet sein, der über den Button Ordner gewählt wurde, und der Dateiname sollte sich aus dem Text der Entschuldigung plus der Erweiterung ».txt« zusammensetzen.
- ★ Der Dialog sollte zwei Filter haben: Textdateien (*.txt) und Alle Dateien (*.*).
- ★ Versucht der Benutzer, eine Entschuldigung zu speichern, bei der Beschreibung oder Reaktion fehlen, sollte das Formular einen Warnung-Dialog anzeigen:

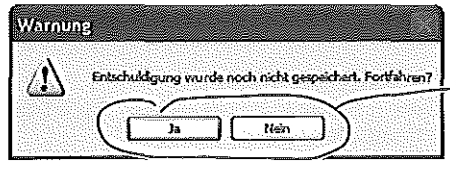
Dieses Symbol lassen Sie anzeigen, indem Sie eine Überladung von MessageBox.Show() verwenden, mit der Sie einen MessageBoxIcon-Parameter übergeben können.



5 Lassen Sie den Öffnen-Button eine Entschuldigung öffnen.

Ein Klick auf den Button sollte den Öffnen-Dialog aufrufen.

- ★ Beim Öffnen des Dialogs sollte der zuvor gewählte Ordner ausgewählt sein.
- ★ Fügen Sie der Klasse Entschuldigung eine `Öffnen()`-Methode hinzu, die eine Entschuldigung aus einer Datei öffnet.
- ★ Nutzen Sie `Convert.ToDateTime()`, um das gespeicherte Datum in das DateTimePicker-Steuerelement zu laden.
- ★ Versucht der Benutzer, ein Entschuldigung zu öffnen, obwohl die aktuelle nicht gespeichert wurde, sollte dieser Dialog angezeigt werden:



Einen Ja/NEIN-Dialog zeigen Sie mit einer Überladung von MessageBox.Show() an, der Sie den Parameter MessageBoxButtons.YesNo übergeben können. Klickt der Benutzer auf »Nein«, liefert Show() DialogResult.No.

6 Lassen Sie den Button Schnelle Ausrede eine zufällige Entschuldigung laden.

Klickt der Benutzer auf den Button Schnelle Ausrede, wird zufällig eine Entschuldigung im aktuellen Entschuldigungsorder gewählt und geöffnet.

- ★ Das Formular muss eine `Random`-Instanz festhalten und an einen der überladenen Konstruktoren von `Entschuldigung` übergeben.
- ★ Wurde die aktuelle Entschuldigung noch nicht gespeichert, sollte die gleiche Warnung angezeigt werden wie beim Öffnen-Button.



Lösungen zu den Übungen

Erstellen Sie die Entschuldigungsverwaltung, damit Brian die Entschuldigungen für die Arbeit verwalten kann.

```
private Entschuldigung dieEntschuldigung = new Entschuldigung();
private string gewählterOrdner = "";
private bool formGeändert = false;
Random zufall = new Random();

private void ordner_Click(object sender, EventArgs e) {
    folderBrowserDialog1.SelectedPath = gewählterOrdner;
    DialogResult ergebnis = folderBrowserDialog1.ShowDialog();
    if (ergebnis == DialogResult.OK) {
        gewählterOrdner = folderBrowserDialog1.SelectedPath;
        speichern.Enabled = true;
        öffnen.Enabled = true;
        schnelleAusrede.Enabled = true;
    }
}

private void speichern_Click(object sender, EventArgs e) {
    if (String.IsNullOrEmpty(beschreibung.Text) || String.IsNullOrEmpty(reaktion.Text)) {
        MessageBox.Show("Geben Sie eine Entschuldigung und ein Ergebnis an",
            "Speichern nicht möglich", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
        return;
    }
    saveFileDialog1.InitialDirectory = gewählterOrdner;
    saveFileDialog1.Filter = "Textdateien (*.txt)|*.txt|Alle Dateien (*.*)|*.*";
    saveFileDialog1.FileName = beschreibung.Text + ".txt";
    DialogResult ergebnis = saveFileDialog1.ShowDialog();
    if (ergebnis == DialogResult.OK) {
        dieEntschuldigung.Speichern(saveFileDialog1.FileName);
        FormularAktualisieren(false);
        MessageBox.Show("Entschuldigung gespeichert");
    }
}

private void öffnen_Click(object sender, EventArgs e) {
    if (GeändertPrüfen()) {
        openFileDialog1.InitialDirectory = gewählterOrdner;
        openFileDialog1.Filter = "Textdateien (*.txt)|*.txt|Alle Dateien (*.*)|*.*";
        openFileDialog1.FileName = beschreibung.Text + ".txt";
        DialogResult ergebnis = openFileDialog1.ShowDialog();
        if (ergebnis == DialogResult.OK) {
            dieEntschuldigung = new Entschuldigung(openFileDialog1.FileName);
            FormularAktualisieren(false);
        }
    }
}

private void schnelleAusrede_Click(object sender, EventArgs e) {
    if (GeändertPrüfen()) {
        dieEntschuldigung = new Entschuldigung(zufall, gewählterOrdner);
        FormularAktualisieren(false);
    }
}
```

Diese Felder halten die aktuelle Entschuldigung, den gewählten Ordner, den Bearbeitungsstatus der aktuellen Entschuldigung und ein Random-Objekt für die schnelle Ausrede fest.

Nachdem der Benutzer einen Ordner gewählt hat, wird dieser gespeichert, und die drei anderen Buttons werden aktiviert.

Die beiden senkrechten Striche heißen ODER – der Ausdruck ist true, wenn beschreibung leer ODER reaktion leer ist.

Hier werden die Filter für den Speichern-Dialog gesetzt.

Das sorgt dafür, dass im Aufklappmenü »Dateityp« zwei Zeilen auftauchen: eine für Textdateien (*.txt) und eine für Alle Dateien (*.*).

Über den DialogResult-Wert, den die Öffnen- und Speichern-Dialoge liefern, prüfen Sie, ob der Benutzer »OK« oder »Abbrechen« anklickte.

```
private bool GeändertPrüfen() {
    if (formGeändert) {
        DialogResult ergebnis = MessageBox.Show(
              "Entschuldigung wurde noch nicht gespeichert. Fortfahren?",
               "Warnung", MessageBoxButtons.YesNo, MessageBoxIcon.Warning);
        if (ergebnis == DialogResult.No)
            return false;
    }
    return true;
}

private void beschreibung_TextChanged(object sender, EventArgs e) {
    dieEntschuldigung.Beschreibung = beschreibung.Text;
    FormularAktualisieren(true);
}

private void reaktion_TextChanged(object sender, EventArgs e) {
    dieEntschuldigung.Reaktion = reaktion.Text;
    FormularAktualisieren(true);
}

private void letzteVerwendung_ValueChanged(object sender, EventArgs e) {
    dieEntschuldigung.LetzteVerwendung = letzteVerwendung.Value;
    FormularAktualisieren(true);
}

public class Entschuldigung {
    public string Beschreibung;
    public string Reaktion;
    public DateTime LetzteVerwendung;
    public string Dateipfad;
    public Entschuldigung() {
        Dateipfad = "";
    }
    public Entschuldigung(string dateipfad) {
        DateiÖffnen(dateipfad);
    }
    public Entschuldigung(Random zufall, string ordner) {
        string[] dateinamen = Directory.GetFiles(ordner, "*.txt");
        DateiÖffnen(dateinamen[zufall.Next(dateinamen.Length)]);
    }
    private void DateiÖffnen(string dateipfad) {
        this.Dateipfad = dateipfad;
        using (StreamReader reader = new StreamReader(dateipfad)) {
            Beschreibung = reader.ReadLine();
            Reaktion = reader.ReadLine();
            LetzteVerwendung = Convert.ToDateTime(reader.ReadLine());
        }
    }
    public void Speichern(string dateipfad) {
        using (StreamWriter writer = new StreamWriter(dateipfad))
        {
            writer.WriteLine(Beschreibung);
            writer.WriteLine(Reaktion);
            writer.WriteLine(LetzteVerwendung.ToString());
        }
    }
}
```

MessageBox.Show() liefert ebenfalls einen DialogResult-Wert, den wir prüfen können.

Hier sind die drei Changed-Event-Handler für die Eingabefelder. Wird einer von ihnen ausgelöst, wurde die Entschuldigung geändert. Dann wird die erste Entschuldigung-Instanz. aktualisiert und anschließend FormularAktualisieren(), aufgerufen, um an den Text in der Titelleiste den Asterisk anzuhängen und Geändert auf true zu setzen.

Wird FormularAktualisieren() true übergeben, wird das Formular nur als geändert markiert, ohne die Eingabeelemente zu aktualisieren.

Der Button Schnelle Ausrede nutzt Directory.GetFiles(), um alle Textdateien im gewählten Ordner in ein Array zu lesen, und wählt dann zufällig ein Element des Arrays aus.

Wir haben immer eine using-Anweisung verwendet, wenn wir einen Stream öffnen. Das sichert, dass die Dateien immer geschlossen werden.

Hier wirkt sich die using-Anweisung aus. Wir deklarieren den StreamWriter in einer using-Anweisung, damit Close() automatisch für uns aufgerufen wird.

Beim Schreiben von Dateien muss man in der Regel viele Entscheidungen treffen

Sie werden eine Menge Programme schreiben, die nur eine einzige Eingabe benötigen, die eventuell aus einer Datei stammt, und auf Basis dieser Eingabe entscheiden, was sie tun müssen. Hier ist Code, der eine lange `if`-Anweisung verwendet – das ist ziemlich typisch. Er prüft die Variable `teil` und schreibt auf Basis des verwendeten Enum-Werts Zeilen in die Datei. Es gibt viele Optionen, also auch viele `else if`:

```
enum Körperteil {
    Kopf,
    Schultern,
    Knie,
    Zehen
}
```

Hier ist ein Enum – wir möchten eine Variable mit allen vier Membern vergleichen und, je nachdem, eine andere Zeile in den StreamWriter schreiben. Und passt keiner von ihnen, schreiben wir einen noch anderen Text.

```
private void TeileSchreiben(Körperteil teil, StreamWriter writer)
{
  if (teil == Körperteil.Kopf)
    writer.WriteLine("der Kopf ist haarig");
  else if (teil == Körperteil.Schultern)
    writer.WriteLine("die Schultern sind breit");
  else if (teil == Körperteil.Knie)
    writer.WriteLine("die Knie sind knubbelig");
  else if (teil == Körperteil.Zehen)
    writer.WriteLine("die Zehen sind winzig");
  else
    writer.WriteLine("ein Teil ist unbekannt");
  }
}
```

Schreiben wir eine Folge von if/else-Anweisungen, müssen wir diesen >>if (teil ==[option])<< immer wieder wiederholen.

Das abschließende else darf ran, wenn wir keinen Treffer finden.



KOPF-NUSS

Was kann schiefgehen, wenn Sie Code wie diesen mit so vielen if/else-Anweisungen schreiben? Denken Sie beispielsweise an Tippfehler, falsche Klammern, einfache Anführungszeichen usw.

Wählen Sie Ihre Optionen mit einer switch-Anweisung

Dass man eine Variable mit einer Menge verschiedener Werte vergleichen muss, ist ein Muster, auf das Sie immer wieder stoßen werden. Beim Lesen und Schreiben von Dateien tritt es besonders häufig auf. Es ist sogar so häufig, dass C# eine besondere Anweisung bietet, die genau für solche Fälle gedacht ist.

Mit einer **switch-Anweisung** können Sie eine Variable auf kompakte und lesbare Weise mit mehreren Werten vergleichen. Hier ist eine switch-Anweisung, die genau das Gleiche macht wie die Folge von `if/else`-Anweisungen auf der letzten Seite.

Die switch-Anweisung ist nicht speziell für Dateien gedacht. Sie ist ein nützliches C#-Werkzeug, das wir hier verwenden können.

Eine switch-Anweisung vergleicht EINE Variable mit MEHREREN möglichen Werten.

```
enum Körperteil
{
    Kopf,
    Schultern,
    Knie,
    Zehen,
}

private void TeileSchreiben(Körperteil teil, StreamWriter writer)  {
    switch (teil) {
        case Körperteil.Kopf:
            writer.WriteLine("der Kopf ist haarig");
            break;
        case Körperteil.Schultern:
            writer.WriteLine("die Schultern sind breit");
            break;
        case Körperteil.Knie:
            writer.WriteLine("die Knie sind knubbelig");
            break;
        case Körperteil.Zehen:
            writer.WriteLine("die Zehen sind winzig");
            break;
        default:
            writer.WriteLine("ein Teil ist unbekannt");
            break;
    }
}
```

Sie beginnen mit dem Schlüsselwort switch. Auf dieses folgt die Variable, die mit mehreren möglichen Werten verglichen werden soll.

Den Inhalt der switch-Anweisung bildet eine Folge von case-Anweisungen (Fällen), die den switch-Testausdruck mit einem bestimmten Wert vergleichen.

Jeder Fall muss mit >>break;<< enden.

Jeder dieser Fälle besteht aus dem Schlüsselwort case und einem Testausdruck, auf den ein Doppelpunkt folgt. Darauf folgen beliebig viele Anweisungen, die mit einer >>break;<<-Anweisung abgeschlossen werden. Diese Anweisungen werden ausgeführt, wenn der case-Ausdruck dem switch-Ausdruck entspricht.

switch-Anweisungen können mit einem default:-Block enden, der ausgeführt wird, wenn keiner der anderen Fälle passt.

Schreiben und lesen Sie mit switch Ihre Spielkarten in eine bzw. aus einer Datei

Das Ausgeben einer Karte ist leicht – schreiben Sie einfach eine Schleife, die die Namen aller Karten in eine Datei schreibt. Folgende Methode macht genau das. Fügen Sie sie einfach der Klasse Kartenstapel hinzu:

```
public void KartenSchreiben(string Filename) {
    using (StreamWriter writer = new StreamWriter(Filename)) {
        for (int i = 0; i < karten.Anzahl; i++) {
            writer.WriteLine(karten[i].Name);
        }
    }
}
```

Aber wie sieht es mit dem Wiedereinlesen der Datei aus? Das ist nicht ganz so einfach. Dabei erweist sich die switch-Anweisung als praktisch.

Mit der switch-Anweisung können Sie einen Wert mit vielen anderen Werten vergleichen und in Abhängigkeit vom passenden Wert unterschiedliche Anweisungen ausführen lassen.

```
Karte.Farben farbe;
switch (farbeString) {
    case "Karo":
        farbe = Karte.Farben.Karo;
        break;
    case "Herz":
        farbe = Karte.Farben.Herz;
        break;
    case "Pik":
        farbe = Karte.Farben.Pik;
        break;
    case "Kreuz":
        farbe = Karte.Farben.Kreuz;
        break;
    default:
        MessageBox.Show(farbeString + " ist ungültig!");
}
```

Die switch-Anweisung beginnt mit einem Wert, mit dem verglichen wird. Diese switch-Anweisung wird dann von einer Methode aufgerufen, die die Kartenfarbe als String gespeichert hat.

Jede dieser case-Zeilen vergleicht einen Wert mit dem Wert in der switch-Zeile. Sind die beiden Werte gleich, werden alle nachfolgenden Zeilen bis zum nächsten break ausgeführt.

Die default-Zeile folgt am Ende. Passt keine der case-Zeilen, werden stattdessen die Anweisungen nach default ausgeführt.

Ein überladener Kartenstapel()-Konstruktor, der einen Kartenstapel aus einer Datei einliest

Mit einer switch-Anweisung können Sie einen neuen Konstruktor für die Klasse Kartenstapel erstellen, die Sie im letzten Kapitel geschrieben haben. Dieser Konstruktor liest eine Datei ein und prüft jede Zeile auf eine Karte. Jede gültige Karte wird dem Stapel hinzugefügt.

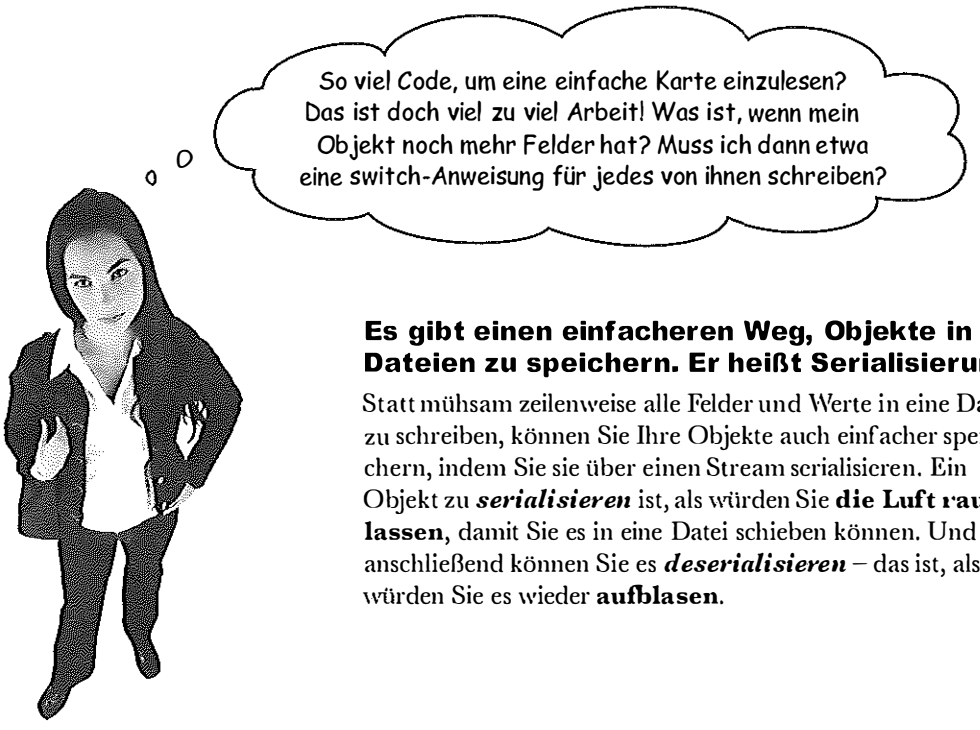
Es gibt eine Methode der Klasse String, die sich als praktisch erweisen wird: `Split()`. Mit ihr können Sie einen String in ein Array von Teil-Strings zerlegen, indem Sie ein `char[]` mit Trennzeichen übergeben, das verwendet wird, um den String aufzuspalten.

```
public Kartenstapel(string Filename) {
    karten = new List<Karte>();
    StreamReader reader = new StreamReader(Filename);
    while (!reader.EndOfStream) {
        bool ungültigeKarte = false;
        string karte = reader.ReadLine();
        string[] kartenteile = karte.Split(new char[] { ' ' });
        Karte.Werte wert = Karte.Werte.Ass;
        switch (kartenteile[1]) {
            case "Ass": wert = Karte.Werte.Ass; break;
            case "Zwei": wert = Karte.Werte.Zwei; break;
            case "Drei": wert = Karte.Werte.Drei; break;
            case "Vier": wert = Karte.Werte.Vier; break;
            case "Fünf": wert = Karte.Werte.Fünf; break;
            case "Sechs": wert = Karte.Werte.Sechs; break;
            case "Sieben": wert = Karte.Werte.Sieben; break;
            case "Acht": wert = Karte.Werte.Acht; break;
            case "Neun": wert = Karte.Werte.Neun; break;
            case "Zehn": wert = Karte.Werte.Zehn; break;
            case "Bauer": wert = Karte.Werte.Bauer; break;
            case "Dame": wert = Karte.Werte.Dame; break;
            case "König": wert = Karte.Werte.König; break;
            default: ungültigeKarte = true; break;
        }
        Karte.Farben farbe = Karte.Farben.Herz;
        switch (kartenteile[0]) {
            case "Karo": farbe = Karte.Farben.Karo; break;
            case "Herz": farbe = Karte.Farben.Herz; break;
            case "Pik": farbe = Karte.Farben.Pik; break;
            case "Kreuz": farbe = Karte.Farben.Kreuz; break;
            default: ungültigeKarte = true; break;
        }
        if (!ungültigeKarte) {
            karten.Add(new Karte(farbe, wert));
        }
    }
}
```

Diese Zeile sagt C#, dass es den String `nächsteKarte` an einem Bindestrich aufspalten soll. Das teilt den String `"Kreuz-Sechs"` in das Array `{"Kreuz", "Sechs"}` auf.

Diese switch-Anweisung vergleicht das zweite Wort in der Zeile mit den Kartenwerten. Findet sie einen passenden Wert, wird er der Variablen wert zugewiesen.

Das Gleiche tun wir für das erste Wort in der Zeile, nur wandeln wir das in eine Farbe um.



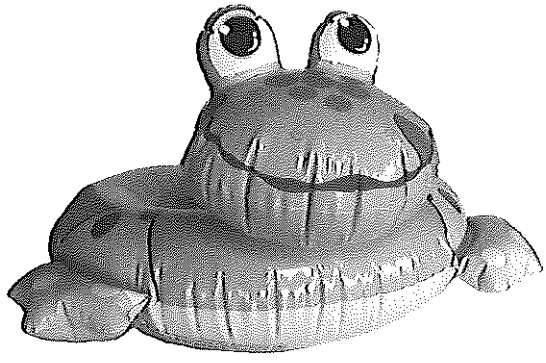
Es gibt einen einfacheren Weg, Objekte in Dateien zu speichern. Er heißt Serialisierung.

Statt mühsam zeilenweise alle Felder und Werte in eine Datei zu schreiben, können Sie Ihre Objekte auch einfacher speichern, indem Sie sie über einen Stream serialisieren. Ein Objekt zu ***serialisieren*** ist, als würden Sie **die Luft rauslassen**, damit Sie es in eine Datei schieben können. Und anschließend können Sie es ***deserialisieren*** – das ist, als würden Sie es wieder **aufblasen**.

Was bei der Serialisierung eines Objekts passiert

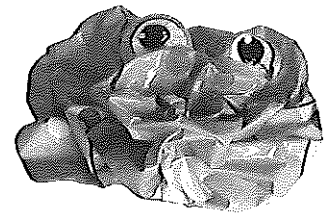
Es scheint, als müssten mit einem Objekt einige ziemlich geheimnisvolle Dinge passieren, damit Sie es aus dem Heap nehmen und in eine Datei stecken können. Aber eigentlich ist das ein ziemlich einfache Geschichte.

1 Objekt auf dem Heap

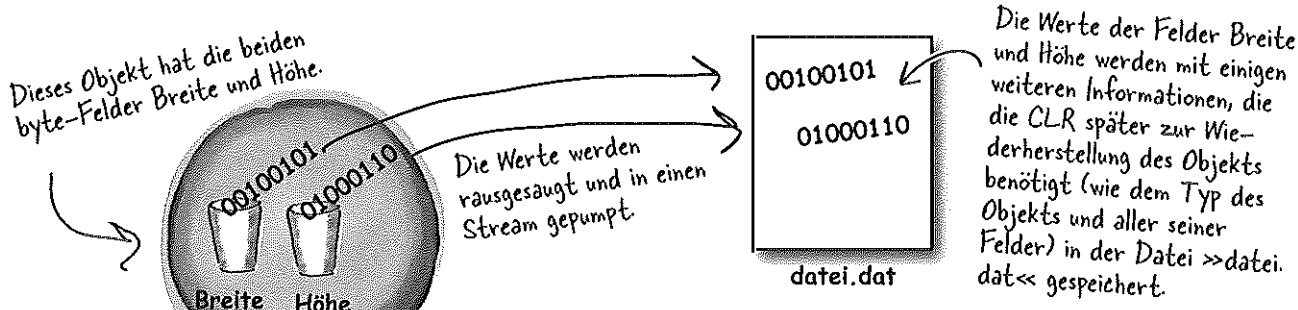


Erstellen Sie eine Instanz einer Klasse, besitzt diese einen **Zustand**. Das, was ein Objekt »weiß«, ist das, was eine Instanz einer Klasse von einer anderen Instanz der gleichen Klasse unterscheidet.

2 Objekt serialisiert



Serialisiert C# ein Objekt, **speichert es den gesamten Zustand des Objekts**, damit eine identische Instanz später wieder auf dem Heap zum Leben erweckt werden kann.



3 Und später ...

Später – vielleicht Tage später und in einem anderen Programm – können Sie sich die Datei nehmen und das Objekt darin **deserialisieren**. Das zieht das ursprüngliche Objekt aus der Datei und stellt es **genau so wieder her, wie es war**, mit allen intakten Feldern und Werten.

Objekt wieder auf dem Heap

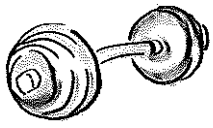


Aber was genau IST der Zustand eines Objekts? Was muss gespeichert werden?

Wir wissen bereits, dass **ein Objekt seinen Zustand in Feldern speichert**. Wird ein Objekt serialisiert, muss jedes dieser Felder in der Datei gespeichert werden.

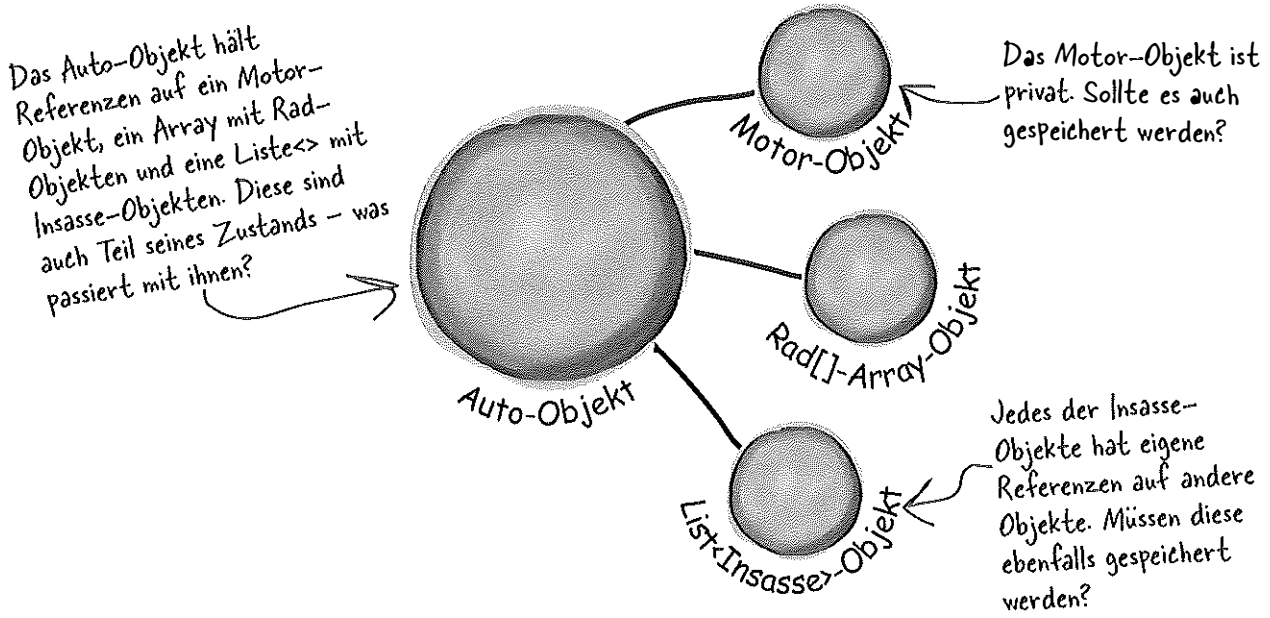
Serialisierung beginnt, interessant zu werden, wenn Ihre Objekte komplizierter werden. 37 und 70 sind byte-Werte – also Werttypen, die so, wie sie sind, in Dateien geschrieben werden können. Aber was ist, wenn ein Objekt eine Instanzvariable hat, die selbst eine *Objektreferenz* ist? Was ist mit einem Objekt, das fünf Instanzvariablen hat, die Objektreferenzen sind? Was ist, wenn auch diese Objekte selbst wieder Instanzvariablen haben?

Denken Sie einen Augenblick darüber nach. Welche Teile eines Objekts sind möglicherweise einzigartig? Überlegen Sie, was wiederhergestellt werden muss, um ein Objekt zu erhalten, das mit dem identisch ist, das gespeichert wurde. Irgendwie muss alles auf dem Heap in die Datei geschrieben werden.



KRAFT-TRAINING

Was muss mit diesem Auto-Objekt geschehen, damit es so gespeichert wird, dass es später in seinem ursprünglichen Zustand wiederhergestellt werden kann? Nehmen wir an, das Auto hat drei Insassen, eine 3-Liter-Maschine und Allwetterreifen ... sind diese Dinge nicht Teil seines Zustands? Was sollte mit ihnen passieren?

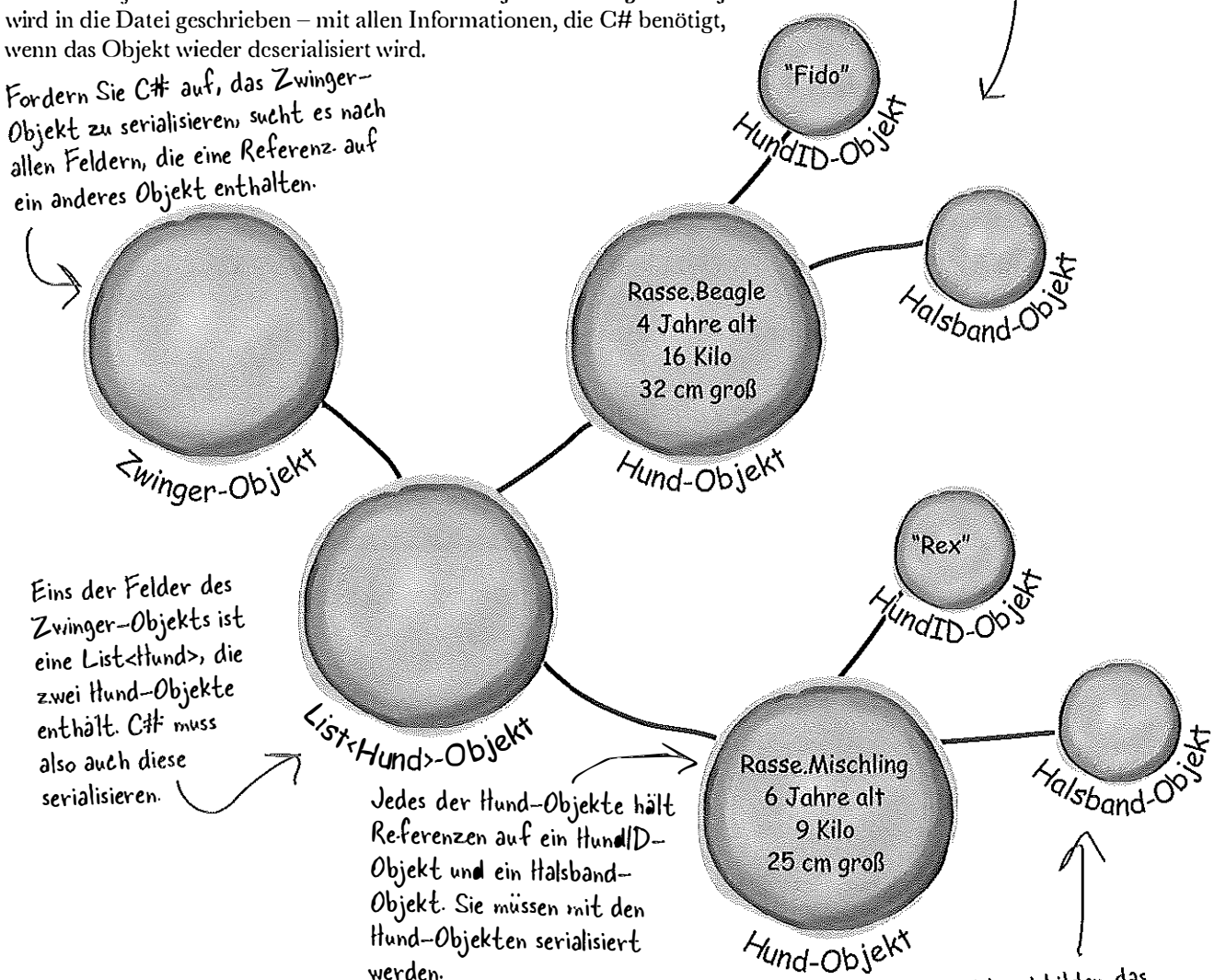


Wird ein Objekt serialisiert, werden die Objekte, die es referenziert, ebenfalls serialisiert ...

... und alle Objekte, auf die *diese* verweisen, und alle Objekte, auf *die diese anderen Objekte* verweisen und so weiter. Aber machen Sie sich keine Gedanken – es klingt zwar kompliziert, passiert aber alles automatisch. C# beginnt mit dem Objekt, das Sie serialisieren möchten, und sucht in seinen Feldern nach anderen Objekten. Dann macht es das Gleiche für jedes davon. Jedes Objekt wird in die Datei geschrieben – mit allen Informationen, die C# benötigt, wenn das Objekt wieder deserialisiert wird.

Manche Leute bezeichnen eine solche Gruppe verknüpfter Objekte als »Graphen«.

Fordern Sie C# auf, das Zwinger-Objekt zu serialisieren, sucht es nach allen Feldern, die eine Referenz auf ein anderes Objekt enthalten.



Eins der Felder des Zwinger-Objekts ist eine List<Hund>, die zwei Hund-Objekte enthält. C# muss also auch diese serialisieren.

Jedes der Hund-Objekte hält Referenzen auf ein HundID-Objekt und ein Halsband-Objekt. Sie müssen mit den Hund-Objekten serialisiert werden.

HundID und Halsband bilden das Ende der Kette – sie halten keine Referenzen auf weitere Objekte.

Mit Serialisierung können Sie ein ganzes Objekt auf einmal lesen oder schreiben

Ein Objekt kann man leicht in eine Datei kopieren oder aus einer lesen. Sie können es serialisieren oder deserialisieren.

Sie sind nicht darauf beschränkt, Text zeilenweise in Dateien zu schreiben oder aus ihnen zu lesen. Mit **Serialisierung** können Ihre Programme ganze Objekte in Dateien schreiben und wieder daraus lesen ... und all das mit nur ein paar Zeilen Code! Sie müssen lediglich ein paar Vorarbeiten leisten – setzen Sie an den Anfang der Klasse, die serialisiert werden soll, eine `[Serializable]`-Zeile. Aber dann können Sie auch wirklich direkt loslegen.

Sie brauchen ein BinaryFormatter-Objekt

Möchten Sie ein Objekt serialisieren – *egal was für ein* Objekt –, müssen Sie zuerst immer eine `BinaryFormatter`-Instanz erstellen. Das ist einfach – und kann mit einer einzigen Zeile Code gemacht werden (und einer zusätzlichen `using`-Zeile am Anfang der Klassendatei).

```
using System.Runtime.Serialization.Formatters.Binary;
...
BinaryFormatter formatter = new BinaryFormatter();
```

Erstellen Sie jetzt einen Stream zum Schreiben und Lesen

Nutzen Sie die **`Serialize()`**-Methode des BinaryFormatter-Objekts, um ein Objekt in einen Stream zu schreiben.

Die Methode File.Create() erzeugt eine neue Datei. Eine bestehende können Sie mit File.OpenWrite() öffnen.

```
Stream ausgabe = File.Create(dateiname);
formatter.Serialize(ausgabe, serialisierungsobjekt);
ausgabe.Close();
```

Mit einer using-Anweisung würde Close() automatisch aufgerufen.

Die Methode Serialize() nimmt ein Objekt und schreibt es in einen Stream. Das ist viel einfacher, als eine eigene Methode zu schreiben, um es auszugeben.

Haben Sie ein Objekt in eine Datei serialisiert, nutzen Sie die **`Deserialize()`**-Methode des BinaryFormatter-Objekts, um es wieder einzulesen. Die Methode liefert eine object-Referenz. Sie müssen die Ausgabe also auf den Typ casten, den die Referenzvariable hat, der Sie sie zuweisen.

```
Stream eingabe = File.OpenRead(dateiname);
EinObjekt obj = (EinObjekt)formatter.
Deserialize(eingabe);
eingabe.Close();
```

Nutzen Sie Deserialize(), um ein Objekt aus einem Stream zu lesen, dürfen Sie nicht vergessen, den Rückgabewert auf den passenden Typ zu casten.

Soll eine Klasse serialisierbar sein, müssen Sie sie mit dem Attribut [Serializable] markieren

Attribute sind besondere Tags, die Sie in C#-Klassen einfügen können. Über sie speichert C# **Metadaten** zu Ihrem Code oder Informationen dazu, wie der Code genutzt oder behandelt werden soll. **Stellen Sie unmittelbar vor die Deklaration einer Klasse das Attribut `[Serializable]`**, sagen Sie C#, dass Ihre Klasse für die Serialisierung geeignet ist. Es kann nur mit Klassen verwendet werden, deren Felder Werttypen sind (wie `int`, `string` oder `enum`) oder andere serialisierbare Klassen. Vergessen Sie das Attribut bei einer Klasse, die Sie serialisieren möchten, oder schließen Sie ein Feld mit einem Typ ein, der nicht serialisierbar ist, liefert Ihr Programm einen Fehler, wenn Sie versuchen, es auszuführen. ***Sehen Sie selbst ...***

Tun Sie das!

1 Erzeugen Sie eine Klasse und serialisieren Sie sie.

Erinnern Sie sich an die Klasse Typ aus Kapitel 3? Serialisieren wir Tim, damit wir in einer Datei speichern können, wie viel Geld er in der Tasche hat, auch nachdem das Programm geschlossen wurde.

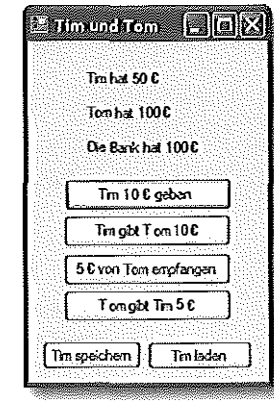
```
[Serializable]
public class Typ
```

Hier ist der Code, der sie in eine Datei mit dem Namen »TypDatei.dat« speichert – fügen Sie dem Formular einen »Tim speichern«- und einen »Tim laden«-Button hinzu:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
...

private void timSpeichern_Click(object sender, EventArgs e)
{
  using (Stream ausgabe = File.Create("TypDatei.dat")) {
    BinaryFormatter formatter = new BinaryFormatter();
    formatter.Serialize(ausgabe, tim);
  }
}
private void timSpeichern_Click(object sender, EventArgs e)
{
  using (Stream eingabe = File.OpenRead("TypDatei.dat")) {
    BinaryFormatter formatter = new BinaryFormatter();
    tim = (Typ)formatter.Deserialize(eingabe);
  }
  FormularAktualisieren();
}
```

Diese beiden `using`-Zeilen brauchen Sie. Die erste ist für die Datei- und Stream-Klassen, die zweite für die Serialisierung.

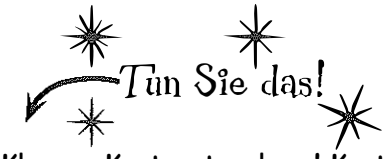


2 Führen Sie das Programm aus und spielen Sie damit.

Hätte Tim in seinen Transaktionen mit Tom während der Laufzeit des Programms 200 € angesammlt, wäre es ziemlich ärgerlich, diese zu verlieren, nur weil das Programm beendet wird. Jetzt kann Ihr Programm Tim in eine Datei speichern und jederzeit wiederherstellen.

Serialisieren und deserialisieren wir einen Kartenstapel

Nehmen Sie einen Kartenstapel und schreiben Sie ihn in eine Datei. C# macht das Serialisieren von Objekten zu einem Klacks. Sie brauchen nur einen Stream, schon können Sie Ihre Objekte speichern.



1. **Erstellen Sie ein Projekt und fügen Sie diesem die Klassen Kartenstapel und Karte hinzu.**
Klicken Sie im Projektmappen-Explorer rechts auf das Projekt, wählen Sie »Hinzufügen/Vorhandenes Element« und fügen Sie die Klassen `Karte` und `Kartenstapel` hinzu, die Sie in Kapitel 7 erstellt haben. Außerdem benötigen Sie die beiden Comparer-Klassen für Karten, da `Kartenstapel` diese nutzt. Die IDE kopiert diese Dateien in Ihr Projekt – denken Sie daran, die `namespace`-Zeilen am Anfang der Klassendateien so zu ändern, dass sie dem Namen des neuen Projekts entsprechen.

2. **Markieren Sie die Klassen als serialisierbar.**
Markieren Sie alle Klassen, die Sie dem Projekt hinzugefügt haben, mit dem Attribut `[Serializable]`.

Tun Sie das nicht, lässt C#: Sie die Klassen nicht in eine Datei serialisieren.

3. **Fügen Sie dem Formular ein paar nützliche Methoden hinzu.**
Die Klasse ZufallsStapel erzeugt einen Stapel mit zufälligen Karten, und die Methode Karten-Ausgeben gibt die Karten aus und zeigt sie auf der Konsole an.

```
Random zufall = new Random();
private Kartenstapel ZufallsStapel(int anzahl) {
    Kartenstapel derStapel = new Kartenstapel(new Karte[] { });
    for (int i = 0; i < anzahl; i++)
    {
        derStapel.Hinzufügen(new Karte(
            (Karte.Farben)zufall.Next(4),
            (Karte.Werte)zufall.Next(1, 14)));
    }
    return derStapel;
}
```

Das erzeugt einen leeren Stapel, dem dann zufällig Karten hinzugefügt werden. Dazu wird die Klasse Karte aus dem letzten Kapitel verwendet.

```
private void KartenAusgeben(Kartenstapel ausgabeStapel, string Title) {
    Console.WriteLine(Title);
    while (ausgabeStapel.Anzahl > 0)
    {
        Karte karte = ausgabeStapel.Ausgeben(0);
        Console.WriteLine(karte.Name);
    }
    Console.WriteLine("--------------------");
}
```

Die Methode KartenAusgeben() gibt alle Karten im Stapel aus und zeigt sie auf der Konsole an.

4 Die Vorbereitungen sind gemacht ... serialisieren wir jetzt den Stapel.

Fügen Sie zunächst Buttons hinzu, um einen Zufallsstapel in eine Datei zu serialisieren und wieder aus ihr zu lesen. Prüfen Sie die Ausgabe in der Konsole, um sicherzustellen, dass der geschriebene Stapel dem entspricht, den Sie wieder einlesen.

```
private void button1_Click(object sender, EventArgs e) {
    Kartenstapel stapel = ZufallsStapel(5);
    using (Stream ausgabe = File.Create("Stapel1.dat")) {
      BinaryFormatter bf = new BinaryFormatter();
      bf.Serialize(ausgabe, stapel);
    }
    KartenAusgeben(stapel, "In die Datei geschrieben");
}
private void button2_Click(object sender, EventArgs e) {
    Kartenstapel stapel;
    using (Stream eingabe = File.OpenRead("Stapel1.dat")) {
      BinaryFormatter bf = new BinaryFormatter();
      stapel = (Kartenstapel)bf.Deserialize(eingabe);
    }
    KartenAusgeben(stapel, "Aus der Datei gelesen");
}
```

Der BinaryFormatter nimmt ein beliebiges mit dem Attribut Serializable markiertes Objekt und schreibt es mit der Methode Serialize() in einen Stream.

Die Deserialize()-Methode liefert einen Wert des Typs object. Das ist der allgemeine Typ, von dem alle C#-Objekte erben. Dieses müssen wir auf den Typ Kartenstapel casten.

5 Serialisieren Sie jetzt mehrere Stapel aus der gleichen Datei

Ist ein Stream einmal geöffnet, können Sie so viel hineinschreiben, wie Sie wollen. In eine Datei können Sie so viele Objekte serialisieren, wie Sie müssen. Fügen Sie jetzt zwei weitere Buttons hinzu, um eine zufällige Zahl von Stapeln in die Datei zu serialisieren. Prüfen Sie anhand der Ausgabe, ob alles in Ordnung ist.

```
private void button3_Click(object sender, EventArgs e) {
    using (Stream ausgabe = File.Create("Stapel1.dat")) {
      BinaryFormatter bf = new BinaryFormatter();
      for (int i = 1; i <= 5; i++) {
        Kartenstapel stapel = ZufallsStapel(zufall.Next(1,10));
        bf.Serialize(ausgabe, stapel);
        KartenAusgeben(stapel, "Kartenstapel " + i + " geschrieben");
      }
    }
}

private void button4_Click(object sender, EventArgs e) {
    using (Stream eingabe = File.OpenRead("Stapel1.dat")) {
      BinaryFormatter bf = new BinaryFormatter();
      for (int i = 1; i <= 5; i++) {
        Kartenstapel stapel = (Kartenstapel)bf.Deserialize(eingabe);
        KartenAusgeben(stapel, "Kartenstapel " + i + " gelesen");
      }
    }
}
```

Sie können nacheinander mehrere Objekte in einen Stream serialisieren.

Beachten Sie, wie die Zeile einen Stapel liest und mit (Kartenstapel) die Ausgabe von Deserialize() auf Kartenstapel castet. Das liegt daran, dass Deserialize() ein Objekt des Typs object liefert.

Die Anzahl der Objekte, die Sie in einen Stream serialisieren, ist nicht beschränkt. Sie müssen nur dafür sorgen, dass die eingelesenen Objekte auf den richtigen Typ gecastet werden.

6 Sehen Sie sich die geschriebene Datei an.

Öffnen Sie Stapel1.dat im Editor. Das ist vielleicht nicht unbedingt die ideale Urlaubslektüre, aber es enthält alle Informationen, die erforderlich sind, um Ihren Kartenstapel wiederherzustellen.

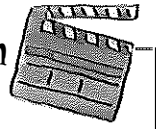


Moment. Nein, mir gefällt es nicht, meine Objekte in eine seltsame Datei zu stecken, die wie Müll aussieht, wenn ich sie öffne. Als ich den Stapel als String speicherte, konnte ich mir die Ausgabe im Editor ansehen und alles darin lesen. Sollte mir C# diese Dinge nicht etwas lesbarer liefern?

In eine Datei serialisierte Objekte, werden in einem binären Format geschrieben.

Aber das bedeutet nicht, dass es nicht entziffert werden kann – es ist bloß kompakt. Deswegen können Sie in Dateien mit serialisierten Objekten die Strings erkennen: Das ist die kompakteste Art, Strings in eine Datei zu schreiben – als Strings. Aber eine Zahl lesbar zu schreiben wäre Platzverschwendung. Ein `int` kann in 4 Bytes gespeichert werden. Es wäre also seltsam, würde C# die Zahl 49.369.144 als lesbaren String mit 8 Zeichen speichern – 10 Zeichen, schließt man die Kommata ein. Das wäre Platzverschendung!

Hinter den Kulissen



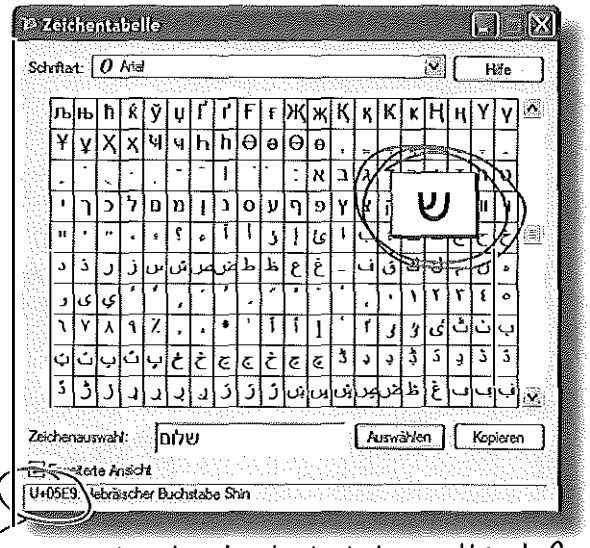
.NET nutzt **Unicode** für die Kodierung von Zeichen in Bytes. Glücklicherweise bietet Windows ein nettes kleines Werkzeug, mit dem wir uns ansehen können, wie Unicode funktioniert. Öffnen Sie die Zeichenpalette (Sie finden sie im Start-Menü unter Zubehör oder über Start/Ausführen und »charmap.exe«).

Sehen Sie sich die Zeichen und Symbole an, die in den Sprachen der Welt verwendet werden, erkennen Sie, wie viele unterschiedliche *Dinge* in eine Datei geschrieben werden müssen, um Text zu speichern. Deswegen **kodiert** .NET alle Zeichen mit Unicode. Kodieren bedeutet, dass die symbolischen Daten (wie der Buchstabe H) in Bytes (die Zahl 72) umgewandelt werden. Das muss es, weil alle Buchstaben, Zahlen, Enums und andere Daten auf der Festplatte oder im Speicher in Bytes festgehalten werden. Und mit der Zeichenpalette können Sie sehen, wie Buchstaben in Zahlen verwandelt werden.

Wählen Sie die Schrift Arial und scrollen Sie nach unten, bis Sie zu den hebräischen Buchstaben kommen.

Klicken Sie auf den Buchstaben, wird in der Statusleiste seine Unicode-Nummer angezeigt. Der Buchstabe Shin hat die Nummer 05E9. Das ist eine Hexadezimalzahl – oder kurz: »Hex-Zahl«.

Diese können Sie mit dem Windows-Rechner umwandeln: Öffnen Sie ihn, wechseln Sie in den Wissenschaftlich-Modus, klicken Sie auf den Radiobutton »Hex«, geben Sie »05E9« ein und klicken Sie dann auf »Dec« – es ist 1.513.



Unicode ist ein Industriestandard, der vom Unicode Consortium entwickelt wird und programm- sowie plattformübergreifend funktioniert. Nehmen Sie sich einen Augenblick Zeit, und gehen Sie zu http://www.unicode.org/.

.NET wandelt Text automatisch in Unicode um



Die beiden C#-Werttypen für die Speicherung von Text – `string` und `char` – halten ihre Daten als Unicode im Speicher. Wenn diese Daten als Bytes geschrieben werden, werden die jeweiligen Unicode-Codenummern in die Datei geschrieben. Starten Sie ein neues Projekt mit einem Formular und drei Buttons und nutzen Sie File.WriteAllBytes() und ReadAllBytes(), um ein Gefühl dafür zu entwickeln, wie Unicode-Daten in eine Datei geschrieben werden.

1 Schreiben Sie einen String in eine Datei und lesen Sie ihn wieder ein.

Nutzen Sie die Methode WriteAllText(), die Sie im Text-Editor verwendet haben, um den ersten Button den String »Eureka!« in die Datei »eureka.txt« schreiben zu lassen. Erzeugen Sie dann ein byte-Array namens eurekaBytes, lesen Sie die Datei darin ein und geben Sie dann die gelesenen Bytes aus:

```
File.WriteAllText("eureka.txt", "Eureka!");
byte[] eurekaBytes = File.ReadAllBytes("eureka.txt");
foreach (byte b in eurekaBytes)
    Console.Write("{0} ", b);
Console.WriteLine();
```

ReadAllBytes() liefert eine Referenz auf ein Byte-Array, das alle Bytes enthält, die aus der Datei gelesen wurden.

In der Ausgabe sehen Sie diese Bytes: `69 117 114 101 107 97 33`. **Öffnen Sie jetzt die Datei im Einfachen Texteditor**, den Sie eben geschrieben haben. Dort steht »Eureka!«.

2 Lassen Sie den zweiten Button die Bytes als Hex-Zahlen anzeigen.

Nicht nur die Zeichentabelle zeigt Hex-Zahlen. Fast alle Sachen, die Sie über die Kodierung von Daten lesen, zeigen Daten hexadezimal. Es ist also nützlich, wenn Sie wissen, wie man damit arbeitet. Geben Sie dem zweiten Button im Formular den gleichen Code wie **dem ersten**, ändern Sie nur folgendermaßen die Zeile Console.Write():

```
Console.Write("{0:x2} ", b);
```

Hexadezimalzahlen nutzen die Zahlen 0 bis 9 und die Buchstaben A bis F, um Zahlen zur Basis 16 darzustellen, 6B entspricht also 107.

Write() gibt den Parameter 0 (der erste nach dem String) dann als Hex-Code mit zwei Zeichen aus. Es werden also die gleichen sieben Zahlen hexadezimal statt dezimal geschrieben: `45 75 72 65 6b 61 21`

3 Lassen Sie den dritten Button hebräische Buchstaben schreiben.

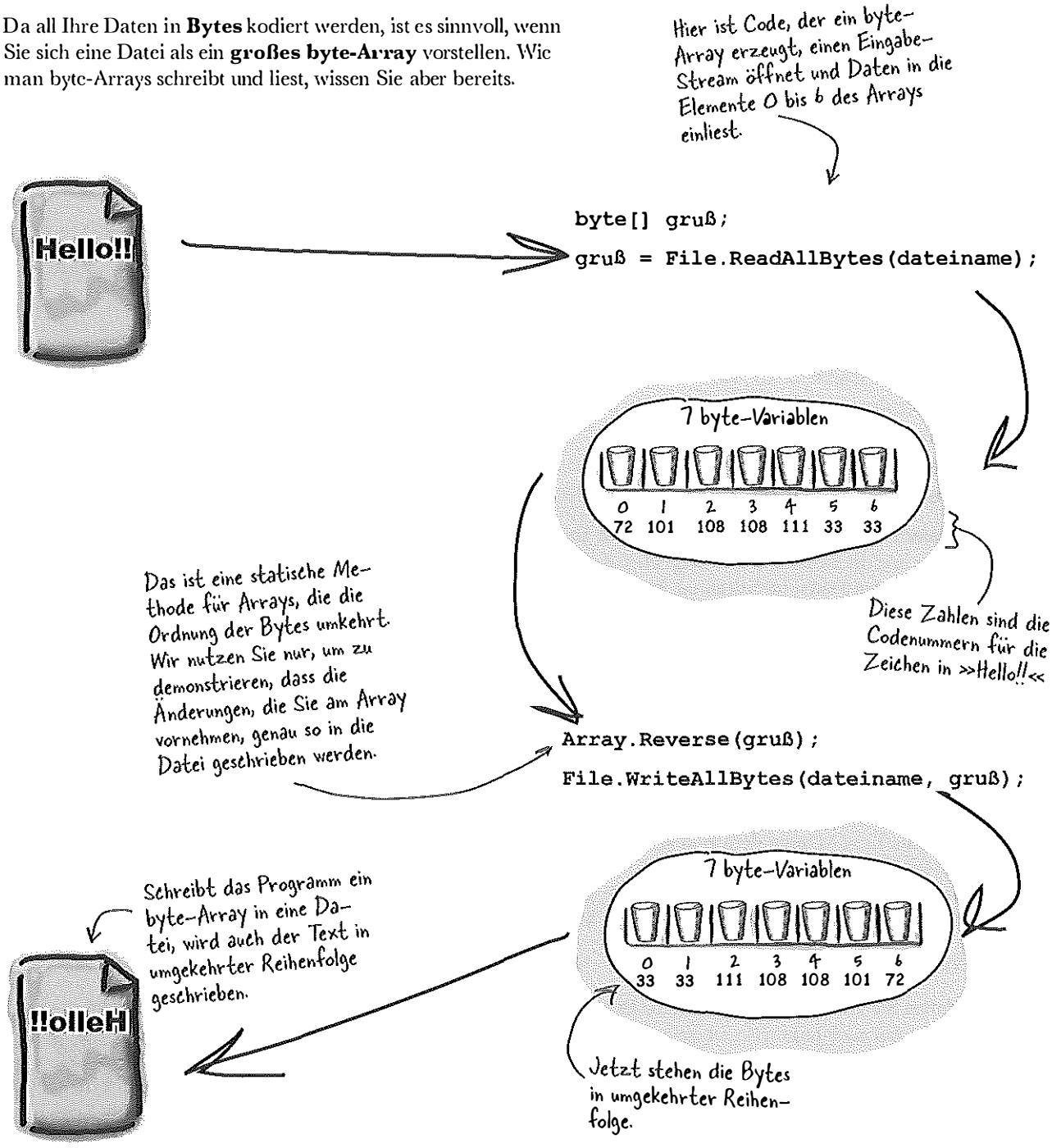
Gehen Sie zur Zeichentabelle, klicken Sie doppelt auf das Zeichen Shin (oder wählen Sie den Button Auswählen). Er wird dann dem Feld »Zeichenauswahl« hinzugefügt. Machen Sie dann das Gleiche für die anderen fünf Buchstaben in »Shalom«: Lamed (U+05DC), Vav (U+05D5) und Mem (U+05DD). Der Code für den dritten Button sieht genau so aus wie der für den zweiten. Ändern Sie den auszugebenden Text, indem Sie in der Zeichentabelle auf »Kopieren« klicken und die Zeichen dann über »Eureka!« kopieren:

```
File.WriteAllText("eureka.txt", "שלום", Encoding.Unicode);
```

Haben Sie gesehen, dass die IDE die Buchstaben **rückwärts** eingefügt hat? Sie weiß, dass man Hebräisch von rechts nach links schreibt, zeigt hebräische Zeichen entsprechend und lässt sie auch so bearbeiten – die Links- und Rechts-Tasten sind vertauscht! Führen Sie jetzt den Code aus und sehen Sie sich die Ausgabe genau an: `ff fe e9 05 dc 05 d5 05 dd 05`. Die ersten beiden Zeichen sind »FF FE«. Das ist die sogenannte Byte Order Mark (BOM), die der Textverarbeitung sagt, in welcher Reihenfolge die beiden Bytes für das Zeichen stehen. Die restlichen Bytes sind die hebräischen Buchstaben – deren Bytes hier in umgekehrter Reihenfolge erscheinen. U+05E9 erscheint also als **e9 05**. Im Text-Editor sieht wieder alles richtig aus!

Mit byte-Arrays können Daten verschoben werden

Da all Ihre Daten in **Bytes** kodiert werden, ist es sinnvoll, wenn Sie sich eine Datei als ein **großes byte-Array** vorstellen. Wie man byte-Arrays schreibt und liest, wissen Sie aber bereits.



Binäre Daten schreiben Sie mit einem BinaryWriter

Sie ***könnten*** alle Ihre Strings, Zeichen und Zahlen in byte-Arrays umwandeln, bevor Sie sie in eine Datei schreiben, aber das wäre ziemlich umständlich. Deswegen gibt Ihnen .NET die nützliche Klasse **BinaryWriter**, die **Ihre Daten automatisch kodiert** und in eine Datei schreibt. Sie müssen nur einen FileStream erzeugen und an den Konstruktor von BinaryWriter übergeben. Dann rufen Sie seine Methoden auf, um Ihre Daten zu speichern. Fügen Sie Ihrem Programm also einen weiteren Button hinzu, damit wir Ihnen zeigen können, wie man BinaryWriter() verwendet.



1. Richten Sie zunächst die zu schreibenden Daten ein.

```
int intWert = 48769414;
string stringWert = "Hello!";
byte[] byteArray = { 47, 129, 0, 116 };
float floatWert = 491.695F;
char charWert = 'E';
```

File.Create() erstellt eine neue Datei – besteht bereits eine mit diesem Namen, wird sie pulverisiert und stattdessen eine neue geschrieben. File.OpenWrite() öffnet eine bestehende Datei und überschreibt die Daten vom Anfang ausgehend.

2. Um den BinaryWriter zu nutzen, müssen Sie erst mit File.Create() einen neuen Stream öffnen:

```
using (FileStream ausgabe = File.Create("binärdaten.dat")) {
  BinaryWriter writer = new BinaryWriter(ausgabe);
}
```

3. Rufen Sie dann einfach die Methode Write() auf. Bei jedem Aufruf werden neue Bytes ans Ende der Datei angehangen, die eine kodierte Version der Daten erhält, die Sie als Parameter übergeben.

```
writer.Write(intWert);
writer.Write(stringWert);
writer.Write(byteArray);
writer.Write(floatWert);
writer.Write(charWert);
```

Jede Write()-Anweisung kodiert einen Wert in Bytes und sendet diese Bytes dann an das FileStream-Objekt. Sie können ihr einen beliebigen Werttyp übergeben. Sie kann alle automatisch kodieren.

Der FileStream hängt die Bytes ans Ende der Datei an.

Spitzen Sie Ihren Bleistift

4. Nutzen Sie jetzt den gleichen Code wie zuvor, um die gerade geschriebenen Daten wieder auf der Datei einzulesen.

```
byte[] geschriebene Daten = File.ReadAllBytes("binärdaten.dat");
foreach (byte b in geschriebene Daten)
    Console.Write("{0:x2} ", b);
Console.WriteLine(" - {0} bytes", geschriebene Daten.Length);
```

Schreiben Sie die Ausgabe unten auf. **Erkennen Sie, welche Bytes** welcher der fünf `Write()`-Anweisungen entsprechen? Markieren Sie die Gruppen mit dem Namen der Variablen.

Ein Tipp: Strings können unterschiedlich lang sein. Der String muss also mit einer Zahl beginnen, die .NET sagt, wie lang er ist. Und die Unicode-Werte für die Zeichen können Sie in der Zeichentabelle nachschlagen.

_____ - ____ Bytes

Die float- und int-Werte brauchen vier Bytes, wenn Sie sie in eine Datei schreiben. long oder double würden sogar jeweils acht Bytes beanspruchen.

Spitzen Sie Ihren Bleistift

Lösung

```
86 29 e8 02 06 48 65 6c 6c 6f 21 2f 81 00 74 f6 d8 f5 43 45 - 20 bytes
```

intWert | stringWert | byteArray | floatWert | charWert

Das erste Byte für den String ist 6 – das ist die Länge des Strings. Mit der Zeichentabelle können Sie die einzelnen Zeichen in >>Hello!<< nachschlagen – diese beginnen mit U+0048 und enden mit U+0021.

Mit dem Windows-Rechner können Sie diese Bytes aus dem Hexadezimalformat ins Dezimalformat umwandeln. Sie sehen dann, dass diese Zahlen im byteArray sind.

char enthält ein Unicode-Zeichen, und 'E' braucht nur ein Byte – es wird als U+0045 kodiert.

Lesen Sie die Daten mit BinaryReader wieder ein

Die Klasse BinaryReader funktioniert genau wie BinaryWriter. Sie erzeugen einen Stream, knüpfen das BinaryReader-Objekt daran und rufen seine Methoden auf. Aber der Reader **weiß nicht, was für Daten sich in der Datei befinden**! Und er kann es auch nicht in Erfahrung bringen. Ihr Float-Wert `491.695F` wurde als `d8 f5 43 45` kodiert. Aber genau diese Bytes können auch den vollkommen gültigen int 1.140.185.334 darstellen. Sie müssen dem BinaryReader also sagen, welche Typen er aus der Datei lesen soll. Fügen Sie Ihrem Formular also einen weiteren Button hinzu und lassen Sie ihn die Daten einlesen, die Sie gerade geschrieben haben.

Vertrauen Sie uns nicht blind. Ersetzen Sie die Zeile, die den float einliest, durch einen Aufruf von ReadInt32(). (Sie müssen den Typ von floatGelesen in int ändern.) Dann können Sie selbst sehen, was aus der Datei gelesen wird.

1. Richten Sie zunächst die FileStream- und BinaryReader-Objekte ein:

```
using (FileStream eingabe = File.OpenRead("binäredaten.dat")) {
    BinaryReader reader = new BinaryReader(eingabe);
```

2. Welche Daten der BinaryReader lesen soll, sagen Sie ihm mithilfe seiner unterschiedlichen Methoden.

```
int intGelesen = reader.ReadInt32();
string stringGelesen = reader.ReadString();
byte[] byteArrayGelesen = reader.ReadBytes(4);
float floatGelesen = reader.ReadSingle();
char charGelesen = reader.ReadChar();
```

BinaryReader() hat eine eigene Methode für jeden Werttyp, der die Daten des richtigen Typs liefert. Die meisten benötigen keine Parameter, aber ReadBytes() nimmt einen, der sagt, wie viele Bytes gelesen werden sollen.

3. Geben Sie dann die gelesenen Daten auf der Konsole aus:

```
    Console.Write("int: {0}  String: {1}  Bytes: ", intGelesen, stringGelesen);
    foreach (byte b in byteArrayGelesen)
        Console.Write("{0} ", b);
    Console.Write(" float: {0}  char: {1} ", floatGelesen, charGelesen);
}
```

So sieht die Ausgabe aus, die auf der Konsole ausgegeben wird:

```
int: 48769414  String: Hello!  Bytes: 47 129 0 116  float: 491.695  char: E
```

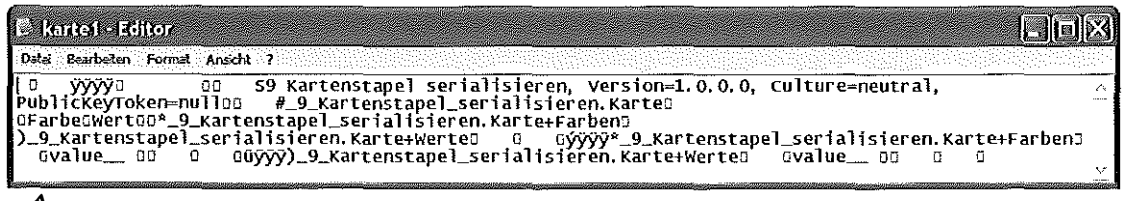
Sie können serialisierte Dateien auch manuell lesen und schreiben

Im Editor geöffnet, sehen serialisierte Dateien nicht so schön aus. Alle Dateien, die Sie schreiben, finden Sie im »bin/Debug«-Ordner Ihres Projekts – nehmen wir uns eine Minute und machen wir uns besser mit der Funktionsweise der Serialisierung vertraut.

Tun Sie das!

1 **Serialisieren Sie zwei Karte-Objekte in verschiedene Dateien.**
Nutzen Sie den bereits geschriebenen Serialisierungscode, um die **Herz-Drei** in `karte1.dat` und die **Pik-Sechs** in `karte2.dat` zu schreiben. Prüfen Sie, ob beide Dateien geschrieben wurden und sich jetzt im angegebenen Ordner befinden und ob beide die gleiche Dateigröße haben. Öffnen Sie sie dann im Editor:

In der Datei sind einige Wörter wie »Karte« (der Name der Klasse) lesbar, aber das meiste ist unlesbar.



2 **Schreiben Sie eine Schleife, die die beiden Dateien vergleicht.**
Wir haben ReadByte() verwendet, um das nächste Byte aus dem Stream zu lesen – es wird ein `int` geliefert, der den Wert des Bytes enthält. Außerdem haben wir das `Length`-Feld des Streams verwendet, um sicherzustellen, dass wir die gesamte Datei lesen.

```
byte[] ersteDatei = File.ReadAllBytes("karte1.dat");
byte[] zweiteDatei = File.ReadAllBytes("karte2.dat");
for (int i = 0; i < ersteDatei.Length; i++)
    if (ersteDatei[i] != zweiteDatei[i])
        Console.WriteLine("Byte {0}: {1} versus {2}",
            i, ersteDatei[i], zweiteDatei[i]);
```

Diese Schleife vergleicht nacheinander die entsprechenden Bytes der beiden Dateien. Findet sie einen Unterschied, gibt sie eine Zeile auf der Konsole aus.

Die beiden Dateien werden in verschiedene Byte-Arrays eingelesen, damit sie Byte-weise verglichen werden können. Da Objekte der gleichen Klasse in zwei verschiedene Dateien serialisiert wurden, sollten die Arrays fast identisch sein ... aber schauen wir uns einfach mal an, WIE identisch sie sind.



Aufgepasst

Wenn Sie eine Datei schreiben, beginnen Sie nicht immer bei null!

Passen Sie auf, wenn Sie File.OpenWrite() verwenden. Es löscht die Datei nicht – sondern beginnt nur, die Daten vom Anfang ausgehend zu überschreiben. Deswegen nutzen wir File.Create() – es erstellt eine neue Datei.

Wir sind noch nicht fertig - blättern Sie um!

Die Dateien an den Punkten ändern, an denen sie sich unterscheiden

Die Schleife, die Sie gerade geschrieben haben, hebt genau die Punkte hervor, an denen sich die Dateien mit den serialisierten Karten unterscheiden. Da die einzigen Unterschiede zwischen den beiden Objekten die Felder Farbe und Wert sind, sollte das auch der einzige Unterschied zwischen den Dateien sein. Wenn wir die Bytes finden, die die Werte für Farbe und Wert enthalten, sollten wir diese **ändern können, um eine neue Karte** mit beliebigen Werten für Farbe und Wert zu erzeugen.

3. **Schauen Sie in der Konsole nach, wo sich die beiden Dateien unterscheiden.**
Die Konsole sollte zeigen, dass zwei Bytes unterschiedlich sind:

```
Byte 327: 1 versus 2
Byte 396: 3 versus 6
```

Und das scheint sinnvoll zu sein! Gehen Sie zum `Karte.Farben`-Enum aus dem letzten Kapitel, werden Sie sehen, dass der Wert für Herz 1 ist und der für Pik 3. Das ist also der erste Unterschied. Und der zweite Unterschied – 3 versus 6 – ist offensichtlich der Wert der Karten. Sie sehen vielleicht andere Byte-Werte, aber das ist nicht überraschend: Vielleicht verwenden Sie ja einen anderen Namensraum, der die Länge der Datei verändert.

Auch der Namensraum ist Teil der serialisierten Datei. Wenn Sie einen anderen Namensraum verwenden, dann haben Sie hier anderen Byte-Numrnern.

Hmm, repräsentiert Byte 32.7 in der serialisierten Datei die Farbe, sollten wir die Farbe ändern können, indem wir die Datei einlesen, dieses Byte ändern und die Datei wieder speichern. (Denken Sie daran, dass Ihre eigene serialisierte Datei die Farbe eventuell an einem anderen Ort speichert.)

4. **Schreiben Sie Code, der manuell eine Datei mit einem Karo-König erzeugt.**
Wir nehmen eins der Arrays, die wir eingelesen haben, ändern es so, dass es eine neue Karte enthält, und speichern es wieder in einer Datei.

```
ersteDatei[327] = (byte)Karte.Farben.Karo;
ersteDatei[396] = (byte)Karte.Values.König;
File.Delete("karte3.dat");
File.WriteAllBytes("karte3.dat", ersteDatei);
```

Wenn Sie in Schritt 3 andere Nummern ermittelt haben, müssen Sie diese auch hier verwenden.

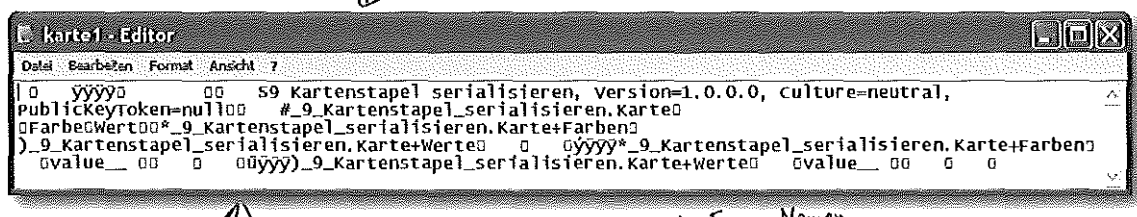
Deserialisieren Sie die Karte aus `karte3.dat` und prüfen Sie, ob es der Karo-König ist.

Da Sie jetzt wissen, welche Bytes die Farbe und den Wert enthalten, können Sie nur diese Bytes im Array ändern, bevor es in karte3.dat gespeichert wird.

Die Arbeit mit Binärdateien kann kompliziert sein

Was machen Sie, wenn Sie eine Datei vor sich haben und nicht sicher sind, was sich darin befindet? Sie wissen nicht, von welcher Anwendung sie erstellt wurde, und wissen auch sonst nichts über sie – aber wenn Sie sie im Editor öffnen, sieht es nach einem Haufen Müll aus. Was ist, wenn Sie alle anderen Möglichkeiten ausgeschöpft haben, sich die Datei aber unbedingt ansehen müssen? Es ist ziemlich offensichtlich, dass der Editor nicht das richtige Werkzeug ist.

Hier ist die serialisierte Karte im Editor. Das ist völlig unbrauchbar.

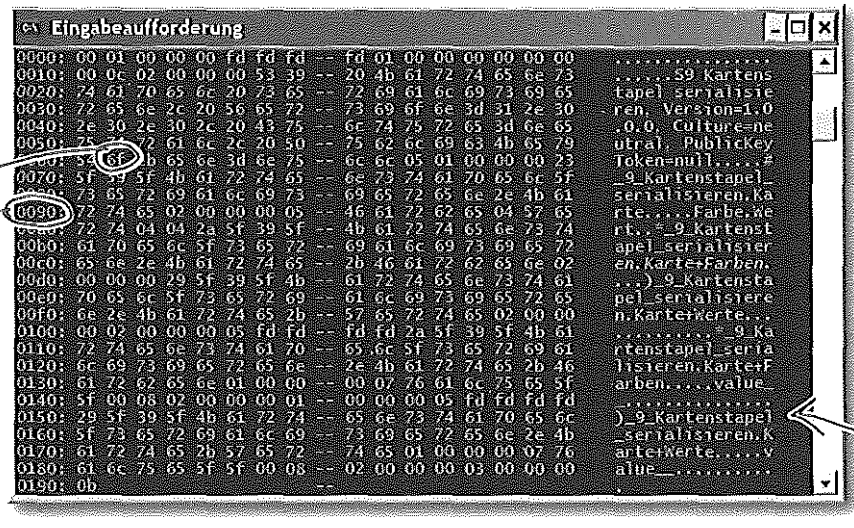


Ein paar Dinge können Sie erkennen – die Enum-Namen (>>Farbe<< und >>Wert<<) und den Namen des Namensraums (>>9 Kartenstapel serialisieren<<). Aber das bringt uns nichts.

Es gibt eine weitere Möglichkeit – es ist ein Format, das als »Hex-Dump« bezeichnet wird und ein Standardverfahren für einen Blick auf binäre Daten ist. Es ist entschieden informativer, als die Datei im Editor zu betrachten. Hexadezimal kann man bequem die Bytes in einer Datei einsehen. Jedes Byte wird hexadezimal mit zwei Zeichen dargestellt, Sie können also eine Menge Daten auf sehr kleinem Raum in einem Format betrachten, in dem man leicht Muster erkennen kann. Außerdem ist es nützlich, binäre Daten in Zeilen anzuzeigen, die 8, 16 oder 32 Bytes lang sind, weil die meisten binären Daten in Happen von 4, 8, 16 oder 32 Bytes aufgeteilt werden ... wie alle Typen in C#. Ein `int` benötigt beispielsweise 4 Bytes und ist, auf die Festplatte serialisiert, 4 Bytes lang. So sieht die gleiche Datei als Hex-Dump aus, der mit jedem der verfügbaren freien Hex-Dump-Programme für Windows erzeugt werden kann.

Sie können sofort die numerischen Werte der einzelnen Bytes erkennen.

Die Zahl am Anfang der einzelnen Zeilen ist der Offset (oder Abstand vom Dateianfang) des ersten Bytes in der Zeile.



Sie sehen den ursprünglichen Text immer noch, aber die ganzen Müllzeichen wurden entfernt.

Mit Datei-Streams ein Hex-Dump-Programm erstellen

Ein **Hex-Dump** ist eine *hexadezimale* Darstellung des Inhalts einer Datei. Das ist ein gebräuchliches Mittel, mit dem sich Programmierer Einblick in die internen Strukturen einer Datei verschaffen. Die meisten Betriebssysteme werden mit einem entsprechenden Programm ausgeliefert. Windows unglücklicherweise nicht. Erstellen wir also unser eigenes!

Wie man einen Hex-Dump erstellt

Beginnen wir mit einem wohlbekannten Text:

```
Die Würde des Menschen ist unantastbar. Sie zu achten und zu schützen ist ...
```

Ein Hex-Dump dieses Texts würde so aussehen:

Wieder sehen Sie unmittelbar die numerischen Werte der einzelnen Bytes.

```
0000: 44 69 65 20 57 fd 72 64 -- 65 20 64 65 73 20 4d 65    Die W.rde des Me
0010: 6e 73 63 68 65 6e 20 69 -- 73 74 20 75 6e 61 6e 74    nschen ist unant
0020: 61 73 74 62 61 72 2e 20 -- 53 69 65 20 7a 75 20 61    astbar. Sie zu a
0030: 63 68 74 65 6e 20 75 6e -- 64 20 7a 75 20 73 63 68    chten und zu sch
0040  fd 74 7a 65 6e 20 69 73 -- 74 20 2e 2e 2e             .tzen ist ...
```

Wir haben jeder Zeile den Offset des ersten Bytes in der Datei vorangestellt.

Und wir müssen den Müll durch Punkte ersetzen.

Jede dieser Zahlen – 57, 65, 6F – ist der Wert eines Bytes in der Datei. Einige dieser »Zahlen« enthalten Buchstaben, weil es *hexadezimale* Werte (oder Hex-Werte) sind. Das ist eine andere Notation für Zahlen. Anstelle von zehn Ziffern von 0 bis 9 werden 16 Ziffern von 0 bis 9 zuzüglich der Buchstaben A bis F verwendet.

Jede Zeile in unserem Hex-Dump repräsentiert 16 Zeichen aus der Eingabe, aus der sie generiert wurde. Die ersten vier Zeichen in unserem Hex-Dump sind die Verschiebung in der Datei – die erste Zeile beginnt beim Zeichen 0, die nächste beim Zeichen 16 (oder Hex 10), dann folgt Zeichen 32 (Hex 20) usw. (Andere Hex-Dumps sehen etwas anders aus, aber das ist für uns ausreichend.)

Mit Hex-Zahlen arbeiten

Hex-Zahlen können Sie direkt in Ihrem Programm angeben – stellen Sie der Zahl einfach nur die Zeichen `0x` voran:

```
int j = 0x20;
MessageBox.Show("Der Wert ist " + j);
```

Hängen Sie mit dem +-Operator eine Zahl an einen String an, wird dieser in eine Dezimalzahl umgewandelt. Mit der statischen Methode `String.Format()` können Sie Ihre Zahl stattdessen in einen hexadezimal formatierten String verwandeln:

```
string h = String.Format("{0:x2}", j);
```

String.Format() nutzt Parameter wie Console.WriteLine(). Sie müssen also nichts Neues lernen, um damit zu arbeiten.

StreamReader und StreamWriter reichen vollkommen aus

Unser Hex-Dump-Programm schreibt in eine Datei, und da es nur Text schreibt, ist ein StreamWriter vollkommen in Ordnung. Aber wir können zusätzlich die **ReadBlock()**-Methode von StreamReader nutzen. Diese liest einen Block von Zeichen in ein char-Array ein – Sie geben die Anzahl von Zeichen an, die Sie lesen möchten, und sie liest entweder so viele Zeichen oder, falls in der Datei nicht mehr so viele vorhanden sind, den gesamten Rest der Datei. Da wir auf jeder Zeile 16 Zeichen anzeigen, werden wir Blöcke mit 16 Zeichen einlesen.

Fügen Sie Ihrem Programm also einen weiteren Button hinzu – und knüpfen Sie an ihn dieses Hex-Dump-Programm. Ändern Sie die ersten beiden Zeilen so, dass sie auf echte Dateien auf Ihrer Festplatte zeigen. Beginnen Sie mit einer serialisierten Karte-Datei. Schauen Sie dann, ob Sie den Code so ändern können, dass Speichern- und Öffnen-Dialoge angezeigt werden.

```
using (StreamReader reader = new StreamReader(@"C:\Ordner\EingabeDatei.txt"))
using (StreamWriter writer = new StreamWriter(@"C:\Ordner\AusgabeDatei.txt", false))
{
  int position = 0;

  while (!reader.EndOfStream) {
    char[] puffer = new char[16];
    int geleseneZeichen = reader.ReadBlock(puffer, 0, 16);
    writer.Write("{0}: ", String.Format("{0:x4}", position));
    position += geleseneZeichen;

    for (int i = 0; i < 16; i++) {
      if (i < geleseneZeichen) {
        string hex = String.Format("{0:x2}", (byte)puffer[i]);
        writer.Write(hex + " ");
      }
      else
        writer.Write("   ");
      if (i == 7) { writer.Write("-- "); }
      if (puffer[i] < 32 || puffer[i] > 250) { puffer[i] = '.'; }
    }
    string pufferInhalt = new string(puffer);
    writer.WriteLine("  " + pufferInhalt.Substring(0, geleseneZeichen));
  }
}
```

Das **EndofStream**-Feld des StreamReader liefert false, wenn in der Datei keine zu lesenden Zeichen verbleiben.

Dieser ReadBlock()-Aufruf liest bis zu 16 Zeichen in ein **char**-Array ein.

Die statische Methode String.Format wandelt Zahlen in Strings um. "{0:x4}" sagt Format(), dass der zweite Parameter – hier **position** – als 4-Zeichen-Hex-Zahl geschrieben werden soll.

Diese Schleife durchläuft die Zeichen und gibt sie in einer Zeile der Ausgabe aus.

Einige der Zeichen mit einem Zeichenwert kleiner 32 sind nicht druckbar, deswegen ersetzen wir sie mit einem Punkt.

Ein **char[]**-Array können Sie in einen String umwandeln, indem Sie es dem überladenen Konstruktor für **string** übergeben.

Strings bieten die Methode **Substring**, die einen Teil des Strings liefert. Hier liefert sie die ersten **geleseneZeichen**-Zeichen des Strings, beginnend mit der Position 0, dem Anfang des Strings. (Gehen Sie noch einmal nach oben an die Stelle, an der **geleseneZeichen** gesetzt wird – die Methode **ReadBlock()** liefert die Anzahl der Zeichen, die in das Array eingelesen werden.)

Es gibt keine Dummen Fragen

F: Warum musste ich die Datei nicht mit `Close()` schließen, nachdem ich `File.ReadAllText()` und `File.WriteAllText()` verwendet habe?

A: Die Klasse `File` bietet mehrere praktische statische Methoden, die automatisch eine Datei öffnen, Daten lesen oder schreiben und die Datei anschließend automatisch schließen. Neben `ReadAllText()` und `WriteAllText()` gibtes noch`ReadAllBytes()` und `WriteAllBytes()`, die mit Byte-Arrays arbeiten, sowie `ReadAllLines()` und `WriteAllLines()`, die String-Arrays lesen und schreiben, in denen jeder String eine Zeile aus der Datei ist. All diese Methoden öffnen und schließen die Streams automatisch. Sie können die gesamten Dateioperationen also in einer einzigen Anweisung abwickeln.

F: Wenn `FileStream`Methoden zum Lesen und Schreiben bietet, warum muss ich dann überhaupt `StreamReader` oder `StreamWriter` verwenden?

A: Die Klasse `FileStream` ist wirklich praktisch, wenn es ums Schreiben oder Lesen von Bytes in Binärdateien geht. Ihre Methoden zum Schreiben und Lesen arbeiten mit Bytes und Byte-Arrays. Aber viele Programme arbeiten ausschließlich mit Textdateien, wie die erste Version des Entschuldigungsgenerators, die Strings in Dateien schrieb. Dann sind `StreamReader` und `StreamWriter` praktisch. Beide bieten Methoden, die speziell für das Lesen und Schreiben von Text gedacht sind. Möchten Sie eine Zeile aus einer Textdatei lesen, müssten Sie ohne sie erst ein Byte-Array einlesen und dann eine Schleife aufbauen, die in diesemArray nach einem Zeilenumbruch sucht – Sie sehen also, dass sie uns das Leben leichter machen.

F: Wann sollte Ich `File` nutzen und wann `FileInfo`?

A: Der Hauptunterschied zwischen `File` und `FileInfo` ist, dass die Methoden in `File` statisch sind, sie müssen also keine Klasseninstanz erstellen. `FileInfo` andererseits verlangt, dass sie es mit einem Dateinamen initialisieren. In einigen Fällen wäre das mühevoller, beispielsweise wenn Sie nur eine Dateioperation durchführen müssen (z.B. eine Datei löschen oder verschieben). Müssen Sie auf einer Datei viele Operationen durchführen, ist `FileInfo` effizienter, weil Sie den Dateinamen nur einmal übergeben müssen. Was Sie verwenden, sollten Sie anhand der jeweiligen Situation entscheiden. Genauer gesagt: Nutzen Sie `File`, wenn Sie eine Dateioperation durchführen; nutzen Sie `FileInfo`, wenn Sie hintereinander viele Dateioperationen durchführen müssen.

F: Augenblick. Wieso wurde »Eureka!« mit nur einem Byte pro Zeichen geschrieben, während die hebräischen Buchstaben jeweils zwei Bytes pro Zeichen in Anspruch nahmen? Und was war dieses »FF FE«-Ding vor den Bytes?

A: Weil wir einen kleinen Trick verwendet haben, um Ihnen den Unterschied zwischen zwei **eng verwandten** Unicode-Kodierungen zu zeigen. Für den hebräischen Text haben wir mit dem dritten Parameter für WriteAllText() (Encoding.Unicode) explizit die Unicode-Kodierung angefordert. Der technische Name für diese spezielle Kodierung für den Unicode-Zeichensatz ist UTF16. In ihr werden alle 65.536 Zeichen jeweils mit zwei Bytes dargestellt, die den Zeichennummern im Unicode-Zeichensatz entsprechen. Sie wird deswegen auch einfach als Unicode-Kodierung bezeichnet. Die dem Text voranstehende Byte-Folge FF FE ist die sogenannte Byte-Ordnung-Markierung, BOM. Sie gibt an, in welcher Reihenfolge die beiden Bytes für ein Zeichen stehen.

Bei Texten, die mit einer geringen Anzahl von Zeichen geschrieben werden, kann diese 2-Byte-Kodierung Platzverschwendung sein. Das Unicode-Konsortium hat deswegen die Kodierung UTF8 entwickelt, die die Zeichen des Unicode-Zeichensatzes mit ein bis vier Bytes kodiert. UTF8 kodiert die ersten 127 Zeichen des Unicode-Zeichensatzes mit einem Byte, die Zeichen 128 bis 2.047 mit zwei Bytes und alle weiteren Zeichen mit vier Bytes. Die 24 Buchstaben des lateinischen Alphabets, Zahlen, gebräuchliche Interpunktionszeichen und einige Standardzeichen (wie geschweifte Klammern und andere Dinge, die Sie auf Ihrer Tastatur sehen), die eine niedrige Unicode-Nummer zwischen 0 und 127 haben, werden mit einem Byte dargestellt. (Da diese den Nummern im alten ASCII-Zeichensatz entsprechen, hat das den Vorteil, dass alte ASCII-kodierte Texte weiterverwendet werden können.) Zeichen wie deutsche Umlaute oder die von uns verwendeten hebräischen Buchstaben werden mit zwei Bytes dargestellt, chinesische Schriftzeichen usw. mit vier Bytes. UTF8 ist die wichtigste und gebräuchlichste Unicode-Kodierung. Es ist die Kodierung, die das .NET Framework standardmäßig verwendet.

F: Warum heißt das Ding Byte-Ordnung-Markierung?

A: Erinnern Sie sich, dass die Bytes für unsere Werte in umgekehrter Reihenfolge standen? Der Unicode-Wert für Shin ist U+05E9 und wurde als E9 05 in der Datei gespeichert. Das nennt man »Little Endian«. Kehren Sie zu dem Code zurück, der diese Bytes geschrieben hat, und ändern Sie den dritten Parameter für WriteAllText() in Encoding.BigEndianUnicode. Das sagt, dass die Daten in »Big Endian« geschrieben werden sollen. Die Bytes werden dann umgekehrt. Diesmal werden die Bytes in der Folge »05 E9« geschrieben. Außerdem sehen Sie eine andere BOM: »FE FF«. Und Ihr Einfacher Texteditor ist so schlau, dass er beides lesen kann!



Ändern Sie Brians Entschuldigungsgenerator so, dass er binäre Dateien mit serialisierten Entschuldigung-Objekten anstelle von Textdateien verwendet.

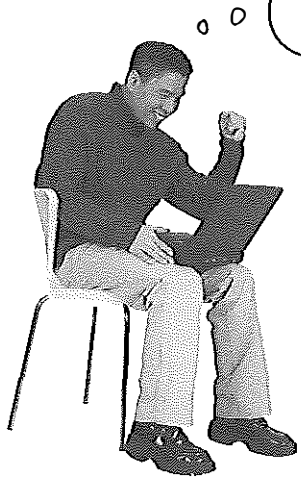
4. **Machen Sie die Klasse Entschuldigung serialisierbar.**
Markieren Sie die Klasse Entschuldigung mit dem Attribut [Serializable], um sie serialisierbar zu machen. Außerdem müssen Sie diese using-Zeile hinzufügen:
`using System.Runtime.Serialization.Formatters.Binary;`

2. **Lassen Sie die Methode Entschuldigung.Speichern() die Entschuldigung serialisieren.**
Schreibt die Methode Speichern() eine Datei in den Ordner, sollte sie keinen StreamWriter verwenden, sondern eine Datei öffnen und sich selbst in diese serialisieren. Sie müssen herausfinden, wie sich die aktuelle Klasse deserialisieren kann.

Tipp: Welches Schlüsselwort können Sie in einer Klasse verwenden, um die Klasse selbst zu referenzieren?

3. **Lassen Sie die Methode Entschuldigung.DateiÖffnen() eine Entschuldigung deserialisieren.**
Sie müssen ein temporäres Entschuldigung-Objekt erstellen, um aus einer Datei zu deserialisieren, und die Felder dann in die aktuelle Klasse kopieren.

4. **Ändern Sie das Formular so, dass es eine neue Dateierweiterung verwendet.**
Am Formular müssen Sie nur eine kleine Änderung vornehmen. Da wir nicht mehr mit Textdateien arbeiten, sollten wir die Erweiterung `.txt` nicht mehr verwenden. Ändern Sie die Dialoge, die Standard-Dateinamen und den Code zur Suche von Verzeichnissen, damit diese mit `*.entsch`-Dateien arbeiten.



Wow, das war echt leicht! Der gesamte Code zum Speichern und Öffnen von Entschuldigungen steckt in der Klasse Entschuldigung. Ich musste nur die Klasse ändern – das Formular musste ich fast nicht anrühren. Das ist fast so, als würde sich das Formular überhaupt nicht dafür interessieren, wie die Klasse ihre Daten speichert. Es übergibt nur den Dateinamen und weiß, dass alles ordentlich gespeichert wird.

Das stimmt! Ihr Code ließ sich leicht ändern, weil die Klase gut gekapselt ist.

Wenn Ihre Klasse ihre internen Operationen vor dem Rest des Programms verbirgt und nur Verhalten veröffentlicht, die veröffentlicht werden müssen, nennt man sie **gut gekapselt**. In der Entschuldigungsverwaltung braucht das Formular keine Informationen dazu, wie Entschuldigungen in einer Datei gespeichert werden. Es übergibt der Klasse Entschuldigung bloß den Dateinamen, und die Klasse kümmert sich um den Rest. Das macht es sehr leicht, große Änderungen daran vorzunehmen, wie Ihre Klasse mit Dateien arbeitet. Je besser Ihre Klassen gekapselt sind, umso leichter lassen sie sich später ändern.

Erinnern Sie sich daran, dass Kapselung eins der vier Kernprinzipien von OOP ist? Hier ist ein Beispiel dafür, wie diese Prinzipien Ihre Programme verbessern.



LÖSUNGEN ZU DEN ÜBUNGEN

Ändern Sie Brians Entschuldigungsgenerator so, dass er binäre Dateien mit serialisierten Entschuldigung-Objekten anstelle von Textdateien verwendet.

Im Formular müssen Sie nur drei Zeilen ändern: zwei Zeilen im Click-Event des Speichern-Buttons und zwei in dem des Öffnen-Buttons – sie sorgen nur dafür, dass die Dialoge die Erweiterung entschuldigung verwenden.

```
private void speichern_Click(object sender, EventArgs e) {
  // vorhandener Code
  saveFileDialog1.Filter =
    "Entschuldigung-Dateien (*.entsch)|*.entsch|Alle Dateien(*.*)|*.*";
  saveFileDialog1.FileName = beschreibung.Text + ".entsch";
  // vorhandener Code
}
private void öffnen_Click(object sender, EventArgs e) {
  // vorhandener Code
  openFileDialog1.Filter =
    "Entschuldigung-Dateien (*.entsch)|*.entsch|Alle Dateien(*.*)|*.*";
  // vorhandener Code
}
```

Hier reichen gewöhnliche Speichern- und Öffnen-Dialoge.

Im Formular muss nur die Dateinamenerweiterung geändert werden, die an die Klasse Entschuldigung übergeben wird.

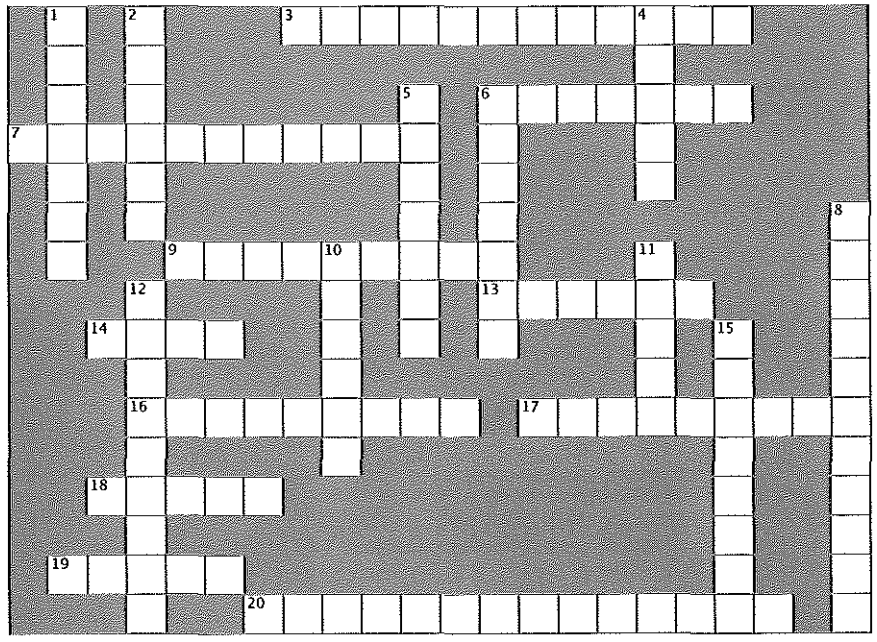
```
[Serializable]
public class Entschuldigung {
    public string Beschreibung;
    public string Reaktion;
    public DateTime LetzteVerwendung;
    public string Dateipfad;
    public Entschuldigung() {
        Dateipfad = "";
    }
    public Entschuldigung(string dateipfad) {
        DateiÖffnen(dateipfad);
    }
    public Entschuldigung(Random zufall, string ordner) {
        string[] dateinamen = Directory.GetFiles(ordner, "*.entsch");
        DateiÖffnen(dateinamen[zufall.Next(dateinamen.Length)]);
    }
    private void DateiÖffnen(string dateipfad) {
        this.Dateipfad = dateipfad;
        BinaryFormatter formatter = new BinaryFormatter();
        Entschuldigung tempExcuse;
        using (Stream eingabe = File.OpenRead(dateipfad)) {
            tempExcuse = (Entschuldigung)formatter.Deserialize(eingabe);
        }
        Beschreibung = tempExcuse.Beschreibung;
        Reaktion = tempExcuse.Reaktion;
        LetzteVerwendung = tempExcuse.LetzteVerwendung;
    }
    public void Speichern(string dateipfad) {
        BinaryFormatter formatter = new BinaryFormatter();
        using (Stream ausgabe = File.OpenWrite(dateipfad)) {
            formatter.Serialize(ausgabe, this);
        }
    }
}
```

Der Konstruktor für das Laden einer zufälligen Entschuldigung muss nach der Erweiterung >>.entsch<< anstelle der Erweiterung >>*.txt<< suchen.

Wir übergeben >>this<<, weil dieses Objekt serialisiert werden soll.



Datei-Kreuzworträtsel

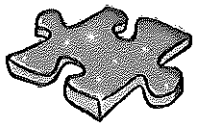


Waagerecht

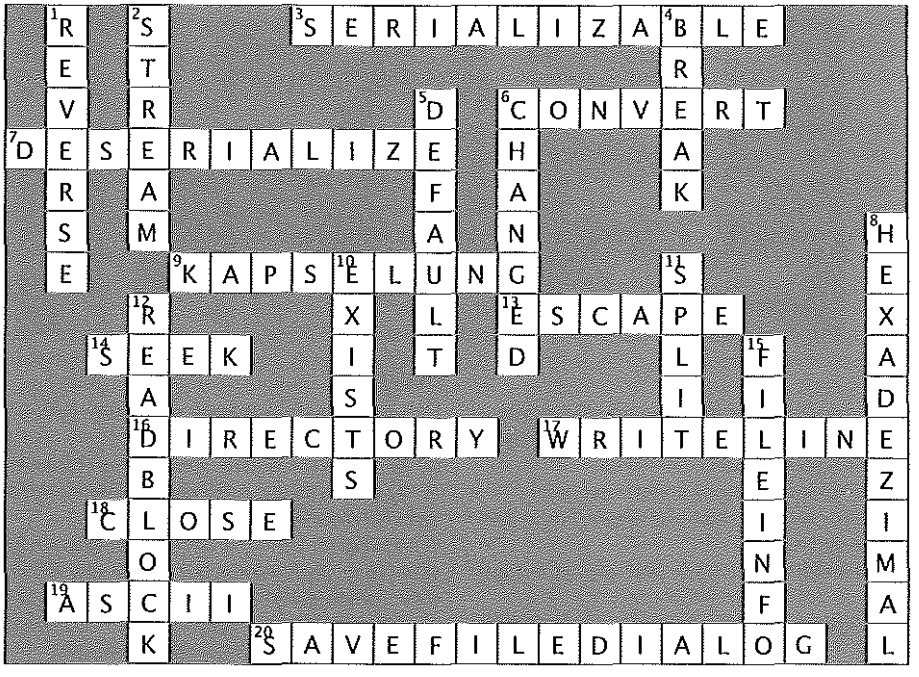
3. Geben Sie dieses Attribut nicht an, um anzuzeigen, dass eine Klasse in einen Stream geschrieben werden kann, meldet BinaryFormatter einen Fehler.
6. Diese Klasse hat viele statische Methoden wie ToDateTime(), über die Sie Daten von einem Typ in den anderen umwandeln können.
7. Diese BinaryFormatter-Methode liest ein Objekt aus einem Stream.
9. Dieses OOP-Prinzip vereinfacht die Wartung Ihres Codes.
13. \n und \r sind Beispiele für derartige Sequenzen.
14. Die Stream-Methode, mit der Sie zu einem beliebigen Punkt im Stream springen können.
16. Diese Klasse bietet viele statische Methoden zur Arbeit mit Verzeichnissen.
17. Diese Methode sendet Text und einen Zeilenumbruch an einen Stream.
18. Rufen Sie diese Methode nicht auf, kann es passieren, dass ein Stream gesperrt bleibt und von anderen Programmen nicht geöffnet werden kann.
19. Ein Kodierungssystem, das nur 256 Zeichen referenzieren kann.
20. Ein nicht sichtbares Steuerelement, über das Sie einen standardmäßigen »Speichern unter«-Dialog einblenden können.

Senkrecht

1. Diese statische Methode der Klasse Array liefert das Array rückwärts.
2. Die akstrakte Klasse, von der FileStream erbt.
4. Diese Anweisung zeigt das Ende eines Falls in einer switch-Anweisung an.
5. Mit diesem Schlüsselwort geben Sie in einer switch-Anweisung die Anweisungen an, die ausgeführt werden sollen, wenn keiner der anderen Fälle passt.
6. Dieses Event wird abgesetzt, wenn Daten in einem Eingabesteuerelement verändert werden.
8. Zahlen zur Basis 16.
10. Eine Methode in der Klasse File, die prüft, ob sich eine bestimmte Datei auf dem Laufwerk befindet.
11. Über diese Methode, die jeder String hat, können Sie ihn anhand von Trennzeichen in ein Array aufspalten.
12. Die StreamReader-Methode, die Daten in ein char[]-Array einliest.
15. Mit dieser Klasse können Sie alle Operationen in der Klasse File auf einer bestimmten Datei durchführen.



Datei-Kreuzworträtsel, Lösung



10 Exception-Handling

Fehler-Prävention



Programmierer sind keine Feuerlöscher.

Sie haben sich auf die Hinterbeine gesetzt, einen Haufen trockener Handbücher und ein paar ansprechende Von Kopf bis Fuß-Bücher durchgearbeitet und haben den Gipfelpunkt Ihres Berufsstands erreicht: **Meisterprogrammierer**. Aber Sie erhalten immer noch Beschwerden, weil **Ihr Programm abstürzt** oder **sich nicht so verhält, wie es sich verhalten soll.** Nichts kann einem die Programmierstimmung so verhageln wie die Forderung, einen seltsamen Fehler zu beheben ... aber mit **Exception-HandlIng** kann sich Ihr Code **um Probleme kümmern,** die aufkommen. Und Sie können auf diese Probleme sogar reagieren und dafür sorgen, **dass das Programm welterläuft**.

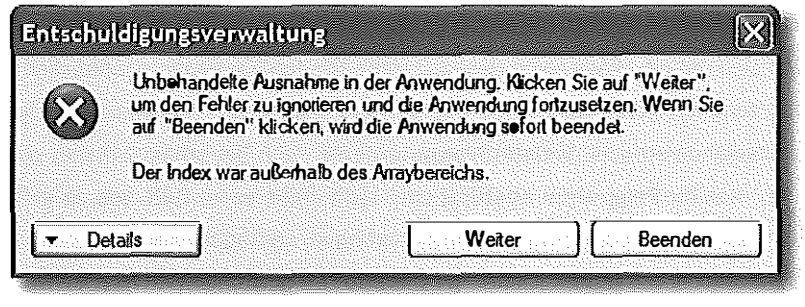
Brians Entschuldigungen müssen mobil werden

Kürzlich wurde Brian in die Abteilung International versetzt. Jetzt jettet er quer durch die ganze Welt. Aber er muss sich weiterhin um seine Entschuldigungen kümmern. Deswegen hat er das von Ihnen erstellte Programm auf seinem Laptop installiert und nimmt es mit, wo immer er hingeht.



Aber das Programm funktioniert nicht

Brian klickt auf den Button »Schnelle Ausrede« und erhält einen ziemlich hässlichen Fehler. Irgendwie konnten seine Entschuldigungen nicht gefunden werden. Warum das?



Eine unbehandelte Exception ... das muss ein Problem sein, das wir nicht berücksichtigt haben.

Spitzen Sie Ihren Bleistift

Hier ist ein weiteres Beispiel für fehlerhaften Code. Dieser Code löst fünf verschiedene Exceptions aus. Die entsprechenden Fehlermeldungen sehen Sie rechts. Ordnen Sie die Fehlermeldungen den Codezeilen zu, die sie verursachen. Hinweise finden Sie in den Fehlermeldungen.

```
public static void BienenVerwaltung() {
    object dieBiene = new HonigBiene(36.5, "Willi");
    float honigMenge = (float)dieBiene;

    HonigBiene andereBiene = new HonigBiene(12.5, "Maja");
    double bienenname = double.Parse(andereBiene.Name);

    double honigGesamt = 36.5 + 12.5;
    string versorgteBienen = "";
    for (int i = 1; i < (int) honigGesamt; i++) {
        versorgteBienen += i.ToString();
    }
    float f =
      float.Parse(versorgteBienen);

    int drohnen = 4;
    int königinnen = 0;
    int drohnenProKönigin = drohnen / königinnen;

    andereBiene = null;
    if (drohnenProKönigin < 10) {
        andereBiene.ArbeitErledigen();
    }
}
```

double.Parse("32") parst einen String und liefert einen double-Wert wie 32.

Eine Referenz, die auf nichts zeigt, hat den besonderen Wert `null`. Wird eine Referenz auf `null` gesetzt, sagt das C#, dass sie auf nichts mehr zeigt.

1

OverflowException wurde nicht behandelt.

Der Wert für ein einzelnes Byte war zu groß oder zu klein.

2

NullReferenceException wurde nicht behandelt.

Der Objektverweis wurde nicht auf eine Objektinstanz festgelegt.

3

InvalidCastException wurde nicht behandelt.

Die angegebene Umwandlung ist ungültig.

4

DivideByZeroException wurde nicht behandelt.

Es wurde versucht, durch 0 (null) zu teilen.

Hinweise zur Fehlerbehebung:

Der Wert des Nenners darf vor dem Ausführen einer Division nicht 0 sein.

Für diese Ausnahme die allgemeine Hilfe aufrufen.

Weitere Hilfeinformationen online suchen...

5

FormatException wurde nicht behandelt.

Die Eingabezeichenfolge hat das falsche Format.

Hinweise zur Fehlerbehebung:

Stellen Sie sicher, dass die Methodenargumente das richtige Format aufweisen.

Bearbeiten Sie beim Konvertieren einer Zeichenfolge in DateTime diese so, dass das Datum vor dem Einfügen aller Variablen in das DateTime-Objekt übernommen wird.

Für diese Ausnahme die allgemeine Hilfe aufrufen.

Weitere Hilfeinformationen online suchen...

Aktionen:

Details anzeigen...

Ausnahmedetail in die Zwischenablage kopieren

Spitzen Sie Ihren Bleistift
Lösung

Sie sollten den Codezeilen die Fehlermeldungen zuordnen, die sie verursachen.

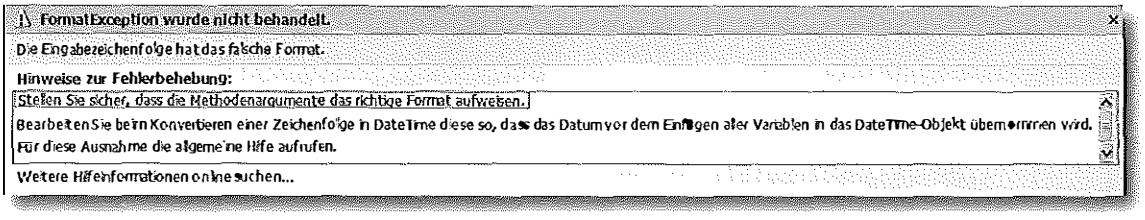
```
object dieBiene = new HonigBiene(36.5, "Willi");
float honigMenge = (float)dieBiene;
```

C# hat keinen Schimmer, wie es ein HonigBiene-Objekt in einen float umwandeln soll. Ein entsprechender Versuch führt deswegen zu einer InvalidCastException.



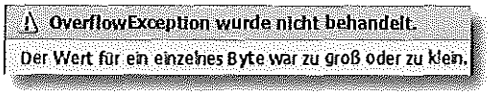

```
HonigBiene andereBiene = new HonigBiene(12.5, "Maja");
double bienenname = double.Parse(andereBiene.Name);
```

Die Methode Parse() erwartet einen String in einem bestimmten Format. Bei dem String »Maja« weiß sie nicht, wie sie ihn umwandeln soll. Deswegen wird eine FormatException ausgelöst.




```
double honigGesamt = 36.5 + 12.5;
string versorgteBienen = "";
for (int i = 1; i < (int) honigGesamt; i++) {
    versorgteBienen += i.ToString();
}
float f = float.Parse(versorgteBienen);
```

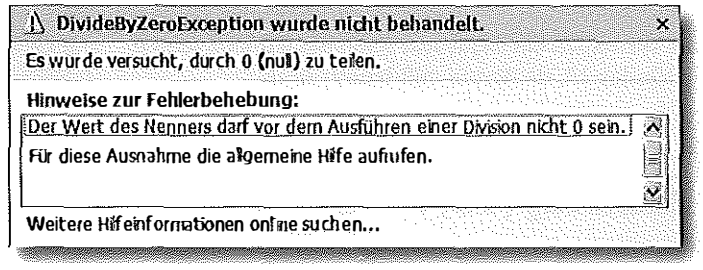
Die for-Schleife erzeugt einen String namens versorgteBienen, der eine Zahl mit mehr als 60 Ziffern enthält. Ein so große Zahl kann ein float unmöglich speichern. Ein solcher Versuch führt zu einer OverflowException.



Diese Exceptions erhalten Sie nie nacheinander. Das Programm löst die erste Exception aus und hält dann an. Die zweite Exception erhalten Sie erst, wenn Sie die erste repariert haben.

```
int drohnen = 4;
int königinnen = 0;
int drohnenProKönigin = drohnen / königinnen;
```

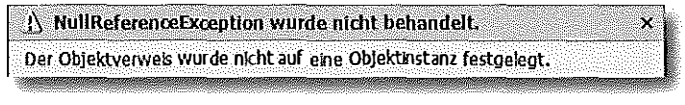
Eine DevideByZeroException kann man sehr leicht auslösen. Teilen Sie einfach eine Zahl durch null.



Wird eine Zahl durch null geteilt, wird immer dieser Typ von Exception ausgelöst. Selbst wenn Sie den Wert von `königinnen` nicht kennen, können Sie den Fehler verhindern, indem Sie vor der Verwendung sicherzustellen, dass der Wert nicht null ist.

```
andereBiene = null;
if (drohnenProKönigin < 10) {
    andereBiene.ArbeitErledigen();
}
```

Wird die Referenz andereBiene auf null gesetzt, sagt das C#, dass diese auf nichts zeigt. Statt auf ein Objekt zeigt sie auf nichts. Indem es eine NullReferenceException auslöst, sagt Ihnen C#, dass es kein Objekt gibt, dessen ArbeitErledigen()-Methode aufgerufen werden kann.



Der DevideByZero-Fehler musste nicht passieren. Das da etwas nicht in Ordnung ist, können Sie schon erkennen, wenn Sie sich den Code ansehen. Das Gleiche gilt für die anderen Exceptions. Diese Fehler ließen sich verhindern – und je mehr Sie über Exceptions wissen, umso leichter verhindern Sie, dass Ihr Code abstürzt.

Löst Ihr Programm eine Exception aus, erzeugt .NET ein Exception-Objekt

Gerade sahen Sie, wie .NET Ihnen sagt, dass in Ihrem Programm etwas schiefgelaufen ist: über **Exceptions**. In C# wird beim Auftreten einer Exception ein Objekt erzeugt, das das Problem repräsentiert. Die entsprechende Klasse heißt `Exception` (und deswegen verwenden wir hier auch den Begriff Exception statt der deutschen Übertragung Ausnahme).

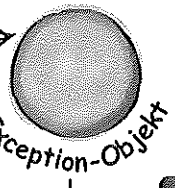
Nehmen Sie beispielsweise an, Sie haben ein Array mit 4 Elementen und versuchen, auf das 16. zuzugreifen (Index 15, da Arrays null-basiert sind).

```
int[] dasArray = {3, 4, 1, 11};
int derWert = dasArray[15];
```

Dieser Code wird offensichtlich zu Problemen führen.

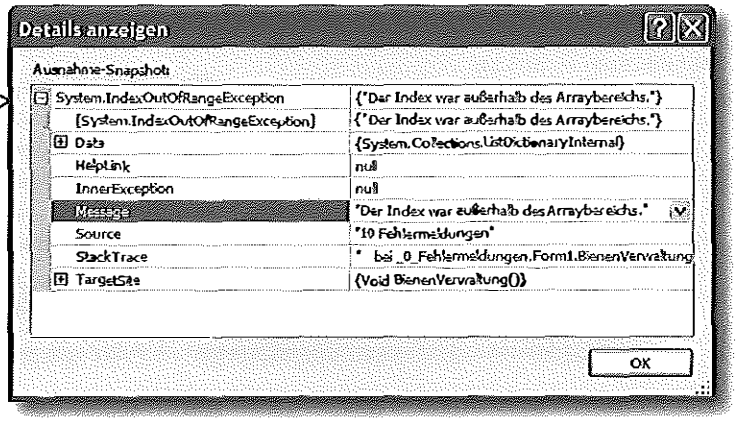
> Aus-nah-me, Nomen
> Eine Person oder eine Sache, die von einer allgemeinen Aussage ausgeschlossen oder einer Regel nicht unterworfen ist. *Normalerweise hasst Tim Obst, aber Erdbeeren sind eine* ***Ausnahme****.*

Sobald Ihr Programm auf eine unbehandelte Exception stößt, generiert es ein Objekt mit allen Daten, die es besitzt.



Diese Informationen können Sie einsehen, indem Sie auf den Link >>Details anzeigen<< im Fenster für die unbehandelte Exception klicken.

Das Exception-Objekt enthält eine Nachricht, die Ihnen sagt, was schiefgelaufen ist, und eine Liste aller Aufrufe an den Systemspeicher, die zu dem Ereignis führten, das die Exception ausgelöst hat.



.NET erzeugt dieses Objekt, weil es Ihnen Informationen dazu liefern will, was die Exception verursacht. Sie müssen eventuell Code reparieren oder ein paar Änderungen daran vornehmen, wie eine bestimmte Situation in Ihrem Programm behandelt wird.

Hier zeigt Ihnen eine **`IndexOutOfRangeException`** an, dass Sie einen Fehler haben: Sie versuchen, in einem Array auf eine Position zuzugreifen, die außerhalb des Bereichs liegt. Sie erhalten auch Informationen, die genau angeben, an welcher Stelle Ihres Codes der Fehler auftritt, die es vereinfachen, das Problem aufzufinden (selbst wenn Sie Tausende Zeilen Code haben).

Es gibt keine Dummen Fragen

F: Warum gibt es so viele verschiedene Arten von Exceptions?

A: Man kann auf unterschiedlichste Weise Code schreiben, bei dem C# einfach nicht weiß, wie es damit umgehen soll. Es wäre sehr schwer, Probleme zu beheben, würde ihnen Ihr Programm nur eine allgemeine Fehlermeldung liefern (»in Zeile 37 trat ein Fehler auf«). Probleme lassen sich viel leichter aufspüren und beheben, wenn man genau weiß, was für eine Art Fehler aufgetreten ist.

F: Was genau *ist* den eine Exception?

A: Es ist ein Objekt (ein Exception-Objekt), das .NET erzeugt, wenn es ein Problem gibt (mehr dazu gleich).

F: Moment? Das ist ein *Objekt*?

A: Ja, eine Exception ist ein **Objekt**. Die Eigenschaften dieses Objekts geben Ihnen Informationen zur Exception. Beispielsweise hat es die Eigenschaft Message, die einen nützlichen String enthält wie »Die angegebene Umwandlung war ungültig« und »Der Wert für ein einzelnes Byte war zu groß oder zu klein«, der verwendet wird, um das Exception-Fenster aufzubauen. .NET erzeugt Exceptions, um Ihnen so viele Informationen wie möglich darüber zu liefern, was genau vor sich ging, als die Anweisung ausgeführt wurde, die die Exception auslöste.

F: Hm. Verstehe ich trotzdem nicht. Tut mir leid. Also, warum gibt es so viele verschiedene Arten von Exceptions?

A: Weil es so viele verschiedene Wege gibt, auf denen sich Ihr Code unerwartet verhalten kann. Es gibt eine Menge Situationen, die dazu führen, dass Ihr Code einfach abstürzt. Es wäre ziemlich schwer, diese Probleme zu beheben, wenn Sie nicht wüssten, warum der Fehler passierte. Indem es den Umständen angemessen unterschiedliche Arten von Exceptions erzeugt, liefert Ihnen .NET eine ganze Menge sehr nützliche Informationen, die Ihnen helfen, das Problem auszumachen und zu beheben.

F: Exceptions sollen mir also helfen und nicht einfach Kopfschmerzen bereiten?

A: Ja! Exceptions sollen Ihnen helfen, das Unerwartete zu erwarten. Viele Menschen sind frustriert, wenn ihr Code eine Exception auslöst. Aber wenn Sie sich daran erinnern, dass Ihnen .NET mit der Exception hilft, Fehler in Ihrem Programm aufzuspüren und zu beheben, sind sie eine große Hilfe, wenn Sie herauszufinden versuchen, was dazu führt, dass Ihr Programm hochgeht.

F: Wenn mein Code eine Exception auslöst, liegt das nicht notwendigerweise daran, dass Ich einen Fehler gemacht habe?

A: Genau. Manchmal sind Ihre Daten nicht so, wie Sie sie erwarten – beispielsweise wenn sich eine Methode mit einem Array herumschlagen muss, das viel kleiner oder größer ist, als Sie ursprünglich erwartet haben. Und vergessen Sie nicht, dass Ihre Programme von Menschen verwendet werden und sich diese gern auf nicht planbare Weise verhalten. Mit Exceptions hilft Ihnen .NET, diese unerwarteten Situationen zu verarbeiten, damit ihr Programm weiterläuft und nicht einfach abstürzt oder eine nutzlose kryptische Fehlermeldung liefert.

F: Nachdem ich wusste, wonach Ich suchen musste, war ziemlich klar, warum der Code auf der vorangehenden Seite zu Fehlern führt. Sind alle Exceptions so leicht zu finden?

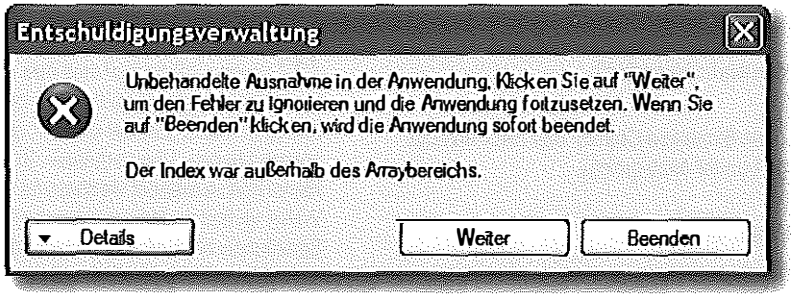
A: Nein. Unglücklicherweise gibt es Gelegenheiten, bei denen Ihr Code Probleme birgt, die sich nur sehr schwer entdecken lassen, indem man den Code einfach nur betrachtet. Deswegen bietet Ihnen die IDE ein sehr nützliches Werkzeug, das als **Debugger** bezeichnet wird. Dieser ermöglicht Ihnen, Ihr Programm anzuhalten, zeilenweise auszuführen und dabei die Werte der einzelnen Variablen und Felder einzusehen. Das macht es Ihnen erheblich leichter herauszufinden, wo sich Ihr Code anders verhält, als Sie es erwartet haben. So haben Sie die beste Chance, das Problem aufzuspüren und zu beheben – oder sogar von Anfang an zu verhindern.

Exceptions helfen Ihnen, Situationen zu finden und zu reparieren, bei denen sich Ihr Code nicht wie erwartet verhält.

Brians Code machte etwas Unerwartetes

Als Brian seine Entschuldigungsverwaltung schrieb, hat er nicht erwartet, dass ein Benutzer versuchen könnte, eine zufällige Entschuldigung aus einem leeren Verzeichnis auszuwählen.

1. Das Problem trat auf, als Brian seine Entschuldigungsverwaltung auf einen leeren Ordner auf seinem Laptop zeigen ließ und dann auf den Button Schnelle Ausrede klickte. Schauen wir uns das an und probieren wir herauszufinden, was schiefgelaufen ist. Hier ist die Fehlermeldung, die erschien, als er das Programm außerhalb der IDE laufen ließ:



2. Gut, das ist ein Ansatzpunkt. Die Meldung sagt uns, dass der Index außerhalb der Array-Grenzen lag. Suchen wir also im Code für den Event-Handler des Buttons Schnelle Ausrede nach einem Array:

```
private void schnelleAusredeButton_Click(object sender, EventArgs e) {
    if (GeändertPrüfen() == true) {
        dieEntschuldigung = new Entschuldigung(zufall, ordner);
        FormularAktualisieren(false);
    }
}
```

3. Hm ... hier gibt es keine Arrays. Aber es wird mit einem der überladenen Konstruktoren ein neues `Entschuldigung`-Objekt erzeugt. Vielleicht gibt es im Konstruktorcode ein Array:

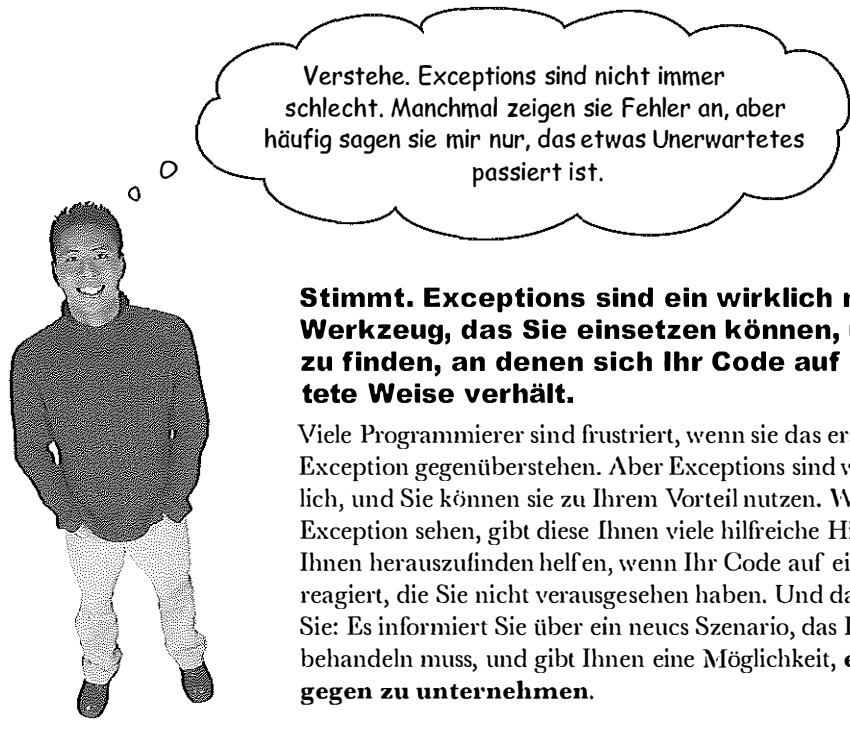
```
public Entschuldigung(Random zufall, string ordner) {
    string[] dateinamen = Directory.GetFiles(ordner, "*.excuse");
    DateiÖffnen(dateinamen[zufall.Next(dateinamen.Length)]);
}
```

Bingo! Da ist das Array. Wir haben es wohl mit einem Index versucht, der hinter dem Ende des Arrays liegt.

4 Es stellt sich heraus, dass `Directory.GetFiles()` ein leeres Array liefert, wenn Sie ein Verzeichnis angeben, das keine Dateien enthält. Aber das können wir ja testen! Wir müssen also nur eine Prüfung einbauen, **die sicherstellt, dass das Verzeichnis nicht leer ist**, bevor wir eine Datei öffnen. Dann wird das hässliche Fenster, dass die unbehandelte Exception meldet, durch einen informativen Dialog ersetzt.

```
private void schnelleAusredeButton_Click(object sender, EventArgs e) {
    string[] dateinamen = Directory.GetFiles(gewählterOrdner, "*.entsch");
    if (dateinamen.Length == 0) {
        MessageBox.Show("Geben Sie einen Ordner mit entschuldigung-Dateien an",
                        "Keine Entschuldigungen gefunden");
    } else {
        if (GeändertPrüfen() == true) {
            dieEntschuldigung = new Entschuldigung(zufall, gewählterOrdner);
            FormularAktualisieren(false);
        }
    }
}
```

Indem wir prüfen, ob der Ordner Entschuldigungen enthält, bevor wir das Entschuldigung-Objekt erstellen, können wir verhindern, dass die Exception ausgelöst wird – und gleich auch eine hilfreiche Meldung anzeigen.



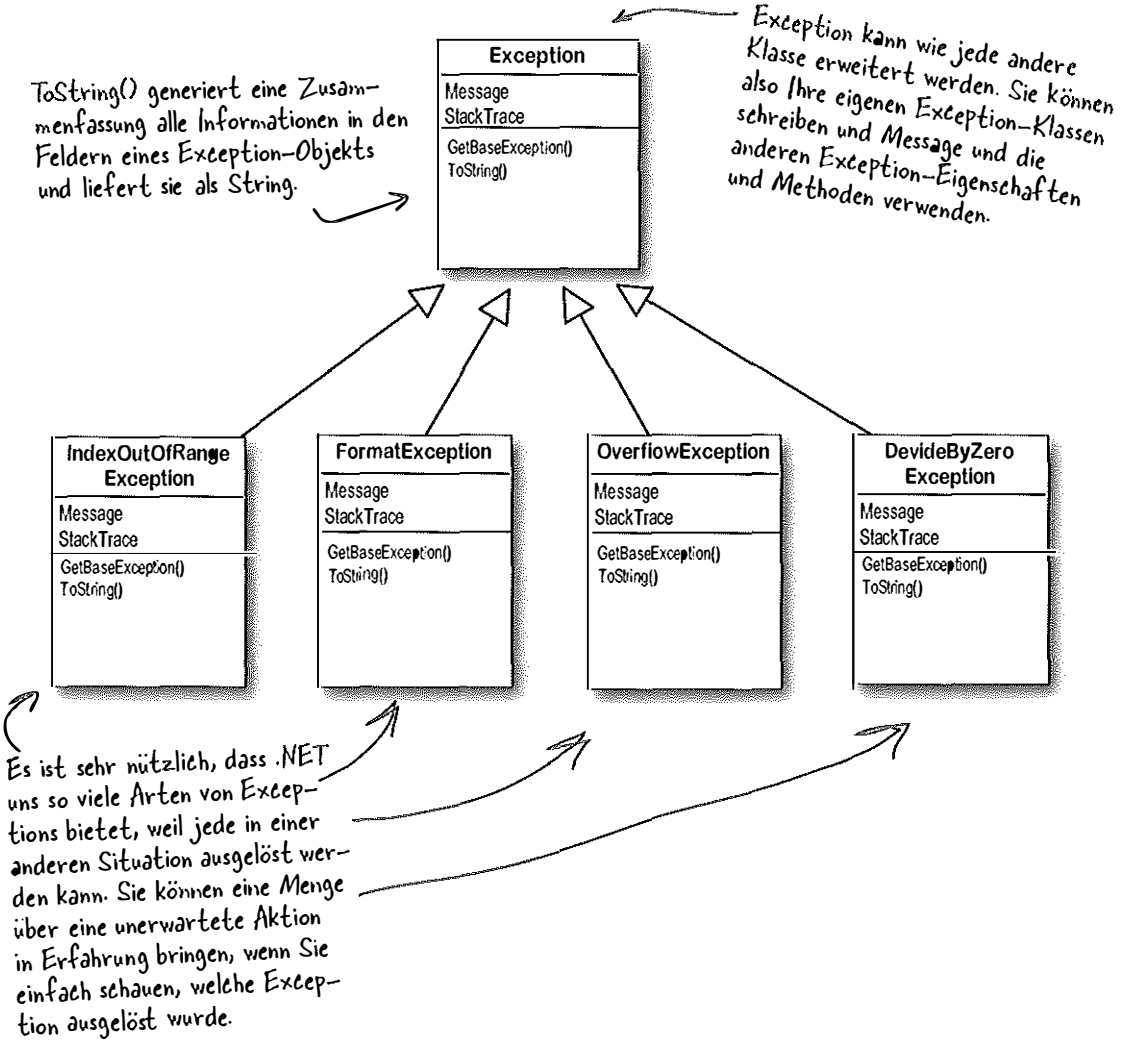
Stimmt. Exceptions sind ein wirklich nützliches Werkzeug, das Sie einsetzen können, um Punkte zu finden, an denen sich Ihr Code auf unerwartete Weise verhält.

Viele Programmierer sind frustriert, wenn sie das erste mal einer Exception gegenüberstehen. Aber Exceptions sind wirklich nützlich, und Sie können sie zu Ihrem Vorteil nutzen. Wenn Sie eine Exception sehen, gibt diese Ihnen viele hilfreiche Hinweise, die Ihnen herauszufinden helfen, wenn Ihr Code auf eine Situation reagiert, die Sie nicht verausgesehen haben. Und das ist gut für Sie: Es informiert Sie über ein neucs Szenario, das Ihr Programm behandeln muss, und gibt Ihnen eine Möglichkeit, **etwas dagegen zu unternehmen**.

Alle Exception-Objekte erben von `Exception`

.NET kennt viele verschiedene Exceptions, die es gegebenenfalls melden können muss. Da viele davon diverse gemeinsame Kennzeichen haben, wird dabei Vererbung genutzt. .NET definiert eine Basisklasse namens Exception, von der alle spezifischen Exception-Typen erben.

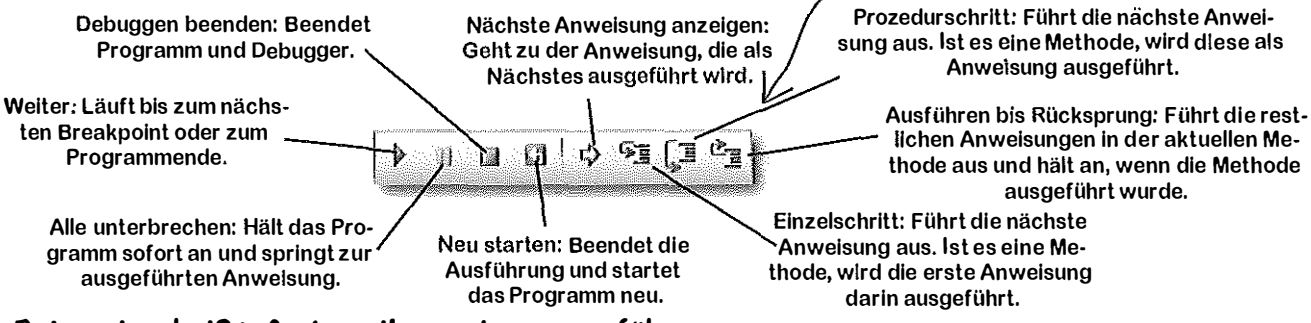
Die Klasse Exception bietet einige nützliche Member. Die Eigenschaft `Message` speichert eine lesbare Meldung über das Problem, und `StackTrace` sagt Ihnen, was im Speicher passierte, als die Exception auftrat, und was ihr voranging. (Es gibt noch weitere, aber das sind die, die wir zunächst verwenden werden.)



Der Debugger hilft Ihnen, Exceptions in Ihrem Code aufzuspüren und zu verhindern

Die Debug-Werkzeugleiste wird nur angezeigt, wenn Sie Ihr Programm in der IDE debuggen. Sie müssen also ein Programm laufen haben, damit Sie diese Werkzeugleistensymbole sehen.

Bevor Sie Ihrem Programm ein Exception-Handling hinzufügen können, müssen Sie wissen, welche Anweisungen in Ihrem Programm die Exception auslösen. Dabei kann der in die IDE eingebaute **Debugger** sehr hilfreich sein. Führen Sie den Debugger aus, blendet die IDE eine Werkzeugleiste mit einigen praktischen Buttons ein. Nehmen Sie sich eine Minute und lassen Sie Ihre Maus über allen schweben, um sich ihre Namen und eine Kurzwahltaste anzeigen zu lassen. (Die Kurzwahltasten werden sich als ziemlich praktisch erweisen.)

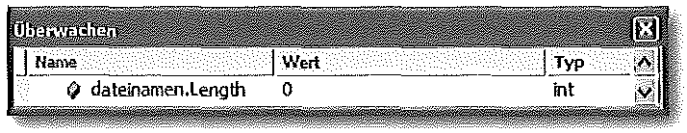


Debugging heißt, Code zeilenweise auszuführen, damit man sieht, was passiert

Führen Sie Ihr Programm in der IDE aus, können Sie es jederzeit anhalten, indem Sie in der Werkzeugleiste auf den Button **Alle unterbrechen** klicken (oder den entsprechenden Befehl im Menü Debuggen wählen). Das bewirkt, dass Ihr Programm **anhält** und Ihnen die Codezeile zeigt, die es jetzt ausführen will. Sie hebt diese Zeile gelb hervor, um Ihnen zu zeigen, dass das die ist, die als Nächstes ausgeführt wird. Klicken Sie auf **Weiter**, wird Ihr Programm fortgesetzt, als hätten Sie es nie angehalten. Aber Sie können Ihren Code auch durchschreiten. Das bedeutet, dass Sie eine Zeile ausführen und dann zur nächsten gehen. Ist die nächste Zeile eine Methode, können Sie **in die Methode hineinspringen**. Das bewirkt, dass der Debugger zur ersten Zeile der Methode springt und sie markiert. Oder Sie können die **Methode überspringen.** Dann wird die Methode als Einheit übersprungen. Befinden Sie sich in einer Methode, können Sie aus ihr **herausspringen**. Das veranlasst den Debugger, die restlichen Anweisungen in der Methode auszuführen und auf der ersten Zeile nach dem Rücksprung aus der Methode anzuhalten.

Unterbrechen Sie im Debugger, hält die IDE Ihr Programm an und hebt die Codezeile gelb hervor, die als Nächstes ausgeführt wird. Sie können dann zeilenweise weitergehen, bis Sie Ihr Problem finden.

Im **Überwachen-Fenster** können Sie außerdem die Werte der Variablen und Felder in Ihrem Code einsehen und ändern. Klicken Sie einfach mit der rechten Maustaste im Code auf eine Variable und wählen Sie »Überwachung hinzufügen«. Sie erscheint dann im Fenster Überwachen. Sie können sie auch direkt in das Fenster Überwachen eingeben. Dort wird dann ihr Wert angezeigt. Ist dieser ein Objekt, können Sie zu seinen einzelnen Feldern hinabsteigen.



Nutzen Sie den Debugger, um herauszufinden, was genau das Problem in der Entschuldigungsverwaltung ist

Nutzen wir den Debugger, um uns das Problem genauer anzusehen, auf das wir in der Entschuldigungsverwaltung gestoßen sind. Das ist eine gute Möglichkeit, etwas Erfahrung mit dem Debugger zu sammeln, weil Sie bereits wissen, wonach Sie suchen. (Das ist ein Luxus, der Ihnen normalerweise nicht vergönnt ist!)



1. **Unterbrechen Sie Ihr Programm mit einem Haltepunkt.**
Sie haben einen Ansatzpunkt – die Exception tritt auf, wenn auf den Button Schnelle Ausrede geklickt wird, nachdem ein leerer Ordner gewählt wurde. Öffnen Sie also den Code für den Button, klicken Sie irgendwo in die erste Zeile der Methode und wählen Sie im Menü Debuggen »Haltepunkt umschalten« (oder drücken Sie F9):

Der Debugger zeigt Ihre Haltepunkte als rote Zeilen mit einem großen roten Punkt am linken Rand.

```
private void schnelleAusrede_Click(object sender, EventArgs e)
{
    string[] dateinamen = Directory.GetFiles(gewählterOrdner, "*.entsch");
    if (dateinamen.Length == 0)
    {                          dateinamen.Length 0
        MessageBox.Show("Geben Sie einen Ordner mit entschuldigung-Dateien an",
                        "Keine Entschuldigungen gefunden");
    }
```

Lassen Sie die Maus über einem Feld oder einer Variablen schweben, zeigt Ihnen die IDE ihren Wert.

Die gelbe Zeile mit dem gelben Pfeil am linken Rand ist die Zeile, die der Debugger als Nächstes ausführen wird.

Die IDE hinterlegt die Zeile rot und setzt einen roten Punkt an den linken Rand. Auf diese Weise sagt Ihnen der Debugger, dass auf dieser Zeile ein **Haltepunkt** gesetzt wurde. Debuggen Sie Ihr Programm in der IDE, wird die Ausführung jetzt auf dieser Zeile angehalten. Probieren Sie es aus – führen Sie das Programm in der IDE aus (indem Sie den »Debuggen starten«-Befehl verwenden, den Sie schon die ganze Zeit nutzen). Reproduzieren Sie das Problem, sollte die Zeile gelb werden und ein gelber Pfeil darauf zeigen. Jetzt wurde das Programm vorübergehend angehalten. Der Menüeintrag »Debuggen starten« hat sich in »Debuggen fortsetzen« verwandelt. Klicken Sie darauf – das Programm sollte an genau der Stelle wieder aufgenommen werden, an der es anhielt: auf der Zeile mit dem Haltepunkt.

Jetzt wissen Sie, warum der IDE-Befehl zur Ausführung eines Programms »Debuggen starten« ist – weil Sie den Debugger nutzen, der in die IDE eingebaut ist.

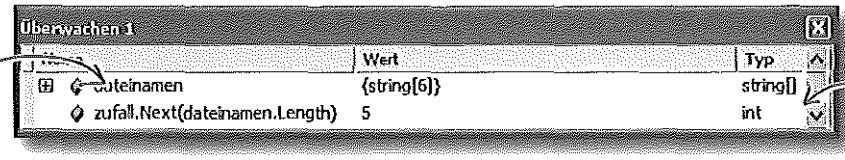
2. **Durchlaufen Sie die Anwendung schrittweise.**
Nutzen Sie den Befehl **Einzelschritt** (indem Sie die Werkzeugleiste oder die F11-Taste verwenden), um die Anwendung zeilenweise zu durchlaufen. Kommen Sie zu der Zeile, die das `Entschuldigung`-Objekt erzeugt, springt der Debugger direkt in den Konstruktor, den Sie repariert haben. Gehen Sie über die erste Zeile, damit die Variable `dateinamen` gesetzt wird. Lassen Sie dann die Maus über der Variablen schweben, um ihren Wert zu setzen.

Sind keine Haltepunkte gesetzt, ist »Debuggen« starten genau das Gleiche wie eine gewöhnliche Ausführung des Programms.

3 Reproduzieren Sie das Problem über das Fenster Überwachen.

Das Fenster **Überwachen** ist eine besonders mächtige Debugger-Komponente, in der Sie die Werte Ihrer Variablen und die Felder Ihrer Objekte überprüfen können. Markieren Sie im Editor `dateinamen.Length`, klicken Sie mit rechts darauf und wählen Sie im Menü »Überwachung hinzufügen«. Der Ausdruck wird dann dem Überwachen-Fenster hinzugefügt (das über »Überwachen« unter Fenster im Menü Debuggen aufgerufen werden kann). Fügen Sie dann alle Teile der Anweisung hinzu: zuerst `zufall.Next(dateinamen.Length)`. Es sollte dann ungefähr so aussehen wie hier, wobei die Zahl davon abhängig ist, wie viele Dateien in dem Verzeichnis sind, das Sie gewählt haben – da das bei uns fünf waren, hat `dateinamen` fünf Elemente:

Im Überwachen-Fenster werden wir das Problem reproduzieren. Zunächst fügen wir das Array dateinamen hinzu.

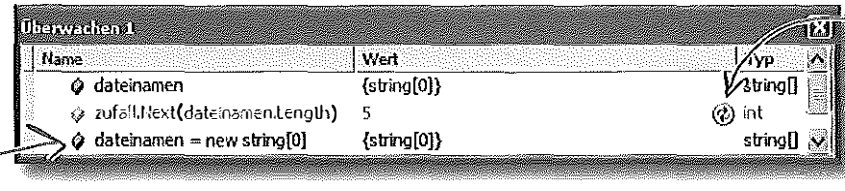


Vor der Überwachung müssen Sie das Programm unterbrechen.

4 Setzen Sie dateinamen auf ein leeres String-Array.

Klicken Sie unter den beiden Variablen doppelt auf den leeren Bereich im Überwachen-Fenster, erhalten Sie einen Cursor. Geben Sie Folgendes ein: **`dateinamen = new string[0]`**. Drücken Sie Enter, ändert sich in der obersten Zeile der Wert von `dateinamen` in `{string[0]}`. Neben der `zufall.Next`-Zeile erscheint das Neuberechnen-Symbol ⟲ – klicken Sie darauf, wird der Wert auf 0 gesetzt. Was ist da passiert? Das Überwachen-Fenster bietet eine weitere nützliche Funktion – Sie können darin **den Wert** der angezeigten Variablen und Felder **ändern**. Und Sie können darin sogar **Methoden ausführen und neue Objekte erzeugen.** Tun Sie das, wird das Neuberechnen-Symbol angezeigt (⟲), damit Sie den Debugger explizit anweisen können, die Zeile erneut auszuführen, da es manchmal zu anderen Ergebnissen kommen kann, wenn eine Methode mehrfach ausgeführt wird (wie bei Random).

Wir wissen, dass das Problem in einem leeren dateinamen-Array liegt, und nutzen das Überwachen-Fenster deswegen, um es auf ein leeres String-Array zu setzen.

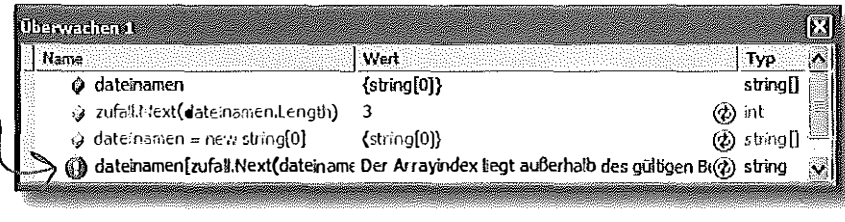


Dieses Symbol sagt, dass der Ausdruck neu ausgewertet werden soll.

5 Reproduzieren Sie das ursprüngliche Problem.

Das ist der Punkt, an dem das Debuggen richtig interessant wird. Fügen Sie dem Debugger eine weitere Zeile hinzu – die Anweisung, die für die Exception verantwortlich war: `dateinamen[zufall.Next(dateinamen.Length)]`. Sobald Sie diese eingeben, wird sie im Überwachen-Fenster ausgewertet ... und wieder wird die Exception ausgelöst. Dass die Exception auftrat, zeigt Ihnen das Fenster, indem es ein Ausrufezeichen und in der Spalte Wert den Text der Exception einblendet.

Mit dem Ausrufezeichen signalisiert Ihnen das Überwachen-Fenster, dass eine Exception auftrat.



Exceptions können Sie im Debugger reproduzieren. Auch auf diese Weise helfen Ihnen aussagekräftige Fehlermeldungen, Ihren Code zu reparieren.

Es gibt keine Dummen Fragen

F: Warum sah Brians unbehandelte Exception anders aus als die in der IDE?

A: Weil ein Programm, das in der IDE läuft, im Debugger ausgeführt wird, der das **Programm anhält** (als hätten Sie auf den Button »Alle unterbrechen« geklickt oder einen Haltepunkt eingefügt), sobald er eine Exception abfängt und diese in einem hilfreichen Fenster anzeigt. Darin können Sie das Exception-Objekt sowie die Felder und Variablen Ihres Programm untersuchen, um den Fehler aufzuspüren.

Brian führte sein Programm nicht in der IDE aus. Er hatte das Programm veröffentlicht und installiert, wie Sie es in Kapitel 1 mit dem Adresslistenprogramm gemacht haben. Sie können Ihr Programm jederzeit außerhalb der IDE laufen lassen, ohne es zu veröffentlichen. Wenn Sie es erstellen, erzeugt Visual Studio eine ausführbare Programmdatei. Gehen Sie einfach in den `bin/`-Ordner Ihres Projektverzeichnisses – in einem seiner Unterverzeichnisse sollten Sie die `exe`-Datei für Ihre Anwendung finden. Führen Sie diese aus, bleiben alle ausgelösten Exceptions unbehandelt und werden in genau so einem Fenster wie bei Brian angezeigt.

F: Ist das alles? Wenn außerhalb der IDE eine Exception auftritt, hält mein Programm einfach an, ohne dass ich dagegen etwas tun könnte?

A: Ja, ihr Programm hält an, wenn es eine ***unbehandelte*** Exception gibt. Aber das heißt nicht, dass alle Ihre Exceptions unbehandelt bleiben müssen! Wir werden ihnen noch eine Menge darüber erzählen, wie Sie in Ihrem Code Exceptions behandeln können. ihre Benutzer müssen niemals mit einer unbehandeiten Exception konfrontiert werden.

F: Wie weiß ich, an welche Steile ich einen Haltepunkt setzen muss?

A: Das ist eine wirklich gute Frage, auf die es leider keine eindeutige Antwort gibt. Löst Code eine Exception aus, beginnt man am besten immer bei der Anweisung, die sie auslöste. Aber in der Regel liegt das eigentliche Problem, dessen Folge die Exception ist, früher im Programm. Beispielsweise könnte eine Anweisung, die einen Division-durch-Null-Fehler auslöst, Werte teilen, die zehn Zeilen zuvor erzeugt und bis dahin nicht verwendet wurden. Es gibt also keine eindeutige Antwort darauf, wo man Haltepunkte einfügen muss, weil alle Probleme anders sind. Aber wenn Sie eine klare Vorstellung davon haben, wie Ihr Code funktioniert, sollten Sie dazu in der Lage sein, einen brauchbaren Anfangspunkt zu finden.

F: Kann ich im Überwachen-Fenster jede x-beliebige Methode ausführen?

A: Ja. Jede Anweisung, die in Ihrem Programm gültig ist, funktioniert im Überwachen-Fenster, selbst Dinge, bei denen eine Ausführung im Überwachen-Fenster überhaupt keinen Sinn macht. Ein Beispiel. Starten Sie ein Programm, halten Sie es an und fügen Sie dann folgende Zeile ins Überwachen-Fenster ein: `System.Threading.Thread.Sleep(2000)`. (Erinnern Sie sich, dass diese Methode bewirkt, dass Ihr Programm zwei Sekunden wartet.) Im wirklichen Leben werden Sie wohl nie Grund haben, das zu tun, aber die Wirkung ist recht interessant: Sie erhalten für zwei Sekunden eine Sanduhr, während die Methode ausgewertet wird. Dann zeigt das Überwachen-Fenster, da `Sleep()` keinen Rückgabewert hat, den Wert »Der Ausdruck wurde ausgewertet und weist keinen Wert auf« an, um Ihnen mitzuteilen, dass nichts zurückgeliefert wurde. Aber der Ausdruck wurde ausgewertet. Und Sie können nicht nur Methoden ausführen, sondern erhalten bei der Eingabe im Fenster auch IntelliSense-Unterstützung. Das ist nützlich, weil Sie erfahren, welche Methoden für ein Objekt verfügbar sind, während Ihr Programm ausgeführt wird.

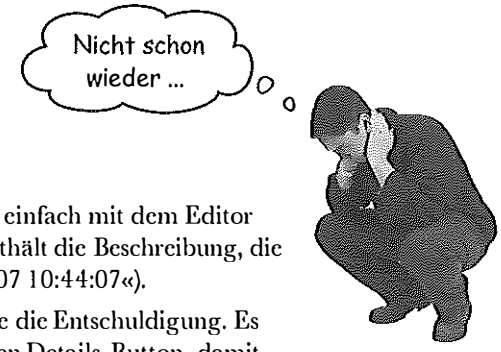
F: Moment! Kann ich im Überwachen-Fenster dann nicht Dinge ausführen, die beeinflussen, wie mein Programm läuft?

A: Sicher! Nicht dauerhaft, aber es kann definitiv die Ausgabe Ihres Programms beeinflussen. Und es reicht sogar aus, im Debugger ***die Maus*** über Feldern schweben zu lassen, um das Verhalten eines Programms zu ändern, weil das Schweben über einer Eigenschaft Ihren ***Getter ausführt***. Haben Sie eine Eigenschaft mit einem Getter, der eine Methode ausführt, führt Schweben über dieser Eigenschaft dazu, dass diese Methode ausgeführt wird. Setzt diese Methode einen Wert in Ihrem Programm, bleibt dieser Wert gesetzt, wenn Sie das Programm weiterlaufen lassen. Und das kann im Debugger zu ziemlich unvorhersehbaren und zufälligen Ergebnissen führen: Man nennt sie **Helsenbugs** (das ist ein Witz, den nur Physiker und Katzen in Kisten verstehen).

Führen Sie Ihr Programm in der IDE aus, führt eine unbehandelte Exception dazu, dass es angehalten wird wie an einem Haltepunkt.

Och nee, der Code hat immer noch Probleme

Brian war mit seiner Entschuldigungsverwaltung zufrieden, nachdem er sich erinnerte, dass er einen Ordner mit einem Haufen Enschuldigungen hatte, die er mit der ersten Programmversion erstellt hatte – aber er hatte vergesssen, dass er diesen Ordner erstellt hatte, ***bevor*** er das Programm mit Serialisierung ausgestattet hatte.



1. Sie können Brians Problem rekonstruieren – erstellen Sie einfach mit dem Editor eine textbasierte Entschuldigungsdatei. Die erste Zeile enthält die Beschreibung, die zweite die Reaktion und die dritte das Datum (»21.12.2007 10:44:07«).

2. Starten Sie die Entschuldigungsverwaltung und öffnen Sie die Entschuldigung. Es wird eine Exception ausgelöst! Klicken Sie diesmal auf den Details-Button, damit wir uns genauer ansehen können, was diese sagt. Achten Sie auf den **Aufrufstapel** – so heißt es, wenn eine Methode von einer anderen aufgerufen wird, die von einer anderen Methode aufgerufen wurde usw.



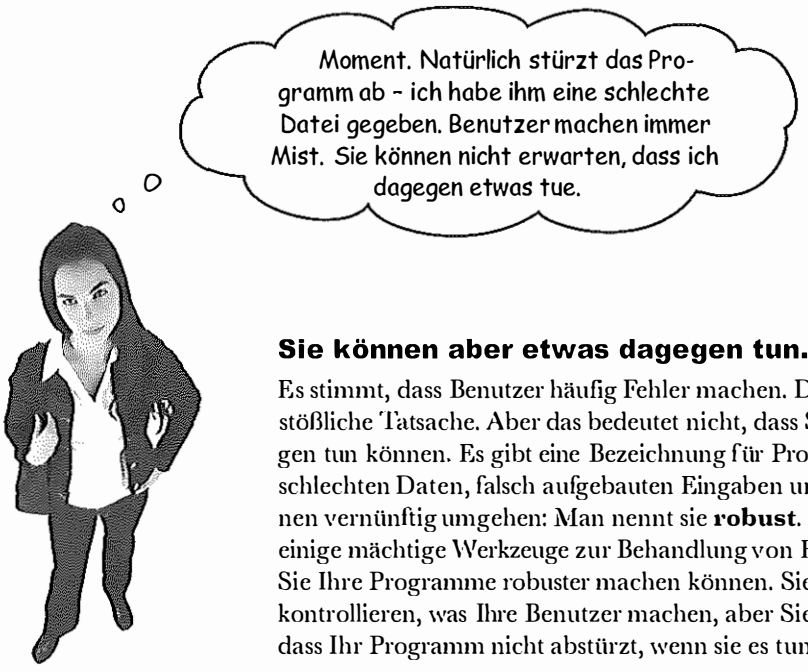
Das Programm meldete eine `SerializationException`. Finden wir die verantwortliche Zeile über den Aufrufstapel?

********** Ausnahmetext **********
System.Runtime.Serialization.**SerializationException**: Der Eingabestream hat kein gültiges binäres Format. Die Startinhalte (in Bytes) sind: 4D-65-69-6E-20-48-75-6E-64-20-68-61-74-20-4B-6F-70 ...
bei System.Runtime.Serialization.Formatters.Binary.SerializationHeaderRecord.Read(__BinaryParser input)
bei System.Runtime.Serialization.Formatters.Binary.__BinaryParser.ReadSerializationHeaderRecord()
bei System.Runtime.Serialization.Formatters.Binary.__BinaryParser.Run()
bei System.Runtime.Serialization.Formatters.Binary.ObjectReader.Deserialize(HeaderHandler handler, __BinaryParser serParser, Boolean fCheck, Boolean isCrossAppDomain, IMethodCallMessage methodCallMessage)
bei System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream serializationStream, HeaderHandler handler, Boolean fCheck, Boolean isCrossAppDomain, IMethodCallMessage methodCallMessage)
bei System.Runtime.Serialization.Formatters.Binary.**BinaryFormatter.Deserialize**(Stream serializationStream)
bei __Entschuldigungsverwaltung_mit_Serialisierung.**Entschuldigung.DateiÖffnen**(String dateipfad) in C:\Dokumente und Einstellungen\Administrator\Eigene Dateien\HeadFirstCsharpCode\9 Excuse Manager - with serialization\9 Excuse Manager - with serialization\Entschuldigung.cs:Zeile 31.
bei __Entschuldigungsverwaltung_mit_Serialisierung.**Entschuldigung..ctor**(String dateipfad) in C:\Dokumente und Einstellungen\Administrator\Eigene Dateien\HeadFirstCsharpCode\9 Excuse Manager - with serialization\9 Excuse Manager - with serialization\Entschuldigung.cs:Zeile 20.
bei __Entschuldigungsverwaltung_mit_Serialisierung.**Form1.schnelleAusredeButton_Click**(Object sender, EventArgs e) in C:\Dokumente und Einstellungen\Administrator\Eigene Dateien\HeadFirstCsharpCode\9 Excuse Manager - with serialization\9 Excuse Manager - with serialization\

Scheinbar gibt es ein Problem mit dem BinaryFormatter – das klingt vernünftig, da er versuchte, eine Textdatei zu deserialisieren.

Vom Aufrufstapel können Sie eine Menge lernen. Er sagt Ihnen, welche Methoden ausgeführt wurden. Sie sehen, dass der Konstruktor (»`.ctor`«) die `DateiÖffnen()`-Methode von Entschuldigung aufrief und er selbst vom »Schnelle Ausrede«-Event-Handler aufgerufen wurde.

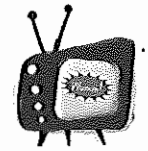
3. Der Details-Button im Exception-Fenster sagt Ihnen also eine Menge zur Problemursache. ***Haben Sie eine Idee, was Sie dagegen tun könnten?***



Sie können aber etwas dagegen tun.

Es stimmt, dass Benutzer häufig Fehler machen. Das ist eine unumstößliche Tatsache. Aber das bedeutet nicht, dass Sie nichts dagegen tun können. Es gibt eine Bezeichnung für Programme, die mit schlechten Daten, falsch aufgebauten Eingaben und anderen Situationen vernünftig umgehen: Man nennt sie **robust**. Und C# gibt Ihnen einige mächtige Werkzeuge zur Behandlung von Exceptions, damit Sie Ihre Programme robuster machen können. Sie *können* zwar *nicht* kontrollieren, was Ihre Benutzer machen, aber Sie *können* sicherstellen, dass Ihr Programm nicht abstürzt, wenn sie es tun.

ro-bust, adj.
Stabil gebaut. Dazu in der Lage, widrigen Umständen zu widerstehen oder sie zu überwinden. *Nach dem Autobahnbrückendesaster suchte das Ingenieursteam nach einem* ***robusteren*** *Entwurf für die neue Brücke.*



BinaryFormatter löst eine Exception aus, wenn mit der serialisierten Datei etwas nicht in Ordnung ist.

Die Entschuldigungsverwaltung bringt man leicht dazu, eine SerializationException auszulösen – geben Sie ihr einfach eine Datei, die kein serialisiertes Entschuldigung-Objekt enthält. Versuchen Sie, ein Objekt aus einer Datei zu deserialisieren, erwartet BinaryFormatter, dass die Datei ein serialisiertes Objekt enthält, das der Klasse entspricht, die zu lesen versucht wird. Enthält die Datei irgendetwas anderes, löst die Methode Deserialize() eine SerializationException aus.

Exceptions mit try und catch behandeln

In C# verwenden Sie `try` und `catch`, um Folgendes zu sagen: »**Versuche** diesen Code und **fange** ihn, falls eine Exception auftritt, mit diesem *anderen* Code **ab**.« Der Codeteil, den Sie versuchen, ist der **Try-Block**, und der Teil, in dem Sie sich um Exceptions kümmern, ist der **Catch-Block**. Im Catch-Block können Sie eine freundliche Fehlermeldung ausgeben, statt das Programm krachend zu Fall kommen zu lassen:

```
private void schnelleAusredeButton_Click(object sender,
EventArgs e) {
  // ... Hier steht der von Ihnen neu eingefügte Code ...
  try {
      if (GeändertPrüfen() == true) {
          dieEntschuldigung =
           new Entschuldigung(zufall, gewählterOrdner);
          FormularAktualisieren(false);
      }
  }
  catch {
      MessageBox.Show(
        "Ihre Datei war ungültig.",
        "Konnte Entschuldigung nicht öffnen");
  }
}
```

Das ist der Try-Block. Das Exception-Handling beginnen Sie mit `try`.

Das Schlüsselwort `catch` bedeutet, dass im unmittelbar folgenden Code ein Exception-Handler steckt.

Stecken Sie den Code, der eine Exception verursachen könnte, in den Try-Block. Tritt keine Exception ein, wird er ganz normal ausgeführt, und die Anweisungen im Catch-Block werden ignoriert. Aber wenn eine Anweisung im Try-Block eine Exception auslöst, wird der Rest des Try-Blocks nicht ausgeführt.

Wird eine Exception ausgelöst, springt das Programm sofort zur catch-Anweisung und beginnt, den Catch-Block auszuführen

Das ist die einfachste Art des Exception-Handlings: das Programm anhalten, eine Meldung ausgeben und dann das Programm weiterlaufen lassen.



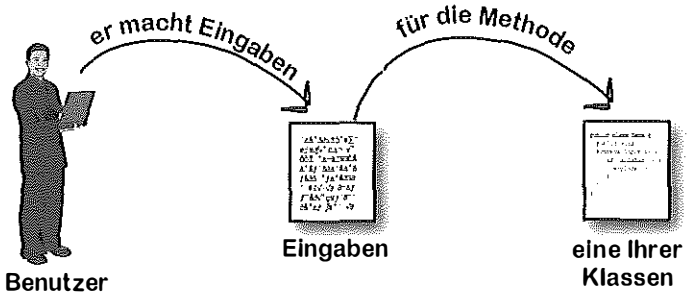
KOPF-NUSS

Wenn Exceptions dazu führen, dass Ihr Code automatisch zum Catch-Block springt, was passiert dann mit den Objekten und Daten, mit denen Sie vor dem Auftreten der Exception gearbeitet haben?

Was passiert, wenn eine Methode gefährlich ist?

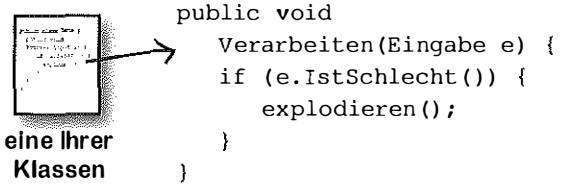
Benutzerverhalten kann man nicht planen. Benutzer füttern Ihr Programm mit den seltsamsten Daten und klicken auf vollkommen unerwartete Weise auf Dinge. Aber eigentlich ist das kein Problem, weil Sie unerwartete Eingaben mit einem guten Exception-Handling verarbeiten können.

1 Nehmen wir an, ein Benutzer nutzt Ihren Code und macht Eingaben, die Sie nicht erwartet haben.



2 Diese Methode macht etwas, das riskant ist, etwas, das zur Laufzeit fehlschlagen könnte.

>>Laufzeit<< heißt: während Ihr Programm läuft. Manche Leute bezeichnen Exceptions als Laufzeitfehler.

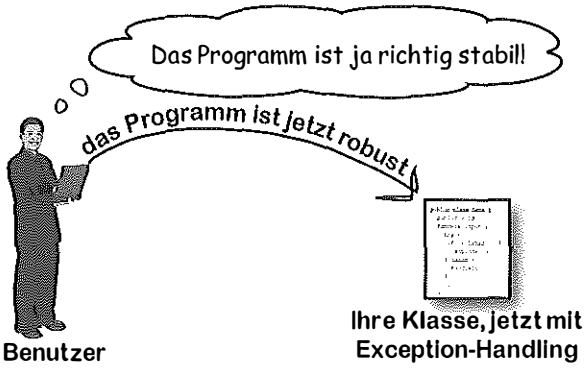



```
public void
    Verarbeiten(Eingabe e) {
    if (e.IstSchlecht()) {
        explodieren();
    }
}
```

3 Sie müssen *wissen*, dass die aufgerufene Methode riskant ist.



4 Dann können Sie Code schreiben, der sich um das Problem kümmert, so*llte* es autreten. Auf diesen Fall müssen Sie vorbereitet sein.



Es gibt keine Dummen Fragen

F: Wann verwende Ich try und catch?

A: Immer wenn Sie riskanten Code schreiben oder Code, der eine Exception auslösen könnte. Das Schwierige ist herauszufinden, welcher Code riskant und welcher sicher ist.

Sie haben bereits gesehen, dass Code, der Benutzereingaben verwendet, riskant sein kann. Benutzer geben Ihnen falsche Dateien, Wörter statt Zahlen oder Namen statt Datumswerten und klicken an die unmöglichsten Stellen. Und ein gutes Programm nimmt all diese Eingaben und arbeitet auf ruhige und vorhersagbare Weise. Vielleicht liefert es dem Benutzer kein Ergebnis, mit dem er etwas anfangen kann, aber es teilt ihm mit, dass es auf ein Problem gestoßen ist, und schlägt eventuell sogar eine Lösung vor.

F: Wie kann ein Programm eine Lösung zu einem Problem vorschlagen, von dem es im Voraus gar nichts wissen kann?

A: Dazu ist der Catch-Block da. Ein Catch-Block wird nur ausgeführt, wenn der Code im Try-Block eine Exception auslöst. Das ist Ihre Chance sicherzustellen, dass der Benutzer erfährt, wenn etwas schiefgelaufen ist, und ihm mitzuteilen, dass das eine Situation ist, die sich eventuell beheben lässt.

Es ist nicht sehr hilfreich, wenn die Entschuldigungsverwaltung bei falschen Eingaben einfach abstürzt. Aber es ist auch keine Hilfe, wenn sie versucht, die Eingabe zu lesen und dann im Formular Müll anzeigt – manche Leute würden sogar sagen, dass das noch schlimmer ist. Aber wenn das Programm eine Fehlermeldung anzeigt, die dem Benutzer sagt, dass die Datei nicht gelesen werden konnte, erhält der Benutzer ein Bild davon, was schiefgelaufen ist, und Informationen, anhand deren er das Problem lösen kann.

F: Wird der Debugger nur eingesetzt, um Exceptions zu reparieren?

A: Nein. Der Debugger ist ein sehr nützliches Werkzeug, das Sie einsetzen können, um den gesamten Code zu untersuchen, den Sie geschrieben haben. Manchmal ist es hilfreich, Code schrittweise durchzugehen und die Werte bestimmter Felder und Variablen zu überprüfen – beispielsweise wenn Sie eine ziemlich komplizierte Methode haben und sichergehen wollen, dass diese ordentlich funktioniert.

Aber wie der Name »Debugger« (Entwanzer) schon andeutet, wird er am häufigsten eingesetzt, um Fehler aufzuspüren und zu beheben. Manchmal sind diese Fehler Exceptions, die ausgelöst werden. Aber häufig werden Sie den Debugger nutzen, um andere Arten von Problemen zu beheben – beispielsweise Code, der ein unerwartetes Ergebnis liefert.

F: Ich glaube, dass mit dem Überwachen-Fenster habe ich noch nicht verstanden. Wozu war das noch mal?

A: Wenn Sie ein Programm debuggen, möchten Sie sich in der Regel genau ansehen, wie sich einige Felder und Variablen ändern. Dazu ist das Überwachen-Fenster nützlich. Fügen Sie Überwachungen für ein paar Variablen hinzu, aktualisiert das Überwachen-Fenster deren Werte bei jedem Schritt in, aus oder über Ihren Code. So können Sie genau überwachen, was mit ihnen nach den einzelnen Anweisungen passiert. Das kann sehr hilfreich sein, wenn Sie versuchen, ein Problem aufzuspüren.

Im Überwachen-Fenster können Sie außerdem beliebige Anweisungen eingeben und auswerten. Aktualisiert die Anweisung eins der Felder oder eine der Variablen in Ihrem Programm, wird auch das ausgeführt. So können Sie die Werte ändern, während Ihr Programm läuft. Auch das kann ein sehr hilfreiches Mittel sein, wenn Sie versuchen, Exceptions oder andere Fehler zu reproduzieren.

Alle Änderungen im Überwachen-Fenster haben lediglich Einfluss auf die Daten im Speicher und bestehen nur, solange das Programm läuft. Starten Sie Ihr Programm neu, werden alle Änderungen rückgängig gemacht.

Der Catch-Block wird nur ausgeführt, wenn der Try-Block eine Exception auslöst. Er gibt Ihnen eine Möglichkeit, dem Benutzer Informationen zu geben, anhand deren er das Problem beheben kann.

Folgen Sie try/catch mit dem Debugger

Ein wichtiger Teil des Exception-Handlings ist es, dass der restliche Code eines Try-Blocks **übersprungen wird**, wenn eine Anweisungdarin eine Exception auslöst. Die Ausführung des Programms springt unmittelbar zur ersten Zeile des Catch-Blocks. ***Aber das müssen Sie uns nicht einfach glauben ...***



1. Stellen Sie sicher, dass Sie den gesamten Code aus diesem Kapitel in den »Schnelle Ausrede«-Button der Entschuldigungsverwaltung integriert haben. Setzen Sie einen Haltepunkt auf die erste Zeile seines Click-Event-Handlers. Führen Sie das Programm dann in der IDE aus. Klicken Sie auf den Ordner-Button und geben Sie einen Ordner an, der eine einzige Datei enthält – und sorgen Sie dafür, dass das **keine gültige Entschuldigung-Datei ist** (das führt dazu, dass eine Exception ausgelöst wird). Klicken Sie auf den Button »Schnelle Ausrede«. Der Debugger sollte das Programm am Haltepunkt anhalten. Klicken Sie sechsmal auf den Button »Prozedurschritt« (oder auf F10), damit Sie zu der Zeile kommen, die den Entschuldigung-Konstruktor aufruft. So sollte Ihr Debugger-Fenster aussehen:

Setzen Sie einen Haltepunkt auf der ersten Zeile im Event-Handler.

Gehen Sie über die Anweisungen, bis die gelbe »Nächste Zeile«-Markierung zeigt, dass mit der nächsten Anweisung das Entschuldigung-Objekt erzeugt wird.

```
private void schnelleAusrede_Click(object sender, EventArgs e) {
    string[] dateinamen = Directory.GetFiles(gewählterOrdner, "*.entsch");
    if (dateinamen.Length == 0){
        MessageBox.Show("Geben Sie einen Ordner mit entschuldigung-Dateien an",
                        "Keine Entschuldigungen gefunden");
    } else {
        try {
            if (GeändertPrüfen() == true)
                dieEntschuldigung = new Entschuldigung(zufall, gewählterOrdner);
                FormularAktualisieren(false);
            }
        }
        catch {
                MessageBox.Show(
                  "Ihre Datei war ungültig.",
                  "Konnte Entschuldigung nicht öffnen");
        }
    }
}
```

Achten Sie darauf, dass Sie im Debugger den Button Prozedurschritt (F10) verwenden, damit er nicht in die Methode GeändertPrüfen() springt.

2. Nutzen Sie Einzelschritt (F11), um in die new-Anweisung zu springen. Der Debugger springt in den Entschuldigung-Konstruktor und platziert die gelbe »Nächste Zeile«-Markierung über der Deklarationszeile in den Code. Drücken Sie Einzelschritt (F11), um in die Methode `DateiÖffnen()` zu springen. Beobachten Sie, was passiert, wenn Sie die Datei deserialisieren.

Wenn Sie in die new-Anweisung springen, die das Entschuldigung-Objekt erzeugt, springt der Debugger zum Konstruktorcode.

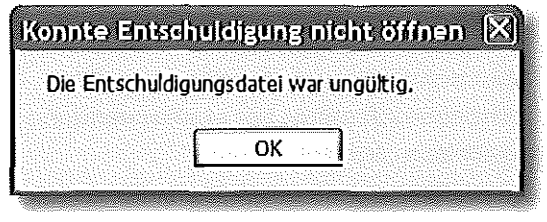
```
public Entschuldigung(Random zufall, string ordner) {
    string[] dateinamen = Directory.GetFiles(ordner, "*.entsch");
    DateiÖffnen(dateinamen[zufall.Next(dateinamen.Length)]);
}
```

3 Sobald der Debugger die `Deserialize()`-Anweisung ausführt, wird die Exception ausgelöst. Das Programm springt dann unmittelbar zur ersten Zeile im Catch-Block. Es **übergeht** den Aufruf von `FormularAktualisieren()` und **springt direkt in den `catch`-Block**.

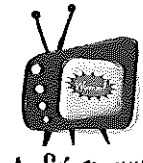
Der Debugger hebt die `catch`-Anweisung mit der gelben »Nächste Zeile«-Markierung hervor und zeigt den restlichen Block in Grau an, um Ihnen zu zeigen, dass er dieses ganze Ding jetzt ausführen wird.

```
private void schnelleAusrede_Click(object sender, EventArgs e) {
    string[] dateinamen = Directory.GetFiles(gewählterOrdner, "*.entsch");
    if (dateinamen.Length == 0) {
        MessageBox.Show("Geben Sie einen Ordner mit entschuldigung-Dateien an",
                        "Keine Entschuldigungen gefunden");
    }
    else {
        try {
            if (GeändertPrüfen() == true) {
                dieEntschuldigung = new Entschuldigung(zufall, gewählterOrdner);
                FormularAktualisieren(false);
            }
        } catch {
            MessageBox.Show(
                "Die Datei war ungültig.",
                "Konnte keine zufällige Entschuldigung öffnen");
        }
    }
}
```

4 Starten Sie das Programm erneut, indem Sie auf den Weiter-Button (oder F5) drücken. Dieser lässt das Programm von der Zeile ab weiterlaufen, die mit der gelben »Nächste Zeile«-Markierung hervorgehoben wird – das ist in diesem Fall der Catch-Block.



Hier ist ein Karrieretipp: In vielen Vorstellungsgesprächen für C#-Programmierer taucht die Frage auf, wie Sie es mit Exceptions in Konstruktoren halten.



Exceptions gehören nicht in Konstruktoren!

Mittlerweile dürfte Ihnen aufgefallen sein, dass ein Konstruktor keinen Rückgabewert hat, nicht einmal `void`. Das liegt daran, dass ein Konstruktor eben nichts zurückliefert. Sein einziger Zweck ist die Initialisierung eines Objekts – und das wirft für das Exception-Handling in Konstruktoren Probleme auf. Wird in einem Konstruktor eine Exception ausgelöst, erhält die Anweisung, die versuchte, die Klasse zu Instantiieren, ***keine Instanz****. Deswegen mussten Sie den try/catch-Block in den Event-Handler für den Button verschieben. Auf diese Weise verhindern wir, dass der Code erwartet, dass dieEntschuldigung ein gültiges Entschuldigung-Objekt enthält, wenn im Konstruktor eine Exception auftrat.*

Nutzen Sie finally, wenn Sie Code haben, der IMMER ausgeführt werden soll

Löst Ihr Programm eine Exception aus, können verschiedene Dinge passieren. Wird die Exception ***nicht*** behandelt, bricht Ihr Programm die Verarbeitung ab und stürzt ab. ***Wird*** die Exception behandelt, springt Ihr Code in den `catch`-Block. Aber was ist mit dem restlichen Code in Ihrem Try-Block? Was ist, wenn der einen Stream schließen oder andere wichtige Ressourcen aufräumen sollte? Dieser Code muss ausgeführt werden, selbst wenn eine Exception auftritt, da der Zustand Ihres Programms ansonsten chaotisch wird. Dann ist ein **`finally`**-Block sehr praktisch. Er kommt nach den `try`- und `catch`-Blöcken. Der **`finally`-Block läuft immer**, egal ob eine Exception ausgelöst wird oder nicht. So würden Sie also das Event-Handling für den Button »Schnelle Ausrede« vervollständigen.

```
private void SchnelleAusredeButton_Click(object sender, EventArgs e) {
    string[] dateinamen = Directory.GetFiles(ordner, "*.excuse");
    if (dateinamen.Length == 0) {
        MessageBox.Show("Geben Sie einen Ordner mit entschuldigung-Dateien an",
                        "Keine Entschuldigungen gefunden");
    } else {
        try {
            if (GeändertPrüfen() == true) {
                dieEntschuldigung = new Entschuldigung(zufall, ordner);
            }
        }
        catch (Exception) {
            dieEntschuldigung = new Entschuldigung();
            dieEntschuldigung.Beschreibung = "";
            dieEntschuldigung.Reaktion = "";
            dieEntschuldigung.LetzteVerwendung = DateTime.Now;
            MessageBox.Show(
              "Die Datei war ungültig.",
              "Konnte keine zufällige Entschuldigung öffnen");
        }
        finally {
            FormularAktualisieren(false);
        }
    }
}
```

Löst der Konstruktor eine Exception aus, können wir nicht in Erfahrung bringen, was in dieEntschuldigung ist. Aber wir wissen, dass keine Entschuldigung-Instanz erzeugt wurde. Der `catch`-Block erzeugt also ein Entschuldigung-Objekt und leert dessen Felder.

Der `finally`-Block sichert, dass `FormularAktualisieren()` ausgeführt wird, egal ob eine Exception ausgelöst wurde oder nicht. `FormularAktualisieren()` wird aufgerufen, wenn der Konstruktor eine Entschuldigung liefert, aber auch, wenn er eine Exception auslöst und die Entschuldigung leert.

Haben Sie bemerkt, dass auf `catch` die Klammer `(Exception)` folgt? Einer `catch`-Anweisung kann eine bestimmte Art von Exception folgen, die sagt, was für Exceptions der Block abfangen soll. Geben Sie **`(Exception)`** an oder lassen Sie die Klammer weg, werden alle Exceptions abgefangen. Möchten Sie nur eine `SerializationException` abfangen, können Sie in der Klammer stattdessen diese Klasse angeben. Oder Sie könnten eine `IOException` verwenden, die **jede Art von Problem mit der Dateieingabe/-ausgabe abfängt**.

Debuggen Sie jetzt das

1. Aktualisieren Sie den Event-Handler für den Button »Schnelle Ausrede« mit dem Code auf der letzten Seite. Setzen Sie dann einen Haltepunkt auf die erste Zeile der Methode und debuggen Sie das Programm.

2. Führen Sie das Programm normal aus und prüfen Sie, ob der Button »Schnelle Ausrede« funktioniert, wenn Sie den Ordner für das Programm auf ein Verzeichnis mit normalen Entschuldigung-Dateien setzen. Der Debugger sollte am Haltepunkt anhalten:

Befinden sich die gelbe Markierung und der Haltepunkt auf der gleichen Zeile, zeigt die IDE Ihnen einen gelben Pfeil auf einem großen roten Punkt.

```
private void schnelleAusrede_Click(object sender, EventArgs e) {
    string[] dateinamen = Directory.GetFiles(gewählterOrdner, "*.entsch");
    if (dateinamen.Length == 0) {
        MessageBox.Show("Geben Sie einen Ordner mit entschuldigung-Dateien an",
                        "Keine Entschuldigungen gefunden");
    }
    else {
        try {
            if (GeändertPrüfen() == true) {
                dieEntschuldigung = new Entschuldigung(zufall, gewählterOrdner);
                FormularAktualisieren(false);
            }
        }
        catch (Exception) {
            dieEntschuldigung = new Entschuldigung();
            dieEntschuldigung.Beschreibung = "";
            dieEntschuldigung.Reaktion = "";
            dieEntschuldigung.LetzteVerwendung = DateTime.Now;
            MessageBox.Show(
              "Die Datei war ungültig.",
              "Konnte keine zufällige Entschuldigung öffnen");
        }
        finally {
            FormularAktualisieren(false);
        }

    }
}
```

3. Gehen Sie mit Einzelschritten durch den restlichen Code des Event-Handlers und prüfen Sie, ob alles wie erwartet funktioniert. Der `try`-Block sollte bis zum Ende durchlaufen und der `catch`-Block übersprungen werden (weil keine Exceptions ausgelöst wurden). Abschließend sollte der `finally`-Block ausgeführt werden.

4. Setzen Sie den Ordner für das Programm jetzt so, dass er auf ein Verzeichnis mit falschen Entschuldigung-Dateien zeigt, und klicken Sie dann auf den Button Schnelle Ausrede. Zunächst sollte der `try`-Block ausgeführt und dann in den `catch`-Block gesprungen werden, wenn die Exception ausgelöst wird. Sind alle Anweisungen des `catch`-Blocks ausgeführt worden, sollte der `finally`-Block ausgeführt werden.

Es gibt keine Dummen Fragen

F: Moment. Jedes Mal, wenn mein Programm auf eine Exception stößt, setzt es mit der Arbeit aus, wenn ich keinen speziellen Code zur Verarbeitung der Exception schreibe. Wie kann das gut sein?

A: Mit das Beste an Exceptions ist, dass sie offenlegen, dass es Probleme gibt. Denken Sie nur daran, wie leicht es in einer komplexen Anwendung passieren kann, dass Sie den Überblick über all die Objekte verlieren, mit denen ihr Programm arbeitet. Exceptions machen Sie auf die Probleme aufmerksam und helfen Ihnen, ihre Ursachen zu beheben, damit Sie immer wissen, dass Ihr Programm macht, was es machen soll.

Jedes Mal, wenn in Ihrem Programm eine Exception auftritt, ist etwas nicht passiert, was Sie erwartet hätten. Vielleicht zeigte eine Referenz nicht auf das erwartete Objekt oder konnte ein Benutzer einen Wert angeben, den Sie nicht berücksichtigt hatten. Möglicherweise war eine Datei nicht mehr verfügbar, mit der Sie zu arbeiten meinten. Passiert so etwas, ohne dass Sie davon Kenntnis haben, würde Ihr Programm wahrscheinlich seltsame Ausgaben liefern und sich sein gesamtes Verhalten von diesem Punkt an vollkommen von dem unterscheiden, das Sie erwarteten, als Sie es schrieben.

Stellen Sie sich jetzt vor, Sie hätten keine Ahnung, dass der Fehler aufgetreten ist, und Ihre Benutzer wenden sich mit falschen Daten an Sie und sagen Ihnen, dass Ihr Programm instabil sei. Deswegen ist es *gut*, dass Exceptions alles unterbrechen, was das Programm macht. Sie zwingen Sie dazu, sich um das Problem zu kümmern, während es leicht zu finden und zu reparieren ist.

F: Gut ... und was ist dann eine behandelte und was eine unbehandelte Exception?

A: Löst ihr Programm eine Exception aus, sucht die Laufzeitumgebung in Ihrem Code nach einem passenden Catch-Block. Haben Sie einen geschrieben, werden die von ihnen für diese bestimmte Exception vorgegebenen Anweisungen ausgeführt. Da es einen Catch-Block für diesen Fehler gibt, wird diese Exception als eine behandelte Exception betrachtet. Findet die Laufzeit keinen passenden Catch-Block für die Exception, hält sie die Ausführung Ihres Programms an und meldet einen Fehler. Dann bezeichnet man die Exception als unbehandelt.

F: Und wie war das mit der Angabe einer bestimmten Art von Exception? Warum sollte ich das tun?

A: ***Normalerweise*** wollen Sie keine beliebige Exception abfangen. Exceptions an sich abzufangen sollten Sie tunlichst meiden und stattdessen spezifische Typen von Exceptions abfangen. Nehmen wir beispielsweise an, Sie wollten vermeiden, dass die Ktasse Entschuldigung eine `FileNotFoundException` an das Formular weiterreicht – und Sie wollten das so machen, dass beim Versuch, eine Datei zu öffnen, die nicht gefunden werden kann, automatisch eine Entschuld'igung mit diesem Dateinamen erzeugt und mit Default-Werten gefüllt wird. Dann könnten Sie einen `Try`-Block hinzufügen, auf den `catch (FileNotFoundException)` folgt. Wird eine Datei nicht gefunden, kann die Klasse mit diesem Fehler dann umgehen – aber eine IOException oder SerializationException würde nicht abgefangen, sondern erst vom Exception-Handling im Formular verarbeitet. Aber wenn Sie das tun möchten, müssen Sie sicherstellen, dass sich im Aufrufstapel *irgendeine* Methode befindet, die ein alles einfangendes Exception-Handling bietet, andernfalls könnten Exceptions unbehandelt bleiben. Und das würde wieder dazu führen, dass der Benutzer dem hässlichen Absturz wegen einer unbehandelten Exception begegnet.

F: Was passiert, wenn ich ein catch habe, das keine bestimmte Exception angibt?

A: Ein derartiger Catch-Block fängt alle Exceptions aus dem Try-Block ein.

F: Warum sollte ich eine spezifische Exception angeben, wenn ein Catch-Block, bei dem keine bestimmte Exception angegeben ist, alles abfängt?

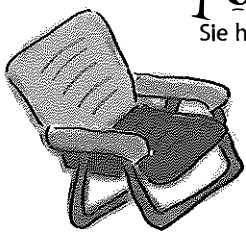
A: Gute Frage. Weil bestimmte Exceptions unterschiedliche Aktionen erfordern könnten, damit Ihr Programm weiterlaufen kann. Eine Exception, die auftritt, wenn Sie durch null teiten, könnte einen Catch-Block haben, in dem Sie einige Zahlenwerte setzen, um ein paar der Daten zu speichern, mit denen Sie arbeiten. Eine NuiiReference-Exception könnte verlangen, dass Sie eine neue Instanz erstellen, damit das Programm weiterlaufen kann.

F: Erfolgt die gesamte Fehlerbehandlung in try/catch/finally-Folgen?

A: Nein. Die Elemente sind variabel. Sie können **mehrere Catch-Blöcke haben**, wenn Sie mit unterschiedlichen Arten von Fehlern klarkommen müssen. Sie können den Catch-Block auch vollständig weglassen. Auch eine try/finally-Folge ist zulässig. Das würde keine Exceptions abfangen, würde aber dafür sorgen, dass der Code im Finally-Block auch dann ausgeführt wird, wenn Sie die Arbeit irgendwo im Try-Block abbrechen. Aber dazu werden wir gleich noch mehr erzählen ...

Unbehandelte Exceptions bedeuten, dass Ihr Programm unvorhersehbar läuft, deswegen wird es dann angehalten.

Pool-Puzzle



Sie haben die ***Aufgabe***, die leeren Zeilen im Code mit den Codeschnipseln aus dem Pool zu füllen. Einzelne Schnipsel **können** mehrfach verwendet werden, und Sie werden nicht alle Schnipsel benötigen. Das ***Ziel*** ist es, eine Klasse zu erstellen, die sich kompilieren lässt und die gezeigte Ausgabe erzeugt.

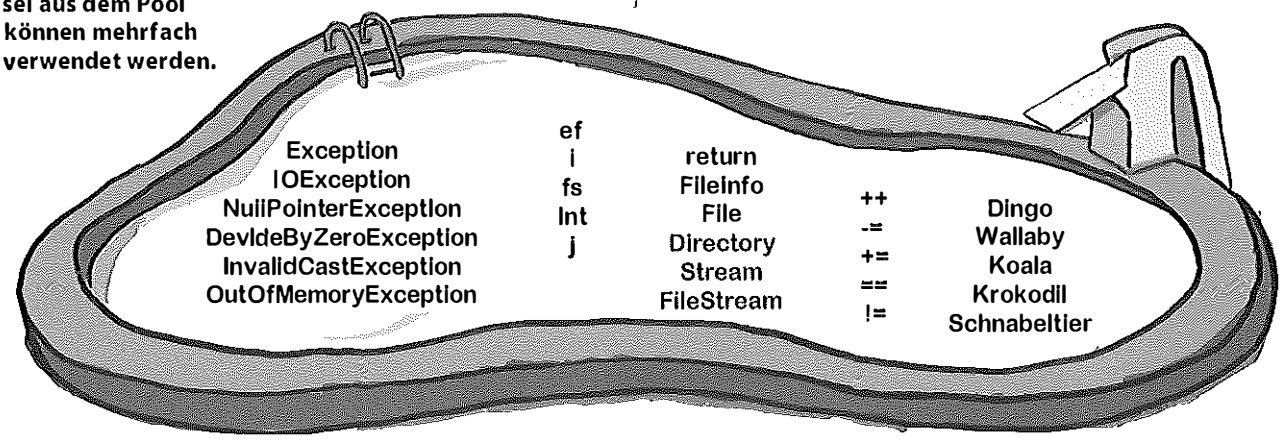
Ausgabe: → `Hallo Kumpel!`

```
using System.IO;
public static void Main() {
    Känguruh king = new Känguruh();
    int Koala = king.Wombat(
           king.Wombat(king.Wombat(1)));
    try {
        Console.WriteLine((15 / Koala)
                  + " Eier pro Kilo");
    }
    catch (_____________________) {
        Console.WriteLine("Hallo Kumpel!");
    }
}
```

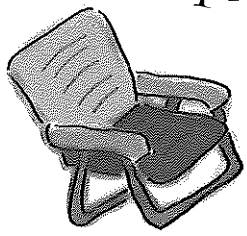
```
public class Känguruh {
    _____________ fs;
    int Krokodil;
    int Dingo = 0;

    public int Wombat(int Wallaby) {
        ________ __;
        try {
            if (_________ > 0) {
                __ = ______.OpenWrite("wobbiegong");
                Krokodil = 0;
            } else if (_________ < 0) {
                Krokodil = 3;
            } else {
                ___ = ______.OpenRead("wobbiegong");
                Krokodil = 1;
            }
        }
        catch (IOException) {
            Krokodil = -3;
        }
        catch {
            Krokodil = 4;
        }
        finally {
            if (_______ > 2) {
                Krokodil ___ Dingo;
            }
            _________ _______;
        }
    }
}
```

Hinweis: Alle Schnipsel aus dem Pool können mehrfach verwendet werden.



Pool-Puzzle, Lösung



```
public static void Main() {
    Känguruh king = new Känguruh();
    int Koala = king.Wombat(king.Wombat(king.Wombat(1)));
    try {
        Console.WriteLine((15 / Koala) + " Eier pro Kilo");
    }
    catch (DevideByZeroException) {
        Console.WriteLine("Hallo Kumpel!");
    }
}

public class Känguruh {
    FileStream fs;
    int Krokodil;
    int Dingo = 0;

    public int Wombat(int Wallaby) {
        Dingo ++;
        try {
            if (Wallaby > 0) {
                fs = File.OpenWrite("wobbiegong");
                Krokodil = 0;
            } else if (Wallaby < 0) {
                Krokodil = 3;
            } else {
                fs = File.OpenRead("wobbiegong");
                Krokodil = 1;
            }
        }
        catch (IOException) {
            Krokodil = -3;
        }
        catch {
            Krokodil = 4;
        }
        finally {
            if (Dingo > 2) {
                Krokodil -= Dingo;
            }
        }
        return Krokodil;
    }
}
```

king.Wombat() wird dreimal aufgerufen und liefert beim dritten Mal null. Das führt dazu, dass WriteLine() eine DevideByZeroException auslöst.

Dieser catch-Block fängt nur Exceptions ab, wenn der Code durch null teilt.

Dass das ein FileStream ist, kann man daran erkennen, dass das Objekt eine OpenRead()-Methode hat und eine IOException auslöst.

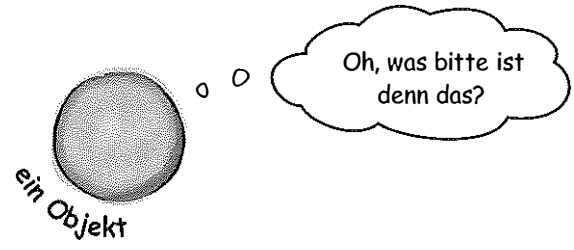
Dieser Code öffnet eine Datei namens >>wobbiegong<< und hält diese nach dem ersten Aufruf offen. Später versucht er erneut, die Datei zu öffnen. Da sie nie geschlossen wurde, wird dann eine IOException ausgelöst.

Sie wissen bereits, dass Dateien immer geschlossen werden müssen, wenn Sie sie nicht mehr benötigen. Tun Sie das nicht, bleibt die Datei gesperrt. Versuchen Sie, sie erneut zu öffnen, erhalten Sie eine IOException.

Nutzen Sie das Exception-Objekt, um Informationen zum Problem zu erhalten

Wir haben bereits gesagt, dass .NET ein `Exception`-Objekt generiert, wenn eine Exception ausgelöst wird. Schreiben Sie einen `catch`-Block, haben Sie Zugriff auf dieses Objekt. Das funktioniert so:

1. Ein Objekt schlendert herum, macht sein Ding, und begegnet plötzlich dem Unerwarteten und ... löst eine Exception aus.

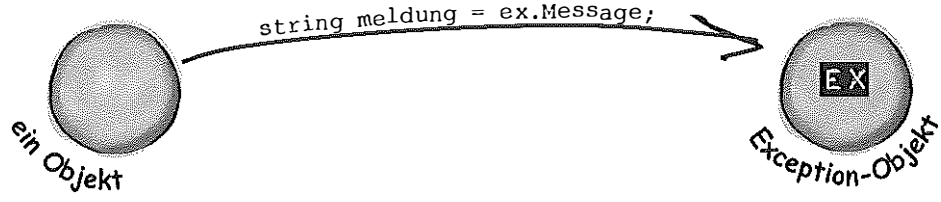


2. Glücklicherweise fängt sein `try/catch`-Block die Exception ab. Im `catch`-Block gab er der Exception-Referenz den Namen **`ex`**.

```
try {
     WasRiskantesTun();
}
catch (Exception ex) {
     string meldung = ex.Message;
     MessageBox.Show(meldung, "Ein Fehler trat auf");
}
```

Wenn Sie im `catch`-Block den Exception-Typ angeben, können Sie einen Variablennamen verwenden, über den Ihr Code das Exception-Objekt dann nutzen kann.

3. Das Exception-Objekt besteht, bis der `catch`-Block fertig ist. Dann verschwindet die Referenz **`ex`**, und das Objekt wird Futter für die Müllabfuhr.



Mehrere Typen von Exceptions mit mehrere Catch-Blöcken abfangen

Sie wissen, dass Sie spezifische Typen von Exceptions abfangen können . . . aber was ist, wenn Sie Code schreiben, in dem mehrere Arten von Problemen auftauchen können? In diesen Fällen möchten Sie vielleicht separaten Code für die unterschiedlichen Arten von Exceptions schreiben. Dann wird die Verwendung von mehreren Catch-Blöcken interessant. Hier ist ein Beispiel mit Code aus der Nektarverarbeitung eines Bienenstocks. Sie können sehen, wie er mehrere Arten von Exceptions abfängt. In einigen Fällen nutzt er Eigenschaften des Exception-Objekts. Die Eigenschaft **Message** wird häufig verwendet. In der Regel enthält sie eine Beschreibung der Exception, die ausgelöst wurde.

Auch über die ToString()-Methode des Exception-Objekts können Sie sich viele nützliche Informationen beschaffen.

```
public void NektarVerarbeiten(NektarKrug krug, Biene arbeiter, StockLog log) {
    try {
        NektarEinheit[] einheiten = arbeiter.KrugLeeren(krug);
        for (int anzahl = 0; anzahl < arbeiter.ErwarteteEinheiten, anzahl++) {
            stream stockLogDatei = log.OpenLogFile();
            arbeiter.LogEintrag(stockLogDatei);
        }
    }
    catch (IndexOutOfRangeException) {
        krug.Geleert = true;
    }
    catch (IOException ex) {
        arbeiter.KöniginBenachrichtigen("Log-Datei beschädigt: " + ex.Message);
    }
    catch (Exception ex) {
        arbeiter.KöniginBenachrichtigen("Unbekannter Fehler: "
            + "Meldung: " + ex.Message + "\r\n"
            + "Aufrufstapel: " + ex.StackTrace + "\r\n"
            + "Daten: " + ex.Data + "\r\n");
    }
    finally {
        krug.Versiegeln();
        arbeiter.ArbeitBeendet();
    }
}
```

Wird das Exception-Objekt nicht verwendet, muss es nicht deklariert werden.

Wenn Sie mehrere catch-Blöcke haben, werden diese nacheinander betrachtet. Hier wird erst geprüft, ob ein Index außerhalb des Bereichs lag. Falls nicht, wird auf eine IOException geprüft. Der letzte catch-Block ist ein alles einfangender Catch-Block, der bei allen Exceptions ausgeführt wird, die noch nicht abgefangen wurden.

Dieser catch-Block weist die Exception der Variablen **ex** zu, über die er dann Informationen abfrufen kann.

Zwei Blöcke können den gleichen Namen (>>ex<<) für die Exception verwenden.

Diese Anweisung nutzt drei Eigenschaften des Exception-Objekts: Message, die die Meldung enthält, die z.B. auch im Exception-Fenster der IDE angezeigt wird (>>Es wurde versucht, durch 0 zu teilen<<), StackTrace, die Ihnen eine Zusammenfassung des Aufrufstapels liefert; und Data, die gelegentlich relevante Daten für die Exception enthält.

Eine Klasse löst eine Exception aus, eine andere fängt sie ab

Natürlich kann auch eine Methode eine Exception auslösen und sie selbst abfangen.

Beim Aufbau einer Klasse wissen Sie nicht immer, wie sie verwendet wird. Manchmal verwenden andere Menschen Ihre Objekte auf eine Weise, die zu Problemen führt – und manchmal machen Sie es auch selbst! Dann werden Exceptions interessant.

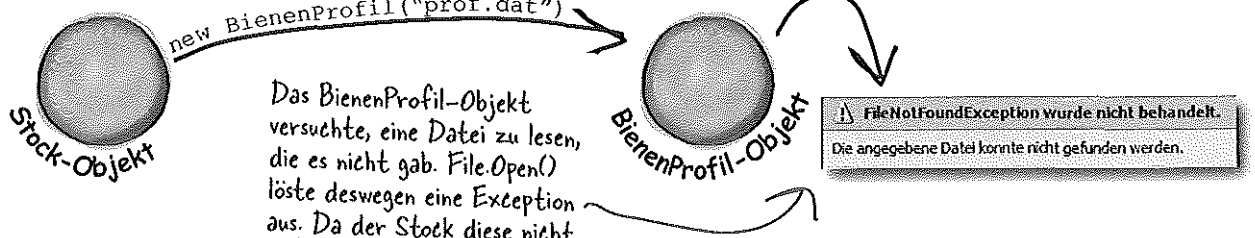
Das Wichtigste beim Auslösen von Exceptions ist, dass Sie sehen, was schieflaufen könnte, damit Sie eine Art Notfallplan in Kraft setzen können. Normalerweise löst eine Methode keine Exception aus, die sie selbst einfängt. In der Regel wird eine Exception in einer Methode ausgelöst und in einer ganz anderen abgefangen – meist sogar in einem anderen Objekt.

Anstelle von diesem ...

Ohne gutes Exception-Handling kann eine Exception das ganze Programm anhalten. So würde das in einem Programm funktionieren, das Bienenprofile für eine Bienenkönigin verwaltet.

Der Konstruktor dieses BienenProfil-Objekts erwartet den Dateinamen für eine Datei mit Profildaten, die er mit File.Open() öffnet. Gibt es ein Problem, geht das Programm hoch.

```
stream = File.Open(profil);
```



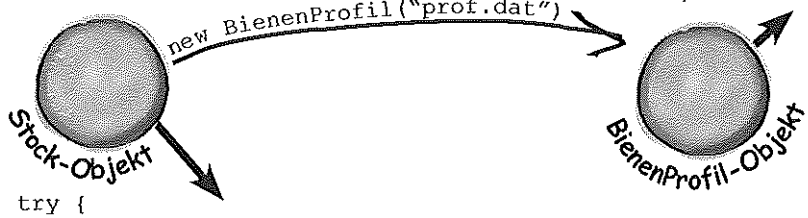
Das BienenProfil-Objekt versuchte, eine Datei zu lesen, die es nicht gab. File.Open() löste deswegen eine Exception aus. Da der Stock diese nicht abfängt, bleibt sie unbehandelt.

Beachten Sie, wie das BienenProfil-Objekt die Exception abfängt, mit der Methode LogEintragSchreiben() protokolliert und dann wieder auslöst, damit sie an den Stock übergeben wird.

```
try {
   stream = File.Open(profil);
} catch (FileNotFoundException ex) {
   LogEintragSchreiben("Konnte " + profil +
    "nicht öffnen: " + ex.Message();
   throw ex;
}
```

... können wir das tun:

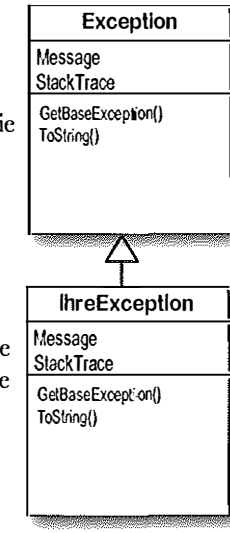
Das BienenProfil-Objekt kann die Exception abfangen und protokollieren. Dann kann es die Exception mit throw wieder auslösen und so an den Stock weiterreichen, der sie einfängt und verarbeitet, damit das Programm weiterlaufen kann.




```
try {
   prof = new BienenProfil("prof.dat");
} catch (FileNotFoundException) {
   Stock.ProfilWiderherstellen("prof.dat");
}
```

Wenn der Stock jetzt versucht, ein BienenProfil-Objekt zu erstellen, und dabei einen ungültigen Dateinamen übergibt, kann er sich darauf verlassen, dass BienenProfil den Fehler protokolliert und ihn dann auf das Problem aufmerksam macht, indem es eine Exception auslöst. Der Stock kann sie abfangen und korrigierend eingreifen – hier erstellt er das Profil neu.

Bienen brauchen eine HonigLeerException



Ihre Klassen können ihre eigenen Exceptions auslösen. Beispielsweise ist es recht verbreitet, dass man, wenn man in einer Methode, die einen Wert erwartet, einen Parameter erhält, der null ist, die gleiche Exception auslöst, die auch eine .NET-Methode auslösen würde:

```
throw new ArgumentException();
```

Aber manchmal möchte Ihr Programm eine Exception auslösen, die eine besondere Bedingung beschreibt, die während seiner Ausführung auftritt. Beispielsweise verbrauchen Bienen, die wir im Stock erstellt haben, in Abhängigkeit von ihrem Gewicht unterschiedliche Mengen Honig. Gibt es keinen Honig mehr, wäre es sinnvoll, dass der Stock eine Exception auslöst. Sie können eine eigene Exception-Klasse erstellen, die diese spezifische Fehlerbedingung beschreibt, indem Sie eine Klasse erstellen, die von `Exception` erbt, und diese Exception auslösen, wenn dieser Fehler auftritt.

```
public class HonigLeerException : System.Exception {
    public HonigLeerException(string message) : base(message) { }
}
public class HonigLieferSystem {
    ...
    public void EierMitHonigFüttern() {
       if (honigMenge == 0) {
            throw new HonigLeerException("Der Stock hat keinen Honig mehr.");
       } else {
            foreach (Ei ei in Eier) {
               ...
}
public partial class Form1 : Form {
...
   private void honigVerbrauchen_Click(object sender, EventArgs e) {
      HonigLieferSystem lieferung = new HonigLieferSystem();
      try {
           lieferung.EierMitHonigFüttern();
      }
      catch (HonigLeerException ex) {
          MessageBox.Show(ex.Message, "Warnung: Setze Stock zurück");
          Stock.Zurücksetzen();
      }
   }
}
```

Sie müssen eine Klasse für Ihre Exception erstellen und diese von System.Exception erben lassen. Beachten Sie, wie wir den Konstruktor überladen, damit wir eine Meldung übergeben können.

Hat der Stock noch Honig, wird die Exception nicht ausgelöst, stattdessen wird dieser Code ausgeführt.

Das löst eine Instanz der neuen Exception-Klasse aus.

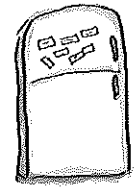
Eine eigene Exception können Sie ebenfalls über ihren Namen abfangen, um dann alles damit zu tun, um sie zu behandeln.

Hat der Stock keinen Honig mehr, kann keine der Bienen mehr arbeiten. Der Simulator kann also nicht mehr fortgesetzt werden. In diesem Fall kann das Programm nur weiterlaufen, wenn der Stock zurückgesetzt wird. Das können wir tun, indem wir im Catch-Block Code zum Zurücksetzen des Stocks definieren.



Exception-Magneten

Ordnen Sie die Magneten so, dass die Anwendung die Ausgabe auf der Konsole vornimmt.



```
public static void Main() {
    Console.Write("Beim");
    ExTestlauf.Null("ja");
    Console.Write(" braucht man ");
    ExTestlauf.Null("nein");
    Console.WriteLine(".");
}

class DieException : Exception { }
```

Ausgabe:

Beim Bauen braucht man Balken.

}

}

}

Alle Werttypen – auch Konstanten wie »ja« oder 37 oder true – haben eingebaute Methoden. "ja".Equals(t) liefert true, wenn der Wert der Variablen t dem String »ja« entspricht.

```
if ("ja".Equals(t)) {
```

```
Console.Write("u");
```

```
Console.Write("k");
```

```
Console.Write("B");
```

```
Console.Write("e");
```

```
Console.Write("n");
```

```
try {
```

```
} catch (DieException) {
```

```
throw new DieException();
```

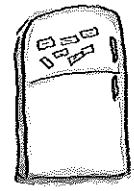
```
} finally {
```

```
riskantesTun(test);
```

```
        Console.Write("l");
    }
}
```

```
public class ExTestlauf {
  public static void Null(string test) {
```

```
static void riskantesTun(String t) {
  Console.Write("a");
```



Exception-Magneten, Lösung

Ordnen Sie die Magneten so, dass die Anwendung diese Ausgabe auf der Konsole vornimmt.

Ausgabe:

Beim Bauen braucht man Balken.

```
public static void Main() {
    Console.Write("Beim");
    ExTestlauf.Null("ja");
    Console.Write(" braucht man ");
    ExTestlauf.Null("nein");
    Console.WriteLine(".");
}

class DieException : Exception { }

public class ExTestlauf {
   public static void Null(string test) {
      try {
         Console.Write("B");
         riskantesTun(test);
         Console.Write("k");
      } catch (DieException) {
         Console.Write("u");
      } finally {
         Console.Write("e");
      }
      Console.Write("n");
   }

   static void riskantesTun(String t) {
      Console.Write("a");
      if ("ja".Equals(t)) {
         throw new DieException();
      }
      Console.Write("l");
   }
}
```

Diese Zeile definiert eine benutzerdefinierte Exception namens `DieException`, die im `catch`-Block des Codes abgefangen wird.

Die Methode `Null()` gibt entweder >>Bauen<< oder >>Balken<< aus, je nachdem, ob ihr über den Parameter `test` >>ja<< oder etwas anderes übergeben wurde.

Der `finally`-Block sichert, dass die Methode immer >>e<< ausgibt. Und das >>n<< wird außerhalb des Exception-Handlings ausgegeben, erfolgt also auch immer.

Diese Zeile wird nur ausgeführt, wenn `riskantesTun()` die Exception nicht auslöst.

Die Methode `riskantesTun()` löst die Exception nur aus, wenn ihr der String >>ja<< übergeben wird.

Punkt für Punkt

- Jede Anweisung kann eine Exception auslösen, wenn zur Laufzeit etwas fehlschlägt.
- Mit `try`/`catch`-Blöcken können Sie Exceptions behandeln. Unbehandelte Exceptions führen dazu, dass das Programm angehalten und eine Fehlermeldung angezeigt wird.
- Jede Exception im `try`-Block führt dazu, dass das Programm sofort zur ersten Anweisung des `catch`-Blocks springt.
- Das `Exception`-Objekt gibt Ihnen Informationen zur eingefangenen Exception. Geben Sie in der `catch`-Anweisung eine `Exception`-Variable an, enthält diese Informationen zur im `try`-Block ausgelösten Exception:

```
try {
  // Anweisungen, die eine Exception
  // auslösen könnten.
} catch (IOException ex) {
  // Wird eine Exception ausgelöst,
  // enthält ex Informationen dazu.
}
```

- Es gibt viele verschiedene Arten von Exceptions. Für jede gibt es eigene Klassen, die von `Exception` erben. Vermeiden Sie es, `Exception` abzufangen – fangen Sie spezifische Exception-Klassen ab.
- Ein `try` kann mehrere `catch`-Blöcke haben:

```
try { ... }
catch (NullReferenceException ex) {
   // Diese Anweisungen werden bei einer
   // NullReferenceException ausgeführt.
}
catch (OverflowException ex) { ... }
catch (Exception ex) {
   // Noch nicht behandelte Exceptions
   // werden in diesem Block behandelt.
}
```

- Mit `throw` kann Ihr Code selbst Exceptions auslösen:

```
throw;

throw new Exception("Exception-Meldung");
```

- Eigene Exception-Typen können Sie erstellen, indem Sie die Basisklasse `Exception` erweitern.

```
class EigeneException : Exception;
```

- Meist müssen Sie nur Exceptions auslösen, für die .NET eingebaute Exception-Klassen wie ArgumentException bietet. Verschiedene Arten von Exceptions liefern Sie Ihren Benutzern, damit Sie **ihnen mehr Informationen liefern können**. Ein Fenster mit der Meldung »Ein unbekannter Fehler trat auf« ist nicht annähernd so hilfreich wie eine Fehlermeldung, die besagt: »Der Entschuldigungsordner ist leer. Wählen Sie einen anderen Ordner, wenn Sie Entschuldigungen lesen möchten.«

Ein einfaches Mittel, um viele Probleme zu vermeiden: `using` gibt Ihnen try und finally kostenlos

Erinnern Sie sich: Für eine in einer >>using<<-Anweisung deklarierte Referenz wird nach Ende des Blocks automatisch Dispose() aufgerufen

Sie wissen bereits, dass `using` ein einfaches Mittel ist, mit dem Sie sicherstellen können, dass Ihre Dateien immer geschlossen werden. Aber Sie wissen nicht, dass das **lediglich ein C#-Kürzel** für `try` und `finally` ist!

```
using (IhreKlasse o
         = new IhreKlasse() ) {

    // Code

}
```

ist das Gleiche wie

```
IhreKlasse o = new IhreKlasse();

try {

    // Code

} finally {

    o.Dispose();
}
```

Mit der using-Anweisung nutzen Sie finally, um sicherzustellen, dass die Methode Dispose() immer aufgerufen wird.

Exception-Vorsorge: Implementieren Sie IDisposable für eigene Aufräumarbeiten

IDisposable ist ein effektives Mittel, um häufige Exceptions und Probleme zu vermeiden. Achten Sie darauf, dass Sie immer using-Anweisungen verwenden, wenn Sie mit einer Klasse arbeiten, die diese Schnittstelle implementiert.

Streams sind klasse, weil in sie bereits Code eingeschlossen ist, der sie schließt, wenn das Objekt nicht mehr benötigt wird. Aber was ist, wenn Sie ein eigenes Objekt haben, das immer etwas tun muss, wenn es nicht mehr benötigt wird? Wäre es nicht wunderbar, wenn Sie eigenen Code schreiben könnten, der immer aufgerufen würde, wenn Ihr Objekt in einer `using`-Anweisung verwendet wird?

In einer »using«-Anweisung können Sie nur Klassen verwenden, die IDisposable implementieren. Andernfalls lässt sich Ihr Programm nicht kompilieren.

Das ermöglicht Ihnen C# mit der Schnittstelle `IDisposable`. Folgendermaßen implementieren Sie `IDisposable` und schreiben in der Methode `Dispose()` Aufräumcode:

Ihre Klasse muss IDisposable implementieren, wenn Sie Ihre Objekte in using-Anweisungen verwenden mö hten.

```
class Nektar : IDisposable {
  private double menge;
  private BienenStock stock;
  private Stream stockLog;
  public Nektar(double menge, BienenStock stock, Stream stockLog) {
    this.menge = menge;
    this.stock = stock;
    this.stockLog = stockLog;
  }
  public void Dispose() {
    Stock.Hinzufügen(menge);
    Stock.LogEintragSchreiben(stockLog,
      menge + "mg Nektar hinzugefügt");
  }
}
```

Die Schnittstelle IDisposable hat nur einen Member: die Methode Dispose(). Was Sie in dieser Methode angeben, wird am Ende der using-Anweisung aufgeführt.

Dieser Code protokolliert die hinzugefügte Menge Nektar. Das ist wichtig und muss erfolgen, deswegen haben wir den Code in die Methode Dispose() gesteckt.

Auf gestapelte using-Anweisungen werden Sie häufig stoßen.

Jetzt können wir mehrere using-Anweisungen verwenden. Nutzen wir zunächst ein eingebautes Objekt, `Stream`, das `IDisposable` implementiert. Dann arbeiten wir mit unserem `Nektar`-Objekt, das ebenfalls `IDisposable` implementiert:

```
using (Stream Log = new File.Write("log.txt"))
using (Nektar nektar = new Nektar(16.3, stock, Log)) {
      Biene.FliegenZu(blume);
      Biene.Ernten(nektar);
      Biene.FliegenZu(stock);
}
```

Das Nektar-Objekt nutzt den Log-Stream, der automatisch am Ende der äußeren using-Anweisung geschlossen wird.

Das Biene-Objekt nutzt das Nektar-Objekt, das am Ende der inneren using-Anweisung automatisch die Menge protokolliert.

Es gibt keine Dummen Fragen

F: Kann Ich nur Objekte von Typen, die `IDisposable` implementieren, in einer usIng-Anwelsung verwenden?

A: Ja. `IDisposable` ist auf die Verwendung in `using`-Anweisungen zugeschnitten. Eine `using`-Anweisung ist, als würden Sie eine neue Instanz einer Klasse erstellen, die allerdings immer die `Dispose()`-Methode aufruft.

F: Kann ich in einem `using`-Block beliebige Anweisungen verwenden?

A: Klar. `using` soll für Sie sichern, dass das erstellte Objekt ordentlich aufgeräumt wird. Aber was Sie mit diesem Objekt zuvor machen, ist altein Ihre Sache. Sie könnten mit `using` auch ein Objekt erzeugen und es innerhalb des Blocks nie verwenden. Aber das wäre recht sinntos, deswegen würden wir Ihnen das nicht empfehlen

F: Kann man `Dispose()` außerhalb von `using`-Anwelsungen aufrufen?

A: Ja. Sie ***müssen*** `using` nicht verwenden. Sie können `Dispose()` auch selbst aufrufen, wenn Sie ein Objekt nicht mehr benötigen. Oder Sie können alle erforderlichen Aufräumarbeiten – wie den Aufruf der `Close()`-Methode von Streams – manuell durchführen. Aber wenn Sie `using` verwenden, macht es Ihren Code leichter verständlich und verhindert Probleme, die auftauchen, wenn Sie Ihre Objekte nicht aufräumen.

F: Sie haben einen »try/flnally«-Block erwähnt. Heißt das, dass ich try und flnally auch ohne catch verwenden kann?

A: Ja! Sie können einen Try-Biock ohne catch, aber mit finally haben. Das sieht so aus:

```
try {
      MachWasRiskantes();
      NochWasRiskantes();
}
finally {
      DasImmerAusführen();
}
```

Löst `MachWasRiskantes()` eine Exception aus, wird der Finally-Biock sofort ausgeführt.

F: Funktioniert Dispose() nur mit Dateien und Streams?

A: Nein. Es gibt viele Klassen, die `IDisposable` implementieren. Wenn Sie eine davon verwenden, sollten Sie immer eine `using`-Anweisung verwenden. (Ein paar davon werden Sie in den nächsten Kapiteln sehen.) Und wenn Sie eine Klasse schreiben, die auf bestimmte Weise aufgeräumt werden muss, können Sie `IDisposable` ebenfalls implementieren.



Wenn try/catch so klasse ist, warum umgibt die IDE nicht einfach alles damit? Dann müssten wir all diese try/catch-Blöcke nicht selbst schreiben.

Sie möchten wissen, welche Art Exception ausgelöst wurde, damit Sie genau diese behandeln können.

Beim Exception-Handling reicht es nicht aus, eine allgemeine Fehlermeldung auszugeben. Beispielsweise könnten wir in der Entschuldigungsverwaltung bei einer `FileNotFoundException` einen Fehler mclden, der vorschlägt, wo sich die richtigen Dateien befinden könnten. Haben wir eine Exception, die sich auf Datenbanken bezieht, könnten wir eine E-Mail an den Datenbankadministrator schicken. All das basiert darauf, dass Sie *bestimmte* Arten von Exceptions abfangen.

Deswegen gibt es so viele Klassen, die von Exception erben, und deswegen möchten Sie eventuell auch Ihre eigenen Exception-Klassen schreiben.

Der SCHLIMMSTE catch-Block: Kommentare

Ein Catch-Block hält Ihr Programm am Laufen, wenn Sie das möchten. Es wird eine Exception ausgelöst, Sie fangen sie ein, und statt eine Fehlermeldung zu liefern und das Programm abzubrechen, machen Sie weiter. Aber manchmal ist das nicht das Richtige.

Sehen Sie sich diese Rechner-Klasse an, die sich die ganze Zeit seltsam zu verhalten scheint. Aber was passiert hier?

```
public class Rechner {

...

  public void Teilen(float zähler, float nenner) {

    try {

      this.quotient = zähler / nenner;

    } catch {

      // Anmerkung von Tim: Irgendwie müssen wir noch verhindern,

      // dass es zu einer Division durch null kommen kann.

    }

  }

}
```

Hier ist das Problem. Ist der Nenner null, führt das zu einer DivideByZeroException.

Aber es gibt einen Catch-Block. Warum erhalten wir also immer noch Fehler?

Der Programmierer dachte, er könnte seine Exception vergraben, indem er einen leeren Catch-Block einsetzt. Aber damit bereitet er nur demjenigen Kopfschmerzen, der später Probleme mit dem Programm aufspüren muss.

Sie sollten Exceptions behandeln, nicht vergraben

Dass Sie Ihr Programm weiterlaufen lassen, bedeutet noch nicht, dass Sie Ihre Exception *behandelt* haben. Der Code oben verhindert, dass der Rechner abstürzt ... zumindest in der Methode `Teilen()`. Aber was ist, wenn anderer Code diese Methode aufruft und versucht, das Ergebnis auszugeben? War der Nenner null, hat die Methode wahrscheinlich einen falschen (und unerwarteten) Wert zurückgeliefert.

Statt einfach einen Kommentar einzufügen und die Exception damit zu begraben, müssen Sie die **Exception behandeln**. Und wenn Sie das Problem nicht behandeln können, ***lassen Sie keine leeren oder nur aus einem Kommentar bestehenden Catch-Blöcke stehen!*** Damit erschweren Sie demjenigen die Arbeit, der das Problem lösen soll. Es ist besser, wenn Sie das Programm weiter Exceptions auslösen lassen, weil man dann leichter herausfinden kann, was schiefgeht.

Vorübergehende Lösungen sind okay (vorübergehend)

Manchmal stoßen Sie auf ein Problem und wissen zwar, dass das wirklich eins ist, Sie wissen aber nicht, was Sie dagegen tun sollen. In solchen Fällen sollten Sie das Problem protokollieren und festhalten, was passiert. Das ist nicht so gut wie eine ordentliche Behandlung der Exception, aber besser als gar nichts.

Hier ist eine vorübergehende Lösung für den Rechner:

```
public class Rechner {

...

  public void Teilen(float zähler, float nenner) {

    try {

      this.quotient = zähler / nenner;

    } catch (Exception ex) {

      StreamWriter sw = new StreamWriter(@"C:\Logs\errors.txt");
      sw.WriteLine(ex.getMessage());
      sw.Close();

    }

  }

}
```

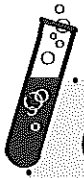
Das muss immer noch repariert werden, aber vorübergehend macht es klar, wo das Problem aufgetreten ist.



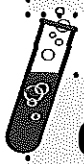
Exceptions zu behandeln ist nicht immer das Gleiche, wie Exceptions zu REPARIEREN.

Es ist nie gut, wenn Ihr Programm hochgeht. Aber noch viel schlimmer ist es, wenn man überhaupt nicht weiß, dass das Programm abstürzt oder was es mit den Daten des Benutzers macht. Deswegen müssen Sie darauf achten, dass Sie sich um die Fehler kümmern, die Sie lösen können, und die protokollieren, die Sie nicht lösen können.

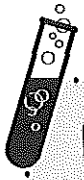
Einige einfache Gedanken zum Exception-Handling



Gestalten Sie Ihren Code so, dass er VERSTÄNDLICH fehlschlägt.



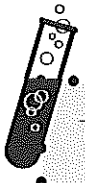
Geben Sie Ihren Benutzern NÜTZLICHE Fehlermeldungen.



Lösen Sie, wenn möglich, eingebaute .NET-Exceptions aus. Verwenden Sie eigene nur, wenn Sie spezifische Informationen liefern müssen.



Denken Sie über Code im Try-Block nach, der übersprungen werden KÖNNTE.



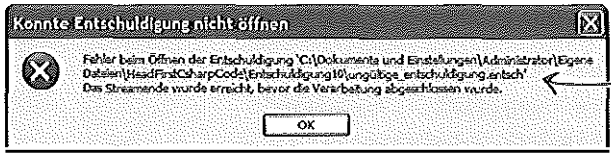
... und am wichtigsten ...

Vermeiden Sie unnötige Dateisystemfehler ... verwenden Sie bei Streams immer einen Using-Block! Immer immer immer!



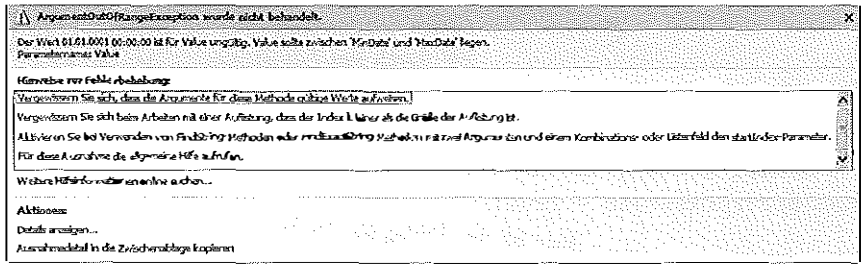
Nutzen Sie das, was Sie über `try/catch/finally` wissen, um das Exception-Handling in Brians Entschuldigungsverwaltung zu verbessern.

1. Fügen Sie dem Click-Event-Handler des Öffnen-Buttons Exception-Handling hinzu. Erstellen Sie einen einfachen try/catch-Block, der ein Meldungsfenster einblendet. Folgendes sollte angezeigt werden, wenn Sie versuchen, eine Datei zu öffnen, die keine Entschuldigungsdatei ist:



Bei einer sehr kleinen Datei erhalten Sie diese Exception-Meldung, bei einer größeren Datei erhalten Sie eine andere ("Der Eingabestream hat kein gültiges Format."). Ihr Meldungsfenster sollte bei beiden Meldungen funktionieren.

2. Sie sind noch nicht fertig. Öffnen Sie die Entschuldigungsverwaltung, wählen Sie einen Ordner und geben Sie Daten in die Felder »Beschreibung« und »Reaktion« ein, **aber keine in das Feld »Letzte Verwendung«**. Versuchen Sie jetzt, einen Ordner zu wählen und die Entschuldigung zu speichern. Haben Sie diese `ArgumentOutOfRange`-Exception erhalten?



Spüren Sie diese Exception mit dem Debugger auf. Diese Exception lässt sich vollkommen vermeiden – können Sie das Programm reparieren und sichern, dass die Exception überhaupt nicht auftritt? *(Tipp: Dazu müssen Sie keinen try/catch-Block hinzufügen. Sie müssen **herausfinden, warum** das Feld »Letzte Verwendung« zu einem Problem führt. Prüfen Sie die Exception-Meldung genau auf Hinweise darauf.)*

3. Eine letzte Sache. Bevor das Programm die `ArgumentOutOfRangeException` auslöste, hat es eine Datei gespeichert. Laden Sie diese Datei, sollten Sie die gleiche Exception erhalten. Erhalten Sie beim Versuch, eine Datei zu öffnen, einen anderen Fehler, ist es keine gültige Entschuldigungsdatei. Fügen Sie einen Exception-Handler ***in den Block ein, den Sie in Schritt 2*** hinzugefügt haben, um sicherzustellen, dass das Programm nicht fehlschlägt, wenn Sie versuchen, eine ungültige Entschuldigungsdatei zu laden (was unter verschiedenen Umständen passieren kann). Dazu müssen Sie Folgendes tun:

   1. Deklarieren Sie über dem try/catch-Block eine Boolesche Variable namens `formularZurücksetzen`. Setzen Sie diese auf true, wenn es eine Exception gibt, und prüfen Sie sie später, um zu sehen, ob das Formular zurückgesetzt werden soll.
   2. Fügen Sie in den gerade dem Öffnen-Button hinzugefügten `try/catch`-Block einen weiteren ein.
   3. Hängen Sie an das äußere `try/catch` einen `finally`-Block an, der das Formular in seinen ursprünglichen Zustand zurückversetzt. Setzen Sie `LetzteVerwendung.Value` auf `DateTime.Now` (das liefert das aktuelle Datum), wenn die Variable `formularZurücksetzen` true ist.



Lösungen zu den Übungen

Nutzen Sie das, was Sie über `try/catch/finally` wissen, um das Exception-Handling in Brians Entschuldigungsverwaltung zu verbessern.

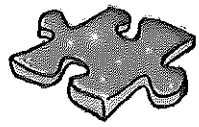
```
private void open_Click(object sender, EventArgs e) {
    if (GeändertPrüfen()) {
        openFileDialog1.InitialDirectory = gewählterOrdner;
        openFileDialog1.Filter =
          "Entschuldigung-Dateien (*.entsch)|*.entsch|Alle Dateien (*.*)|*.*";
        openFileDialog1.FileName = beschreibung.Text + ".excuse";
        DialogResult ergebnis = openFileDialog1.ShowDialog();
        if (ergebnis == DialogResult.OK) {
            bool formularZurücksetzen = false;
            try {
                dieEntschuldigung = new Entschuldigung(openFileDialog1.FileName);
                try {
                    FormularAktualisieren(false);
                }
                catch {
                    MessageBox.Show("Die Entschuldigung-Datei '"
                        + openFileDialog1.FileName + "' ist ungültig",
                        "Konnte Entschuldigung nicht öffnen");
                    formularZurücksetzen = true;
                }
            }
            catch (Exception ex) {
                MessageBox.Show("Fehler beim Öffnen der Entschuldigung '"
                    + openFileDialog1.FileName + "'\n" + ex.Message,
                    "Konnte Entschuldigung nicht öffnen", MessageBoxButtons.OK,
                    MessageBoxIcon.Error);
                formularZurücksetzen = true;
            }
            finally {
                if (formularZurücksetzen) {
                    beschreibung.Text = "";
                    reaktion.Text = "";
                    letzteVerwendung.Value = DateTime.Now;
                }
            }
        }
    }
}
```

Dieser try/catch-Block erzeugt das Fenster mit der Fehlermeldung, wenn über den Entschuldigung-Konstruktor eine Entschuldigung geladen werden soll.

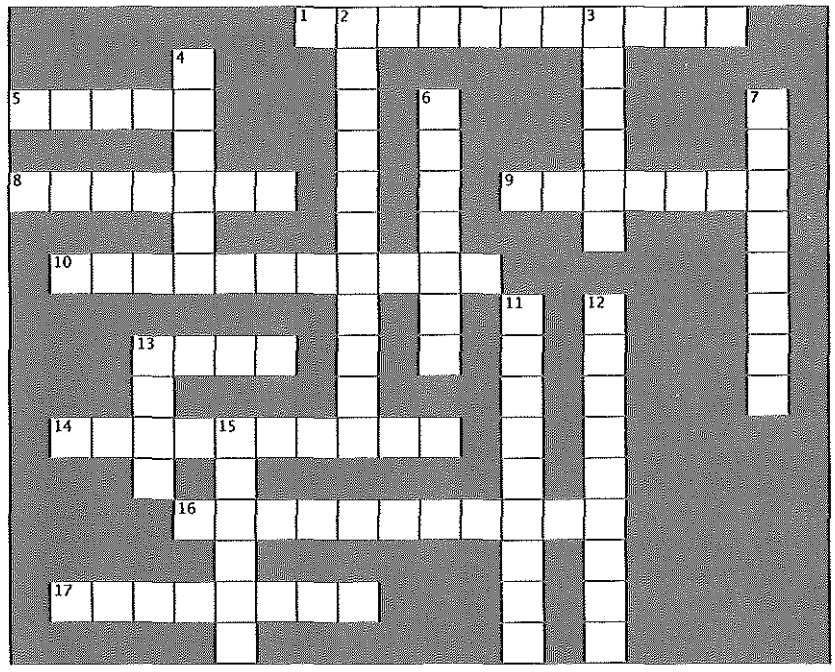
Das ist ein eingebettetes try/catch. Es kümmert sich um Exceptions, die auftreten, wenn eine Datei mit ungültigen Daten geladen wird. Das ist ein anderes Problem als das, was durch den Entschuldigung-Konstruktor entsteht.

Hier ist der Dialog zum äußeren try/catch-Block. Er gibt die Exception-Meldung aus.

Beide catch-Blöcke setzen formularZurück-setzen auf true, damit der finally-Block weiß, dass das Formular zurückgesetzt werden soll. Code, der mit dem finally-Block interagiert, ist in Ordnung, da Sie wissen, dass der finally-Block immer ausgeführt wird.



Exception-Kreuzworträtsel

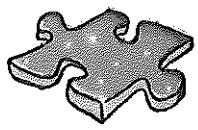


Waagerecht

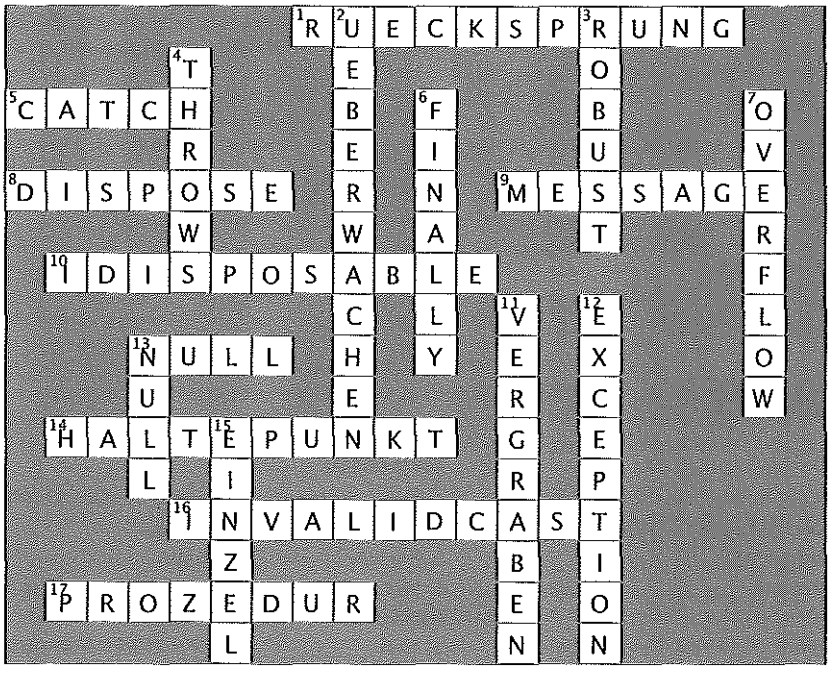
1. »Ausführen bis _____« sagt dem Debugger, dass er alle Anweisungen in der aktuellen Methode ausführen und dann anhalten soll.
5. Ein Try-Block kann mehrere _____-Blöcke haben.
8. Diese Methode wird am Ende eines Using-Blocks immer aufgerufen.
9. Das Feld im Exception-Objekt, das einen String mit einer Beschreibung enthält.
10. Variablen können nur dann in using-Anweisungen deklariert werden, wenn ihr Typ diese Schnittstelle implementiert.
13. Das enthält eine Referenz, wenn sie auf nichts zeigt.
14. Das schalten Sie an, wenn der Debugger anhalten soll, sobald er auf eine bestimmte Codezeile trifft.
16. Eine _____Exception tritt auf, wenn Sie einen Wert auf einen Typ casten, in den er nicht passt.
17. Ist die nächste Anweisung ein Methodenaufruf, führt der Debugger mit »_____schritt« die gesamte Methode aus und hält unmittelbar hinter ihr an.

Senkrecht

2. Das Fenster der IDE, In dem Sie die Werte Ihrer Variablen überprüfen können.
3. Ein derartiges Programm kommt gut mit Fehlern klar.
4. Mit diesem Schlüsselwort können Sie bei einem Problem eine Exception auslösen.
6. Der _____-Block enthält Anweisungen, die nach der Behandlung einer Exception unbedingt ausgeführt werden müssen.
7. Eine _____Exception heißt, dass Sie versucht haben, in eine Variable einen Wert zu quetschen, der nicht hineinpasst.
11. Wenn Sie Ihre Exceptions _____, können sie schwer aufzuspüren sein.
12. Die Basisklasse, von der DivideByZeroException und FormatException erben.
13. Wenn Sie versuchen, dadurch zu teilen, erhalten Sie eine Exception.
15. Wenn die nächste Anweisung ein Methodenaufruf ist, sagt »_____schritt« dem Debugger, dass er die erste Anweisung in dieser Methode ausführen soll.



Exception-Kreuzworträtsel, Lösung



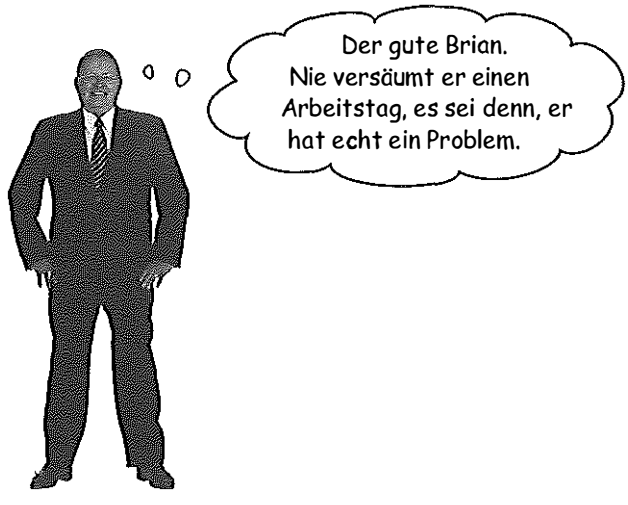
finally verschafft Brian seinen Urlaub ...

Nachdem Brian seine Exceptions im Griff hat, läuft seine Arbeit reibungslos. Er kann sich also endlich seinen wohlverdienten (und von seinem Chef genehmigten) Urlaub nehmen.



... und auch im Büro sieht alles gut aus.

Ihre Fähigkeiten im Exception-Handling haben nicht nur Probleme verhindert. Sie haben auch gesichert, dass Brians Chef keine Ahnung hat, dass jemals etwas schiefgelaufen ist!



Gutes Exception-Handling ist für die Benutzer unsichtbar. Das Programm stürzt nie ab, und wenn es Probleme gibt, werden diese immer sauber und ohne verwirrende Fehlermeldungen behandelt.

11 Events und Delegates

Was Ihr Code macht, wenn Sie nicht gucken



Ihre Objekte beginnen, für sich selbst zu denken.

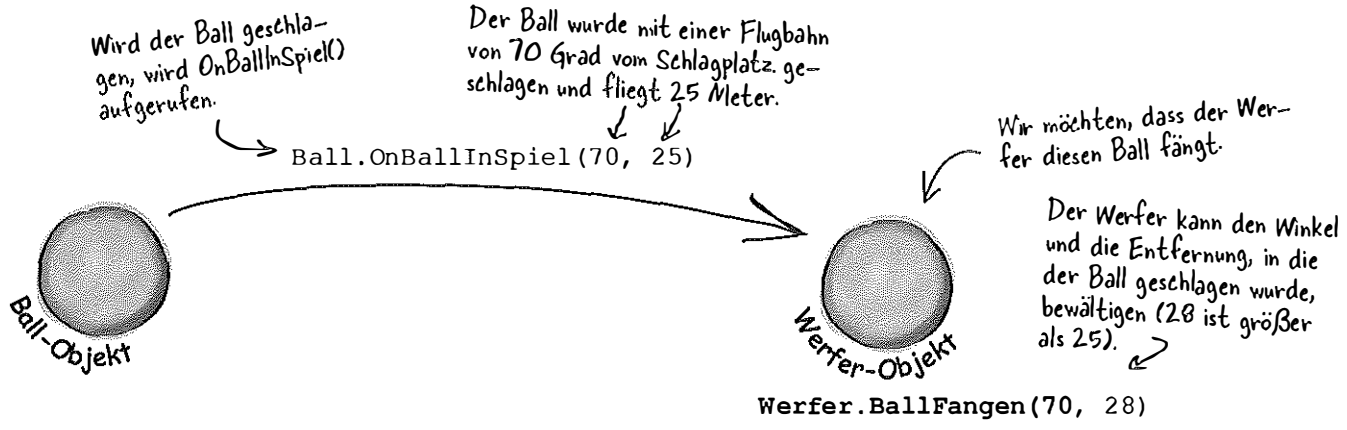
Sie können nicht immer kontrollieren, was Ihre Objekte machen. Manche Dinge passieren einfach. Und wenn sie passieren, sollten Ihre Objekte schlau genug sein, **auf alles eine Antwort zu wissen**, was eintreten könnte. Darum geht es bei Events (oder Ereignissen). Ein Objekt *veröffentlicht* ein Event, andere Objekte *abonnieren* es, und alle arbeiten zusammen, damit die Dinge im Fluss bleiben. Das geht so lange gut, bis zu viele Objekte auf dasselbe Event reagieren. Dann werden sich **Callbacks** als praktisch erweisen.

Möchten auch Sie, dass Ihre Objekte für sich selbst denken?

Angenommen, Sie schreiben einen Baseball-Simulator. Sie modellieren ein Spiel, verkaufen die Software an die Yankees (die haben ein dickes Portmonnaie) und scheffeln die Millionen. Sie erstellen `Ball`-, `Werfer`-, `Schiri`- und `Fan`-Objekte und eine Menge mehr. Sie schreiben Ihren Code so, dass ein `Werfer`-Objekt einen Ball fangen kann.

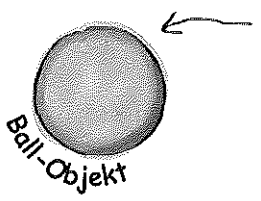
Das ist ein Standardverfahren zur Bezeichnung von Methoden – später werden wir noch mehr dazu erzählen.

Jetzt müssen Sie alles nur noch zusammenknüpfen. Sie fügen `Ball` eine `OnBallInSpiel()`-Methode hinzu und möchten nun, dass Ihr `Werfer`-Objekt mit seiner Event-Handler-Methode darauf reagiert. Nachdem die Methoden geschrieben sind, müssen die einzelnen Methoden nur noch verknüpft werden.



Aber woher weiß ein Objekt, dass es reagieren soll?

Hier ist das Problem. Sie wollen, dass Ihr `Ball`-Objekt sich nur um das Geschlagenwerden kümmert und Ihr `Werfer`-Objekt nur darum, Bälle zu fangen, die in seinen Bereich fliegen. Anders gesagt, Sie möchten nicht, dass der `Ball` dem `Werfer` sagen muss: »Ich komme zu dir.«



Der Ball weiß nicht, welcher Feldspieler ihn aufheben wird ... vielleicht der Werfer oder der Fänger oder der Mann am dritten Base, der sich zum DazwischenSpringen() entschließt.

Ihre Objekte sollen sich nur um sich selbst kümmern, nicht um andere Objekte. Sie trennen die Verantwortlichkeiten der Objekte.

Das heißt nicht, dass Objekte nicht interagieren können. Es bedeutet einfach, dass ein Ball nicht festlegen soll, wer ihn im Feld aufnimmt. Das ist nicht Aufgabe des Balls.

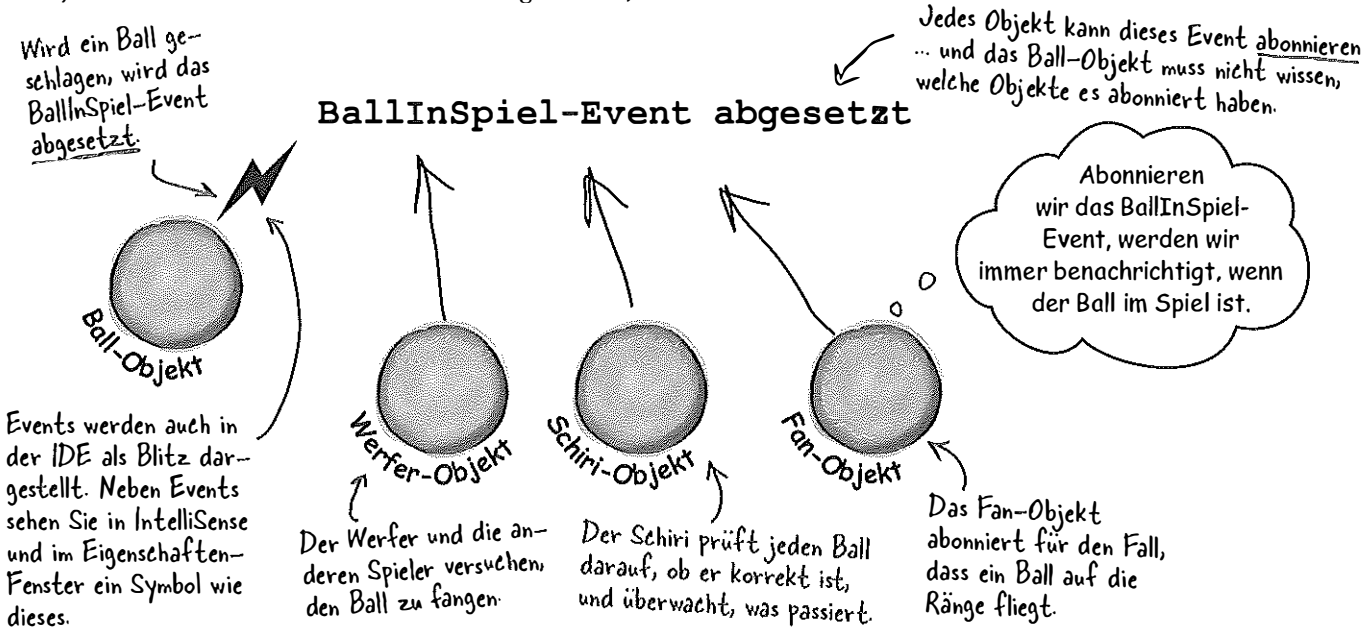
Wenn Events eintreten ... lauschen Objekte

Event, Nomen.
Ein **Ding**, das passiert und meist wichtig ist.
Die Sonnenfinsternis war ein umwerfendes ***Event****.*

Dazu müssen Sie ein **Event** nutzen, wenn der Ball geschlagen wird. Ein Event ist einfach ***etwas, das in Ihrem Programm passiert***, ein Ereignis. Andere Objekte können dann auf dieses Event reagieren – beispielsweise unser `Werfer`-Objekt.

Noch besser ist, dass mehrere Objekte auf ein Event lauschen können. Der Werfer lauscht auf ein Ball-geschlagen-Event und ebenso ein `Fänger`, ein `MannAmDrittenBase`, ein `Schiri` und sogar ein `Fan`. Und alle Objekte können auf das Event unterschiedlich reagieren.

Wir möchten also ein `Ball`-Objekt, das **ein Event absetzen kann**. Dann sollen andere Objekte **diesen bestimmten Typ von Event abonnieren** ... das bedeutet, dass sie darauf lauschen und benachrichtigt werden, wenn dieses Event eintrtitt.



Sie wollen ETWAS mit einem Event TUN? Dann brauchen Sie einen Event-Handler!

»Erfährt« Ihr Objekt von einem Event, können Sie Code einrichten, der daraufhin ausgeführt wird. Derartigen Code bezeichnet man als **Event-Handler**. Ein Event-Handler erhält Informationen zu dem Event und wird jedes Mal ausgeführt, wenn es eintritt.

Das tun wir schon die ganze Zeit. Jedes Mal, wenn Sie auf einen Button klicken, wird ein Event abgesetzt, auf das Ihr Code reagiert.

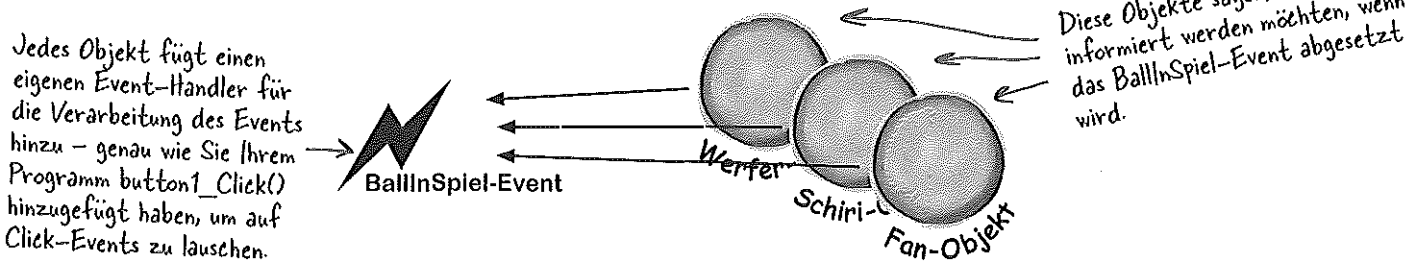
Denken Sie daran, dass all das zur Laufzeit ***ohne Ihr Eingreifen*** erfolgt. Sie schreiben also Code zum Auslösen eines Events sowie Code, der diese Events verarbeitet, und starten Ihre Anwendung. Wird ein Event abgesetzt, tritt dann immer Ihr Event-Handler in Aktion ... *ohne dass Sie etwas tun.* Und das Beste ist, dass Ihre Objekte dann getrennte Verantwortlichkeiten haben. Sie kümmern sich um sich selbst, nicht um andere Objekte.

Ein Objekt setzt ein Event ab, andere lauschen darauf ...

Schauen wir uns an, wie Events, Event-Handler und das Abonnieren in C# funktionieren:

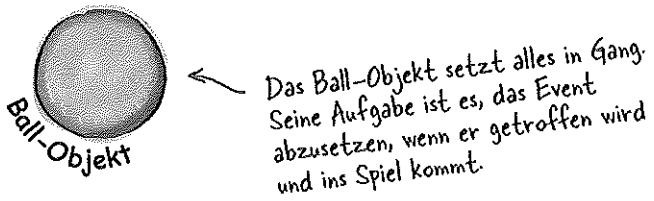
1 **Erst abonnieren die anderen Objekte ein Event.**

Bevor der `Ball` sein `BallInSpiel`-Event absetzen kann, müssen es die anderen Objekte abonnieren. Auf diese Weise sagen sie, dass sie informiert werden wollen, wenn ein `BallInSpiel`-Event eintritt.



2 **Irgendetwas setzt ein Event ab.**

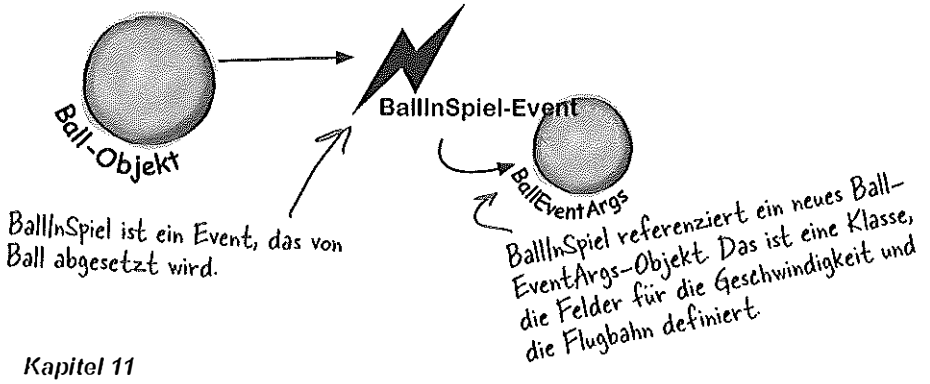
Der Ball wird getroffen. Das `Ball`-Objekt kann ein neues Event absetzen.



Manchmal spricht man auch davon, dass Events feuern oder aufgerufen werden. Das bedeutet das Gleiche wie Absetzen. Diese Begriffe werden synonym verwendet.

3 **Der Ball setzt ein Event ab.**

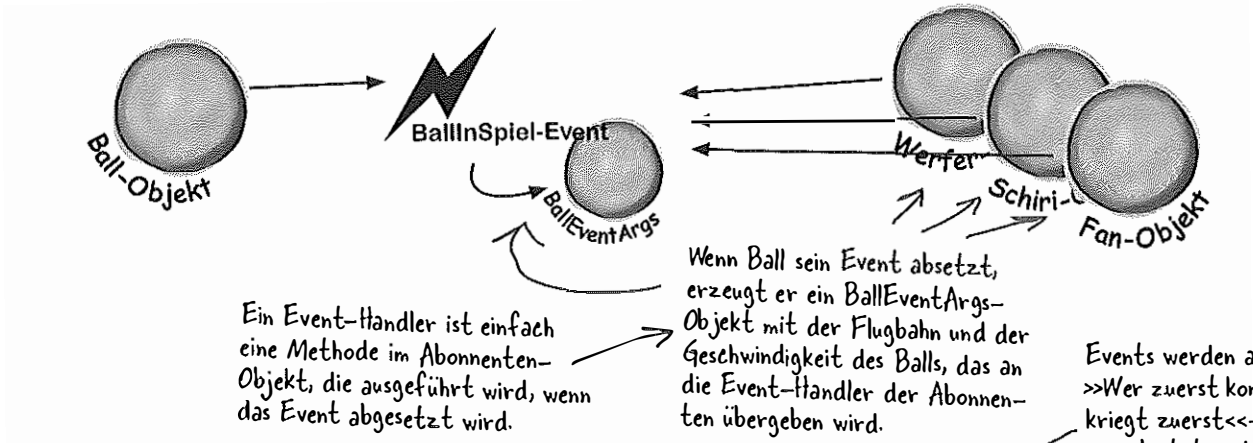
Es wird ein neues Event erzeugt (wie das funktioniert, werden wir gleich erklären). Dieses Event hat einige Argumente wie die Geschwindigkeit des Balls und seine Flugbahn. Diese Argumente sind als Instanz der Klasse EventArgs an das Event geknüpft. Dann wird das Event abgeschickt und ist für jeden verfügbar, der darauf lauscht.



Dann behandeln andere Objekte das Event

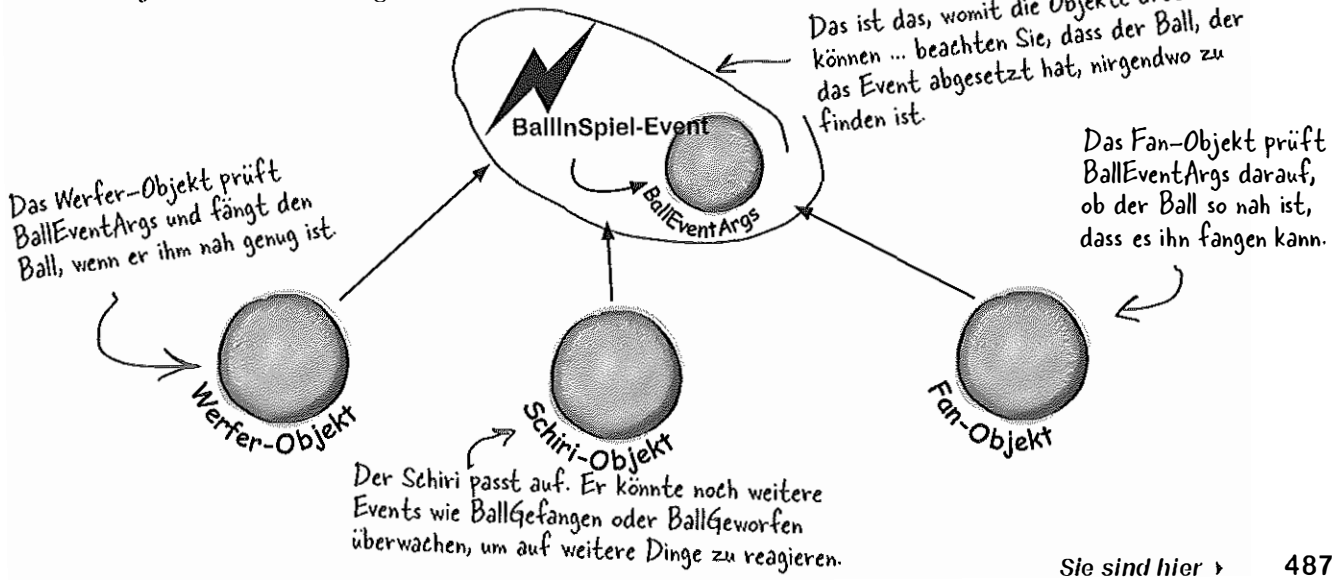
Nachdem ein Event abgesetzt wurde, erhalten alle Objekte, die es abonniert haben, eine Benachrichtigung und können darauf reagieren:

④ **Abonnenten erhalten Benachrichtigungen.**
Da die `Werfer`-, `Schiri`- und `Fan`-Objekte das `BallInSpiel`-Event von `Ball` abonniert haben, werden sie alle benachrichtigt – ihre Event-Handler-Methoden werden nacheinander aufgerufen.



Events werden auf einer >>Wer zuerst kommt, kriegt zuerst<<-Basis verarbeitet – das Objekt, das als Erstes abonniert, wird zuerst benachrichtigt.

⑤ **Alle Objekte verarbeiten das Event.**
Jetzt können `Werfer`, `Schiri` und `Fan` das `BallInSpiel`-Event auf jeweils eigene Art verarbeiten. Aber sie werden nicht alle parallel ausgeführt – die Event-Handler werden nacheinander mit einer Referenz auf ein `BallEventArgs`-Objekt als Parameter aufgerufen.

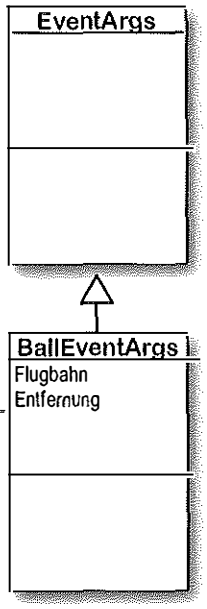


Die Punkte verbinden

Nachdem Sie wissen, was passiert, können wir uns genauer ansehen, wie die Dinge zusammenspielen. Glücklicherweise gibt es nur wenige bewegliche Teile.

Es ist gut (allerdings nicht erforderlich), wenn Ihr Event-Argument von EventArgs erbt. Das ist eine leere Klasse – sie hat keine öffentlichen Member.

Das bedeutet, dass Sie Ihr EventArgs-Objekt upcasten können, wenn Sie es mit einem Event senden müssen, das sich nicht speziell mit ihm befasst.



Ball nutzt diese Eigenschaften, um den Event-Handlern Informationen darüber zu geben, wie der Ball getroffen wurde.

1 Wir brauchen ein Objekt für die Event-Argumente.

Denken Sie daran, dass unser `BallInSpiel`-Event ein paar Argumente mit sich trägt. Wir benötigen also ein sehr einfaches Objekt für diese Argumente. .NET bietet eine Standardklasse dafür, die **`EventArgs`** heißt. Aber diese **hat keine Member.** Sie soll Ihnen nur ermöglichen, an Event-Handler Event-Argument-Objekte zu übergeben, mit denen sie arbeiten können. Hier ist die Klassendeklaration:

```
public class BallEventArgs : EventArgs
```

2 Dann definieren wir das Event in der Klasse, die es absetzt.

Die Klasse Ball hat eine Zeile mit dem **Schlüsselwort `event`** – auf diese Weise informiert sie die anderen Objekte über das Event, damit diese es abonnieren können. Diese Zeile kann sich an beliebiger Stelle der Klasse finden – üblicherweise steht sie in der Nähe der Eigenschaftendeklarationen. Sobald sie sich in der Klasse Ball befindet, können andere Objekte das Ball-Event abonnieren. Diese sieht so aus:

```
public event EventHandler BallInSpiel;
```

Events sind üblicherweise öffentlich. Dieses Event ist in der Klasse Ball definiert, aber wir möchten, dass Werfer, Schiri usw. es referenzieren können. Sie könnten es privat machen, wenn nur Instanzen derselben Klasse es abonnieren können sollen.

Nach dem Schlüsselwort Event kommt `EventHandler`. Das ist kein reserviertes C#-Schlüsselwort – es ist ein Teil von .NET. Sie brauchen es, um den Objekten, die das Event abonnieren, mitzuteilen, wie ihre Event-Handler-Methoden aussehen müssen.

Wenn Sie `EventHandler` nutzen, sagen Sie anderen Objekten, dass ihre EventHandler zwei Parameter erwarten müssen, ein `object` namens `sender` und eine `EventArgs`-Referenz namens `e`.

③ Die abonnierenden Klassen brauchen Event-Handler-Methoden.

Jedes Objekt, das das BallInSpiel-Event von Ball abonnieren will, benötigt einen Event-Handler. Wie Event-Handler funktionieren, wissen Sie bereits – jedes Mal, wenn Sie eine Methode zur Verarbeitung des Click-Events eines Buttons oder das ValueChanged-Event eines NumericUpDown-Elements hinzugefügt haben, hat die IDE Ihrer Klasse eine **Event-Handler-Methode** hinzugefügt. Das BallInSpiel-Event von Ball funktioniert genau so, und Event-Handler sollten Ihnen ziemlich vertraut sein:

```
void ball_BallInSpiel(object sender, EventArgs e)
```

Es gibt keine C#-Regel, die festlegt, dass die Namen Ihrer Event-Handler eine bestimmte Form haben, aber es gibt eine ziemlich standardisierte Namenskonvention: Die Namen sollten aus dem Namen der Objektreferenz, einem Unterstrich und dem Namen des Events bestehen.

Die Klasse, die diese spezielle Event-Handler-Methode enthält, hat eine Ball-Referenz namens `ball`. Ihr BallInSpiel-Event-Handler beginnt deswegen mit »`ball_`«. Darauf folgt der Name des Events, das behandelt wird, »`BallInSpiel`«.

Die Deklaration des BallInSpiel-Events gibt den Typ `EventHandler` vor. Das bedeutet, dass die Methode zwei Parameter akzeptieren muss – ein Objekt namens sender und ein EventArgs-Objekt namens e – und keinen Rückgabewert haben darf.

④ Die einzelnen Objekte abonnieren das Event.

Nachdem wir den Event-Handler eingerichtet haben, müssen die Werfer-, Schiri-, MannAmDrittenBase- und Fan-Objekte ihre Event-Handler registrieren. Jedes von ihnen hat eine spezielle `ball_BallInSpiel`-Methode, die auf jeweils andere Weise auf das Event reagiert. Gibt es eine Ball-Referenz namens `ball`, können die Event-Handler mit dem +=-Operator registriert werden:

```
ball.BallInSpiel += new EventHandler(ball_BallInSpiel);
```

Das sagt C#, dass der Event-Handler mit dem BallInSpiel-Event des Objekts verbunden werden soll, auf das die Referenz ball zeigt.

Mit dem +=-Operator kann ein Event-Handler für ein Event registriert werden.

Dieser Teil gibt die Event-Handler-Methode an, die das Event abonnieren soll.

Die Signatur der Event-Handler-Methode (ihre Parameter und ihr Rückgabewert) muss der entsprechen, die von EventHandler definiert wird. Sonst lässt sich das Programm nicht kompilieren.

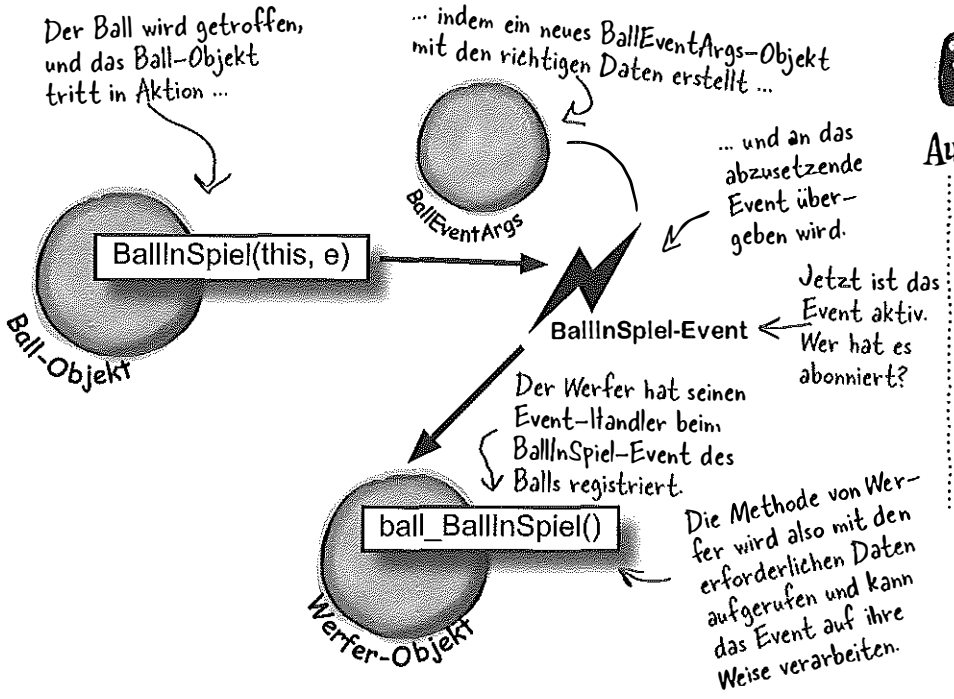
Blättern Sie um, es kommt noch was ... →

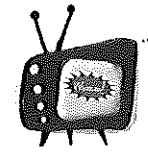
⑤ Ein Ball-Objekt löst sein Event aus, um die Abonnenten darüber zu informieren, dass es im Spiel ist.

Nachdem die Events eingerichtet sind, kann Ball sein **Event absetzen**, um auf etwas zu reagieren, das im Simulator passiert. Das Absetzen des Events ist leicht – es muss nur das BallInSpiel-Event aufgerufen werden.

```
if (BallInSpiel != null)
  BallInSpiel(this, e);
```

e ist ein neues BallEventArgs-Objekt.





Aufgepasst

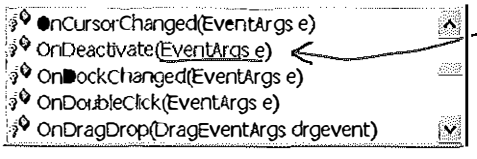
Wird ein Event ohne Handler abgesetzt, wird eine Exception ausgelöst.

Haben keine Objekte dem Event ihre Event-Handler hinzugefügt, ist das Event null. Prüfen Sie also immer, ob Ihr Event ungleich null ist, bevor Sie es absetzen. Tun Sie das nicht, erhalten Sie eine NullReferenceException.

Nutzen Sie Standardnamen für Methoden zum Absetzen von Events

Nehmen Sie sich eine Minute und gehen Sie zu dem Code eines Formulars. Geben Sie an einer Stelle, an der Sie eine Methode definieren, das Schlüsselwort override ein. Drücken Sie danach die Leertaste, springt ein IntelliSense-Fenster auf:

```
override
```



Sehen Sie, dass jede dieser Methoden einen EventArgs-Parameter erwartet? Alle übergeben diesen Parameter an das Event, wenn Sie es absetzen.

Es gibt eine Menge Events, die ein Form-Objekt absetzen kann, und für jedes davon gibt es eine eigene Methode, die es absetzt. Die OnDoubleClick()-Methode setzt das DoubleClick-Event ab, und das ist ihre einzige Daseinsberechtigung. Das Ball-Event wird also der gleichen Konvention folgen: Wir werden dafür sorgen, dass es **eine Methode namens OnBallInSpiel hat**, die ein BallEventArgs-Objekt als Parameter erwartet. Der Baseball-Simulator ruft diese Methode immer dann auf, wenn er das BallInSpiel-Event absetzen muss – entdeckt der Simulator, dass der Ball mit dem Schläger getroffen wurde, erzeugt er eine BallEventArgs-Instanz mit der Flugbahn und der Distanz des Balls und übergibt sie an OnBallInSpiel().

Es gibt keine Dummen Fragen

F: Warum muss ich bei der Deklaration eines Events das Wort EventHandler einschließen? Ich dachte, der Event-Handler wäre das, was die anderen Objekte nutzen, um das Event zu abonnieren.

A: Das stimmt – wenn Sie ein Event abonnieren müssen, schreiben Sie eine Methode, die als Event-Handler bezeichnet wird. Ist Ihnen aber aufgefallen, dass wir das Wort `EventHandler` in der Event-Deklaration (Schritt 2) ***und*** in der Zeile verwendet haben, mit der wir den Event-Handler beim Event registrieren (Schritt 4)? `EventHandler` definiert die **Signatur** des Events – das Wort sagt Objekten, die das Event abonnieren, wie sie ihre Methoden definieren müssen. Es sagt genau, dass eine Methode, die für dieses Event registriert werden soll, zwei Parameter erwarten (eine `object`- und eine `EventArgs`-Referenz) und den Rückgabewert `void` haben muss.

F: Was passiert, wenn ich versuche, eine Methode zu verwenden, die nicht dem von EventHandler definierten Schema entspricht?

A: Dann lässt sich Ihr Programm nicht kompilieren. Der Compiler sichert, dass Sie nie aus Versehen eine inkompatible Methode für ein Event registrieren. Darum ist der Standard-Event-Handler `EventHandler` so nützlich – sobald Sie ihn sehen, wissen Sie genau, wie Ihre Event-Handler-Methode aussehen muss.

F: Moment. »Standard-Event-Handler?« Gibt es noch andere?

A: Ja! Ihre Events ***müssen nicht ein*** Objekt und ein EventArgs übergeben. Sie können alles Mögliche versenden – oder auch nichts! Schauen Sie sich die letzte Zeile im IntelliSense-Fenster auf der gegenüberliegenden Seite an. Sehen Sie, dass die Methode OnDragDrop eine DragEventArgs-Referenz statt einer EventArgs-Referenz erwartet? DragEventArgs erbt von EventArgs, genau wie BallEventArgs. Das DragDrop-Event des Formulars nutzt EventHandler nicht. Es nutzt etwas anderes, DragEventArgs, und wenn Sie es mit Ihrem Event-Handler verarbeiten wollen, muss er eine object- und eine DragEventArgs-Referenz akzeptieren.

Die Parameter des Events werden von etwas definiert, das als *Delegate* bezeichnet wird – EventHandler ist ein Beispiel für Delegates. Aber darüber werden wir in Kürze ausführlicher reden.

F: Dann kann ich meine Event-Handler wohl auch etwas anderes zurückliefern lassen als void, oder?

A: Ja, können Sie. Aber meist ist das kein guter Gedanke. Liefert Ihr Event-Handler nicht void zurück, können Sie Event-Handler nicht *verketten*. Das bedeutet, dass Sie mit einem Event nicht mehrere Handler verbinden können. Da das Verketten eine praktische Funktionalität ist, sollte man seine Event-Handler immer void liefern lassen.

F: Verketten? Was ist das?

A: Darüber können mehrere Objekte das gleiche Event abonnieren – sie ketten ihre Event-Handler nacheinander an das Event. Auch darüber werden wir gleich noch sprechen.

F: Habe ich deswegen += verwendet, als ich meinen Event-Handler hinzugefügt habe? So als würde ich den vorhandenen Handlern einen neuen hinzufügen?

A: Genau! Jedes Mal, wenn Sie einen Event-Handler hinzufügen, sollten Sie `+=` verwenden. So sichern Sie, dass Ihr Handler die vorhandenen Handler nicht ersetzt. Er wird einfach zu einem Element in einer möglicherweise sehr langen Kette von Event-Handlern, die alle auf das gleiche Event lauschen.

F: Warum wird »this« verwendet, wenn das BallInSpiel()-Event ausgelöst wird?

A: Weil das der erste Parameter des Standard-Event-Handlers ist. Haben Sie gesehen, dass jeder Click-Event-Handler den Parameter »object sender« hat? Dieser Parameter ist eine **Referenz auf das Objekt, das das Event absetzt**. Behandeln Sie einen Button-Klick, zeigt `sender` auf den Button, der angeklickt wurde. Behandeln Sie ein BallInSpiel-Event, zeigt `sender` auf das Ball-Objekt, das im Spiel ist – und der Ball setzt den Parameter auf `this`, wenn er das Event absetzt.

EIN Event wird immer von EINEM Objekt abgesetzt.

Auf EIN Event können aber beliebig VIELE Objekte reagieren.

Die IDE erzeugt Event-Handler automatisch

Die meisten Programmierer folgen bei der Benennung ihrer Event-Handler den gleichen Konventionen. Gibt es ein `Ball`-Objekt, das ein `BallInSpiel`-Event hat, und hat die Referenz auf das Objekt den Namen `ball`, hat der Event-Handler üblicherweise den Namen `ball_BallInSpiel()`. Das ist keine festgeschriebene Regel, macht Code für andere Programmierer aber besser lesbar.

Glücklicherweise macht es die IDE Ihnen leicht, Ihren Event-Handlern ordentliche Namen zu geben. Sie hat eine Funktion, die **automatisch Event-Handler-Methoden hinzufügt**, wenn Sie mit einer Klasse arbeiten, die ein Event absetzt. Es sollte nicht zu überraschend sein, dass Ihnen die IDE diese Arbeit abnehmen kann – schließlich ist das genau das, was sie macht, wenn Sie in Ihrem Formular doppelt auf einen Button klicken.



1 **Fügen Sie einer neuen Windows-Anwendung Ball und BallEventArgs hinzu.**
Hier ist die Klasse Ball:

```
public class Ball {
    public event EventHandler BallInSpiel;
    public void OnBallInSpiel(BallEventArgs e) {
        if (BallInSpiel != null)
            BallInSpiel(this, e);
    }
}
```

Und hier ist die Klasse BallEventArgs:

```
public class BallEventArgs : EventArgs {
    public int Flugbahn { get; private set; }
    public int Entfernung { get; private set; }
    public BallEventArgs(int Flugbahn, int Entfernung) {
        this.Flugbahn = Flugbahn;
        this.Entfernung = Entfernung;
    }
}
```

2 **Beginnen Sie, den Konstruktor für Werfer hinzuzufügen.**
Fügen Sie Ihrem Projekt die Klasse Werfer hinzu. Geben Sie ihr einen Konstruktor, der eine `Ball`-Referenz über einen Parameter namens `ball` annimmt. Der Konstruktor enthält nur eine einzige Zeile Code, die den Event-Handler der Klasse für `ball.BallInSpiel` registriert. Beginnen Sie, die Anweisung einzugeben, aber **tippen Sie das += noch nicht**.

```
public Werfer(Ball ball) {
    ball.BallInSpiel
}
```

3 Geben Sie += ein, beendet die IDE die Anweisung für Sie.

Sobald Sie in die Anweisung += eintippen, blendet die IDE einen nützlichen kleinen Kasten ein:

```
public Werfer(Ball ball) {
    ball.BallInSpiel +=
}                        new EventHandler(ball_BallInSpiel);   (Zum Einfügen TAB-Taste drücken)
```

Drücken Sie die Tabulatortaste, beendet die IDE die Anweisung für Sie. Sie sieht dann so aus:

```
public Werfer(Ball ball) {
    ball.BallInSpiel += new EventHandler(ball_BallInSpiel);
}
```

Klicken Sie im Form-Designer doppelt auf einen Button, macht die IDE genau das Gleiche – sie fügt automatisch einen Event-Handler hinzu – nur fügt sie den Code der `InitializeComponent()`-Methode von Form1.Designer.cs hinzu, statt ihn einfach ans Ende der Klassendatei anzuhängen.

4 Die IDE fügt auch Ihren Event-Handler hinzu.

Sie sind noch nicht fertig – Sie brauchen immer noch die Methode, die für das Event registriert wird. Netterweise nimmt die IDE Ihnen auch das ab.

```
new EventHandler(ball_BallInSpiel);
                 Zum Generieren des Handlers "ball_BallInSpiel" in dieser Klasse die TAB-Taste drücken
```

Drücken Sie erneut die Tabulatortaste, damit die IDE der Klasse `Werfer` diese Event-Handler-Methode hinzufügt. Die IDE folgt immer der `referenzName_HandlerName()`-Konvention:

```
void ball_BallInSpiel(object sender, EventArgs e) {
    throw new NotImplementedException();
}
```

Diese NotImplementedException fügt die IDE immer als Platzhalter ein, damit bei der Ausführung eine Exception ausgelöst wird, die Ihnen sagt, dass Sie etwas noch implementieren müssen, das die IDE automatisch eingefügt hat.

5 Beenden Sie den Event-Handler des Werfers.

Jetzt haben Sie das Gerüst des Event-Handlers und müssen nur noch den restlichen Code einbauen. Der Werfer sollte alle niedrigen Bälle fangen und andernfalls das erste Base bewachen.

```
void ball_BallInSpiel(object sender,EventArgs e) {
    if (e is BallEventArgs) {
        BallEventArgs ballEventArgs = e as BallEventArgs;
        if ((ballEventArgs.Entfernung < 30) && (ballEventArgs.Flugbahn < 60))
            BallFangen();
        else
            ErstesBaseBewachen();
    }
}
```

Diese Methoden werden Sie gleich hinzufügen.

Da BallEventArgs eine Unterklasse von EventArgs ist, downcasten wir mit as, damit wir die Eigenschaften verwenden können.



Es ist an der Zeit, das bisher Gelernte anzuwenden. Sie sollen die Klassen Ball und Werfer vervollständigen, eine Fan-Klasse hinzufügen und sicherstellen, dass sie alle in einer sehr einfachen Version Ihres Baseball-Simulators zusammenarbeiten.

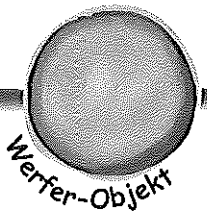
1 Vervollständigen Sie die Klasse Werfer.

Unten sehen Sie, was wir bisher für `Werfer` haben. Ergänzen Sie die Methoden `BallFangen()` und `ErstesBaseBewachen()`. Die eine sollte ausgeben, dass der Werfer den Ball gefangen hat, die andere, dass er zum ersten Base gelaufen ist.

```
public class Werfer {
    public Werfer(Ball ball) {
        ball.BallInSpiel += new EventHandler(ball_BallInSpiel);
    }

    void ball_BallInSpiel(object sender, EventArgs e) {
        if (e is BallEventArgs){
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if ((ballEventArgs.Entfernung < 30) && (ballEventArgs.Flugbahn < 60))
                BallFangen();
            else
                ErstesBaseBewachen();
        }
    }
}
```

Diese Methoden müssen Sie implementieren, um eine Ausgabezeile in die Konsole zu schreiben.



2 Schreiben Sie die Klasse Fan.

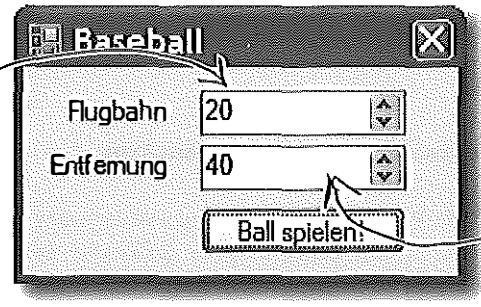
Erzeugen Sie eine andere Klasse namens Fan. Fan sollte in seinem Konstruktor ebenfalls das BallInSpiel-Event abonnieren. Der Event-Handler von Fan sollte prüfen, ob die Entfernung größer als 130 Meter und die Flugbahn größer als 30 ist (ein Homerun), nach einem Handschuh greifen und versuchen, den Ball dann zu fangen. Andernfalls sollte der Fan kreischen und schreien. Schreiben Sie in die Konsole, was der Fan macht.

Schauen Sie sich das Ausgabefenster auf der nächsten Seite an, um zu sehen, was genau ausgegeben werden soll.



8 **Erstellen Sie einen einfachen Simulator.**
Erstellen Sie ein neues Formular. Es sollte zwei NumericUpDown-Steuerelemente enthalten: eins für die Entfernung des Balls und eins für seine Flugbahn. Fügen Sie einen Button mit der Aufschrift »Ball spielen!« hinzu. Wird auf »Ball spielen!« geklickt, wird der Ball mit den Werten in den beiden NumericUpDown-Elementen geschlagen. Ihr Formular sollte so aussehen:

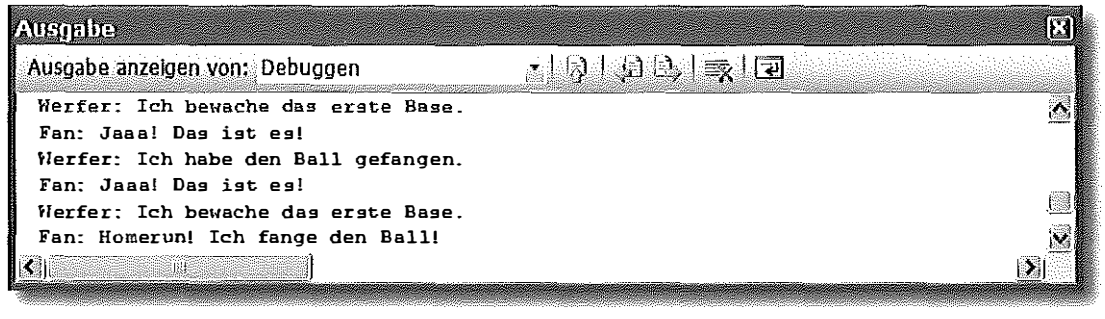
Der Wert für die Flugbahn kann zwischen 0 und 100 liegen. Setzen Sie also die Eigenschaften Minimum auf 0, Maximum auf 100 und Value auf 20.



Vergessen Sie nicht, die `Value`-Eigenschaften in `int`-Werte umzuwandeln, bevor Sie sie verwenden.

Die Entfernung kann zwischen 0 und 150 liegen. Ihr Standardwert ist 40.

4 **Erzeugen Sie die folgende Ausgabe.**
Versuchen Sie, Ihren Simulator dazu zu bringen, mit drei aufeinanderfolgenden Bällen diese Ausgabe zu erzeugen. Schreiben Sie unten die Werte auf, die Sie dazu verwendet haben.



Ball 1:
Flugbahn: .....
Entfernung:.....

Ball 2:
Flugbahn: .....
Entfernung:.....

Ball 3:
Flugbahn: .....
Entfernung:.....



Lösungen zu den Übungen

Es ist an der Zeit, das bisher Gelernte anzuwenden. Sie sollen die Klassen Ball und Werfer vervollständigen, eine Fan-Klasse hinzufügen und sicherstellen, dass sie alle in einer sehr einfachen Version Ihres Baseball-Simulators zusammenarbeiten.

```
public class Ball
{
    public event EventHandler BallInSpiel;
    public void OnBallInSpiel(BallEventArgs e)
    {
        if (BallInSpiel != null)
            BallInSpiel(this, e);
    }
}
```

Die Methode OnBallInSpiel() setzt nur das BallInSpiel-Event ab – muss aber prüfen, ob das Event nicht null ist, da ansonsten eine Exception ausgelöst wird.

Schreibgeschützte automatische Eigenschaften funktionieren in EventArgs sehr gut, da die Event-Handler die ihnen übergebenen Daten nur lesen.

```
public class BallEventArgs : EventArgs {
{
    public int Flugbahn { get; private set; }
    public int Entfernung { get; private set; }
    public BallEventArgs(int Flugbahn, int Entfernung)
    {
        this.Flugbahn = Flugbahn;
        this.Entfernung = Entfernung;
    }
}
```

Der Fan-Konstruktor hängt seinen Event-Handler an das BallInSpiel-Event an.

```
public class Fan {
    public Fan(Ball ball)
    {
        ball.BallInSpiel += new EventHandler(ball_BallInSpiel);
    }

    void ball_BallInSpiel(object sender, EventArgs e)
    {
        if (e is BallEventArgs) {
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if (ballEventArgs.Entfernung > 130 && ballEventArgs.Flugbahn > 30)
                Console.WriteLine("Fan: Homerun! Ich fange den Ball!");
            else
                Console.WriteLine("Fan: Jaaa! Das ist es!");
        }
    }
}
```

Der BallInSpiel-Handler von Fan sucht nach einem Ball, der lang und hoch genug ist.

```
public class Werfer {
    public Werfer(Ball ball) {
        ball.BallInSpiel += new EventHandler(ball_BallInSpiel);
    }
    void ball_BallInSpiel(object sender, EventArgs e) {
        if (e is BallEventArgs) {
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if ((ballEventArgs.Entfernung < 30) && (ballEventArgs.Flugbahn < 60))
                BallFangen();
            else
                ErstesBaseBewachen();
        }
    }
    private void BallFangen() {
        Console.WriteLine("Werfer: Ich habe den Ball gefangen.");
    }
    private void ErstesBaseBewachen() {
        Console.WriteLine("Werfer: Ich bewache das erste Base.");
    }
}
```

Den BallInSpiel-Handler von Werfer haben Sie bereits. Er achtet auf tiefe Bälle.

```
public partial class Form1 : Form {
    Ball ball = new Ball();
    Werfer werfer;
    Fan fan;

    public Form1() {
        InitializeComponent();
        werfer = new Werfer(ball);
        fan = new Fan(ball);
    }

    private void ballSpielen_Click(object sender, EventArgs e) {
        BallEventArgs ballEventArgs = new BallEventArgs(
            (int)flugbahn.Value, (int)entfernung.Value);
        ball.OnBallInSpiel(ballEventArgs);
    }
}
```

Das Formular braucht einen Ball, einen Fan und einen Werfer. Fan und Werfer verknüpft es im Konstruktor mit dem Ball.

Wird auf den Button geklickt, sagt das Formular dem Werfer, dass er den Ball zum Schläger werfen soll. Dieser sagt dem Ball, dass er sein BallInSpiel absetzen soll, das seinerseits die Event-Handler in den Werfer- und Fan-Objekten anstößt.

Hier sind die Werte, die wir verwendet haben, um die Ausgabe zu erhalten. Ihre könnten etwas anders aussehen.

Ball 1:	**Ball 2:**	**Ball 3:**
Flugbahn: 75	Flugbahn: 48	Flugbahn: 40
Entfernung: 33	Entfernung: 28	Entfernung: 150

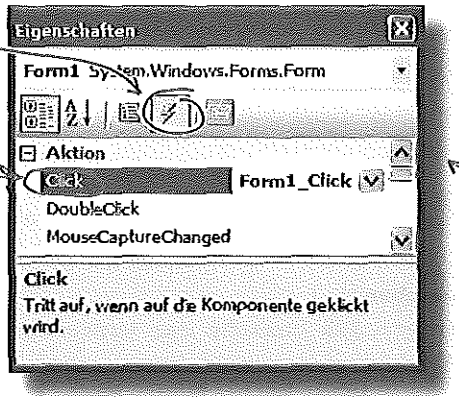
Alle Formulare, die Sie bisher erstellt haben, nutzen Events

Jedes Mal, wenn Sie einen Button erstellen, im Designer darauf doppelklicken und Code für eine Methode wie `button1_Click()` schreiben, arbeiten Sie mit Events.

Tun Sie das!

1. Erstellen Sie eine neue Windows-Anwendung. Gehen Sie zum Eigenschaften-Fenster für das Formular. Oben im Fenster sind Symbole – klicken Sie auf das mit dem Blitz. Damit öffnen Sie **die Events-Seite des Eigenschaften-Fensters**:

Die Events für ein Steuerelement können Sie sehen, indem Sie auf den Events-Button des Eigenschaften-Fensters klicken.



Einen Event-Handler, der angestoßen wird, wenn jemand auf das Formular klickt, können Sie registrieren, indem Sie auf der Events-Seite neben Click Form1_Click auswählen.

Scrollen Sie zu Click und klicken Sie doppelt auf das Wort »Click«. Die IDE fügt dem Formular dann einen neuen Event-Handler hinzu, der angestoßen wird, wenn darauf geklickt wird. Außerdem fügt sie Form1.Designer.cs eine Zeile hinzu, die den Event-Handler für das Event registriert.

2. Doppelklicken Sie in der Events-Seite auf »Click«. Die IDE fügt Ihrem Formular dann automatisch eine Event-Handler-Methode namens Form1_Click hinzu. Ergänzen Sie diese um folgende Codezeile:

```
private void Form1_Click(object sender, EventArgs e) {
    MessageBox.Show("Sie haben gerade auf das Formular geklickt");
}
```

3. Visual Studio hat für Sie aber nicht nur die kleine Methodendeklaration geschrieben. Es hat für Sie auch den Event-Handler beim Click-Event des Formulars registriert. Öffnen Sie Form1.Designer.cs und nutzen Sie die Schnellsuche-Funktion der IDE, um im aktuellen Projekt den Text `Form1_Click` zu suchen. Sie werden diese Codezeile finden:

```
this.Click += new System.EventHandler(this.Form1_Click);
```

Führen Sie das Programm jetzt aus und testen Sie, ob der Code funktioniert!



Event-Handler müssen immer »registriert« werden.

Ziehen Sie einen Button auf ein Formular und fügen diesem eine Methode namens `button1_Click()` *hinzu, die die erforderlichen Parametern aufweist,* ***aber nicht beim Button registriert ist****, wird sie auch nicht aufgerufen. Klicken Sie im Designer doppelt auf den Button, sieht die IDE, dass der Standard-Event-Handler-Name vergeben ist, und fügt für den Button einen Event-Handler namens* `button1_Click_1()` *hinzu.*

Ein Event, viele Handler

Hier ist eine richtig nützliche Sache, die Sie mit Events machen können: Sie können sie so **verketten**, dass ein Event oder Delegate nacheinander mehrere Methoden aufruft. Fügen wir der Anwendung in paar Buttons hinzu, um zu sehen, wie das funktioniert.

4 Ergänzen Sie Ihr Formular um diese Methoden:

```
private void EtwasSagen(object sender, EventArgs e) {
    MessageBox.Show("Etwas");
}
private void EtwasAnderesSagen(object sender, EventArgs e) {
    MessageBox.Show("Etwas anderes");
}
```

5 Fügen Sie Ihrem Formular jetzt zwei Buttons hinzu. Klicken Sie doppelt auf beide Buttons, um ihre Event-Handler hinzuzufügen. Hier ist der Code für beide Event-Handler:

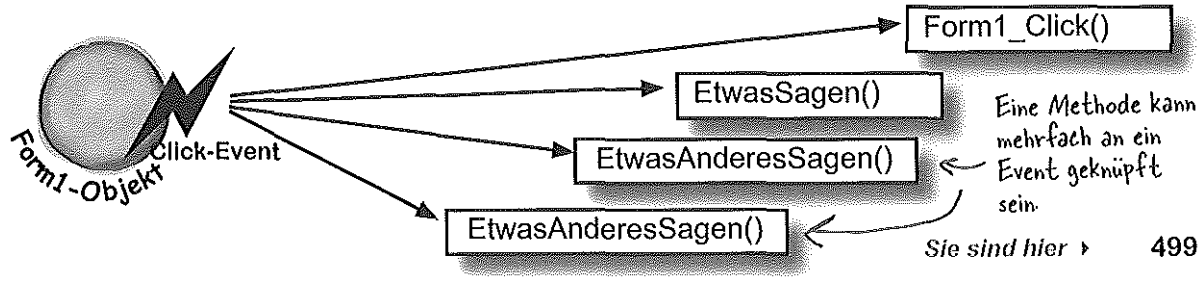
```
private void button1_Click(object sender, EventArgs e) {
    this.Click +=new EventHandler(EtwasSagen);
}
private void button2_Click(object sender, EventArgs e) {
    this.Click +=new EventHandler(EtwasAnderesSagen);
}
```

6 Führen Sie Ihr Programm jetzt aus und tun Sie Folgendes:

- **Klicken Sie auf das Formular** – es erscheint ein Dialog mit dem Text »Sie haben gerade auf das Formular geklickt«.
- **Klicken Sie jetzt auf button1** und dann **wieder auf das Formular**. Es erscheinen zwei Dialoge: erst »Sie haben gerade auf das Formular geklickt« und dann »Etwas«.
- **Klicken Sie zweimal auf button2** und dann **wieder auf das Formular**. Es erscheinen vier Dialoge: »Sie haben gerade auf das Formular geklickt«, »Etwas«, »Etwas anderes« und »Etwas anderes«.

Was ist hier passiert?

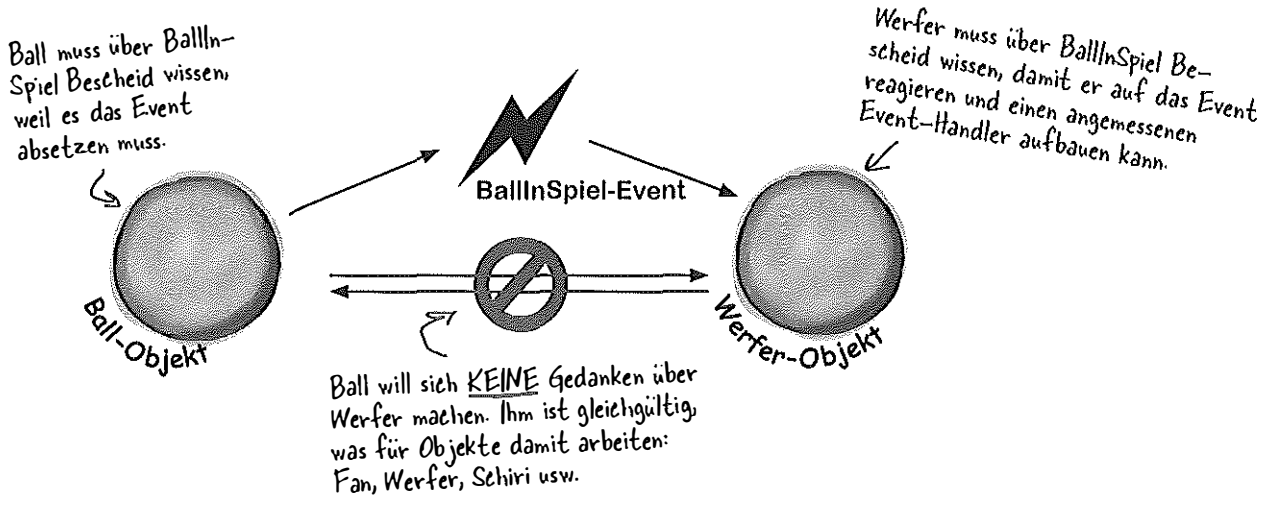
Jedes Mal, wenn Sie auf einen der Buttons geklickt haben, haben Sie eine weitere Methode an das Click-Event des Formulars gekettet – entweder EtwasSagen() oder EtwasAnderesSagen(). Immer wenn Sie auf einen der Buttons klicken, **ketten Sie die gleiche Methode ein weiteres Mal** an das Ereignis. Das Event interessiert nicht, wie viele Methoden an es gekettet sind oder ob die gleiche Methode in seiner Kette mehrfach vorhanden ist. Jedes Mal, wenn das Event abgesetzt wird, ruft es diese einfach nacheinander in der Reihenfolge auf, in der sie angeknüpft wurden.



Event-Sender mit Event-Empfängern verbinden

Eine der schwierigsten Dinge bei Events ist, dass der **Sender** des Events wissen muss, was für eine Art von Event er senden soll – einschließlich der Argumente, die er dem Event übergeben muss. Und der **Empfänger** des Events muss den Rückgabetyp und die Argumente kennen, die seine Event-Handler-Methoden haben müssen.

Aber – und das ist der schwierige Teil – Sie können Sender und Empfänger nicht *zusammenknüpfen*. Sie möchten, dass der Sender das Event sendet und *sich keine Gedanken über die Empfänger macht*. Und der Empfänger soll sich nur für das Event interessieren, *nicht für das Objekt, das es abgesetzt hat*. Sender und Empfänger konzentrieren sich also beide auf das Event und nicht aufeinander.



»Wir werden uns mit Ihnen in Verbindung setzen.«

Sie wissen, was dieser Code macht:

```
Ball aktuellerBall;
```

Er erzeugt eine **Referenzvariable**, die auf ein beliebiges Ball-Objekt zeigen kann. Sie ist nicht an eine bestimmte Ball-Instanz gebunden. Stattdessen kann sie auf jedes Ball-Objekt zeigen – oder null sein und auf überhaupt nichts zeigen.

Ein Event benötigt eine vergleichbare Art von Referenz – aber statt einer, die auf ein Objekt zeigt, benötigt es eine, die **auf eine Methode zeigt**. Jedes Event muss eine Liste der Methoden festhalten, die für es registriert sind. Sie haben bereits gesehen, dass sich diese in anderen Klassen befinden und sogar privat sein können. Wie also hält es all diese Event-Handler-Methoden nach, die es aufrufen muss?

Ein Delegate VERTRITT die eigentliche Methode

Erstellen Sie ein Delegate, müssen Sie nur die Signatur der Methoden angeben, auf die es zeigen kann.

Dieses Delegate kann also verwendet werden, um Methoden zu referenzieren, die ein object und ein EventArgs erwarten und keinen Rückgabewert haben.

Einer der nützlichsten Aspekte von Events ist, dass es, wenn es abgesetzt wird, **keine Ahnung hat**, wessen Event-Handler-Methoden es aufruft. Jeder, der ein Event abonniert, wird aufgerufen. Wie also verwaltet das Event das?

Es nutzt einen C#-Typ namens **Delegate** (oder auch Delegat). Mit einem Delegate können Sie eine Referenzvariable erstellen, die nicht auf eine Klasseninstanz, sondern auf eine **Methode in einer Klasse verweist.**

Sie haben in diesem Kapitel bereits Delegates verwendet! Als Sie das BallInSpiel-Event verwendeten, haben Sie `EventHandler` verwendet. Und `EventHandler` ist eben einfach ein Delegate. Klicken Sie in der IDE mit der rechten Maustaste auf `EventHandler` und wählen »Zu Definition gehen«, würden Sie Folgendes sehen (probieren Sie es selbst):

```
public delegate void EventHandler(object sender, EventArgs e);
```

Das gibt den Rückgabewert der Delegate-Signatur an – er bedeutet, dass ein EventHandler nur auf Methoden zeigen kann, deren Rückgabewert void ist.

Der Name dieses Delegates ist EventHandler.

Tun Sie das!

Ein Delegate fügt Ihrem Projekt einen neuen Typ hinzu

Wenn Sie Ihrem Projekt ein Delegate hinzufügen, erzeugen Sie einen **Delegate-Typ**. Wenn Sie diesen für ein Feld oder eine Variable nutzen, erstellen Sie eine **Instanz** dieses Delegate-Typs. **Erstellen Sie also ein neues Projekt.** Fügen Sie ihm dann eine neue Klassendatei namens LiefertEinenString.cs hinzu. Aber statt in diese eine Klasse einzugeben, stecken Sie in sie nur eine einzige Zeile:

```
public delegate string LiefertEinenString();
```

LiefertEinenString ist ein Delegate-Typ, den Sie Ihrem Projekt hinzugefügt haben. Jetzt können Sie ihn verwenden, um Variablen und Felder zu deklarieren.

Gehen Sie zum Formularcode und fügen Sie dem Formular dieses Feld hinzu:

```
LiefertEinenString eineMethode;
```

eineMethode ist eine Referenz des Delegate-Typs LiefertEinenString.

Erstellen Sie jetzt Ihr Programm – es lässt sich kompilieren! (Sie erhalten eine Warnung, weil ein Feld deklariert, aber nicht verwendet wird – aber das ist in Ordnung.) Als Sie dem Programm das neue Delegate hinzufügten, erstellte es einen neuen Typ namens `LiefertEinenString`. Nutzen Sie diesen Typ, um eine Variable zu deklarieren, können Sie diese Variable auf jede Methode setzen, die keine Parameter erwartet und einen String liefert. Probieren Sie es – fügen Sie Ihrem Code diese Methode hinzu:

```
private string Hallo() {
    return "Hallo!";
}
```

Die Signatur dieser Methode entspricht LiefertEinenString.

Ergänzen Sie einen Button, der diese drei Zeilen hat.
Klicken Sie darauf und schauen Sie, was passiert:

```
eineMethode = new LiefertEinenString(Hallo);
string meldung = eineMethode();
MessageBox.Show(meldung);
```

eineMethode können Sie setzen wie jede andere Variable. Aber rufen Sie sie auf wie eine gewöhnliche Methode, ruft sie die Methode, auf die sie zeigt.

De-le-gat, Nomen.
Eine Person, die ausgesendet wurde, um eine andere zu vertreten.
Die Kanzlerin schickte eine ***Delegation*** *zum Gipfel.*

Delegates in Aktion

Delegates haben nichts Geheimnisvolles – und für sie ist nicht einmal viel Code erforderlich. Nutzen wir sie, um einem Restaurantbesitzer zu helfen, die geheimen Zutaten seines Küchenchefs zu sortieren.



1 **Fügen Sie einem neuen Windows-Projekt ein Delegate hinzu.**
Delegates stehen üblicherweise außerhalb anderer Klassen. Fügen Sie Ihrem Projekt also eine neue Klassendatei hinzu, die Sie GeheimZutatHinzufügen.cs nennen. Sie enthält nur eine Codezeile:

```
public delegate string GeheimZutatAbrufen(int menge);
```

Mit diesem Delegate kann ein Feld erstellt werden, das auf eine beliebige Methode zeigen kann, die einen int-Parameter erwartet und einen String zurückliefert.

2 **Fügen Sie eine Klasse für die Küchenchefin Susanne hinzu.**
Susanne.cs enthält eine Klasse, die die Geheimzutaten der Küchenchefin Susanne festhält. Diese hat eine private Methode namens SusannesGeheimZutat() mit einer Signatur, die GeheimZutatAbrufen entspricht. Außerdem hat sie eine schreibgeschützte Eigenschaft, deren Typ Sie sich genau ansehen sollten. Sie liefert ein GeheimZutatAbrufen. Andere Objekte können diese Eigenschaft also nutzen, um eine Referenz auf Susannes SusannesGeheimZutat()-Methode abzurufen.

Susannes Geheimzutat-Methode erwartet einen int und liefert einen String, der ihre Geheimzutat beschreibt.

```
public class Susanne {
  public GeheimZutatAbrufen GeheimZutatMethode {
    get {
      return new GeheimZutatAbrufen(SusannesGeheimZutat);
    }
  }
  private string SusannesGeheimZutat(int menge) {
    return menge.ToString() + " Gramm Nelken";
  }
}
```

3 **Fügen Sie dann eine Klasse für die Souschefin Anne hinzu.**
Annes Methode funktioniert ähnlich wie die von Susanne:

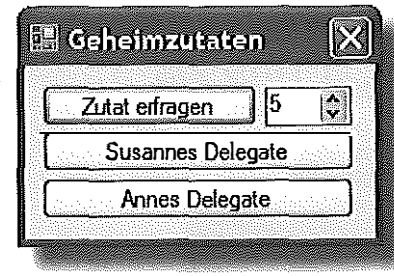
Annes GeheimZutatAbrufen-Eigenschaft liefert eine Instanz des Delegates GeheimZutatAbrufen, das auf ihre Geheimzutaten-Methode zeigt.

Annes Geheimzutat-Methode erwartet ebenfalls einen int und liefert einen String, der aber anders aussieht als der von Susanne.

```
public class Anne {
    public GeheimZutatAbrufen AnnesGeheimZutatMethode {
        get {
            return new GeheimZutatAbrufen(AnnesGeheimZutat);
        }
    }
    private string AnnesGeheimZutat(int menge) {
        if (menge < 10)
            return menge.ToString()
                        + " Dosen Sardinen -- das reicht nicht!";
        else
            return menge.ToString() + " Dosen Sardinen";
    }
}
```

4 Erstellen Sie das Formular und ein Delegate.

Erstellen Sie dieses Formular:



Hier ist der Code für das Formular:

```
GeheimZutatAbrufen zutatenMethode = null;
Susanne susanne = new Susanne();
Anne anne = new Anne();

private void zutatNutzen_Click(object sender, EventArgs e) {
    if (zutatenMethode != null)
        Console.WriteLine("Ich füge " + zutatenMethode((int)menge.Value) + " hinzu");
    else
        Console.WriteLine("Ich habe keine Geheimzutat!");
}

private void susanneFragen_Click(object sender, EventArgs e) {
    zutatenMethode = new GeheimZutatAbrufen(susanne.GeheimZutatMethode);
}

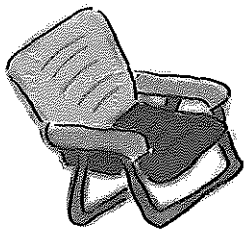
private void anneFragen_Click(object sender, EventArgs e) {
    zutatenMethode = new GeheimZutatAbrufen(anne.AnnesGeheimZutatMethode);
}
```

5 Schauen Sie sich mit dem Debugger an, wie Delegates funktionieren.

Sie haben ein wunderbares Werkzeug – den Debugger der IDE –, das Ihnen gut helfen kann, Delegates in den Griff zu bekommen. Führen Sie die folgenden Schritte durch:

- ★ Starten Sie Ihr Programm. Klicken Sie erst auf den Button »Zutat erfragen«; auf der Konsole sollte die Zeile »Ich habe keine Geheimzutat« erscheinen.
- ★ Klicken Sie auf den Button »Susannes Delegate« – das nimmt das zutatenMethode-Feld des Formulars (das ein GeheimZutatAbrufen-Delegate ist) und setzt es auf das, was Susannes GeheimZutatAbrufen-Eigenschaft liefert. Diese Eigenschaft liefert eine Instanz des Typs GeheimZutatAbrufen, die auf SusannesGeheimZutat()-Methode zeigt.
- ★ Klicken Sie erneut auf den Button »Zutat erfragen«. Jetzt zeigt das zutatenMethode-Feld des Formulars auf SusannesGeheimZutat(), ruft sie auf, übergibt ihr den Wert im NumericUpDown und schreibt die Ausgabe in die Konsole.
- ★ Klicken Sie auf den Button »Annes Delegate«. Er nutzt die Eigenschaft Anne.GeheimZutatAbrufen, um das zutatenMethode-Feld des Formulars auf die AnnesGeheimZutat()-Methode zeigen zu lassen.
- ★ Klicken Sie noch einmal auf »Zutat erfragen«. Jetzt wird Annes Methode aufgerufen.
- ★ **Nutzen Sie an dieser Stelle den Debugger**, um genau zu beobachten, was passiert. Setzen Sie einen Haltepunkt auf die erste Zeile der drei Methoden im Formular. **Starten Sie dann das Programm neu** (zutatenMethode wird dadurch wieder auf null gesetzt) und beginnen Sie erneut mit den fünf vorangehenden Schritten. Nutzen Sie die Einzelschritte (F11), um die einzelnen Codezeilen durchzugehen. Beobachten Sie, was passiert, wenn Sie auf »Zutat erfragen« klicken. Die Ausführung springt gleich in die Klassen Susanne und Anne, je nachdem, auf welche Methode das Feld zutatenMethode zeigt.

Pool-Puzzle



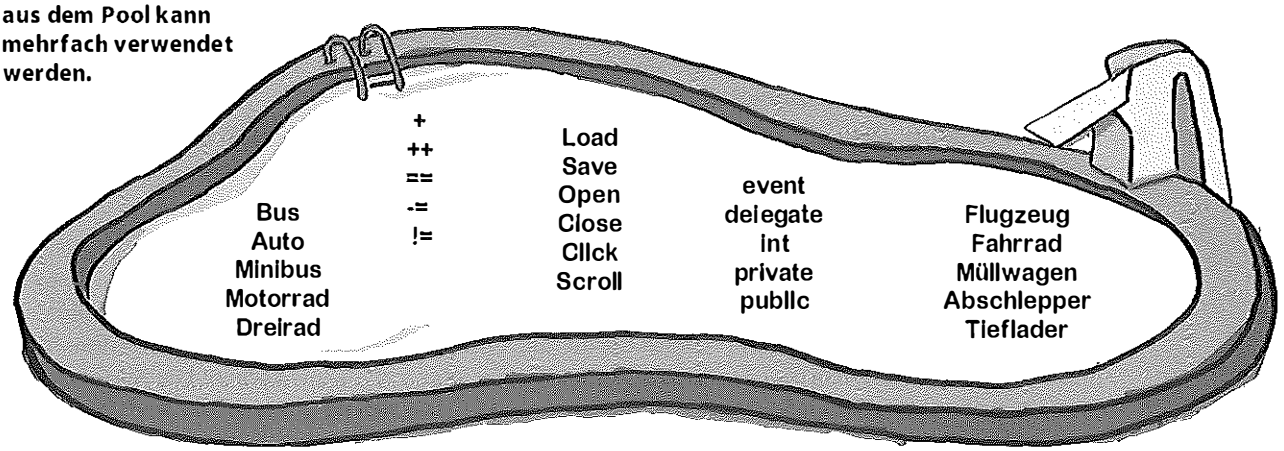
Sie haben die ***Aufgabe***, die leeren Zeilen im Code mit den Codeschnipseln aus dem Pool zu füllen. Einzelne Schnipsel **können** mehrfach verwendet werden, und Sie werden nicht alle Schnipsel benötigen. Das ***Ziel*** ist es, den Code für ein Formular zu vervollständigen, das folgende Ausgabe in die Konsole schreibt, wenn auf den Button `button1` geklickt wird.

Ausgabe

```
Maxe wird dich holen!
```

```
public Form1() {
    InitializeComponent();
    this._____ += new EventHandler(Minibus);
    this._____ += new EventHandler(_____);
}
void Abschlepper(object sender, EventArgs e) {
    Console.Write("wird dich ");
}
void Motorrad(object sender, EventArgs e) {
    button1._____ += new EventHandler(_____);
}
void Fahrrad(object sender, EventArgs e) {
    Console.WriteLine("holen!");
}
void _____(object sender, EventArgs e) {
    button1._____ += new EventHandler(Müllwagen);
    button1._____ += new EventHandler(_____);
}
void _____(object sender, EventArgs e) {
    Console.Write("Maxe ");
}
```

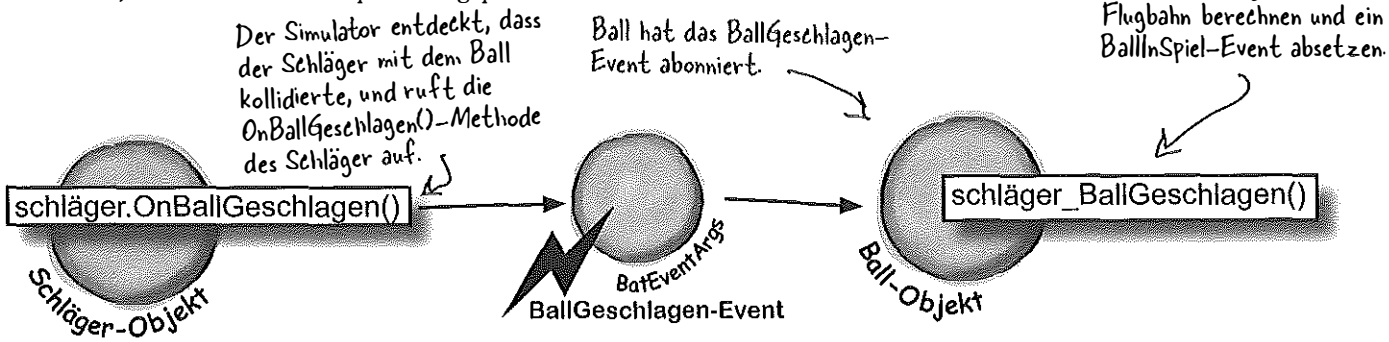
Hinweis: Jedes Teil aus dem Pool kann mehrfach verwendet werden.



Jedes Objekt kann ein öffentliches Event abonnieren ...

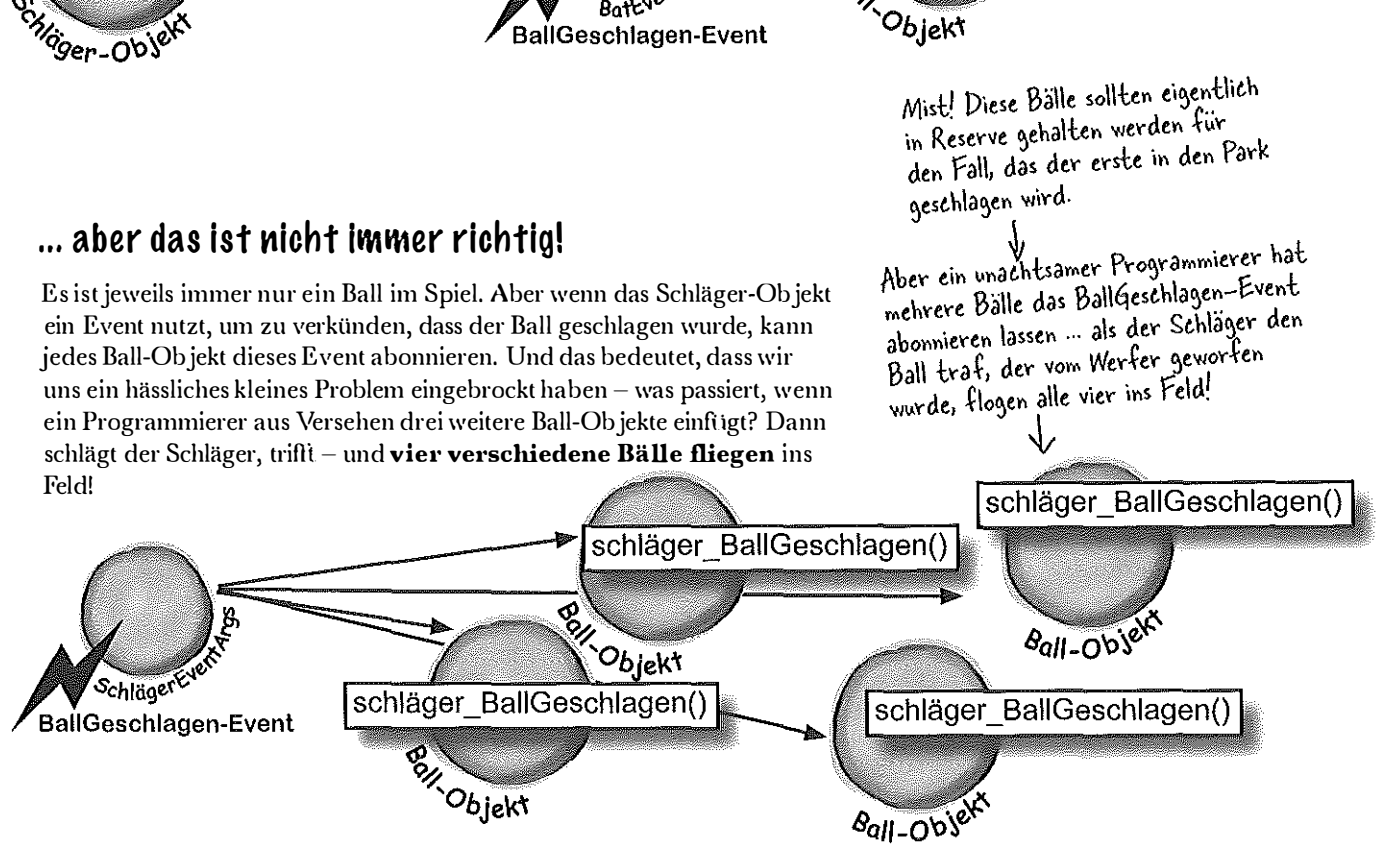
Angenommen, wir fügen unserem Simulator eine neue Klasse hinzu – die Klasse `Schläger`, die dem Gebilde das Event `BallGeschlagen` hinzufügt. Das funktioniert so: Wenn der Simulator entdeckt, dass der Spieler den Ball geschlagen hat, ruft er die `OnBallGeschlagen()`-Methode von `Schläger` auf, die das `BallGeschlagen`-Event absetzt.

Jetzt können wir der Klasse `Ball` eine `schläger_BallGeschlagen`-Methode hinzufügen, die das `BallGeschlagen`-Event von `Schläger` abonniert. Wird der Ball geschlagen, ruft ihr eigener Event-Handler die Methode `OnBallInSpiel()` auf, um ihr eigenes Event, `BallInSpiel`, abzusetzen und die Kettenreaktion in Gang zu bringen. Feldspieler fangen, Fans kreischen, Schiris rufen ... das Spiel ist angepfiffen.

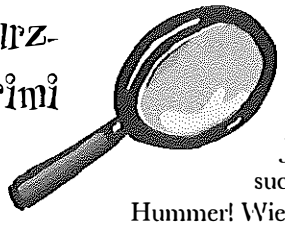


... aber das ist nicht immer richtig!

Es ist jeweils immer nur ein Ball im Spiel. Aber wenn das Schläger-Objekt ein Event nutzt, um zu verkünden, dass der Ball geschlagen wurde, kann jedes Ball-Objekt dieses Event abonnieren. Und das bedeutet, dass wir uns ein hässliches kleines Problem eingebrockt haben – was passiert, wenn ein Programmierer aus Versehen drei weitere Ball-Objekte einfügt? Dann schlägt der Schläger, trifft – und **vier verschiedene Bälle fliegen** ins Feld!



Kurz-Krimi



Der Fall des Goldenen Hummers

Heiner »Leichtfuß« Heimweh ist SchatzSucher. Er war einem der begehrtesten Schätze unter den so seltenen und ungewöhnlichen Wassergetier-Juwelen auf der Spur: einem Jade-gefassten, leuchtenden Goldhummer. Aber was war mit den ganzen anderen Schatzsuchern? In ihren Konstruktoren erhielten auch sie doch einen Verweis auf eben diesen Hummer! Wie konnte Heiner den Schatz als ***Erster*** aufspüren?

In einem gestohlenen Klassendiagramm entdeckte Heiner, dass die Klasse GoldHummer ein InDeckung-Gehen-Event absetzt, wenn sich ihr jemand nähert. Dieses enthielt sogar eine NeuerOrtArgs-Instanz mit Informationen zu dem Ort, zu dem sich der Hummer bewegt. Dieses Event kannten die anderen Schatzsucher nicht, und Heiner sah sich schon nach dem Hummer greifen.

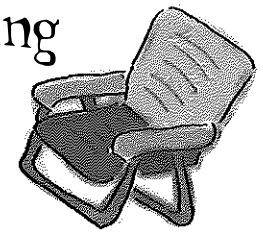
Heiner fügte seinem Konstruktor Code hinzu, der auf seiner GoldHummer-Referenz die Methode schatz_inDeckungGehen() als Event-Handler für das InDeckungGehen-Event registrierte. Dann schickte er einen billigen Schnüffler aus, sich an den Hummer ranzumachen, und wartete, dass der Hummer flüchtete, sich verbarg und das InDeckungGehen-Event absetzte – und seiner schatz_InDeckungGehen509()-Methode damit alle erforderlichen Informationen lieferte.

Zunächst verlief alles nach Plan. Heiner erhielt den neuen Ort und eilte los, um sich den Hummer zu schnappen. Doch angekommen, musste er erstaunt mit ansehen, wie sich drei andere SchatzSucher um den Hummer stritten.

Wie konnten die anderen Schatzsucher Heiner ausstechen?

→ Antworten auf Seite 511.

Pool-Puzzle, Lösung



Der Konstruktor hängt die beiden Event-Handler an das Load-Event. Sie werden angestoßen, sobald das Formular geladen ist.

Ein Klick auf den Button, ruft die drei registrierten Event-Handler auf.

Die zwei Load-Event-Handler registrieren drei separate Event-Handler für das Click-Event des Buttons.

```
public Form1() {
    InitializeComponent();
    this.Load += new EventHandler(Minibus);
    this.Load += new EventHandler(Motorrad);
}
void Abschlepper(object sender, EventArgs e) {
    Console.Write("wird dich ");
}
void Motorrad(object sender, EventArgs e) {

    button1.Click += new EventHandler(Fahrrad);
}
void Fahrrad(object sender, EventArgs e) {
    Console.WriteLine("holen!");
}
void Minibus(object sender, EventArgs e) {
    button1.Click += new EventHandler(Müllwagen);
    button1.Click += new EventHandler(Abschlepper);
}
void Müllwagen(object sender, EventArgs e) {
    Console.Write("Maxe ");
)
```

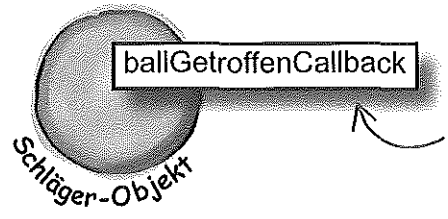
Nutzen Sie Callbacks statt Events, um genau ein Objekt mit einem Delegate zu verbinden

Unser Event-System funktioniert nur, wenn wir lediglich einen `Ball` und einen `Schläger` haben. Haben Sie mehrere `Ball`-Objekte, die alle das öffentliche Event `BallGeschlagen` abonnieren, fliegen alle los, wenn das Event abgesetzt wird. Aber das ist unsinnig ... eigentlich wird nur ein einziges `Ball`-Objekt getroffen. Wir müssen ermöglichen, dass sich der eine Ball, der geschlagen wird, beim Schläger registrieren kann, müssen das aber auf eine Weise tun, die es keinem anderen Ball erlaubt, sich zu registrieren. Das kann man mit einem **Callback** erreichen. Callbacks sind ein Mittel, Delegates so zu verwenden, das gesichert ist, dass das aufrufende Objekt tatsächlich nur die Methode anstößt, die angestoßen werden soll, weil nur eine einzige Methode beim Delegate registriert ist.

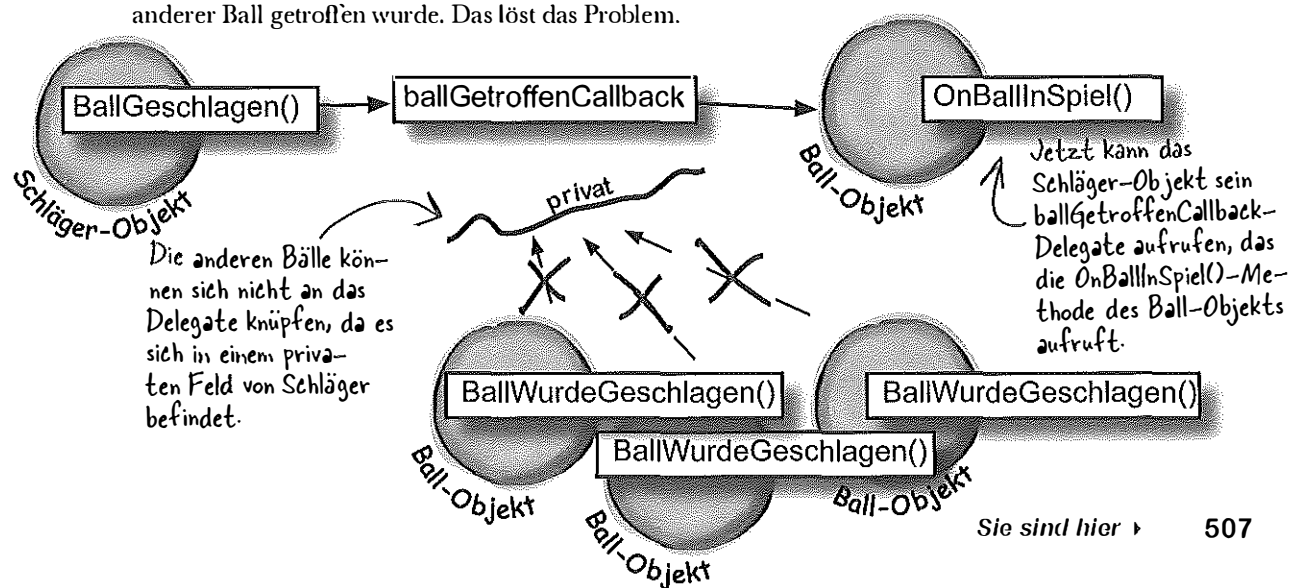
Unser Callback funktioniert so:

1. **Der Schläger hält sein Delegate-Feld privat.**
   Dass sich falsche Ball-Objekte an das Delegate von Schläger hängen, lässt sich am einfachsten verhindern, indem man es privat macht. So kann Schläger steuern, wer sich bei ihm registrieren kann.

2. **Der Konstruktor von Schläger übernimmt ein Delegate, das auf eine Ball-Methode zeigt.**
   Ist der Ball im Spiel, erzeugt er die neue Instanz Schläger und übergibt dem Schläger einen Zeiger auf seine OnBallInSpiel()-Methode. Diese bezeichnet man als **Callback-Method** (Rückruf-Methode), weil Schläger sie nutzt, um eine Methode auf dem Objekt aufzurufen, das es instantiierte.



3. **Wird der Ball getroffen, ruft er die Callback-Methode auf.**
   Aber da das Delegate von Schläger privat ist, kann Schläger 100%ig sicher sein, dass kein anderer Ball getroffen wurde. Das löst das Problem.



Callbacks nutzen Delegates, aber KEINE Events

Callbacks sind eine **andere Verwendung von Delegates**. Dazu brauchen sie keine neuen Schlüsselwörter oder Operatoren. Der Begriff beschreibt nur ein **Muster** – eine Verwendungsweise von Delegates mit Ihren Klasse, bei der ein Objekt einem anderen sagen kann: »Informiere mich, wenn das passiert – aber sag's keinem anderen!«



1 Fügen Sie dem Baseball-Projekt ein weiteres Delegate hinzu.
Da Schläger ein privates Delegate-Feld erhalten soll, das auf die OnBallInSpiel()-Methode von Ball zeigt, brauchen wir ein Delegate mit passender Signatur:

```
public delegate void SchlägerCallback(BallEventArgs e);
```

Das Delegate von Schläger zeigt auf die OnBallInSpiel()-Methode von Ball. Das Callback-Delegate muss also die gleiche Signatur haben wie OnBallInSpiel() – d.h., es muss einen BallEventArgs-Parameter übernehmen und darf nichts zurückliefern.

2 Fügen Sie dem Projekt die Klasse Schläger hinzu.
Die Klasse Schläger ist einfach. Schläger hat eine BallGeschlagen()-Methode, die der Simulator aufruft, wenn ein Ball geschlagen wird. Diese BallGeschlagen()-Methode nutzt das ballGetroffenCallback()-Delegate, um die OnBallInSpiel()-Methode von Ball aufzurufen (oder eine beliebige andere an den Konstruktor übergebene Methode).

```
public class Schläger {
    private SchlägerCallback ballGetroffenCallback;
    public Schläger(SchlägerCallback callbackDelegate) {
        this.ballGetroffenCallback = new SchlägerCallback(callbackDelegate);
    }
    public void BallGeschlagen(BallEventArgs e) {
        if (ballGetroffenCallback != null)
            ballGetroffenCallback(e);
    }
}
```

Prüfen Sie unbedingt jedes Delegate darauf, ob es null ist. Eine null-Referenz würde zu einer Exception führen.

Statt += haben wir = verwendet, weil wir nichts aneinanderhängen müssen. Dieses Delegate wird nur einmal gesetzt. Aber wenn Sie unbedingt += verwenden möchten, würde auch das funktionieren.

3 Wir müssen den Schläger mit einem Ball verbinden.
Wie erhält der Schläger-Konstruktor jetzt eine Referenz auf die OnBallInSpiel()-Methode eines bestimmten Balls? Das ist leicht – rufen Sie einfach die NeuerSchläger()-Methode auf, die Sie Ball hinzufügen müssen:

```
public Schläger NeuerSchläger()
{
    return new Schläger(new SchlägerCallback(OnBallInSpiel));
}
```

Wir setzen das Callback im Schläger-Konstruktor. Aber gelegentlich ist es sinnvoller, die Callback-Methode über eine öffentliche Methode oder den Setter einer Eigenschaft zu setzen.

Die Methode NeuerSchläger() erzeugt ein neues Schläger-Objekt und nutzt das SchlägerDelegate, um ihm eine Referenz auf OnBallInSpiel() zu übergeben. Das ist die Callback-Methode, die der Schläger verwenden wird, wenn er den Ball trifft.

④ Jetzt können wir die Klasse Ball besser kapseln.

Es ist ungewöhnlich, dass eine der `On...`-Methoden zum Auslösen eines Events `public` ist. Das können Sie selbst überprüfen – versuchen Sie, im Formular die OnClick()-Methode des ballSpielen-Buttons aufzurufen. Das wird Ihnen nicht gelingen, weil sie `protected` ist (damit Unterklassen sie überschreiben können). Folgen wir diesem Muster auch mit unserem Ball, indem wir seine OnBallInSpiel()-Methode `protected` machen:

```
protected void OnBallInSpiel(BallEventArgs e)
{
    if (BallInSpiel != null)
        BallInSpiel(this, e);
}
```

Das ist ein wichtiges Standard-Muster, auf das Sie bei der Arbeit mit .NET-Klassen immer wieder stoßen werden. Setzt eine .NET-Klasse ein Event ab, finden Sie fast immer eine geschützte Methode, die mit »On« beginnt.

⑤ Jetzt müssen wir nur noch das Formular einbinden.

Das Formular kann die OnBallInSpiel()-Methode von Ball nicht mehr aufrufen – genau das wollten wir. Deswegen haben wir die Methode Ball.NeuerSchläger() eingerichtet. Jetzt muss das Formular den Ball nach einem neuen Schläger fragen, damit der Ball geschlagen werden kann. Tut es das, stellt Ball sicher, dass die Methode OnBallInSpiel() mit dem Callback des Schlägers verbunden ist.

```
private void ballSpielen_Click(object sender, EventArgs e)
{
    Schläger schläger = ball.NeuerSchläger();
    BallEventArgs ballEventArgs = new BallEventArgs(
        (int)flugbahn.Value, (int)entfernung.Value);
    schläger.BallGeschlagen(ballEventArgs);
}
```

Möchte das Formular einen Ball schlagen, muss es sich vom Ball einen neuen Schläger liefern lassen. Der Ball sichert, dass das Callback mit dem Schläger verknüpft ist. Ruft das Formular die BallGeschlagen()-Methode von Schläger auf, ruft diese die OnBallInSpiel()-Methode von Ball auf, die das Event absetzt.

Führen Sie das Programm jetzt aus – es sollte wie zuvor funktionieren. Aber jetzt ist es vor Problemen **geschützt**, die daraus resultieren, dass mehrere Ball-Objekte auf das gleiche Event lauschen.

Vertrauen Sie uns nicht blind – testen Sie das im Debugger!

Punkt für Punkt

- Wenn Sie Ihrem Projekt ein Delegate hinzufügen, **erstellen Sie einen neuen Typ**, der Referenzen auf Methoden speichert.
- Events informieren über Delegates Objekte darüber, dass Aktionen eingetreten sind.
- Objekte abonnieren die Events anderer Objekte, wenn sie auf etwas reagieren müssen, das in diesen Objekten passiert.
- EventHandler ist ein Delegate-Typ, der bei der Arbeit mit Delegates verbreitet ist.
- An ein Event können Sie mehrere Event-Handler hängen. Deswegen nutzen Sie +=, um einem Event einen Handler zuzuweisen.
- Zur Vermeidung von NullReferenceExceptions sollten Sie vor der Verwendung von Delegates immer prüfen, ob sie null sind.
- Alle Steuerelemente in der Toolbox nutzen Events, um in Ihren Programmen Dinge geschehen zu lassen.
- Übergibt ein Objekt einem anderen eine Referenz auf eine Methode, damit es – und nur dieses – Informationen zurückliefern kann, bezeichnet man das als Callback.
- Beliebige Methoden können die Events Ihrer Objekte anonym abonnieren, während Ihre Objekte mit Callbacks mehr Steuerungsmöglichkeiten über die Delegates haben, die sie akzeptieren.
- Callbacks und Events nutzen beide Delegates, um Methoden in anderen Objekten zu referenzieren und aufzurufen.
- Der Debugger ist ein sehr nützliches Werkzeug, das Ihnen hilft, die Funktionsweise von Events, Delegates und Callbacks zu verstehen. Nutzen Sie ihn!

Es gibt keine Dummen Fragen

F: Worin unterscheiden sich Callbacks und Events?

A: Events sind ein Teil von C#. Sie bieten Objekten die Möglichkeit, andere Objekte darüber zu informieren, dass etwas Bestimmtes passiert ist. Veröffentlicht ein Objekt ein Event, kann dieses von einer beliebigen Anzahl von Objekten abonniert werden, ohne dass das veröffentlichende Objekt das weiß oder sich dafür interessiert. Setzt ein Objekt ein Event ab, ruft es alle Event-Handler auf, die für dieses registriert sind.

Callbacks werden nicht veröffentlicht. Stattdessen richtet ein Callback eine Beziehung zwischen zwei Klassen ein, die ein Objekt automatisch auf ein anderes reagieren lässt. Callbacks werden üblicherweise privat gehalten, und die Klasse, die das Callback speichert, steuert, wer darauf Zugriff hat. Ein Callback wird häufig im Konstruktor eines Objekts eingerichtet.

F: Callback ist also kein richtiger C#-Typ?

A: Nein, ist es nicht. Ein Callback ist ein ***Muster*** – es ist einfach ein neuer Weg, die Typen, Schlüsselwörter und Werkzeuge zu verwenden, die C# mitbringt. Sehen Sie sich noch einmal den Callback-Code an, den Sie gerade für Schläger und Ball geschrieben haben. Sehen Sie irgendwelche Schlüsselwörter, die Sie bisher noch nicht verwendet haben? Nein!

Es gibt eine Menge Muster, die Sie verwenden können. Und es gibt sogar einen Bereich der Programmierung, der sich mit den sogenannten ***Entwurfsmustern*** befasst. Dort finden Sie Lösungen zu vielen Problemen, über die Sie stolpern werden. Und für Probleme, die immer und immer wieder auftauchen, gibt es eigene Entwurfsmuster, die Sie nutzen können.

F: Callbacks sind also einfach private Events?

A: Nicht ganz. Es scheint einfach, sich das vorzustellen, aber eigentlich sind private Events etwas ganz anderes. Denken Sie daran, was der Zugriffsmodifizierer `private` tatsächlich bedeutet. Markieren Sie ein Klassenmember als `private`, können nur Instanzen der gleichen Klasse darauf zugreifen. Markieren Sie ein Event als `private`, können es also nur Instanzen der gleichen Klasse abonnieren. Das ist ganz anders als bei einem Callback, weil es immer noch einschließt, dass beliebig viele Objekte ein Event anonym abonnieren können.

F: Aber es sieht doch wie ein Event aus, dem nur das Schlüsselwort event fehlt, oder?

A: Der Grund dafür, dass Callbacks und Events so große Ähnlichkeit haben, ist, dass beide **Deiegates** nutzen. Dass sie das tun, ist vernünftig, weil Deiegates das C#-Werkzeug darstellen, um zwischen Objekten Methodenreferenzen auszutauschen.

Aber der große Unterschied zwischen gewöhnlichen Events und Callbacks ist, dass Events einer Klasse ermöglichen, der Welt zu sagen, dass etwas Bestimmtes passiert ist. Ein Callback hingegen wird nie veröffentlicht. Es ist privat, und die aufrufende Methode steuert genau, wen sie aufruft.

Werfen Sie auf der O'Reilly-Website einen Blick auf »Entwurfsmuster von Kopf bis Fuß«. Es ist ein ausgezeichnetes Buch, in dem Sie eine Menge über unterschiedliche Muster erfahren, die Sie auf Ihre eigenen Programme anwenden können:

www.oreilly.de/catalog/hfdesignpatger/

Und schon beim ersten Muster, das Ihnen begegnen wird, finden Sie eine Überschrift mit dem Titel »Herausgeber und Abonnent«. Das sollte Ihnen vertraut erscheinen. Ein Objekt veröffentlicht Informationen, andere Objekte abonnieren sie. Hmmm ...

Der Fall des Goldenen Hummers



Wie konnten die anderen Schatzsucher Heiner ausstechen?

Die Lösung des Problems liegt darin, wie der Schatzsucher seine Beute sucht. Aber erst müssen wir uns genau ansehen, was Heiner in den gestohlenen Diagrammen gefunden hat.

In einem gestohlenen Klassendiagramm entdeckte Heiner, dass die Klasse GoldHummer ein InDeckungGehen-Event absetzt, wenn sich ihr jemand nähert. Dieses enthielt sogar eine NeuerOrtArgs-Instanz mit Informationen zu dem Ort, zu dem sich der Hummer bewegt. Dieses Event kannten die anderen Schatzsucher nicht, und Heiner sah sich schon nach dem Hummer greifen.

```
public class GoldHummer {
    public delegate void Flüchten(NeuerOrtArgs e);
    public event Flüchten InDeckungGehen;
    public void EsKommtEiner() {
        NeuerOrtArgs e = new NeuerOrtArgs("unter dem Stein");
        InDeckungGehen(e);
    }
}
public class NeuerOrtArgs {
    public NeuerOrtArgs(Versteck neuerOrt) {
        this.neuerOrt = neuerOrt;
    }
    private Versteck neuerOrt;
    public Versteck NeuerOrt { get { return neuerOrt; } }
}
```

Jedes Mal, wenn sich jemand dem GoldHummer nähert, setzt seine EsKommtEiner()-Methode das Event InDeckungGehen ab und sucht nach einem neuen Versteck.

Und wie nutzte Heiner die frisch entdeckten Insiderinformationen?

Heiner fügte seinem Konstruktor Code hinzu, der auf seiner GoldHummer-Referenz die Methode schatz_InDeckungGehen() als Event-Handler für das InDeckungGehen-Event registrierte. Dann schickte er einen billigen Schnüffler aus, sich an den Hummer ranzumachen, und wartete, dass der Hummer flüchtete, sich verbarg und das InDeckungGehen-Event absetzte – und seine AufHinweiseAchten()-Methode damit alle erforderlichen Informationen lieferte.

```
public class SchatzSucher {
    public SchatzSucher(GoldHummer schatz) {
        schatz.InDeckungGehen += new GoldHummer.Flüchten(schatz_InDeckungGehen);
    }
    void schatz_InDeckungGehen(NeuerOrtArgs e) {
        HierinGehen(e.NeuerOrt);
    }
    void HierinGehen(Versteck ort) {
        // ... Code, um an einen neuen Ort zu gehen ...
    }
}
```

Heiner hielt sich für gerissen, als er seinen Konstruktor so änderte, dass er einen Event-Handler für das InDeckungGehen-Event registrierte, der seine HierinGehen()-Methode aufruft. Aber er vergaß, dass die anderen Schatzsucher von der gleichen Klasse erben und sein gerissener Code auch ihre Event-Handler an die Kette hängt!

Und das erklärt, warum Heiners Plan nach hinten losging. Als er dem SchatzSucher-Konstruktor den Event-Handler hinzufügte, fügte er ihn unbeabsichtigt ***auch allen anderen SchatzSuchern hinzu!*** Und das bedeutete, dass die Event-Handler aller Schatzsucher für das InDeckungGehen-Event registriert wurden. Als der GoldHummer in Deckung ging, wurden also alle über das Ereignis benachrichtigt. Auch das wäre noch kein Problem gewesen, wäre Heiner der Erste gewesen, der die Nachricht erhält. Aber Heiner hatte keine Möglichkeit, in Erfahrung zu bringen, wann die anderen Schatzsucher informiert würden – hatten diese sich vor ihm registriert, erhielten sie auch vor ihm die Benachrichtigung.

Spitzen Sie Ihren Bleistift

Füllen Sie die Lücken aus, um dieses Whack-a-Mole-Spiel ans Laufen zu bringen. Sie müssen Code einfügen, der die Callbacks ausführt. Geben Sie den Code anschließend in der IDE ein. Oder machen Sie es umgekehrt: Bringen Sie den Code in der IDE ans Laufen und füllen Sie die Lücken anschließend aus. Es wird Ihnen Vergnügen bereiten.

```
public partial class Form1 : Form {
    Maulwurf maulwurf;
    Random zufall = new Random();
    public Form1() {
        InitializeComponent();

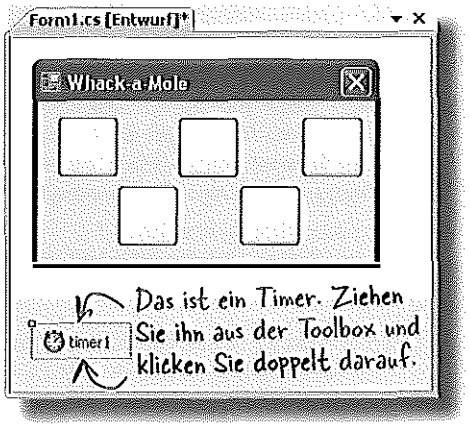
        maulwurf = new Maulwurf(zufall, new Maulwurf._____(_____));
        timer1.Interval = zufall.Next(500, 1000);
        timer1.Start();
    }
    private void timer1_Tick(object sender, EventArgs e) {
        timer1.Stop();
        MaulwurfUmschalten();
    }
    private void MaulwurfUmschalten() {
        if (maulwurf.Verborgen == true)
            maulwurf.Anzeigen();
        else
            maulwurf.WiederVerbergen();
        timer1.Interval = zufall.Next(500, 1000);
        timer1.Start();
    }
    private void MaulwurfCallback(int MaulwurfNr, bool zeigen) {
        if (MaulwurfNr < 0) {
            timer1.Stop();
            return;
        }
        Button button;
        switch (MaulwurfNr) {
            case 0: button = button1; break;
            case 1: button = button2; break;
            case 2: button = button3; break;
            case 3: button = button4; break;
            default: button = button5; break;
        }
        if (zeigen == true)  {
            button.Text = "HAU MICH!";
            button.BackColor = Color.Red;
        } else {
            button.Text = "";
            button.BackColor = SystemColors.Control;
        }
        timer1.Interval = zufall.Next(500, 1000);
        timer1.Start();
    }
    private void button1_Click(object sender, EventArgs e) {
        maulwurf.Getroffen(0);
    }
}
```

Das Formular übergibt dem Maulwurf-Konstruktor ein Delegate, das auf eine Callback-Methode zeigt. Fügen Sie es ein.

Klicken Sie im Formular doppelt auf den Timer (nachdem Sie ihn aus dem Werkzeugkasten gezogen haben), erzeugt die IDE diesen Event-Handler. Timer setzen regelmäßig Tick-Events ab. Diese lernen Sie im nächsten Kapitel kennen.

Diese Methode zeigt oder versteckt den Maulwurf, wenn die Timer-Zeit verstrichen ist.

Dieses switch sorgt dafür, dass der richtige Button seine Farbe ändert.



Fügen Sie bei Eingabe des Codes fünf Button-Event-Handler hinzu. Lassen Sie button2_click() maulwurf.Getroffen(1) aufrufen, button3_click maulwurf.Getroffen(2) usw.

Fügen Sie diese Event-Handler wie üblich hinzu, indem Sie im Formular auf die Buttons doppelklicken.

```
public class Maulwurf {

    public _____ void RausSchauen(int loch, bool zeigen);

    private _____ rausSchauenCallback;
    private bool verborgen;
    public bool Verborgen { get { return verborgen; } }
    private int anzahlTreffer = 0;
    private int anzahlGezeigt = 0;
    private int loch = 0;
    Random zufall;

    public Maulwurf(Random zufall, RausSchauen rausSchauenCallback) {
        if (rausSchauenCallback == null)
            throw new ArgumentException("rausSchauenCallback darf nicht null sein");
        this.zufall = zufall;

        this._____ = _____;
        verborgen = true;
    }

    public void Anzeigen() {
        anzahlGezeigt++;
        verborgen = false;
        loch = zufall.Next(5);

        _____(loch, true);
    }

    public void WiederVerbergen() {
        verborgen = true;

        _____(loch, false);
        GameOverPrüfen();
    }

    public void Getroffen(int gehauenesLoch) {
        if (gehauenesLoch == loch) {
            anzahlTreffer++;
            verborgen = true;
            GameOverPrüfen();

            _____(loch, false);
        }
    }

    private void GameOverPrüfen() {
        if (anzahlGezeigt >= 10) {
            rausSchauenCallback(-1, false);
            MessageBox.Show("Sie haben " + anzahlTreffer + " erzielt", "Gameover");
            Application.Exit();
        }
    }
}
```

Fügen Sie das Delegate und ein Feld zum Speichern des Delegates ein – beide stehen am Anfang der Klasse Maulwurf.

Hier stellen wir sicher, dass das Callback nicht null ist – gegebenenfalls löst Maulwurf eine ArgumentException auf.

Erzeugt das Formular ein neues Maulwurf-Objekt, übergibt es eine Referenz auf seine Callback-Methode. Werfen Sie im Formular einen Blick darauf, wie der Konstruktor aufgerufen wird, und füllen Sie dann die Lücke aus.

Um sich selbst zu zeigen, muss der Maulwurf die Formularmethode aufrufen, die den Maulwurf anzeigt, indem der Button rot gefärbt und darauf der Text »HAU MICH!« angezeigt wird.

Die Methoden WiederVerbergen() und Getroffen() nutzen ebenfalls das Callback-Delegate auf die Methode im Formular.

Das Spiel läuft ab, indem es den Timer nutzt, um ein zufälliges Zeitintervall zwischen einer halben und anderthalb Sekunden zu warten. Ist diese Zeit verstrichen, sagt es dem Maulwurf, dass er sich zeigen soll. Das Formular gibt dem Maulwurf-Objekt ein Callback, über das es dem Formular sagt, dass es einen Maulwurf in einem der Löcher anzeigen oder verbergen soll. Dann wartet das Formular wieder, bevor es dem Maulwurf sagt, dass er sich verbergen soll.

Das Spiel ist vorüber, wenn sich der Maulwurf zehnmal gezeigt hat. Ihre Punktzahl gibt die Anzahl von Treffern an.

Spitzen Sie Ihren Bleistift — Lösung

Sie sollten die Lücken ausfüllen, um das Whack-a-Mole-Spiel ans Laufen zu bringen. Haben Sie den Code in der IDE getestet?

```
public partial class Form1 : Form {
    private void Form1_Load(object sender, EventArgs e)  {

        maulwurf = new Maulwurf(zufall, new Maulwurf.RausSchauen(MaulwurfCallback));
        timer1.Interval = zufall.Next(500, 1000);
        timer1.Start();
    }
}

public class Maulwurf {

    public delegate void RausSchauen(int loch, bool zeigen);

    private RausSchauen rausSchauenCallback;

 ...

    public Maulwurf(Random zufall, RausSchauen rausSchauenCallback) {
        this.zufall = zufall;

        this.rausSchauenCallback = rausSchauenCallback;
        verborgen = true;
    }

    public void Anzeigen() {
        anzahlGezeigt++;
        verborgen = false;
        loch = zufall.Next(5);

        rausSchauenCallback(loch, true);
    }

    public void WiederVerbergen() {
        verborgen = true;

        rausSchauenCallback(loch, false);
        GameOverPrüfen();
    }

    public void Getroffen(int gehauenesLoch) {
        if (gehauenesLoch == loch) {
            anzahlTreffer++;
            verborgen = true;
            GameOverPrüfen();

            rausSchauenCallback(loch, false);
        }
```

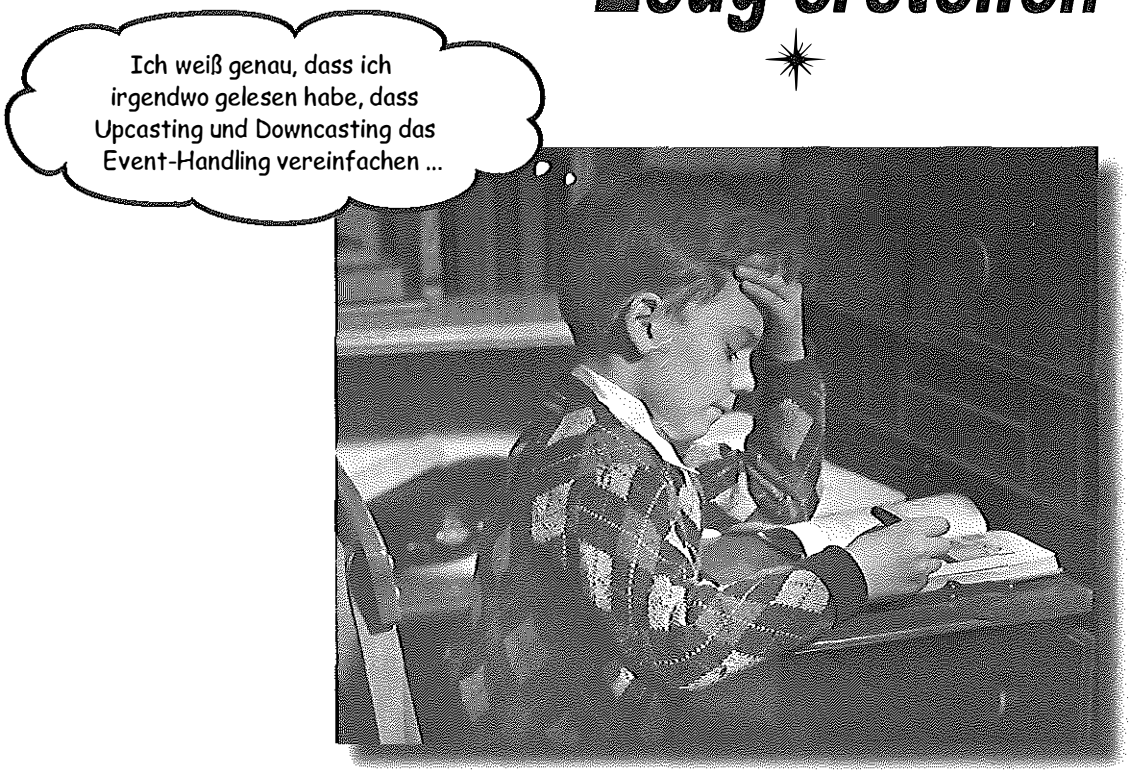
Hier übergibt das Formular eine Referenz auf seine MaulwurfCallback()-Methode in den Maulwurf-Konstruktor, damit der Maulwurf die Methode aufrufen kann.

Hier definiert der Maulwurf das Delegate und nutzt es, um ein privates Feld einzurichten, das eine Referenz auf eine Methode im Formular speichert, die die Farbe und den Text eines Buttons ändert.

Wenn das Formular eine neue Maulwurf-Instanz erzeugt, übergibt es dem Konstruktor als Parameter eine Referenz auf die Methode MaulwurfCallback(). Diese Zeile im Konstruktor kopiert diese Referenz in das Feld rausSchauenCallback. Die Methoden von Maulwurf können dieses Feld nutzen, um die MaulwurfCallback()-Methode im Formular aufzurufen.

Wenn sich der Maulwurf zeigt, sich wieder verbirgt oder gehauen wird, nutzt das Maulwurf-Objekt sein rausSchauenCallback-Delegate-Feld, um die Formularmethode aufzurufen, die die Farbe und den Text eines der Buttons ändert.

Wissen, Macht und cooles Zeug erstellen

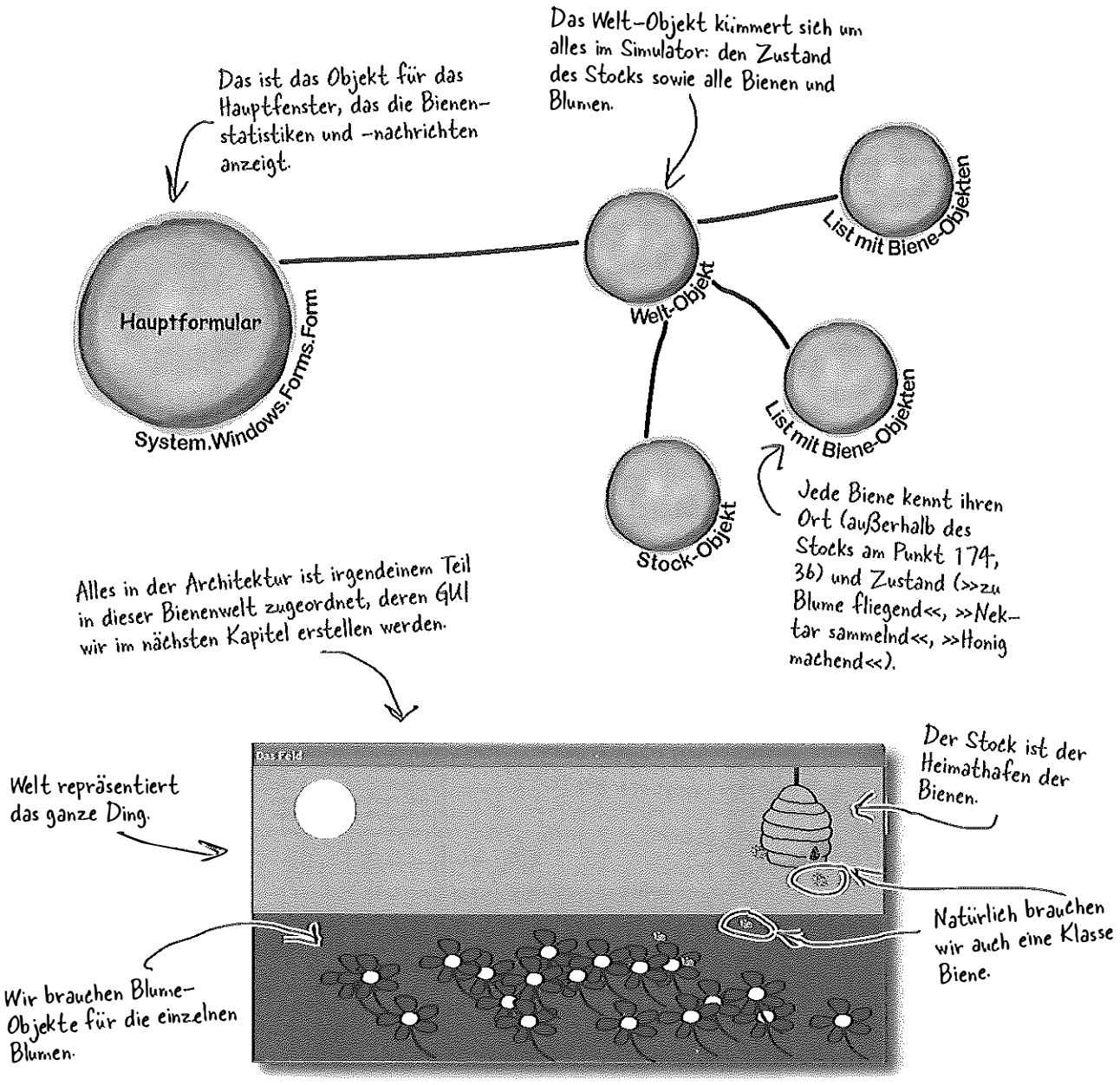


Nur Lernen bringt nichts, man muss das Gelernte auch ANWENDEN.

Erst wenn Sie tatsächlich Code schreiben, wissen Sie, ob Sie die schwereren C#-Konzepte wirklich *verstanden* haben. In diesem Kapitel werden wir einige neue Feinheiten kennenlernen: **Timer** und die Verarbeitung von Collections mit **LINQ** (um nur ein paar zu nennen). Außerdem werden wir eine **richtig komplexe Anwendung** aufbauen und damit prüfen, ob Sie alles, was Sie in den vorangegangenen Kapiteln gelernt haben, auch wirklich beherrschen. Schnallen Sie sich an ... jetzt stürzen wir uns in den Aufbau **cooler Software**.

Die Architektur des Bienenstock-Simulators

Hier ist die Architektur für den Bienen-Simulator. Obwohl der Simulator eine Menge Bienen steuern muss, ist das allgemeine Objektmodell recht einfach.



Den Bienenstock-Simulator erstellen

Etwas derart Komplexes haben wir bislang natürlich noch nicht aufgebaut. Wir werden deswegen mehrere Kapitel benötigen, um alle Teile zusammenzufügen. Auf diesem Weg werden Sie Ihrem Werkzeugkasten Timer, LINQ und eine Menge grafischer Fertigkeiten hinzufügen.

Folgendes werden wir in diesem Kapitel tun (und im nächsten kommt noch mehr dazu):

1. **Wir erstellen eine Klasse Blume, die reift, Nektar produziert und irgendwann welkt und stirbt.**

2. **Wir erstellen eine Klasse Biene, die unterschiedliche Zustände (bei einer Blume Nektar sammeln, zum Stock zurückkehren) haben kann und auf Basis ihres Zustands weiß, was sie tun muss.**

3. **Wir erstellen eine Klasse Stock, die einen Eingang, einen Ausgang, eine Kinderstube für Jungbienen und eine Honigfabrik hat, in der der gesammelte Nektar zu Honig verarbeitet wird.**

4. **Wir erstellen eine Klasse Welt, die Stock, Blumen und Bienen zu jedem gegebenen Moment verwaltet.**

5. **Wir erstellen ein Hauptformular, das Statistiken von den anderen Klassen sammelt und die Welt in Gang hält.**

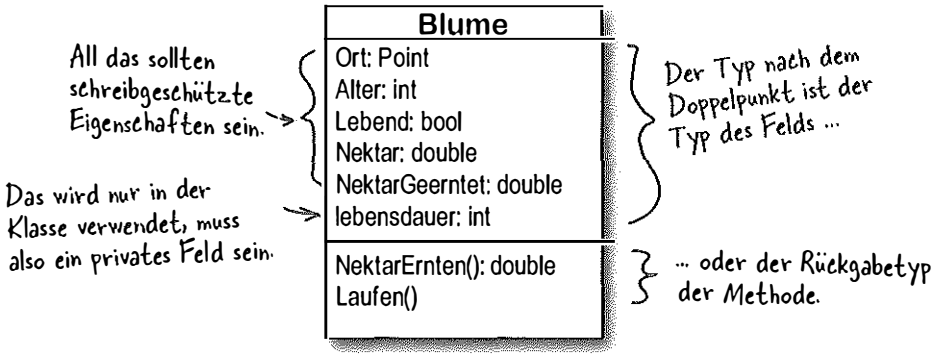


Beginnen wir gleich mit dem Code. Zunächst brauchen wir eine Klasse Blume. Die Klasse Blume hat einen Ort, der durch einen Point definiert wird, ein Alter und eine Lebensdauer. Mit der Zeit altert die Blume. Überschreitet das Alter die Lebensdauer, stirbt die Blume. Das sind die Regeln, die Sie umsetzen sollen.

Ein »Klassengerüst« beinhaltet die Deklarationen für Felder, Eigenschaften und Methoden ohne Implementierung.

1 Schreiben Sie das Codegerüst für Blume.

Unten ist das Klassendiagramm für `Blume`. Schreiben Sie das Grundgerüst für die Klasse. `Ort`, `Alter`, `Lebend`, `Nektar` und `NektarGeerntet` sind **automatische Eigenschaften**. `NektarGeerntet` ist schreibbar, die vier anderen sind **schreibgeschützt**. Lassen Sie die Methoden für den Augenblick leer. Wir werden gleich zu ihnen zurückkommen.



2 Fügen Sie der Klasse einige Konstanten hinzu.

Wir brauchen eine Menge Konstanten für Blumen. Fügen Sie der Klasse `Blume` sechs hinzu:

- `MinLebensdauer`: die kürzeste Lebensdauer einer Blume
- `MaxLebensdauer`: die längste Lebensdauer einer Blume
- `AnfangsNektar`: mit wie viel Nektar eine Blume beginnt
- `MaxNektar`: wie viel Nektar eine Blume maximal aufnehmen kann
- `NektarZuwachsRunde`: wie viel Nektar der Blume jede Runde hinzugefügt wird
- `NektarGesammeltRunde`: wie viel Nektar in einer Runde gesammelt wird

FYI: Konstanten werden im Klassendiagramm üblicherweise nicht angezeigt.

Die Typen für die Konstanten sollten Sie anhand ihrer möglichen Werte ermitteln können. Blumen leben zwischen 15.000 und 30.000 Runden und beginnen mit 1,5 Einheiten Nektar. Sie können bis zu 5 Einheiten Nektar speichern. In jeder Runde erhält eine Blume 0,01 Einheiten Nektar hinzu, und in einer Runde können 0,3 Einheiten gesammelt werden.

In jeder Klasse, die `Point` nutzt, müssen Sie `using System.Drawing;` angeben.

3 Erstellen Sie den Konstruktor.

Der Konstruktor für `Blume` sollte einen `Point` übernehmen, der den Ort der Blume angibt, sowie eine Instanz der Klasse `Random`. Sie sollten diese Argumente einsetzen, um den Ort der Blume zu bestimmen. Setzen Sie dann das Alter auf 0, Lebend auf true und den Nekter auf die anfängliche Nektarmenge für eine Blume. Der Nektar wurde noch nicht geerntet, setzen Sie also auch dieses Feld entsprechend. Ermitteln Sie zum Schluss die Lebensdauer der Blume. Hier ist eine Codezeile, die Ihnen helfen kann:

```
lebensdauer = zufall.Next(MinLebensdauer, MaxLebensdauer + 1);
```

Das funktioniert nur, wenn Ihre Felder und Konstanten sowie die Argumente für den Blume-Konstruktor die entsprechenden Namen haben.

4 Schreiben Sie Code für die Methode NektarErnten().

Jedes Mal, wenn diese Methode aufgerufen wird, sollten Sie prüfen, ob die in der Blume verbleibende Nektarmenge kleiner ist als die Nektarmenge, die pro Runde gesammelt wird. Liefern Sie 0 zurück, sollte das der Fall sein. Andernfalls müssen Sie den in einer Runde gesammelten Betrag vom verbleibenden Nekter in der Blume abziehen. Und vergessen Sie nicht, diesen Betrag dem Feld `NektarGeerntet` hinzuzufügen, das die gesamte Nektarmenge nachhält, die von dieser einen Blume geerntet wurde.

Tipp: Wir haben in dieser Methode NektarGesammeltRunde, Nektar und NektarGeerntet verwendet, aber nichts anderes.

5 Schreiben Sie den Code für die Methode Laufen().

Diese Methode setzt die Blume in Gang. Gehen Sie davon aus, dass ein Aufruf dieser Methode einer Runde entspricht. Aktualisieren Sie also das Alter der Blume entsprechend. Sie müssen auch prüfen, ob das Alter der Blume ihre Lebensdauer übersteigt. Ist das der Fall, stirbt die Blume.

Bleibt die Blume am Leben, müssen Sie ihr die Nektarmenge hinzufügen, die eine Blume in jeder Runde erhält. Achten Sie darauf, dass die maximale Nektarmenge, die die Blume speichern kann, nicht überschritten wird.

Das endgültige Produkt ist animiert: Kleine Bilder von Bienen fliegen umher. Die Methode Laufen() wird in jedem Frame der Animation aufgerufen. Pro Sekunde werden mehrere gezeigt.

Die Antworten stehen auf der nächsten Seite ... versuchen Sie, Ihren Code fertigzustellen, und kompilieren Sie ihn, bevor Sie umblättern.



Lösungen zu den Übungen

Sie sollten die Klasse Blume für den Bienenstock-Simulator erstellen.

Blume
Ort: Point
Alter: int
Lebend: bool
Nektar: double
NektarGeerntet: double
lebensdauer: int
NektarErnten(): double
Laufen()

```
public class Blume {
    private const int MinLebensdauer = 15000;
    private const int MaxLebensdauer = 30000;
    private const double AnfangsNektar = 1.5;
    private const double MaxNektar = 5.0;
    private const double NektarZuwachsRunde = 0.01;
    private const double NektarGesammeltRunde = 0.3;

    public Point Ort { get; private set; }
    public int Alter { get; private set; }
    public bool Lebend { get; private set; }
    public double Nektar { get; private set; }
    public double NektarGeerntet { get; set; }
    private int lebensdauer;

    public Blume(Point ort, Random zufall) {
        Ort = ort;
        Alter = 0;
        Lebend = true;
        Nektar = AnfangsNektar;
        NektarGeerntet = 0;
        lebensdauer = zufall.Next(MinLebensdauer, MaxLebensdauer + 1);
    }

    public double NektarErnten() {
        if (NektarGesammeltRunde > Nektar)
            return 0;
        else {
            Nektar -= NektarGesammeltRunde;
            NektarGeerntet += NektarGesammeltRunde;
            return NektarGesammeltRunde;
        }
    }

    public void Laufen() {
        Alter++;
        if (Alter > lebensdauer)
            Lebend = false;
        else {
            Nektar += NektarZuwachsRunde;
            if (Nektar > MaxNektar)
                Nektar = MaxNektar;
        }
    }
}
```

Ort, Alter, Lebend und Nektar sind schreibgeschützte automatische Eigenschaften.

NektarGeerntet muss für andere Klassen zugreifbar sein.

Blumen haben eine zufällige Lebensdauer, damit sich das Blumenbeet nicht an einem Zeitpunkt vollkommen ändert.

Eine Biene ruft NektarErnten() auf, um Nektar von einer Blume zu erhalten. Eine Biene kann immer nur eine kleine Menge Nektar ernten, muss also mehrere Runden bei der Blume bleiben, bis der gesamte Nektar geerntet wurde.

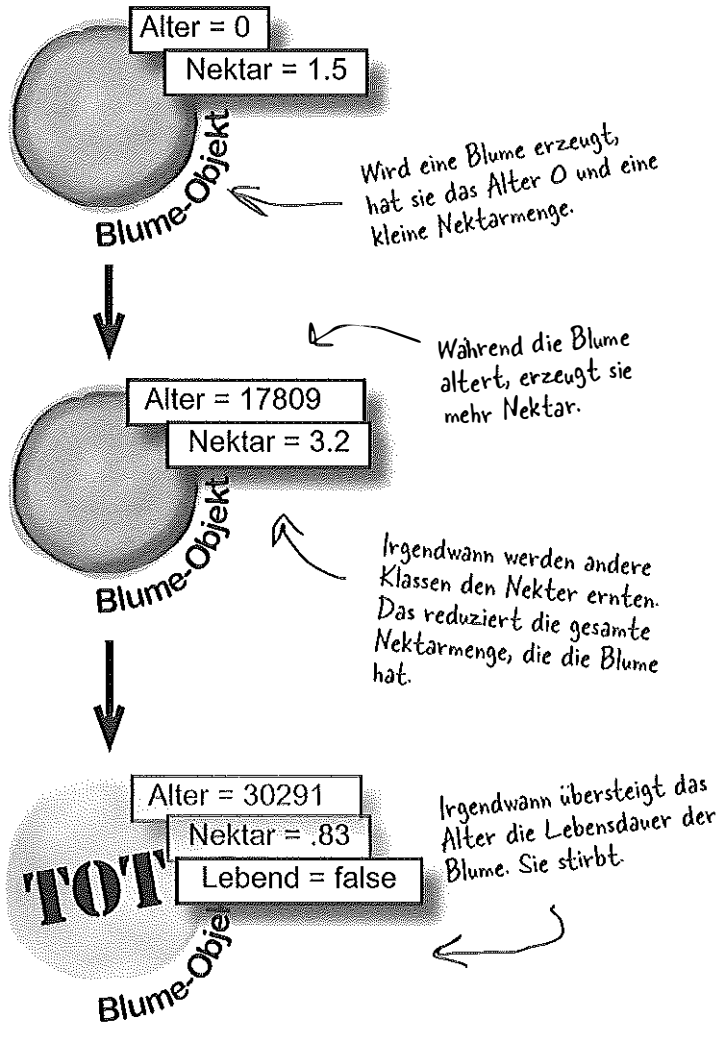
Die Methode Laufen() wird in jedem Frame der Simulator-Animation aufgerufen. Das sorgt dafür, dass die Blume in jedem Frame nur wenig altert – während der Simulator läuft, wird das Alter mit der Zeit steigen.

Achten Sie darauf, dass der Blume nach ihrem Tod kein Nektar mehr hinzugefügt wird.

`Point` befindet sich im Namensraum System.Drawing. Achten Sie darauf, der Datei `using System.Drawing;` hinzuzufügen.

Leben und Sterben einer Blume

Unsere Blume durchlebt einen Kreislauf, sie lebt, bildet Nektar, ihr Nektar wird geerntet, und irgendwann stirbt sie:



Es gibt keine Dummen Fragen

F: NektarGeerntet wird in der Klasse nur an der Stelle verwendet, an der wir es Inkrementieren. Wozu dient dieses Feld?

A: Gut aufgepasst! Wir planen etwas voraus. Irgendwann wird der Simulator die Blumen überwachen und für die Statistik festhalten, wie viel Nektar insgesamt geerntet wurde. Lassen Sie das Feld stehen. Andere Klassen werden es bald verwenden.

F: Warum sind das alles schreibgeschützte automatische Eigenschaften?

A: Erinnern Sie sich an Kapitel 5 und daran, dass man nicht alles zeigen soll? Das ist immer ein gutes Verfahren. Um diese Werte können sich die Blumen kümmern, deswegen haben wir sie schreibgeschützt gemacht. Andere Objekte wie die Bienen und der Stock sollten sie lesen, aber nicht ändern können.

F: Mein Code sieht anders aus. Habe ich etwas falsch gemacht?

A: In Ihren Methoden kann der Code in einer anderen Abfolge stehen. Solange er auf gleiche Weise ***funktioniert*** wie unserer, ist das in Ordnung. Das ist ein anderer Aspekt der Kapselung: Die Interna einer Klasse sind für andere Klassen nicht wichtig, solange die Klasse macht, was sie machen soll.



KOPF-NUSS

Wenn Laufen() das Alter der Blume um 1 erhöht und die Lebensdauer zwischen 15.000 und 30.000 liegt, bedeutet dies, dass Laufen() für jede Blume mindestens 15.000-mal aufgerufen wird, bevor sie stirbt. Wie würden Sie erreichen, dass eine Methode so oft aufgerufen wird? Was wäre, wenn es 10 Blumen gäbe oder 100 oder 1000?

Nun brauchen wir die Klasse Biene

Jetzt können die Blumen abgeerntet werden. Wir brauchen also eine `Biene`-Klasse. Unten ist der grundlegende Code für `Biene`. Die `Biene` kennt ihr Alter und weiß, ob sie sich im Stock befindet oder nicht und wie viel Nektar sie sammeln kann. Wir haben auch eine Methode hinzugefügt, um die Biene zu einem bestimmten Zielpunkt zu bewegen.

```
public class Biene {
  private const double Honigverbrauch = 0.5;
  private const int Schrittweite = 3;
  private const double MinNektarInBlume = 1.5;
  private const int Arbeitsalter = 1000;

  public int Alter { get; private set; }
  public bool InStock { get; private set; }
  public double NektarGesammelt { get; private set; }

  private Point ort;
  public Point Ort { get { return ort; } }

  private int ID;
  private Blume zielBlume;

  public Biene(int id, Point ort) {
    this.ID = id;
    Alter = 0;
    this.ort = ort;
    InStock = true;
    zielBlume = null;
    NektarGesammelt = 0;
  }

  public void Laufen(Random zufall) {
    Alter++;
  }
```

Wie bei Blume müssen einige bienenspezifische Konstanten definiert werden.

Über MinNektarInBlume ermittelt die Biene, welche Blumen abgeerntet werden können.

Für den Ort haben wir ein Hintergrundfeld verwendet. Hätten wir eine automatische Eigenschaft genutzt, könnte ZuOrtBewegen() die Point-Member nicht direkt setzen (>>Ort.X -= Schrittweite<<).

Jeder Biene wird eine eindeutige ID-Kennziffer zugewiesen.

Eine Biene benötigt eine ID und einen Ausgangsort.

Bienen beginnen im Stock, haben noch keine Zielblume und noch keinen Nektar.

Bevor wir fertig sind, müssen wir Laufen() noch eine Menge weiteren Code hinzufügen. Aber das hier ist ein Anfang.

Hier nutzen wir Math.Abs(), um den Absolutwert der Differenz zwischen dem Ziel und dem aktuellen Ort zu berechnen.

```
private bool ZuOrtBewegen(Point ziel) {
    if (ziel != null) {
        if (Math.Abs(ziel.X - ort.X) <= Schrittweite &&
            Math.Abs(ziel.Y - ort.Y) <= Schrittweite)
            return true;
        if (ziel.X > ort.X)
            ort.X += Schrittweite;
        else if (ziel.X < ort.X)
            ort.X -= Schrittweite;
        if (ziel.Y > ort.Y)
            ort.Y += Schrittweite;
        else if (ziel.Y < ort.Y)
            ort.Y -= Schrittweite;
    }
    return false;
}
```

Zunächst ermittelt die Methode, ob die Entfernung zum Ziel bereits kleiner als die Schrittweite ist.

Hat die Biene kein Ziel, ist das entsprechende Feld null. Wir bewegen uns also nur auf ein Ziel zu, wenn ziel NICHT null ist.

Sind wir noch nicht nah genug, bewegen wir uns um die Schrittweite auf das Ziel zu.

Wir liefern false, wenn wir noch nicht am Ziel sind. Dann müssen wir uns weiterbewegen.

Die Methode ZuOrtBewegen() verschiebt den aktuellen Ort, indem die X- und Y-Werte des Ort-Felds angepasst werden. Sie liefert true, wenn die Biene ihr Ziel erreicht hat.



Bienen müssen eine Menge Dinge tun. Unten ist eine Liste. Erstellen Sie ein neues Enum namens `BienenZustand`, das von `Biene` verwendet wird. Außerdem sollten Sie in `Biene` eine schreibgeschützte automatische Eigenschaft namens `AktuellerStatus` erstellen, mit der der Zustand von `Biene` festgehalten wird. Setzen Sie den Anfangsstatus von Bienen auf Unbeschäftigt und fügen Sie in die Methode `Laufen()` eine switch-Anweisung ein, die einen Fall für jedes Element des Enums hat.

Enum-Element	Was das Element bedeutet
Untätig	Die Biene macht nichts.
ZuBlumeFliegend	Die Biene fliegt zu einer Blume.
NektarSammelnd	Die Biene sammelt Nektar in einer Blume.
ZuStockFliegend	Die Biene fliegt zum Stock zurück.
HonigMachend	Die Biene macht Honig.
InRente	Die Biene hat die Flügel an den Haken gehängt.



Lösungen zu den Übungen

Bienen müssen eine Menge Dinge tun. Unten ist eine Liste. Erstellen Sie ein neues Enum namens `BienenZustand`, das von `Biene` verwendet wird. Außerdem sollten Sie in `Biene` eine schreibgeschützte automatische Eigenschaft namens `AktuellerStatus` erstellen, mit der der Zustand von `Biene` festgehalten wird. Setzen Sie den Anfangsstatus von Bienen auf Unbeschäftigt und fügen Sie in die Methode `Laufen()` eine switch-Anweisung ein, die einen Fall für jedes Element des Enums hat.

```
public enum BienenStatus {
  Untätig,
  ZuBlumeFliegend,
  NektarSammelnd,
  ZuStockFliegend,
  HonigMachend,
  InRente
}
```

Hier ist das Enum mit den verschiedenen Bienenstatus.

```
public class Biene {
  // Konstantendeklarationen
  // Felddeklarationen

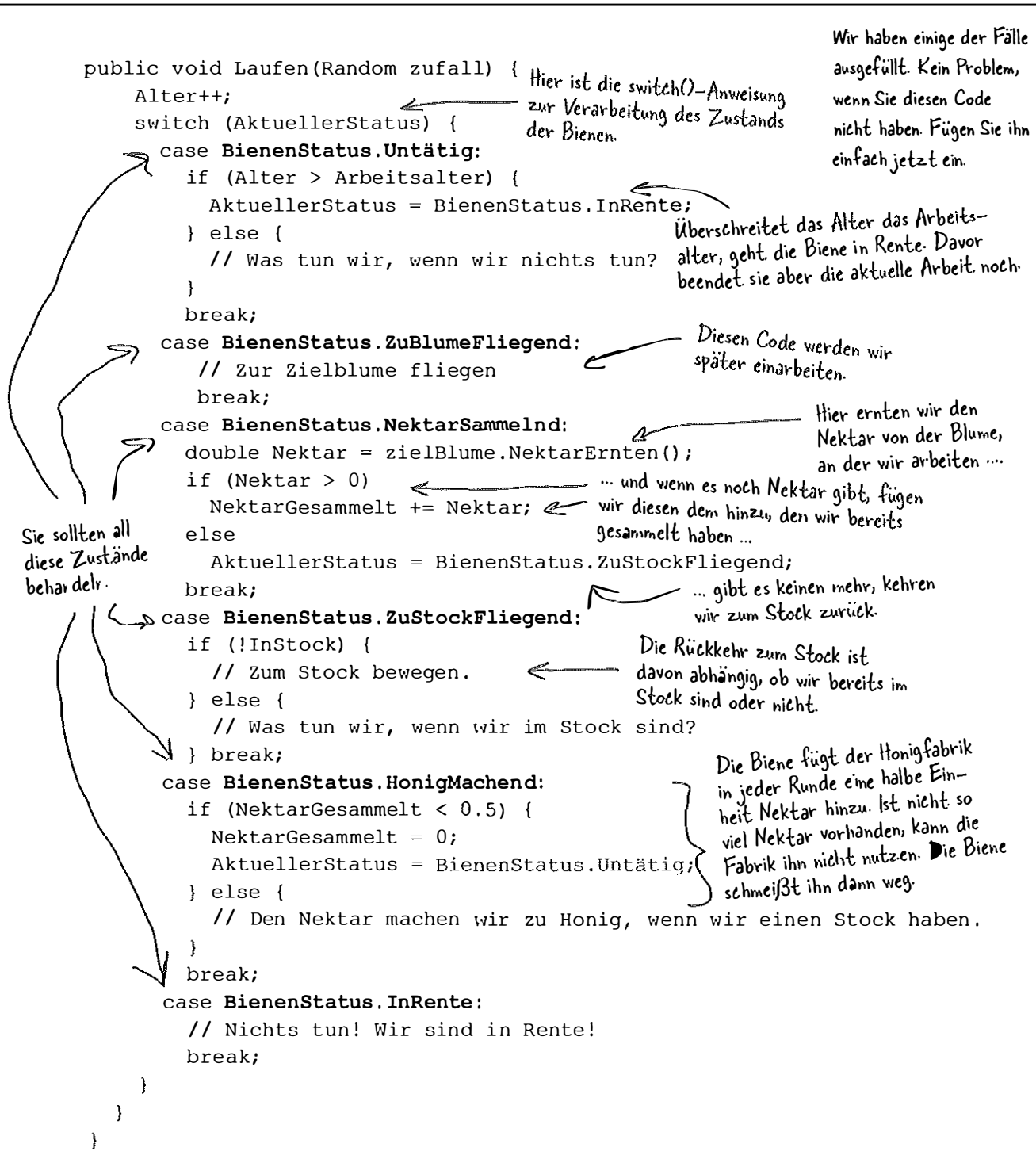
  public BienenStatus AktuellerStatus { get; private set; }

  public Biene(int ID, Point anfangsort) {
    this.ID = ID;
    Alter = 0;
    ort = anfangsort;
    InStock = true;
    AktuellerStatus = BienenStatus.Untätig;
    zielBlume = null;
    NektarGesammelt = 0;
  }
```

Außerdem brauchen wir ein Feld, das den Status der Biene festhält.

Zunächst macht die Biene nichts.

Haben Sie daran gedacht, am Anfang der Datei `using System.Drawing;` hinzuzufügen (weil sie `Point` verwendet)?

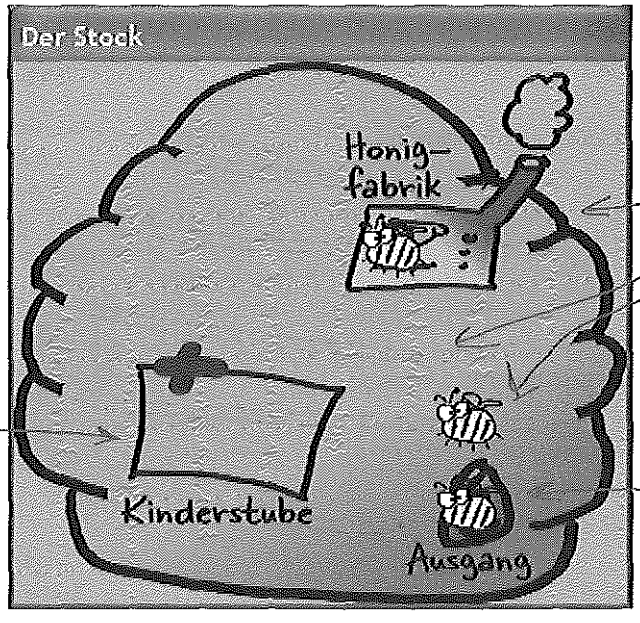



```
public void Laufen(Random zufall) {
    Alter++;
    switch (AktuellerStatus) {
      case BienenStatus.Untätig:
        if (Alter > Arbeitsalter) {
          AktuellerStatus = BienenStatus.InRente;
        } else {
          // Was tun wir, wenn wir nichts tun?
        }
        break;
      case BienenStatus.ZuBlumeFliegend:
        // Zur Zielblume fliegen
        break;
      case BienenStatus.NektarSammelnd:
        double Nektar = zielBlume.NektarErnten();
        if (Nektar > 0)
          NektarGesammelt += Nektar;
        else
          AktuellerStatus = BienenStatus.ZuStockFliegend;
        break;
      case BienenStatus.ZuStockFliegend:
        if (!InStock) {
          // Zum Stock bewegen.
        } else {
          // Was tun wir, wenn wir im Stock sind?
        } break;
      case BienenStatus.HonigMachend:
        if (NektarGesammelt < 0.5) {
          NektarGesammelt = 0;
          AktuellerStatus = BienenStatus.Untätig;
        } else {
          // Den Nektar machen wir zu Honig, wenn wir einen Stock haben.
        }
        break;
      case BienenStatus.InRente:
        // Nichts tun! Wir sind in Rente!
        break;
    }
  }
}
```

Geben wir den Bienen ein Zuhause

Wir haben Bienen und Blumen mit Nektar. Wir müssen noch Code schreiben, der die Bienen den Nektar sammeln lässt. Aber bevor wir das tun, müssen wir uns überlegen, wo die Bienen überhaupt erst erstellt werden! Und wohin sie den Nektar bringen! Wir brauchen dringend eine `Stock`-Klasse.

Der Stock ist allerdings nicht nur der Ort, an den Bienen zurückkehren. Er enthält selbst mehrere Orte, die separate Punkte in der Welt darstellen. Es gibt einen Ein- und Ausgang sowie eine Kinderstube, in der weitere Bienen geboren werden, und eine Honigfabrik, in der der Nektar in Honig umgewandelt wird.



Alle Orte sind separat. Und so wie die Bienen vom Stock zu den Blumen fliegen können, können sie auch von einem dieser Orte zum anderen gehen.

Neue Bienen werden in der Kinderstube erzeugt, die auch ihr Anfangsort ist.

Bienen fliegen durch den Eingang hinein und am Ausgang wieder heraus. Das läuft alles sehr geordnet ab.

Der Stock läuft mit Honig

Die andere große Sache, um die sich der Stock kümmern muss, ist die gespeicherte Honigmenge. Der Stock braucht Honig, damit er läuft, und wenn neue Bienen erstellt werden, ist ebenfalls Honig erforderlich. Zusätzlich muss die Honigfabrik den Nektar nehmen, den die Bienen sammeln, und in Honig umwandeln. Aus jeder einkommenden Nektar-Einheit kann eine Vierteleinheit Honig erstellt werden.

Denken Sie darüber einen Augenblick nach ... während die Zeit verstreicht, braucht der Stock Honig, um am Laufen zu bleiben und weitere Bienen zu erstellen. Inzwischen bringen Bienen weiteren Nektar, der in Honig umgewandelt wird und dafür sorgt, dass die Dinge etwas länger laufen.

All das sollen Sie (mit etwas Hilfe unsererseits) im Code für den Simulator modellieren.



Den Code für Stock sollen **Sie** schreiben.

Stock
Honig: double orte: Dictionary<string, Point> bienenAnzahl: int
OrteInitialisieren() HonigHinzufügen(Nektar: double): bool HonigVerbrauchen(menge: double): bool NeueBiene(zufall: Random) Laufen(zufall: Random) OrtAbrufen(ort: string): Point

1 Schreiben Sie das Codegerüst für Stock.
Wie wir es bei `Blume` gemacht haben, sollten Sie bei `Stock` mit einem Grundgerüst beginnen. Das Klassendiagramm sehen Sie rechts. Machen Sie `Honig` zu einer schreibgeschützten automatischen Eigenschaft und `orte` sowie `bienenAnzahl` zu privaten Feldern, da sie nur intern genutzt werden.

2 Definieren Sie die Konstanten für den Stock.
Sie benötigen eine Konstante für die anfängliche Zahl an Bienen (6), die Honigmenge, mit der der Stock beginnt (3.2), die maximale Honigmenge, die der Stock speichern kann (15), und das Umwandlungsverhältnis von Nektar zu Honig (.25), die maximale Zahl von Bienen (8) und die minimale Honigmenge, die erforderlich ist, damit der Stock neue Bienen erzeugen kann (4).

Sie müssen sich für alle gute Namen sowie die Typen überlegen. Bei den Typen sollten Sie nicht nur die Werte bedenken, die sie speichern, sondern auch die Werte, mit denen die Konstanten verwendet werden. double-Werte passen am besten mit anderen double-Werten zusammen, int-Werte mit anderen int-Werten.

3 Schreiben Sie Code für die Arbeit mit Orten.
Schreiben Sie zunächst die Methode `OrtAbrufen()`. Sie sollte einen String annehmen, diesen im Dictionary orte nachschlagen und den entsprechenden Punkt zurückliefern. Wird er nicht gefunden, sollte eine ArgumentException ausgelöst werden.

Schreiben Sie dann `OrteInitialisieren()`. Diese Methode sollte dem Stock die folgenden Plätze hinzufügen.

- Eingang bei (600, 100)
- Kinderstube bei (95, 174)
- HonigFabrik bei (157, 98)
- Ausgang bei (194, 213)

Diese Koordinaten entsprechen Punkten in dem 2-D-Raum, den der Stock einnimmt. Später müssen wir uns darum kümmern, dass der Simulator den Stock all diese Punkte einschließen lässt.

In dieser Simulation gehen wir davon aus, dass es nur einen Stock mit festgelegten Punkten gibt. Möchten Sie mehrere Stöcke haben, könnten Sie diese Punkte in Bezug auf den Stock angeben anstatt in Bezug auf die gesamte Welt.

4 Erstellen Sie den Stock-Konstruktor.
Wird ein Stock erstellt, sollte der Honig auf die anfängliche Honigmenge für alle Stöcke gesetzt werden. Die Orte im Stock sollten eingerichtet und eine neue Random-Instanz muss erstellt werden. Dann sollte für jede Biene, die anfangs im Stock ist, `NeueBiene()` aufgerufen und dabei die gerade erstellte Random-Instanz übergeben werden

NeueBiene() braucht ein Random-Objekt, weil sie dem Kinderstubenort einen zufälligen Wert hinzufügt, damit die Bienen nicht alle übereinander erscheinen.



Lösungen zu den Übungen

Den Code für Stock sollen **Sie** schreiben.

Denken Sie an >>using System.Drawing;<<, da dieser Code Point nutzt.

Eventuell haben Sie andere Namen für die Konstanten. Das ist okay, wenn Sie im restlichen Code bei diesen Namen bleiben.

```
public class Stock {
  private const int BienenZuAnfang = 6;
  private const double HonigZuAnfang = 3.2;
  private const double MaxHonig = 15.0;
  private const double NektarHonigVerhältnis = .25;
  private const double MinHonigFürBienenErstellung = 4.0;
  private const int MaxBienen = 8;

  private Dictionary<string, Point> orte;
  private int bienenAnzahl = 0;

  public double Honig { get; private set; }

  private void OrteInitialisieren() {
    orte = new Dictionary<string, Point>();
    orte.Add("Eingang", new Point(600, 100));
    orte.Add("KinderStube", new Point(95, 174));
    orte.Add("HonigFabrik", new Point(157, 98));
    orte.Add("Ausgang", new Point(194, 213));
  }

  public Point OrtAbrufen(string ort) {
    if (orte.Keys.Contains(ort))
      return orte[ort];
    else
      throw new ArgumentException("Unbekannter Ort: " + ort);
  }

  public Stock() {
    Honig = HonigZuAnfang;
    OrteInitialisieren();
    Random zufall = new Random();
    for (int i = 0; i < BienenZuAnfang; i++)
      NeueBiene(zufall);
  }

  public bool HonigHinzufügen(double nektar) { return true; }
  public bool HonigVerbrauchen(double menge) { return true; }
  private void NeueBiene(Random zufall) { }
  public void Laufen(Random zufall) { }
}
```

MaxHonig machen wir zu einem double, da diese Konstante in Beziehung zu HonigZuAnfang und Honig steht. Da das beides double-Werte sind, sollte auch MaxHonig einer sein.

Erinnern Sie sich noch an Dictionaries? Unseres speichert einen Point, auf den über einen String zugegriffen wird.

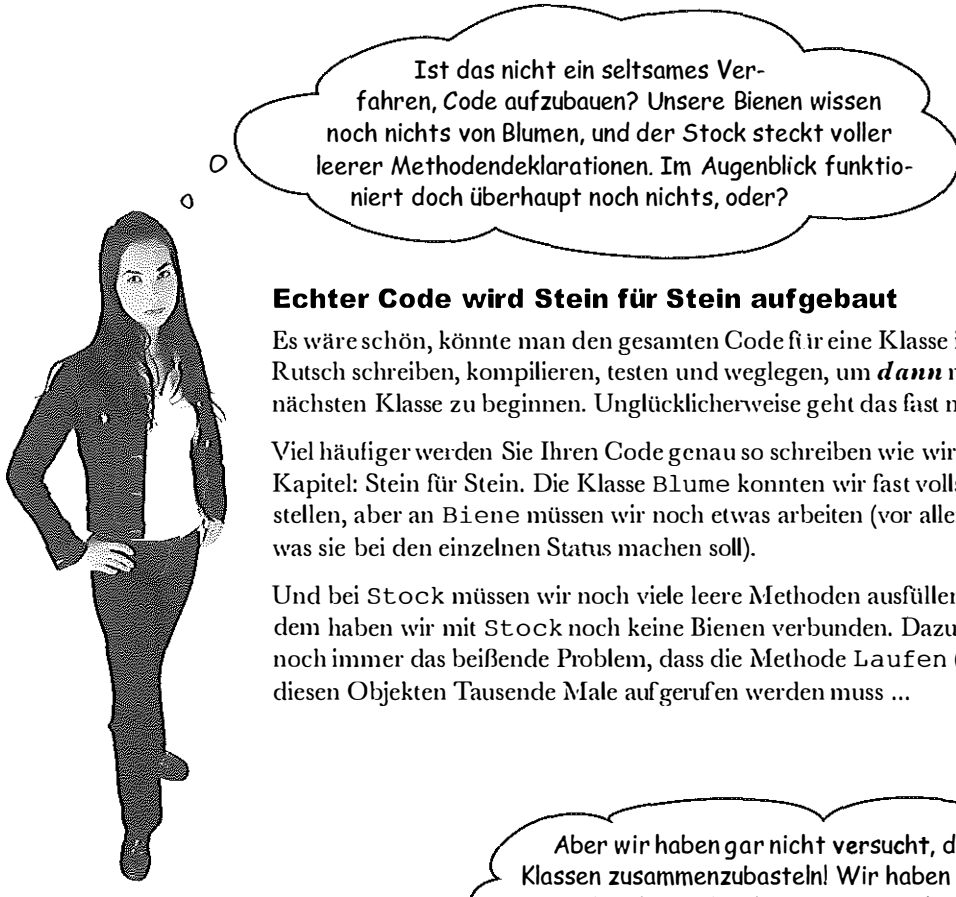
Vergessen Sie nicht, das Dictionary-Objekt zunächst zu instanziieren.

Der Rest der Methode ist ziemlich einfach.

Diese Methode schützt andere davor, bei der Arbeit mit dem Dictionary orte etwas zu ändern, das nicht geändert werden sollte. Das ist ein Beispiel für Kapselung.

NeueBiene() soll einmal für jede Biene aufgerufen werden, mit der der Stock anfängt.

Hier haben wir noch keinen Code. Aber Sie sollten die leeren Methoden als Platzhalter eingefügt haben.



Echter Code wird Stein für Stein aufgebaut

Es wäre schön, könnte man den gesamten Code für eine Klasse in einem Rutsch schreiben, kompilieren, testen und weglegen, um ***dann*** mit der nächsten Klasse zu beginnen. Unglücklicherweise geht das fast nie.

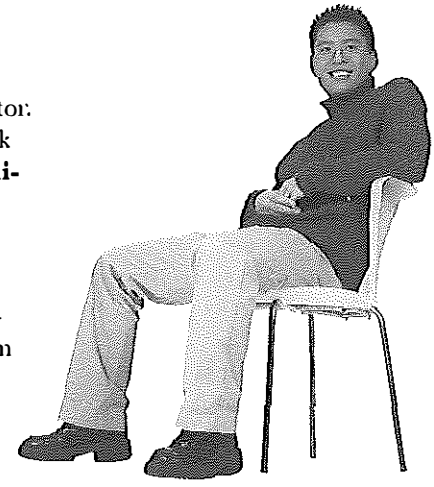
Viel häufiger werden Sie Ihren Code genau so schreiben wie wir in diesem Kapitel: Stein für Stein. Die Klasse `Blume` konnten wir fast vollständig erstellen, aber an `Biene` müssen wir noch etwas arbeiten (vor allem ihr sagen, was sie bei den einzelnen Status machen soll).

Und bei `Stock` müssen wir noch viele leere Methoden ausfüllen. Außerdem haben wir mit `Stock` noch keine Bienen verbunden. Dazu gibt es noch immer das beißende Problem, dass die Methode `Laufen()` in all diesen Objekten Tausende Male aufgerufen werden muss ...

Erst entwerfen Sie, dann erstellen Sie

Unser Ausgangspunkt bei diesem Projekt war, dass wir genau wussten, was wir aufbauen möchten: einen Bienenstock-Simulator. Und wir wissen eine Menge darüber, wie Bienen, Blumen, Stock und Welt zusammenwirken. Deswegen haben wir mit der **Architektur** begonnen, die uns gesagt hat, wie unsere Klassen miteinander arbeiten. Dann konnten wir zu den einzelnen Klassen weitergehen und sie einzeln entwerfen.

Projekte laufen in der Regel glatter, wenn Sie eine klare Vorstellung davon haben, was Sie aufbauen wollen, **bevor** Sie mit dem tatsächlichen Aufbau beginnen. Das erscheint einem ziemlich trivial und selbstverständlich, aber in Bezug auf das endgültige Produkt macht es einen ganz entscheidenden Unterschied.



Die Klasse Stock ergänzen

Kehren wir zur Klasse `Stock` zurück und ergänzen wir ein paar der fehlenden Methoden:

```
public class Stock {
  // Konstantendeklarationen
  // Felddeklarationen

  // OrteInitialisieren()
  // OrtAbrufen()
  // Stock-Konstruktor

  public bool HonigHinzufügen(double nektar) {
    double neuerHonig = nektar * NektarHonigVerhältnis;
    if (neuerHonig + Honig > MaxHonig)
        return false;
    Honig += neuerHonig;
    return true;
  }
  public bool HonigVerbrauchen(double menge) {
    if (menge > Honig)
      return false;
    else {
      Honig -= menge;
      return true;
    }
  }
  private void NeueBiene(Random zufall) {
    bienenAnzahl++;
    int r1 = zufall.Next(100) - 50;
    int r2 = zufall.Next(100) - 50;
    Point anfangsort = new Point(orte["KinderStube"].X + r1,
                                 orte["KinderStube"].Y + r2);
    Biene neueBiene = new Biene(bienenAnzahl, anfangsort);
    // Haben wir ein System, müssen wir diesem die Biene hinzufügen.
  }
  public void Laufen(Random zufall) { }
}
```

Erst müssen wir herausfinden, in wie viel Honig dieser Nektar umgewandelt werden kann ...

... und dann müssen wir prüfen, ob es im Stock noch genug Platz für diese Menge Honig gibt.

Gibt es noch Platz, fügen wir dem Stock den Honig hinzu.

Diese Methode nimmt eine Honigmenge und versucht, sie aus den Vorräten des Stocks zu verbrauchen.

Gibt es im Stock nicht genug Honig, um diese Anforderung zu erfüllen, liefern wir false.

Gibt es genug, wird die Menge aus dem Vorrat des Stocks entfernt und true geliefert.

Diese Methode ist privat ... nur Stock-Instanzen können Bienen erzeugen.

Das erstellt einen Punkt, der in X- und Y-Richtung jeweils bis zu 50 Einheiten vom Ort der Kinderstube entfernt ist.

Erzeugt eine neue Biene am angegebenen Ort.

Wir werden NeueBiene() bald fertigstellen und dann auch die Methode Laufen() füllen ...

Die Laufen()-Methode des Stocks

Wir haben bereits eine `Laufen()`-Methode für `Blume` und eine für `Biene` geschrieben (auch wenn wir hier noch weiteren Code einfügen müssen). Dies ist die `Laufen()`-Methode für `Stock`:

```
public void Laufen(Random zufall) {
  if (Honig > MinHonigFürBienenErstellung)
    NeueBiene(zufall);
}
```

Die einzige Bedingung (zumindest im Moment) ist, dass der Stock genug Honig haben muss, um weitere Bienen zu erzeugen.

Die Random-Instanz, die Laufen() übergeben wurde, wird an die Methode NeueBiene() weitergereicht.

Unglücklicherweise ist das nicht sehr realistisch. In einem beschäftigten Stock passiert es häufig, dass die Königin keine Zeit hat, weitere Bienen zu erzeugen. Wir haben keine Klasse `Bienenkönigin`. Setzen wir also einfach voraus, dass mit einer Wahrscheinlichkeit von 10% eine neue Biene erstellt wird, wenn im Stock ausreichend Honig vorhanden ist. Das können wir so modellieren:

```
public void Laufen(Random zufall) {
  if (Honig > MinHonigFürBienenErstellung
      && zufall.Next(10) == 1) {
    NeueBiene(zufall);
  }
}
```

Das ist ein einfaches Mittel, eine 1-zu-10-Chance dafür zu modellieren, dass eine Biene erzeugt wird. Die Methode liefert eine Zufallszahl zwischen 0 und 9. Ist die Zahl 1, wird die Biene erzeugt.

Man könnte das machen, damit man den Seed-Wert von Random speichern kann – das böte die Möglichkeit, eine bestimmte Simulation noch einmal auszuführen ... wenn Sie das später tun möchten!

Es gibt keine Dummen Fragen

F: Der Stock kann also beliebig viele Bienen erstellen?

A: Zurzeit kann er das – zumindest liegt die Obergrenze sehr hoch –, aber Sie haben recht, sehr realistisch ist das nicht. Später werden wir uns das noch einmal vornehmen und eine Einschränkung formulieren, die in unserer Simulator-Welt nur eine begrenzte Zahl Bienen erlaubt.

F: Könnte man diese Random-Instanz nicht in einer Klasseneigenschaft speichern, anstatt sie an NeueBiene() zu übergeben?

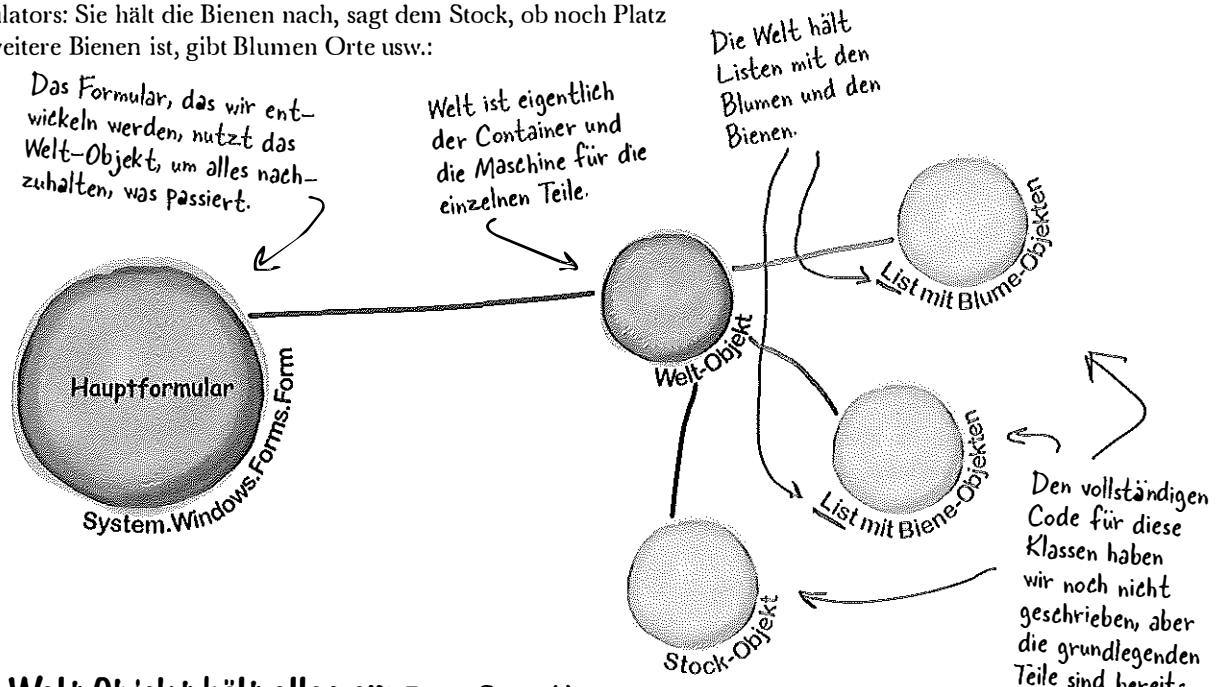
A: Natürlich könnte man das. NeueBiene könnte dann diese Eigenschaft anstelle des Parameters verwenden. Beide Lösungen sind gleich gut, und welche Sie verwenden, bleibt Ihnen überlassen.

F: Ich verstehe immer noch nicht, wie all diese Laufen()-Methoden aufgerufen werden.

A: Das ist nicht schlimm, dazu wollten wir gerade kommen. Erst benötigen wir allerdings noch ein weiteres Objekt: die Klasse Welt, die alles im Griff hat, was im Stock passiert, und alle Bienen und Blumen festhält.

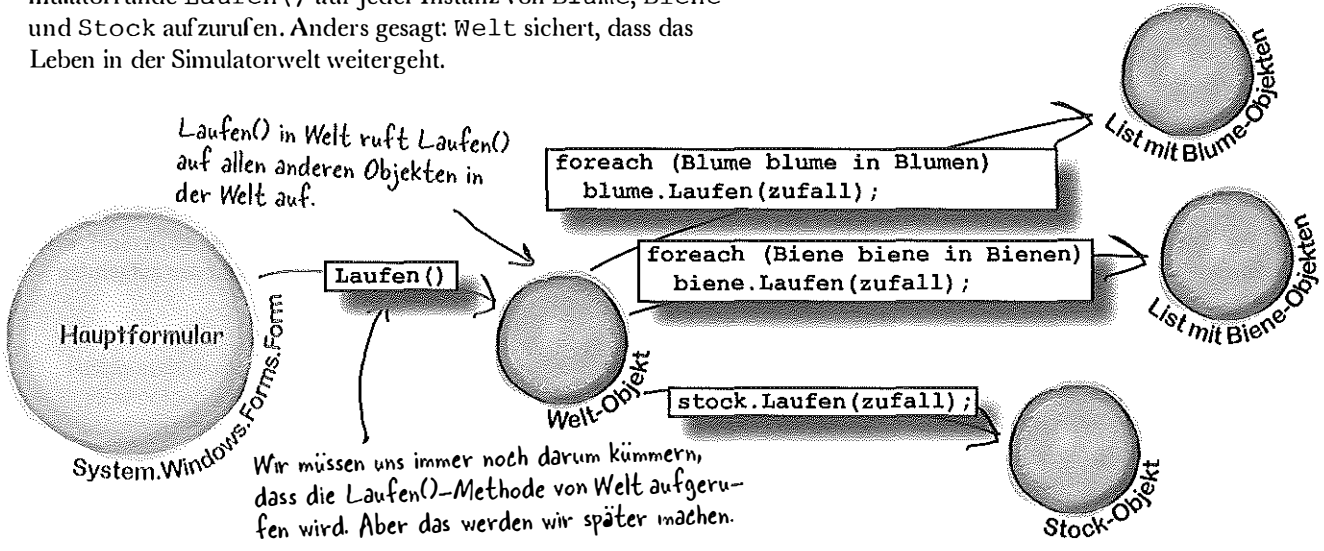
Wir sind bereit für die Welt

Nachdem wir die Klassen `Stock`, `Biene` und `Blume` eingerichtet haben, können wir endlich die Klasse `Welt` aufbauen. `Welt` kümmert sich um die gesamte Koordination der einzelnen Teile unseres Simulators: Sie hält die Bienen nach, sagt dem Stock, ob noch Platz für weitere Bienen ist, gibt Blumen Orte usw.:



Das Welt-Objekt hält alles am `Laufen()`

Eine der größten Aufgaben des `Welt`-Objekts ist es, in jeder Simulatorrunde `Laufen()` auf jeder Instanz von `Blume`, `Biene` und `Stock` aufzurufen. Anders gesagt: `Welt` sichert, dass das Leben in der Simulatorwelt weitergeht.

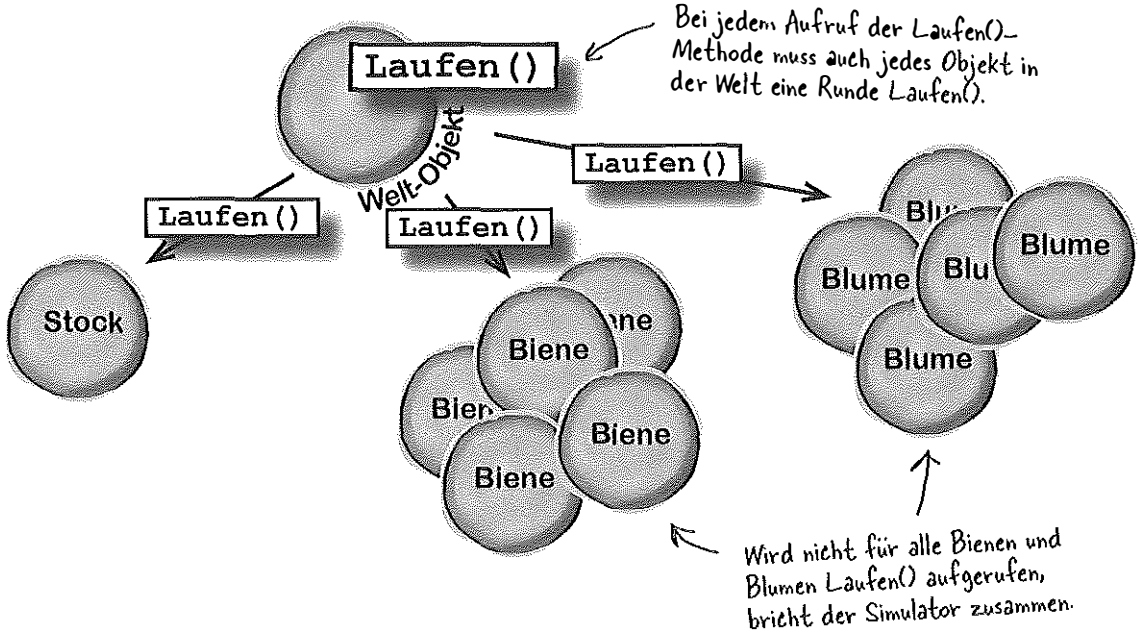


Wir bauen ein rundenbasiertes System

Die `Laufen()`-Methoden in unseren Objekten sollen die einzelnen **Runden** oder jeden **Zyklus** in unserem Simulator abwickeln. Eine Runde bedeutet hier einfach nur eine willkürliche Zeitspanne ... eine Runde könnte beispielsweise 10 Sekunden dauern oder 60 Sekunden, aber auch 10 Minuten.

Die einzelnen »Runden« werden als ein Frame der Animation gezeichnet. Die Welt muss sich in jeder Runde also nur ein kleines bisschen ändern.

Der wichtigste Punkt ist, dass eine Runde alle Objekte in der Welt betrifft. Der Stock altert eine »Runde«, während er prüft, ob weitere Bienen erzeugt werden müssen. Dann vollziehen alle Bienen ihre Runden, bewegen sich einen kleinen Schritt auf ihre Zielblume zu oder führen eine kleine Aktion aus und altern. Anschließend sind die Blumen dran, produzieren etwas Nektar und werden ebenfalls etwas älter. Und genau das macht die Welt: Sie stellt sicher, dass jedes Mal, wenn ihre `Laufen()`-Methode aufgerufen wird, alle Objekte in der Welt ihrerseits in Aktion kommen.



Spitzen Sie Ihren Bleistift

Eins der wichtigen Prinzipien der objektorientierten Programmierung, das wir im Simulator verwendet haben, ist die Kapselung (blättern Sie zur Auffrischung zu Kapitel 5 zurück). Sehen Sie sich den bisher entwickelten Code an und versuchen Sie, jeweils zwei Beispiele für alle erstellten Klassen zu finden.

Stock	Biene	Blume
1.	1.	1. .....
.....	.....	.....
2.	2.	2. .....
.....	.....	.....

Hier ist der Code für Welt

Die Klasse Welt ist eigentlich eine der einfacheren Klassen in unserem Simulator. Hier ist die Grundlage für unseren Code. Aber wenn Sie genauer hinsehen, wird Ihnen auffallen, dass einige Dinge fehlen (die Sie gleich hinzufügen müssen):

```
using System.Drawing;
public class Welt {
    private const double ProNeuerBlumeGeernteterNektar = 50.0;
    private const int FeldMinX = 15;
    private const int FeldMinY = 177;
    private const int FeldMaxX = 690;
    private const int FeldMaxY = 290;

    public Stock Stock;
    public List<Biene> Bienen;
    public List<Blume> Blumen;

    public Welt() {
        Bienen = new List<Biene>();
        Blumen = new List<Blume>();
        Random zufall = new Random();
        for (int i = 0; i < 10; i++)
            NeueBlume(zufall);
    }

    public void Laufen(Random zufall) {
        Stock.Laufen(zufall);

        for (int i = Bienen.Count - 1; i >= 0; i--) {
            Biene biene = Bienen[i];
            biene.Laufen(zufall);
            if (biene.AktuellerStatus == BienenStatus.InRente)
                Bienen.Remove(biene);
        }

        double gesamtGeernteterNektar = 0;
        for (int i = Blumen.Count - 1; i >= 0; i--) {
            Blume blume = Blumen[i];
            blume.Laufen();
            gesamtGeernteterNektar += blume.NektarGeerntet;
            if (!blume.Lebend)
                Blumen.Remove(blume);
        }
```

Diese Konstanten definieren die Grenzen des Felds, auf dem sich Blumen befinden können.

Jede Welt hat einen Stock, eine Liste mit Bienen und eine Liste mit Blumen.

Wenn wir eine neue Welt erzeugen, initialisieren wir unsere Listen, erzeugen einen neuen Stock und fügen dann 10 neue Blumen hinzu.

Das ist einfach ... wir rufen auf Stock Laufen() auf und übergeben eine Random-Instanz

Wir durchlaufen die aktuellen Bienen und rufen auf ihnen Laufen() auf.

Ist die Biene in Rente, müssen wir sie aus der Welt entfernen.

Wir durchlaufen alle Blumen und rufen auf ihnen Laufen() auf.

Wir müssen festhalten, wie viel Nektar in dieser Runde gesammelt wurde. Diesen erhalten wir, indem wir den Nektar zusammenrechnen, der von den einzelnen Blumen gesammelt wurde.

Wie bei den Bienen entfernen wir alle Blumen, die während dieser Runde gestorben sind.

Spitzen Sie Ihren Bleistift
Lösung

Eins der wichtigen Prinzipien der objektorientierten Programmierung, das wir im Simulator verwendet haben, ist die Kapselung (blättern Sie zur Auffrischung zu Kapitel 5 zurück). Sehen Sie sich den bisher entwickelten Code an und versuchen Sie, jeweils zwei Beispiele für alle erstellten Klassen zu finden.

Das sind die, die wir gefunden haben. Sind Ihnen andere aufgefallen?

Stock

1. Das orte-Dictionary des Stocks ist privat.
2. Er gibt den Bienen eine Methode zum Hinzufügen von Honig.

Biene

1. Der Ort der Biene ist schreibgeschützt.
2. Ebenso Alter. Andere Klassen können sie nicht schreiben

Blume

1. Bietet eine Methode zum Nektarsammeln.
2. lebend ist privat.

```
        if (gesamtGeernteterNektar > ProNeuerBlumeGeernteterNektar) {
            foreach (Blume blume in Blumen)
                blume.NektarGeerntet = 0;
            NeueBlume(zufall);
        }
    }

    private void NeueBlume(Random zufall)
    {
        Point ort = new Point(zufall.Next(FeldMinX, FeldMaxX),
                              zufall.Next(FeldMinY, FeldMaxY));
        Blume neueBlume = new Blume(ort, zufall);
        Blumen.Add(neueBlume);
    }
}
```

Während sie Nektar ernten, bestäuben Bienen die Blumen. Haben sie genug Nektar von den Blumen geerntet, haben sie so viele bestäubt, dass die Welt eine neue Blume hinzufügen kann.

Gibt es im Feld genug Nektar, erzeugt die Welt eine neue Blume.

Das kümmert sich darum, einen zufälligen Ort im Feld zu wählen ...

... und dann wird an diesem Punkt eine neue Blume hinzugefügt.

Es gibt keine Dummen Fragen

F: Warum verwenden Sie zum Entfernen toter Blumen und Bienen keine foreach-Schleifen?

A: Weil man in einer foreach-Schleife **keine Elemente** aus einer Auflistung **entfernen** kann, über die iteriert wird. Tun Sie das, löst .NET eine Exception aus.

F: Warum beginnen die for-Schleifen dann am Ende und zählen runter bis 0?

A: Weil die for-Schleifen die Nummerierung der Listen bewahren müssen. Nehmen wir an, Sie würden am Anfang einer Liste mit fünf Blumen beginnen und entdecken, dass eine der Blumen in der Mitte tot ist. Entfernen Sie dann die Blume am Index 3, enthält die Liste nur noch vier Blumen. An den Index 3 wandert eine andere Blume – und diese wird übersprungen, weil die Schleife als Nächstes Index 4 untersucht.

Beginnt die Schleife am Ende, wurde die Blume, die an den leeren Platz rutscht, bereits untersucht. Es kann also keine Blume mehr übersehen werden.



Nachdem unsere vier Kernklassen eingerichtet sind, müssen wir noch ein paar Dinge machen, um sie zu verknüpfen. Befolgen Sie die Schritte unten. Dann sollten Sie funktionierende Biene-, Stock-, Blume- und Welt-Klassen haben. Aber machen Sie sich darauf gefasst, dass Sie Änderungen an fast jeder Klasse vornehmen müssen, bis Sie damit fertig sind.

1. **Aktualisieren Sie Biene so, dass sie eine Stock- und eine Welt-Referenz übernimmt.**
Wir haben jetzt Klassen für `Stock` und `Welt`. `Biene`-Objekte müssen über beide informiert sein. Aktualisieren Sie Ihren Code so, dass im Biene-Konstruktor Referenzen auf Stock- und Welt-Objekte übernommen und zur späteren Verwendung gespeichert werden.

2. **Aktualisieren Sie Stock so, dass eine Welt übernommen wird.**
Genau wie eine `Biene` ihren `Stock` kennen muss, muss ein `Stock` seine `Welt` kennen. Aktualisieren Sie den `Stock`-Konstruktor so, dass er eine `Welt`-Referenz übernimmt und speichert. Außerdem sollten Sie den Bienenerstellungscode in `Stock` so anpassen, dass er dem Biene-Konstruktor eine Referenz auf sich (den `Stock`) und die `Welt` übergibt.

3. **Aktualisieren Sie Welt so, dass sie dem Stock eine Referenz auf sich übergibt.**
Aktualisieren Sie die Klasse `Welt` so, dass sie bei der Erstellung von `Stock` eine Referenz auf sich selbst übergibt.



STOPP! An diesem Punkt sollten Sie Ihren Code kompilieren können. Geht das nicht, ist es besser, ihn jetzt zu überprüfen und eventuelle Fehler zu beheben, bevor Sie weitermachen.

4. **Richten Sie eine Obergrenze für die Zahl Bienen ein, die der Stock erstellen kann.**
Die Klasse Stock besitzt die Konstante MaxBienen, die festlegt, wie viele Bienen der Stock unterstützen kann (in und außerhalb des Stocks). Da der Stock jetzt Zugriff auf die Welt hat, sollten Sie diese Einschränkung erzwingen können.

Tipp: Schauen Sie sich das Umfeld des Codes an, in dem Sie Bienen erstellen. In Stock gibt es zwei Stellen mit Code, der sich darauf bezieht. Geben Sie also acht.

5. **Informieren Sie die Welt, wenn der Stock Bienen erstellt.**
Die Klasse Welt hält alle Bienen nach, die es gibt. Wenn der Stock eine neue Biene erzeugt, muss diese Biene der Gesamtliste hinzugefügt werden, die die Welt pflegt.

Es gibt keine Dummen Fragen

F: Warum haben Sie in der OrtAbrufen()-Methode von Stock eine Exception ausgelöst?

A: Weil wir eine Möglichkeit brauchten, mit ungeeigneten Daten umzugehen. Der Stock hat ein paar Orte, aber über den Parameter für OrtAbrufen() können beliebige Strings übergeben werden. Was passiert, wenn das Programm einen Fehler hat, der bewirkt, dass ein ungültiger String (ein leerer String oder ein Ortsname, der sich nicht im Dictionary orte befindet) als Parameter übergeben wird? Was soll die Methode dann zurückliefern?

Wenn man einen ungültigen Parameter hat und nicht klar ist, was man mit diesem tun soll, ist es meist ein guter Gedanke, eine ArgumentException auszulösen. Das macht die Methode OrtAbrufen() folgendermaßen:

```
throw new ArgumentException(
      "Unbekannter Ort: " + ort);
```

Diese Anweisung bewirkt, dass die Klasse Stock eine ArgumentException mit der Meldung »Unbekannter Ort:« ausgibt, die den Ort angibt, der nicht gefunden werden konnte.

Das ist hilfreich, weil es Sie unmittelbar darauf aufmerksam macht, wenn ein ungeeigneter Ortsparameter an die Methode übergeben wird. Und indem Sie den Parameter in die Exception-Meldung einschließen, liefern Sie sich selbst wertvolle Informationen, die Ihnen helfen, das Problem zu debuggen.

F: Was bringt es, alle Orte in `Point`-Objekten zu speichern, obwohl wir gar nichts zeichnen?

A: Jede Biene hat einen Ort, ganz gleich, ob Sie sie auf dem Bildschirm an diesem Ort zeichnen oder nicht. Das `Biene`-Objekt soll nachhalten, wo es sich in der Welt befindet. Jedes Mal, wenn seine `Laufen()`-Methode aufgerufen wird, muss es sich ein kleines Stück näher zu seinem Ziel bewegen.

Auch wenn wir noch kein Bild der Biene zeichnen, muss die Biene dennoch wissen, wo sie sich im Stock oder im Feld befindet, weil sie wissen muss, ob sie an ihrem Ziel angekommen ist oder nicht.

F: Warum speichern wir den Ort dann in einem `Point` und nicht in etwas anderem? Sind `Point`s nicht speziell zum Zeichnen gedacht?

A: Ja, **`Point`**-Instanzen werden von allen grafischen Steuerelementen für ihre Location-Eigenschaften verwendet. Aber dass .NET sie auf diese Weise verwendet, bedeutet nicht, dass sie uns nicht dabei helfen können, unsere Orte nachzuhalten. Natürlich hätten wir eine `BienenOrt`-Klasse mit int-Feldern namens X und Y erstellen können. Aber es gibt keinen Grund dafür, das Rad noch einmal neu zu erfinden, wenn uns C# und .NET **`Point`** kostenlos zur Verfügung stellen!

Es ist fast immer einfacher, eine vorhandene Klasse, die Ihren Anforderungen im WESENTLICHEN entspricht, einem anderen Zweck zuzuführen oder zu erweitern, statt von Grund auf eine neue Klasse zu schreiben.



LÖSUNGEN ZU DEN ÜBUNGEN

Nachdem unsere vier Kernklassen eingerichtet sind, müssen wir noch ein paar Dinge machen, um sie zu verknüpfen. Befolgen Sie die Schritte unten. Dann sollten Sie funktionierende Biene-, Stock-, Blume- und Welt-Klassen haben. Aber machen Sie sich darauf gefasst, dass Sie Änderungen an fast jeder Klasse vornehmen müssen, bis Sie damit fertig sind.

1 **Aktualisieren Sie Biene so, dass sie eine Stock- und eine Welt-Referenz übernimmt.**
Wir haben jetzt Klassen für `Stock` und `Welt`. `Biene`-Objekte müssen über beide informiert sein. Aktualisieren Sie Ihren Code so, dass im Biene-Konstruktor Referenzen auf Stock- und Welt-Objekte übernommen und zur späteren Verwendung gespeichert werden.

```
public class Biene {
  // Bestehende Konstantendeklarationen
  // Bestehende Felddeklarationen
  private Welt welt;
  private Stock stock;

  public Biene(int ID, Point anfangsort, Welt welt, Stock stock) {
    // Bestehender Code
    this.welt = welt;
    this.stock = stock;
  }
}
```

Das ist ziemlich einfach ... nehmen Sie diese Werte und weisen Sie sie den Feldern zu.

2 **Aktualisieren Sie Stock so, dass eine Welt übernommen wird.**
Genau wie eine `Biene` ihren `Stock` kennen muss, muss ein `Stock` seine `Welt` kennen. Aktualisieren Sie den `Stock`-Konstruktor so, dass er eine `Welt`-Referenz übernimmt und speichert. Außerdem sollten Sie den Bienenerstellungscode in `Stock` so anpassen, dass er dem Biene-Konstruktor eine Referenz auf sich (den `Stock`) und die `Welt` übergibt.

```
public class Stock {
  private Welt welt;

  public Stock(Welt welt) {
    this.welt = welt;
    // Bestehender Code.
  }
  public void NeueBiene(Random zufall) {
    // Restlicher Code zur Bienenerstellung.
    Biene neueBiene = new Biene(bienenAnzahl, anfangsort, welt, this);
  }
}
```

Weiterer elementarer Code ... die Referenz wird übernommen und dem Feld zugewiesen. Sie müssen welt ZUERST zuweisen, weil es im restlichen Konstruktor benötigt wird.

Neue Bienen brauchen jetzt eine Referenz auf die Welt und den Stock.

Sollten Sie das nicht ans Laufen kriegen, können Sie den Code für diese Übung (und alle anderen) unter http://examples.oreilly.de/german_examples/hfcsharpger herunterladen.

3 **Aktualisieren Sie Welt so, dass sie dem Stock eine Referenz auf sich übergibt.**
Aktualisieren Sie die Klasse `Welt` so, dass sie bei der Erstellung von `Stock` eine Referenz auf sich selbst übergibt.

```
public Welt() {
  Bienen = new List<Biene>();
  Blumen = new List<Blume>();
  Stock = new Stock(this);
  Random zufall = new Random();
    for (int i = 0; i < 10; i++)
      NeueBlume(zufall);
}
```

Das übergibt die Referenz an den Stock.

4 **Richten Sie eine Obergrenze für die Zahl Bienen ein, die der Stock erstellen kann.**
Die Klasse Stock besitzt die Konstante MaxBienen, die festlegt, wie viele Bienen der Stock unterstützen kann (in und außerhalb des Stocks). Da der Stock jetzt Zugriff auf die Welt hat, sollten Sie diese Einschränkung erzwingen können.

```
public void Laufen(Random zufall) {
  if (welt.Bienen.Count < MaxBienen
       && Honig > MinHonigFürBienenErstellung
       && zufall.Next(10) == 1) {
    NeueBiene(zufall);
  }
}
```

Jetzt können Sie das Welt-Objekt nutzen, um zu prüfen, wie viele Bienen es insgesamt gibt, und das mit der Maximalzahl an Bienen für diesen Stock vergleichen.

Wir haben den Vergleich an den Anfang gesetzt. Gibt es keinen Platz für Bienen, ist es sinnlos zu prüfen, ob es genug Honig gibt.

5 **Informieren Sie die Welt, wenn der Stock Bienen erstellt.**
Die Klasse Welt hält alle Bienen nach, die es gibt. Wenn der Stock eine neue Biene erzeugt, muss diese Biene der Gesamtliste hinzugefügt werden, die die Welt pflegt.

```
private void NeueBiene(Random zufall) {
  bienenAnzahl++;
  // Den Anfangspunkt berechnen
  Point startpunkt = // In der Nähe der Kinderstube beginnen
  Biene neueBiene = new Biene(bienenAnzahl, startpunkt, welt, this);
  welt.Bienen.Add(neueBiene);
}
```

Wir fügen die neue Biene der Gesamtliste der Welt hinzu.

Das ist einer der Gründe dafür, dass wir in Stock eine Welt-Referenz brauchen.

Den Bienen ein Verhalten geben

In unseren Klassen fehlt noch ein großer Codeteil: die `Laufen()`-Methode von `Biene`. Ein paar der Zustände konnten wir bereits schreiben, aber es stehen noch einige aus (`Untätig` ist unvollständig, `ZuBlumeFliegend` und Teile von `HonigMachend` fehlen).

Erledigen wir jetzt die verbleibenden Status:

```
public void Laufen(Random zufall) {
    Alter++;
    switch (AktuellerStatus) {
        case BienenStatus.Untätig:
            if (Alter > Arbeitsalter)
                AktuellerStatus = BienenStatus.InRente;
            else if (welt.Blumen.Count > 0
                && stock.HonigVerbrauchen(Honigverbrauch)) {
                Blume blume =
                  welt.Blumen[zufall.Next(welt.Blumen.Count)];
                if (blume.Nektar >= MinNektarInBlume && blume.Lebend) {
                    zielBlume = blume;
                    AktuellerStatus = BienenStatus.ZuBlumeFliegend;
                }
            }
            break;
        case BienenStatus.ZuBlumeFliegend:
            if (!welt.Blumen.Contains(zielBlume))
                AktuellerStatus = BienenStatus.ZuStockFliegend;
            else if (InStock) {
                if (ZuOrtBewegen(stock.OrtAbrufen("Ausgang"))) {
                    InStock = false;
                    ort = stock.OrtAbrufen("Eingang");
                }
            }
            else
                if (ZuOrtBewegen(zielBlume.Ort))
                    AktuellerStatus = BienenStatus.NektarSammelnd;
            break;
        case BienenStatus.NektarSammelnd:
            double nektar = zielBlume.NektarErnten();
            if (nektar > 0)
                NektarGesammelt += nektar;
            else
                AktuellerStatus = BienenStatus.ZuStockFliegend;
            break;
```

Hat die Biene nichts zu tun, muss sie eine andere Blume suchen, die sie abernten kann.

Prüfen, ob noch Blumen übrig sind, und dann Honig verbrauchen, um die Sache am Laufen zu halten. Andernfalls geht die Simulation nicht mehr weiter.

Wir brauchen eine lebende Blume mit Nektar.

Geht alles auf, kann die Biene zur neuen Blume fliegen.

Sicherstellen, dass die Blume nicht gestorben ist, während die Biene zu ihr fliegt.

Deswegen haben wir der Biene eine Referenz auf den Stock übergeben.

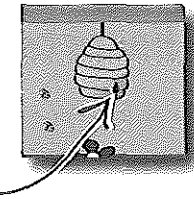
Erreicht die Biene den Ausgang, hat sie den Stock verlassen. Der Ort wird aktualisiert. Da die Biene jetzt auf dem Feld ist, sollte sie in der Nähe des Eingangs ausfliegen.

Wurde der Stock verlassen, fliegt die Biene auf die Blume zu und beginnt, Nektar zu sammeln, wenn sie sie erreicht hat.



Das ist der Ausgang. Wenn der Stock den Ort >>Ausgang<< speichert, entspricht das dem Punkt auf dem Stock-Formular, an dem sich das Bild des Ausgangs befindet.

Das ist der Eingang. Fliegen die Bienen in den Stock zurück, fliegen sie auf den Eingang des Stocks auf dem Feld-Formular zu.



Deswegen speichert das Dictionary orte zwei separate Orte >>Ausgang<< und >>Eingang<<.

```
        case BienenStatus.ZuStockFliegend:
            if (!InStock) {
                if (ZuOrtBewegen(stock.OrtAbrufen("Eingang"))) {
                    InStock = true;
                    ort = stock.OrtAbrufen("Ausgang");
                }
            }
            else
                if (ZuOrtBewegen(stock.OrtAbrufen("HonigFabrik")))
                    AktuellerStatus = BienenStatus.HonigMachend;
            break;
        case BienenStatus.HonigMachend:
            if (NektarGesammelt < 0.5) {
                NektarGesammelt = 0;
                AktuellerStatus = BienenStatus.Untätig;
            }
            else
                if (stock.HonigHinzufügen(0.5))
                    NektarGesammelt -= 0.5;
                else
                    NektarGesammelt = 0;
            break;
        case BienenStatus.InRente:
            // Nichts tun! Wir sind in Rente!
            break;
    }
}
```

Wurde der Stock erreicht, werden der Ort und das Feld InStock aktualisiert.

Ist die Biene im Stock, geht sie zur HonigFabrik.

Sie versucht, den Nektar im Stock abzuliefern.

Falls der Stock den Nektar nutzen kann, um Honig herzustellen ...

... muss er der Biene entnommen werden.

Ist der Stock voll, liefert HonigHinzufügen() false. Die Biene wirft dann den restlichen Nektar weg, damit sie wieder aus dem Stock fliegen kann.

Ist die Biene in Rente, muss sie nur warten, bis der Stock sie von der Liste entfernt. Dann kann sie auf Kreuzfahrt gehen.



KOPF-NUSS

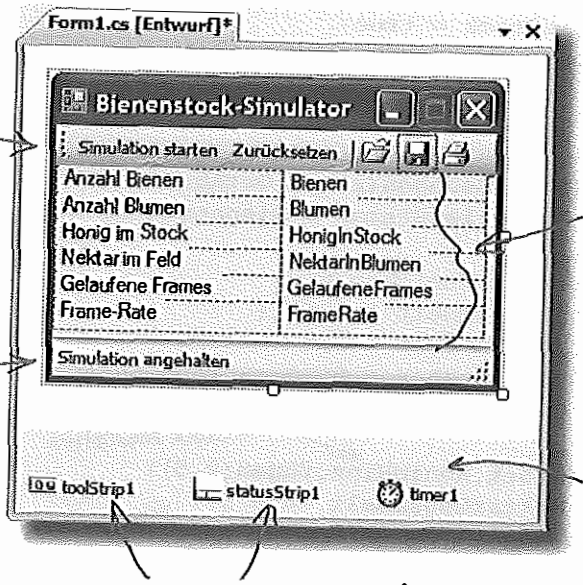
Angenommen, Sie möchten den Simulator so ändern, dass der Flug zur Blume bzw. zum Stock zurück jeweils nur zwei Runden dauert. Welche ***Methoden*** welcher Klassen müssten Sie ändern, um dieses neue Verhalten einzurichten (über den erforderlichen Code müssen Sie sich nicht den Kopf zerbrechen)?

Das Hauptformular bringt die Welt ans Laufen()

Gut, Sie wissen bereits, dass die Welt jede Runde um einen Frame weiterrückt, wenn ihre `Laufen()`-Methode aufgerufen wird. Aber wer ruft diese `Laufen()`-Methode auf? Das Hauptformular natürlich! Jetzt ist es an der Zeit, es zu entwerfen.

Fügen Sie Ihrem Projekt ein neues Formular hinzu. Lassen Sie es aussehen wie das Formular unten. Wir nutzen einige neue Steuerelemente, aber wir werden diese auf den nächsten paar Seiten erklären.

Das ToolStrip-Steuerelement fügt oben in das Formular eine Werkzeugleiste ein. Die beiden Buttons können Sie über das Aufklappmenü hinzufügen, das auf dem Formular erscheint. Setzen Sie die DisplayStyle-Eigenschaft beider Buttons auf Text.



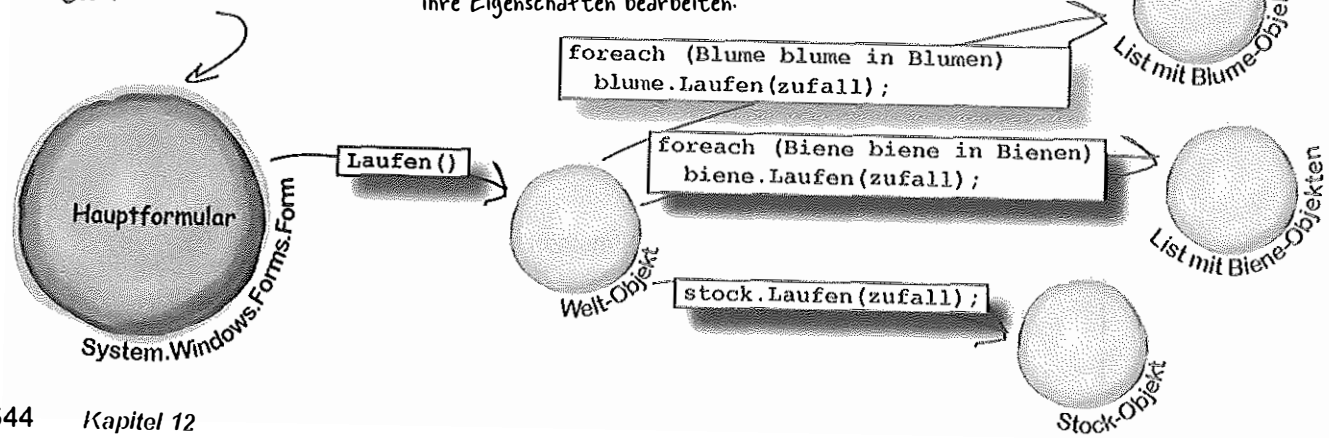
Jedes dieser Labels befindet sich in einer Zelle eines TableLayoutPanel-Steuerelements. Diese gestalten Sie genau wie eine Tabelle in Microsoft Word. Klicken Sie auf den kleinen schwarzen Pfeil, um Zeilen und Spalten hinzuzufügen, zu entfernen und zu verändern.

Fügen Sie unten ein StatusStrip-Element ein, um eine Statusleiste anzeigen zu lassen. Nutzen Sie das Aufklappmenü, um ihm ein StatusLabel hinzuzufügen.

Fügen Sie dem Formular ein Timer-Steuerelement hinzu. Es taucht darauf überhaupt nicht auf – es ist eine nicht sichtbare Komponente, die der Form-Designer im Bereich unter dem Formular als Symbol anzeigt.

Das ToolStrip-Steuerelement fügt oben in Ihr Formular eine Werkzeugleiste ein und StatusStrip unten eine Statusleiste. Beide erscheinen auch als Symbole im Bereich unterhalb des Formulars. Darüber können Sie ihre Eigenschaften bearbeiten.

Endlich kommen wir zu dem Code, der die Welt bewegt.



Wir können die Welt nutzen, um Informationen zu erhalten

Jetzt möchten wir all diese Steuerelemente aktualisieren. Aber wir brauchen nicht für jedes einen Click-Handler. Stattdessen werden wir eine einzige Methode verwenden, die die verschiedenen Statistiken im Simulator-Fenster aktualisiert.

Das zeigt an, wie viel Zeit in einer Runde verstreicht ... diesen Parameter müssen wir von einem anderen Ort hereingeben, zu dem wir in ein paar Seiten kommen werden.

```
private void StatistikAktualisieren(TimeSpan frameDauer) {
    Bienen.Text = welt.Bienen.Count.ToString();
    Blumen.Text = welt.Blumen.Count.ToString();
    HonigInStock.Text = String.Format("{0:f3}", welt.Stock.Honig);
    double nektar = 0;
    foreach (Blume blume in welt.Blumen)
        nektar += blume.Nektar;
    NektarInBlumen.Text = String.Format("{0:f3}", nektar);
    GelaufeneFrames.Text = gelaufeneFrames.ToString();
    double milliSekunden = frameDauer.TotalMilliseconds;
    if (milliSekunden != 0.0)
        FrameRate.Text = string.Format("{0:f0} ({1:f1}ms)",
                                       1000 / milliSekunden, milliSekunden);
    else
        FrameRate.Text = "N/A";
}
```

Meist geht es hier nur darum, Daten von der Welt abzurufen und die Labels zu aktualisieren.

Achten Sie darauf, dass die Namen der Labels im Formular Ihrem Code entsprechen.

Geben Sie den ersten Parameter als Zahl ohne Nachkommastellen an, dann ein Leerzeichen, dann den zweiten Parameter mit einer Nachkommastelle und (in Klammern) die Buchstaben >>ms<<.

Die Frame-Rate ist die Anzahl von Frames, die pro Sekunde ausgeführt werden. Wir nutzen ein TimeSpan-Objekt, um zu speichern, wie lange die Ausführung des Frames dauerte. Dann teilen wir 1000 durch die Anzahl von Millisekunden, die die Ausführung des Frames dauerte – das gibt uns die Gesamtmillisekundenzahl für die Ausführung des letzten Frames.

Fügen Sie diese Methode in Form1 ein.

Ähh! Wo kommt denn das Welt-Objekt her ... das haben wir doch noch gar nicht erstellt? Und was ist mit dem ganzen Zeit- und Frame- Krempel?

Dieser Code nutzt die String-Methode Format(), die Sie im Hex-Dump verwendet haben. Aber statt mit >>x2<< Hex-Werte auszugeben, nutzen Sie >>f3<<, um eine Zahl mit drei Nachkommastellen auszugeben.

Erstellen wir eine Welt

Sie haben recht, wir müssen ein `Welt`-Objekt erstellen. Ergänzen Sie den Konstruktor Ihres Formulars um diese Zeilen:

```
public Form1() {
  InitializeComponent();
  welt = new Welt();
}
```

Fügen Sie Ihrem Formular dann ein privates `Welt`-Feld namens `welt` hinzu.

Jetzt bleibt nur noch der zeitbezogene Code. Wir haben ja immer gesagt, dass wir eine Möglichkeit brauchen, in Welt immer und immer wieder `Laufen()` auszuführen ... und das klingt, als bräuchten wir eine Art Timer.



Dazu werden wir bei der Erstellung des TimeSpan-Objekts noch mehr sagen.

Timer lösen Events immer wieder aus

Nehmen Sie sich etwas Zeit und schauen Sie sich die Funktionsweise von Timern in einem neuen Projekt an. Dann widmen wir uns wieder dem Simulator und wenden das Gelernte an.

Erinnern Sie sich daran, wie Sie die Windhunde mit einer Schleife animiert haben? Dafür gibt es aber noch einen besseren Weg! Ein **Timer** ist eine besonders nützliche Komponente, die ein Event immer wieder auslöst, bis zu tausendmal pro Sekunde.

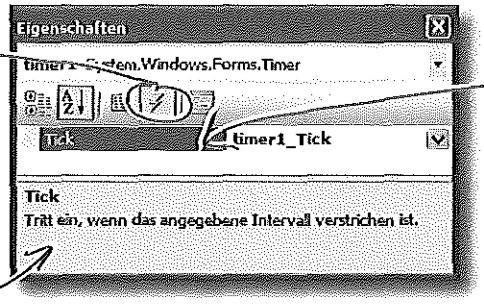
Tun Sie das!

1. **Erstellen Sie ein neues Projekt mit einem Timer und drei Buttons.**
Sie müssen Ihr aktuelles Projekt nicht schließen – öffnen Sie einfach eine neue Visual Studio-Instanz und beginnen Sie ein neues Projekt. Ziehen Sie einen Timer und drei Buttons auf das Formular. Klicken Sie auf das Timer-Symbol unten im Designer und setzen Sie seine Interval-Eigenschaft auf 1000 (Millisekunden). Es sagt dem Timer, dass er sein Tick-Event sekündlich absetzen soll.

2. **Öffnen Sie das Eigenschaften-Fenster und klicken Sie auf den Button Events.**
(Wissen Sie noch? Der Events-Button sieht wie ein Blitz aus und ermöglicht Ihnen, die Events für die Formularsteuerelemente zu setzen.) Timer hat nur ein Event, Tick. **Klicken Sie auf das Timer**-Symbol im Designer und **klicken Sie dann doppelt auf die Zeile** auf der Events-Seite. Die IDE erstellt nun eine Handler-Methode und registriert sie automatisch bei der Eigenschaft für das Event.

Über den Events-Button im Fenster Eigenschaften können Sie mit den Events für Ihre Steuerelemente arbeiten.

Der untere Bereich des Fensters bietet eine Beschreibung des Events.



Der Timer hat nur das Tick-Event. Klicken Sie doppelt darauf, erstellt die IDE automatisch einen Event-Handler.

3. **Schreiben Sie Code für die Buttons und den Event-Handler.**
Dieser Code wird Sie besser verstehen lassen, wie Timer funktionieren.

```
private void timer1_Tick(object sender, EventArgs e) {
    Console.WriteLine(DateTime.Now.ToString());
}
private void einAusSchalten_Click(object sender, EventArgs e) {
    if (timer1.Enabled)
        timer1.Enabled = false;
    else
        timer1.Enabled = true;
}
private void startTimer_Click(object sender, EventArgs e){
    timer1.Start();
    Console.WriteLine("Enabled = " + timer1.Enabled);
}
private void stopTimer_Click(object sender, EventArgs e) {
    timer1.Stop();
    Console.WriteLine("Enabled = " + timer1.Enabled);
}
```

Diese Anweisung schreibt Datum und Uhrzeit in die Ausgabe. Prüfen Sie im Ausgabefenster, ob das Tick-Event sekündlich (alle 1000 Millisekunden) abgesetzt wird.

Über diese Buttons können Sie mit der Enabled-Eigenschaft und den Methoden Start() und Stop() spielen. Der erste setzt Enabled auf true oder false, und die anderen beiden rufen Start() bzw. Stop() auf.

Die Enabled-Eigenschaft aktiviert bzw. deaktiviert den Timer.

Die Methode Start() startet den Timer und setzt Enabled auf true. Die Methode Stop() stoppt den Timer und setzt Enabled auf false.

Im Hintergrund nutzt Timer ein Delegate

Das Tick-Event ist ein durchschnittlicher, gewöhnlicher Event-Handler genau wie die, die Klicks auf Buttons verarbeiten.

Wie sagen C# und .NET dem Timer, was er bei einem Tick machen soll? Wie sorgt ein Timer-Tick dafür, dass die Methode `timer1_Tick()` ausgeführt wird? Damit sind wir wieder bei **Events** und **Delegates**, über die wir im letzten Kapitel gesprochen haben. Nutzen Sie die »Gehe zu Definition«-Funktion der IDE, damit Sie Ihr Wissen zur Funktionsweise von Event-Handler-Delegates auffrischen können.

4. **Klicken Sie mit rechts auf Ihre timer1-Variable und wählen Sie »Gehe zu Definition«.** »Gehe zu Definition« lässt die IDE automatisch an die Stelle des Codes springen, an der die Variable `timer1` definiert wird. Die IDE führt Sie zu dem Code, den sie erstellt hat, um `timer1` in Form1.Designer.cs als Eigenschaft des Form1-Objekts zu deklarieren. Scrollen Sie in der Datei nach oben, bis Sie auf diese Zeile stoßen:

```
this.timer1.Tick += new System.EventHandler(this.timer1_Tick);
```

Das ist das Tick-Event des Timers. Sie haben es so eingerichtet, dass es alle 1000 Millisekunden eintritt.

Hier ist eins der System-Delegates: der grundlegende Event-Handler. Es ist ein Delegate ... ein Zeiger auf eine oder mehrere Methoden.

Hier ist die Methode, die Sie gerade geschrieben haben, timer1_Tick(). Auf diese soll das Delegate zeigen.

6. **Klicken Sie jetzt rechts auf EventHandler und wählen Sie »Gehe zu Definition«.** Die IDE springt automatisch in den Code, der `EventHandler` definiert. Sehen Sie sich den Namen der neuen Registerkarte an, in der Ihnen der Code gezeigt wird: »EventHandler [aus Metadaten]«. Das bedeutet, dass der Definitionscode für EventHandler nicht in Ihrem Code steckt. Er ist in das .NET Framework eingebaut, und die IDE hatte eine »gefälschte« Codezeile generiert, um Ihnen zu zeigen, wie er dargestellt wird:

```
public delegate void EventHandler(object sender, EventArgs e);
```

Events haben den Typ EventHandler. Das Tick-Event zeigt jetzt auf die Methode time 1_Tick().

Deswegen erwartet in C# jeder Event-Handler einen object- und einen EventArgs-Parameter – das ist die Form des Delegates, das .NET für das Event-Handling definiert.



KOPF-NUSS

Was für Code würden Sie schreiben, um die Laufen()-Methode von Welt in unserer Bienenstock-Simulation alle 500 Millisekunden auszuführen?

Geben Sie dem Simulator einen Timer

Fügen wir dem Simulator einen Timer hinzu. Sie haben bereits ein Timer-Steuerelement, das wahrscheinlich `timer1` heißt. Statt uns von der IDE eine `timer1_Tick()`-Methode generieren zu lassen, können wir den Timer auch manuell mit einem Event-Handler namens FrameAusführen() verknüpfen:

DateTime & TimeSpan

.NET nutzt die Klasse DateTime, um Zeitinformationen zu speichern. Ihre Now-Eigenschaft liefert das aktuelle Datum und die aktuelle Uhrzeit. Den Unterschied zwischen zwei Zeiten kann man mit einem TimeSpan-Objekt ermitteln: Ziehen Sie ein DateTime-Objekt von einem anderen ab, erhalten Sie ein TimeSpan-Objekt mit der Zeitdifferenz.

TimeSpan hat Eigenschaften wie Days, Hours, Seconds und Milliseconds, über die Sie die Zeitdifferenz in unterschiedlichen Einheiten messen können.

```
public partial class Form1 : Form {
  Welt welt;
  private Random zufall = new Random();
  private DateTime start = DateTime.Now;
  private DateTime ende;
  private int gelaufeneFrames = 0;

  public Form1() {
    InitializeComponent();
    welt = new Welt();

    timer1.Interval = 50;
    timer1.Tick += new EventHandler(FrameAusführen);
    timer1.Enabled = false;
    StatistikAktualisieren(new TimeSpan());
  }

  private void StatistikAktualisieren(TimeSpan frameDauer) {
    // Bereits erstellter Code zum Aktualisieren der Statistik.
  }

  public void FrameAusführen(object sender, EventArgs e) {
    gelaufeneFrames++;
    welt.Laufen(zufall);
    ende = DateTime.Now;
    TimeSpan frameDauer = ende - start;
    start = ende;
    StatistikAktualisieren(frameDauer);
  }
}
```

Die Welt-Eigenschaft sollten Sie noch einfügen.

Das werden wir verwenden, um herauszufinden, wie lange der Simulator zu einem bestimmten Zeitpunkt gelaufen ist.

Wir möchten festhalten, wie viele Frames – oder Runden – verstrichen sind.

Eine Sekunde hat 1000 Millisekunden, der Timer tickt also 20-mal pro Sekunde.

Läuft alle 50 Millisekunden.

Wir setzen den Event-Handler auf unsere eigene Methode, FrameAusführen().

Timer beginnt ausgeschaltet.

Außerdem beginnen wir, indem wir die Statistiken mit einer neuen TimeSpan (keine Zeit verstrichen) aktualisieren.

Den Frame-Zähler erhöhen und auf welt.Laufen() aufrufen.

Dann ermitteln wir, wie viel Zeit verstrichen ist, seit der letzte Frame gelaufen ist.

Schließlich aktualisieren wir die Statistiken erneut mit der neuen Zeitdauer.



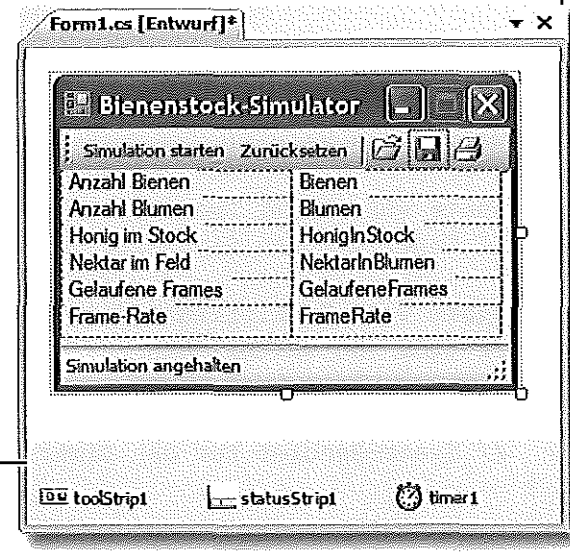
ÜBUNG

Schreiben Sie die Event-Handler für die ToolStrip-Buttons **simulationStarten** und **zurücksetzen**. Das sollten diese Buttons bewirken.

Haben Sie den ToolStrip und den StatusStrip noch nicht aus der Toolbox gezogen, sollten Sie es jetzt tun.

1. Auf dem ersten Button soll »Simulation starten« stehen. Ein Klick darauf sollte die Simulation starten und die Beschriftung des Buttons in ››Simulation anhalten« ändern. Wird die Simulation angehalten, muss auf dem Button »Simulation fortsetzen« stehen.

2. Auf dem zweiten Button soll ››Zurücksetzen« stehen. Durch einen Klick darauf sollte die Welt neu erstellt werden. Ist der Timer aktuell angehalten, sollte der Text auf dem ersten Button wieder in ››Simulation starten« geändert werden.



Auf diese Frage gibt es keine eindeutige Antwort – Sie sollen einfach darüber nachdenken, was wir noch tun müssen.

Spitzen Sie Ihren Bleistift

Was, meinen Sie, muss in dieser Phase des Simulators noch gemacht werden? Versuchen Sie, das Programm auszuführen. Schreiben Sie alles auf, um das wir uns noch kümmern müssen, bevor wir zum grafischen Krempel weitergehen.

.....

.....

.....

.....

.....

.....

.....

Es gibt keine Dummen Fragen

F: Wir haben die ganze Zeit von »Runden« gesprochen, aber jetzt verwenden Sie stattdessen Frames. Was ist der Unterschied?

A: Das ist ein rein begrifflicher Unterschied. Wir befassen uns immer noch mit Runden: kleinen Zeiteinheiten, in denen jedes Objekt in der Welt einmal agieren kann. Aber da wir bald einige Hochleistungsgrafiken einrichten werden, haben wir begonnen, den Begriff Frames zu verwenden, wie es bei grafikintensiven Spielen üblich ist.



Lösungen zu den Übungen

Sie sollten die Event-Handler für die Buttons ››Simulation starten« und »Zurücksetzen« schreiben.




```
public partial class Form1 : Form {
  // Felddeklarationen

  public Form1() {
    InitializeComponent();
    welt = new Welt();
  }
  private void Form1_Load(object sender, tArgs e) {
    // Code, der den Simulator in Gang setzt.
  }
  private void StatistikAktualisieren(TimeSpan frameDauer) {
    // Code von weiter oben zur Aktualisierung der Statistiken.
  }
  public void FrameAusführen(object sender, EventArgs e) {
    // Event-Handler für den Timer.
  }

  private void simulationStarten_Click(object sender, EventArgs e) {
    if (timer1.Enabled) {
      toolStrip1.Items[0].Text = "Simulation fortsetzen";
      timer1.Stop();
    } else {
      toolStrip1.Items[0].Text = "Simulation anhalten";
      timer1.Start();
    }
  }

  private void zurücksetzen_Click(object sender, EventArgs e) {
    gelaufeneFrames = 0;
    welt = new Welt()
    if (!timer1.Enabled)
      toolStrip1.Items[0].Text = "Simulation starten";
  }
}
```

Achten Sie darauf, dass Sie den Formularelementen die entsprechenden Namen geben.

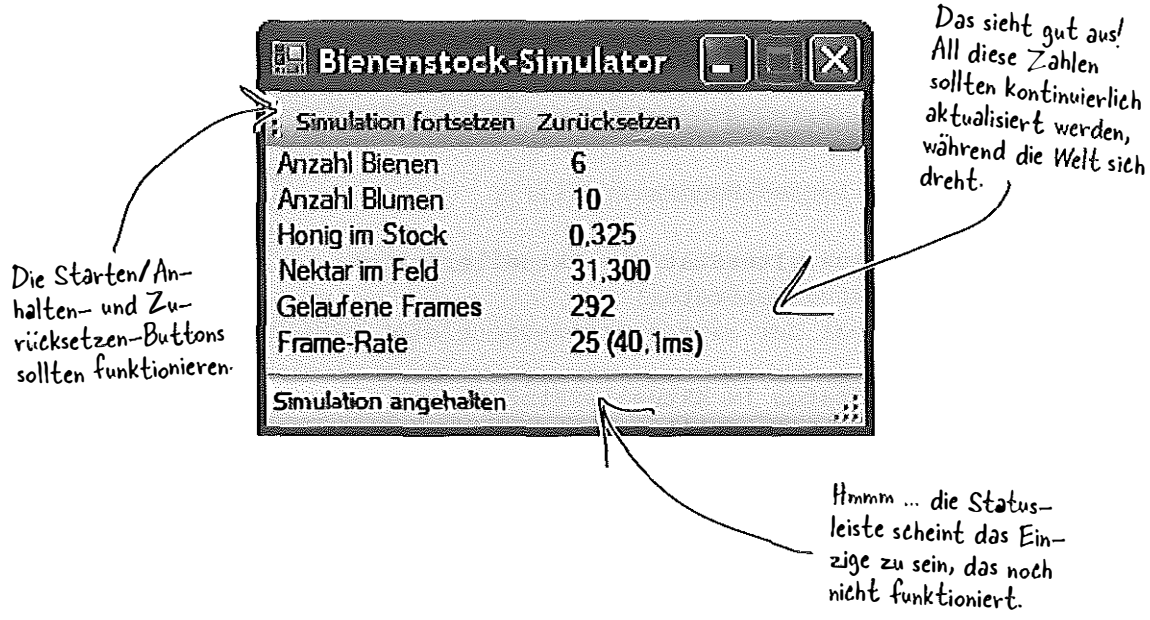
Den Timer umschalten und den Button-Text aktualisieren.

Zum Zurücksetzen des Simulators muss die Welt neu erstellt und die Anzahl von gelaufeneFrames zurückgesetzt werden.

Den Text auf dem ersten Button müssen wir nur ändern, wenn er >>Simulation fortsetzen<< sagt. Sagt er >>Simulation anhalten<<, muss er nicht geändert werden.

Testlauf

Sie haben einen Haufen Arbeit erledigt. Kompilieren Sie Ihren Code, beheben Sie eventuelle Tippfehler und starten Sie den Simulator. Wie sieht es aus?





Hier ist Ihre Möglichkeit, alles zusammenzubringen, was Sie gelernt haben. Wir müssen den Bienen erlauben, dem Simulator mitzuteilen, was sie tun. Wenn sie das tun, sollte der Simulator die Statusmeldungen aktualisieren.

Diesmal sollen Sie nicht nur den meisten Code selbst schreiben, sondern auch herausfinden, wie der zu schreibende Code aussehen soll. Wie können Sie dem Simulator eine Methode geben, die jedes Mal aufgerufen wird, wenn ein Biene ihren Status ändert?

* Gut. Einen weiteren Tipp. Bis auf eine müssen Sie alle Klassen ändern, damit das funktioniert.

Um Ihnen etwas zu helfen, haben wir die Methode geschrieben, die dem Formular hinzugefügt werden muss. Die Klasse Biene sollte diese Methode jedes Mal aufrufen, wenn sich Ihr Status ändert:

```
private void MeldungSenden(int ID, string nachricht) {
  statusStrip1.Items[0].Text = "Biene " + ID + ": " + nachricht;
}
```



Lösungen zu den Übungen

Ihre Aufgabe war es, eine Möglichkeit zu finden, damit die Bienen dem Simulator mitteilen können, was sie gerade tun.

Das haben wir der Klasse Biene hinzugefügt.

Wir nutzen ein Callback, um die Biene-Objekte mit der MeldungSenden()-Methode des Formulars zu verknüpfen.

```
public class Biene {
  // Unser bestehender Code.
  public delegate void BienenMeldung(int ID, string Message);
  public BienenMeldung Melder;

  public void Laufen(Random zufall) {
    Alter++;
    BienenStatus alterStatus = AktuellerStatus;
    switch (AktuellerStatus) {
     // Der Rest der switch-Anweisung bleibt gleich.
    }
    if (alterStatus != AktuellerStatus
        && Melder != null)
            Melder(ID, AktuellerStatus.ToString());
  }
}
```

BienenMeldung ist unser Delegate. Die Signatur entspricht der MeldungSenden()-Methode, die wir für das Formular geschrieben haben.

Wird der Status der Biene geändert, rufen wir die Methode auf, auf die das BienenMeldung-Delegate zeigt, und teilen dieser Methode die Statusänderung mit.

Hier sind die Änderungen, die wir an Stock vorgenommen haben.

```
public class Stock {
  // Unser bestehender Code.
  public Biene.BienenMeldung Melder;

  public Stock(Welt welt, Biene.BienenMeldung Melder) {
    this.Melder = Melder;
    // Der bestehende Konstruktorcode.
  }

  public void NeueBiene(Random zufall) {
    // Bestehender NeueBiene()-Code.
    Biene neueBiene = new Biene(bienenAnzahl, anfangsort, welt, this);
    neueBiene.Melder += this.Melder;
    welt.Bienen.Add(neueBiene);
  }
}
```

Auch der Stock braucht ein Delegate, damit er bei der Erstellung der Biene in NeueBiene() die aufzurufende Methode weiterreichen kann.

NeueBiene() muss jetzt sicherstellen, dass jede Biene die Methode für das Delegate erhält.

Auch an Welt müssen ein paar Dinge geändert werden.

```
public class Welt {
  // Bestehender Code.

  public Welt(Biene.BienenMeldung melder) {
    Bienen = new List<Biene>();
    Blumen = new List<Blume>();
    Stock = new Stock(this, melder);
    Random zufall = new Random();
    for (int i = 0; i < 10; i++)
        NeueBlume(zufall);
  }
}
```

Welt braucht kein eigenes Delegate. Sie reicht die aufzurufende Methode einfach an die Stock-Instanz weiter.

Zum Abschluss das aktualisierte Formular. Alles, was wir nicht zeigen, bleibt gleich.

```
public partial class Form1 : Form {
  // Felddeklarationen

  public Form1() {
    InitializeComponent();
    welt = new Welt(new Biene.BienenMeldung(MeldungSenden));
    // Der restliche Konstruktor von Form1.
  }

  private void zurücksetzen_Click(object sender, EventArgs e) {
    gelaufeneFrames = 0;
    welt = new Welt(new Biene.BienenMeldung(MeldungSenden));
    if (!timer1.Enabled)
        toolStrip1.Items[0].Text = "Simulation starten";
  }

  private void MeldungSenden(int ID, string meldung) {
    statusStrip1.Items[0].Text = "Biene " + ID + ": " + meldung;
  }
}
```

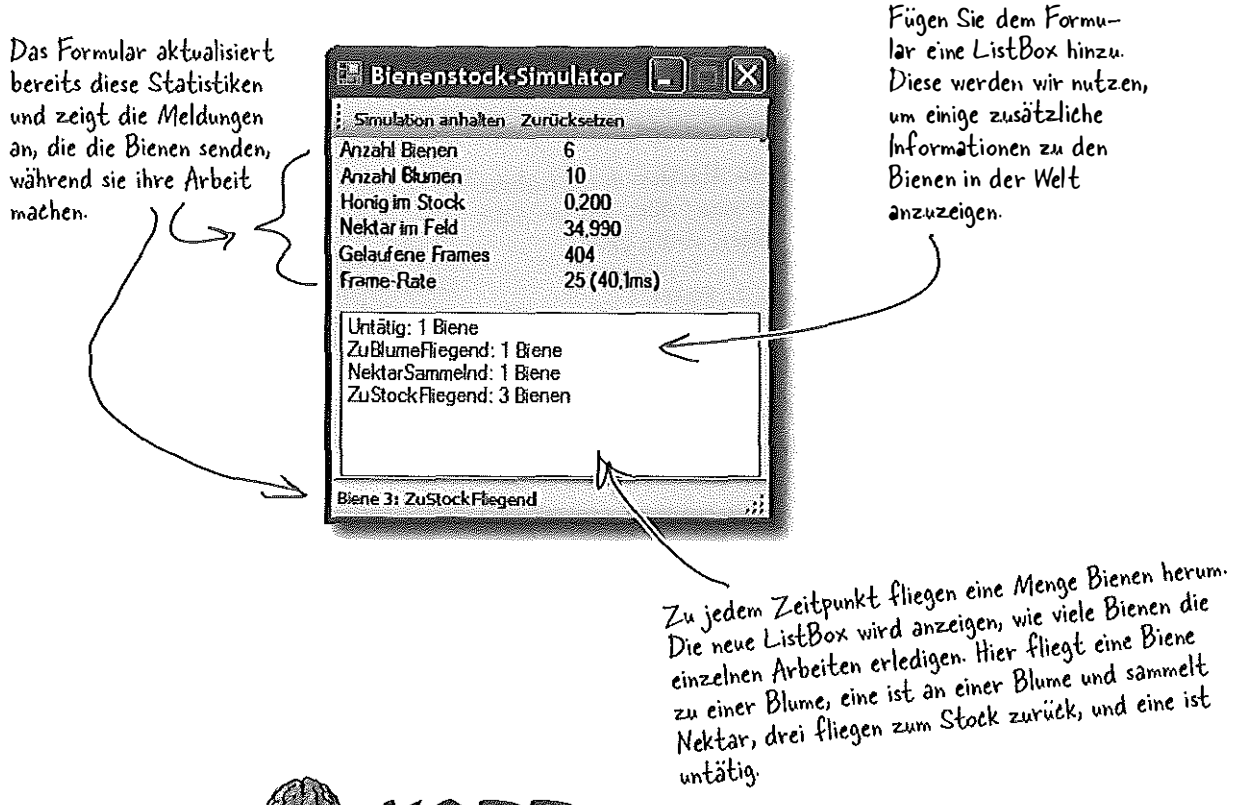
Wir erzeugen auf der Klasse Biene ein neues Delegate (achten Sie darauf, dass Sie BienenMeldung als public deklariert haben) und lassen es auf die Methode MeldungSenden() zeigen.

Hier geschieht das Gleiche ... die Welt wird mit der Methode erzeugt, die die Bienen zurückrufen sollen.

Diese Methode hatten wir Ihnen geliefert ... denken Sie daran, auch diese hinzuzufügen.

Arbeiten wir mit Gruppen von Bienen

Jetzt sollten die Bienen um den Stock und auf dem Feld herumschwirren, und Ihre Simulation sollte laufen! Cool, oder? Aber da wir die grafische Seite des Simulators noch nicht eingerichtet haben – das werden wir im nächsten Kapitel tun –, haben wir bislang nur die Meldungen, die die Bienen über das Callback an das Hauptformular senden. Fügen wir also weitere Informationen darüber hinzu, was die Bienen tun.



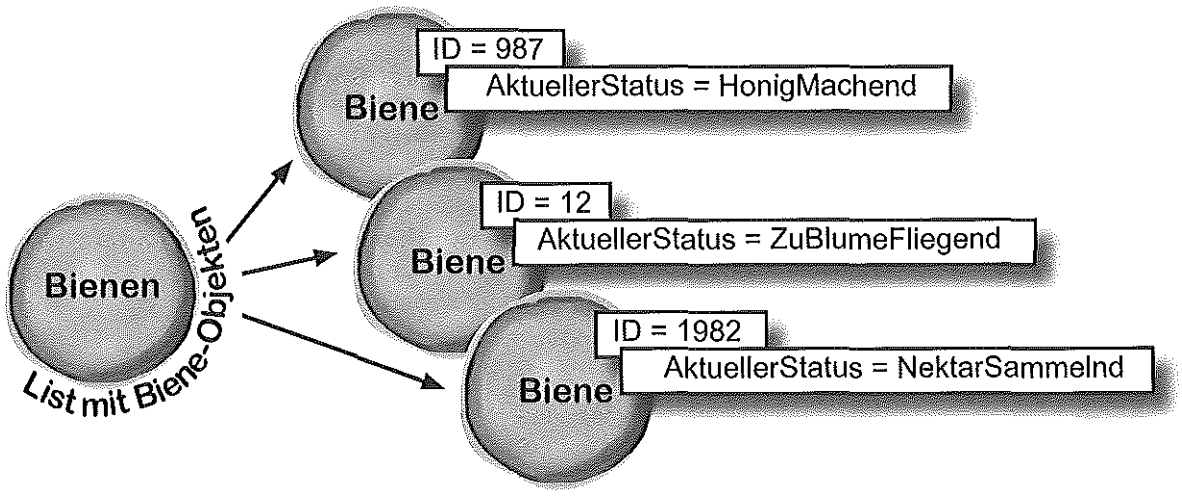


KOPF-NUSS

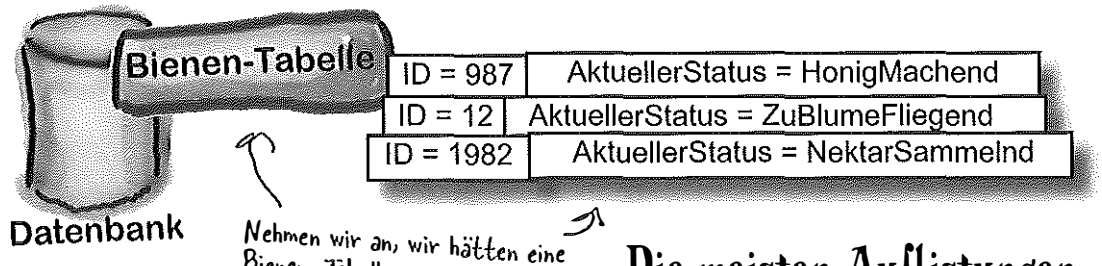
Sie wissen jetzt genug, um die Informationen zu sammeln, die Sie benötigen, um die ListBox zu füllen – nehmen Sie sich einen Augenblick Zeit und überlegen Sie, wie das funktionieren könnte. Aber es ist etwas komplizierter, als es zunächst erscheinen mag. Was bräuchten Sie, um herauszufinden, wie viele Bienen sich in den unterschiedlichen `Biene.Status`-Zuständen befinden?

Eine Auflistung sammelt ... DATEN

Unsere Bienen sind in einer List, einem der .NET-Auflistungstypen, gespeichert. Und eigentlich speichern Auflistungstypen einfach nur Daten ... genau wie eine Datenbank. Jede Biene ist wie eine Zeile mit Daten, mit einem Status, einer ID und so weiter. So sehen unsere Bienen als eine Sammlung von Objekten aus:



In den Feldern der Biene-Objekte sind eine Menge Daten gesammelt. Eine Sammlung von Objekten könnte man sich *beinahe* wie die Zeilen in einer Datenbank vorstellen. Jedes Objekt enthält die Daten in seinen Feldern auf ganz ähnliche Weise, wie die Zeilen einer Datenbank die Daten in ihren Spalten enthalten.



Die meisten Auflistungen – insbesondere wenn sie Objekte enthalten – kann man sich als Datenspeicher vorstellen, die ganz ähnlich wie Datenbanken sind.



Was bringt es, wenn ich mir Auflistungen wie Datenbanken **vorstelle**, wenn ich sie nicht wie Datenbanken **behandeln** kann? Was für eine Zeitverschwendung ...

Was wäre, wenn Sie Collections, Datenbanken und sogar XML-Dokumente mit der gleichen Syntax abfragen könnten?

C# bietet ein sehr nützliches Feature namens **LINQ** (was für **L**angnage **IN**tegrated **Q**uery steht). Der Gedanke hinter LINQ ist, Ihnen eine Möglichkeit zu bieten, die Daten in Arrays, Lists, Stacks, Queues oder anderen Arten von Auflistungen in einer einzigen Operation zu verarbeiten.

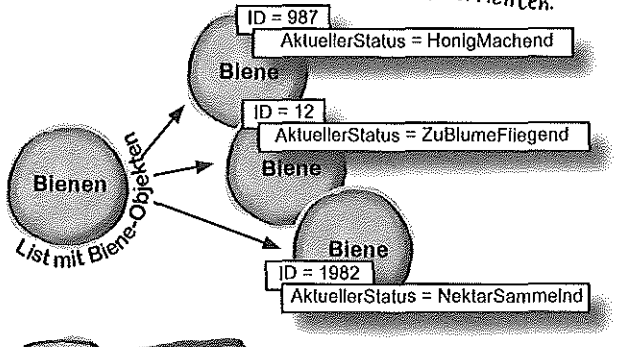
Aber das ganz Besondere an LINQ ist, dass Sie bei der Arbeit mit Auflistungen die gleiche Syntax verwenden können wie bei der Arbeit mit Datenbanken.

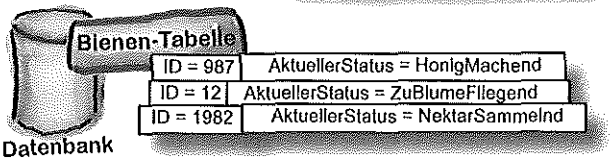
Einen Großteil der Arbeit in Kapitel 15 werden wir mithilfe von LINQ verrichten.

Diese LINQ-Abfrage funktioniert bei Daten einer Auflistung und denen einer Datenbank im Wesentlichen gleich.

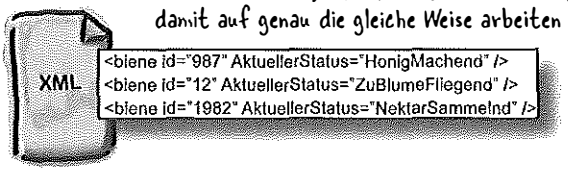
```
var bienenGruppen =
  from biene in welt.Bienen
  group biene by biene.AktuellerStatus
  into bienenGruppe
  orderby bienenGruppe.Key
  select bienenGruppe;
```

LINQ





Hätten wir unsere Bienendaten in einer Datenbank oder einer XML-Datei, könnte LINQ damit auf genau die gleiche Weise arbeiten



LINQ vereinfacht die Arbeit mit den Daten in Auflistungen und Datenbanken

In Kürze werden wir ein vollständiges Kapitel mit LINQ verbringen, können hier aber bereits LINQ und ein Code-Fertiggericht verwenden, um unserem Simulator einige zusätzliche Funktionalitäten hinzuzufügen. Ein Code-Fertiggericht ist Code, den Sie einfach nur eintippen, aber nicht verstehen müssen. Wie er funktioniert, werden Sie in Kapitel 15 lernen.



Code-Fertiggericht

```
private void MeldungSenden(int ID, string meldung) {
    statusStrip1.Items[0].Text = "Biene " + ID + ": " + meldung;
    var bienenGruppen =
      from biene in welt.Bienen
      group biene by biene.AktuellerStatus into bienenGruppe
      orderby bienenGruppe.Key
      select bienenGruppe;
    listBox1.Items.Clear();
    foreach (var gruppe in bienenGruppen) {
        string s;
        if (gruppe.Count() == 1)
            s = "";
        else
            s = "n";
        listBox1.Items.Add(gruppe.Key.ToString() + ": "
                           + gruppe.Count() + " Biene" + s);
        if (gruppe.Key == BienenStatus.Untätig
         && gruppe.Count() == welt.Bienen.Count()
         && gelaufeneFrames > 0) {
            listBox1.Items.Add("Simulation beendet: Alle Bienen untätig");
            toolStrip1.Items[0].Text = "Simulation beendet";
            statusStrip1.Items[0].Text = "Simulation beendet";
            timer1.Enabled = false;
        }
    }
}
```

Das ist eine LINQ-Abfrage. Sie nimmt alle Bienen in der Bienen-List und gruppiert sie über AktuellerStatus.

Der Key einer Gruppe ist der aktuelle Status der Biene. In dieser Reihenfolge werden die Bienen im Formular angezeigt.

Achten Sie darauf, dass die ListBox in Ihrem Formular den gleichen Namen hat.

bienenGruppen kommt aus der LINQ-Abfrage. Wir können die Elemente zählen und über sie iterieren.

Dieser Codeteil sichert, dass der Plural stimmt, d.h. der Text >>1 Biene<< bzw. >>3 Bienen<< lautet.

Fügen Sie abschließend der ListBox den Gruppenstatus (den Schlüssel) und die Anzahl hinzu.

Ein weiteres hübsches Feature. Da wir wissen, wie viele Bienen untätig sind ...

... können wir prüfen, ob ALLE Bienen untätig sind. Ist das der Fall, hat der Stock keinen Honig mehr. Wir sollten die Simulation also beenden.

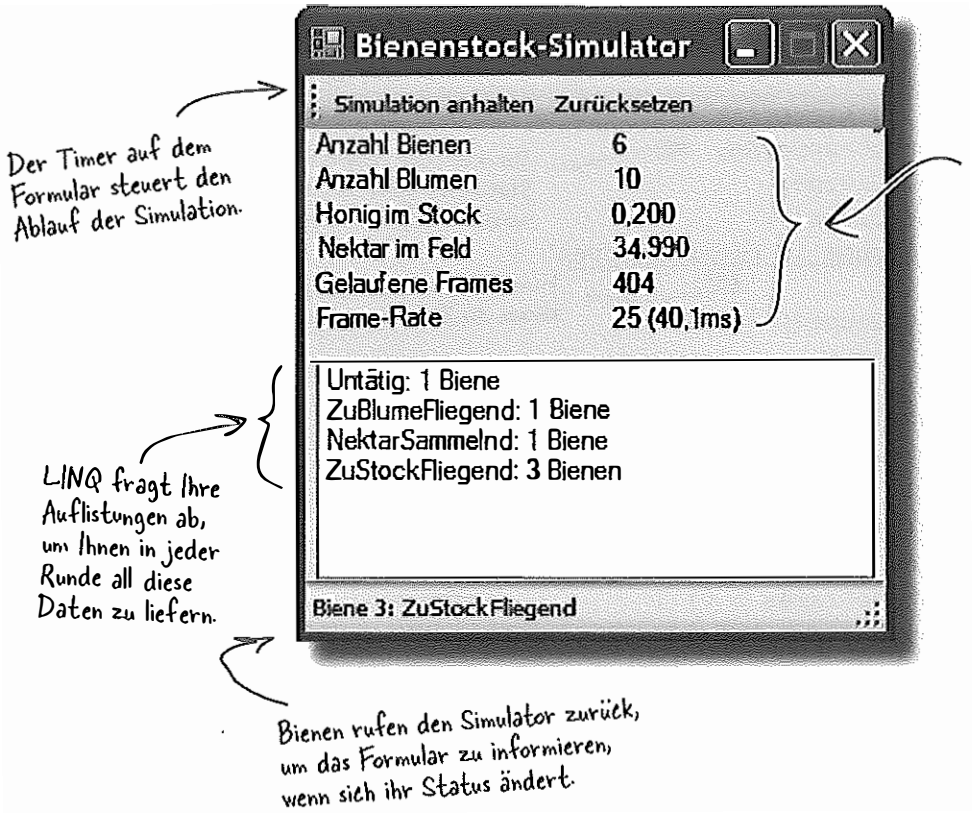


In den folgenden Kapiteln werden Sie noch mehr zu LINQ lernen.

Sie müssen sich die LINQ-Syntax nicht merken und auch nicht versuchen, sich das jetzt alles in den Kopf zu hämmern. In Kapitel 15 werden Sie erheblich mehr Übung in der Arbeit mit LINQ bekommen.

Testlauf (Teil 2)

Legen Sie los. Kompilieren Sie Ihren Code und führen Sie das Projekt aus. Überprüfen Sie die Syntax, wenn Sie Fehler erhalten, insbesondere die des neuen LINQ-Codes. Starten Sie dann den Simulator!



Eine letzte Herausforderung: Öffnen und Speichern

Jetzt sind wir fast bereit, die Grafiken anzugehen und unseren Simulator zu einem grafischen Augenschmaus zu machen. Vorher sollten wir aber noch eine Sache mit dieser Version machen: das Laden, Speichern und Drucken von Bienen-Statistiken ermöglichen.

Fügen Sie den Drucken-Button jetzt hinzu – im nächsten Kapitel werden wir ihn dazu bringen, eine Statusseite für den Stock zu drucken.

1. **Fügen Sie Symbole für Öffnen, Speichern und Drucken hinzu.**
Das ToolStrip-Steuerelement bietet ein sehr nützliches Feature – es kann automatisch grafische Buttons mit Standardsymbolen aufnehmen: Neu, Öffnen, Speichern, Drucken, Ausschneiden, Kopieren, Einfügen und Hilfe. Klicken Sie einfach mit der rechten Maustaste auf das ToolStrip-Symbol unten im Form Designer-Fenster und wählen Sie »Standardelemente einfügen«. Klicken Sie dann auf das erste Element – es ist das »Neu«-Symbol – und löschen Sie es. Behalten Sie die drei folgenden Elemente. Das sind die, die wir brauchen (Öffnen, Speichern und Drucken). Nach diesen folgt ein Trennzeichen. Das können Sie entweder entfernen oder zwischen den Zurücksetzen-Button und den Speichern-Button verschieben. Löschen Sie dann die restlichen Buttons.

2. **Fügen Sie die Event-Handler für die Buttons hinzu.**
Die Standard-Buttons haben die Namen openToolStripButton, saveToolStripButton und printToolStripButton. Klicken Sie doppelt auf sie, um die Event-Handler hinzuzufügen.



Fügen Sie den Code hinzu, der die Buttons Speichern und Öffnen funktionsfähig macht.

1. **Lassen Sie den Speichern-Button die Welt in eine Datei serialisieren.** Halten Sie den Timer an (Sie können ihn nach dem Speichern neu starten). Setzen Sie **`Melder`** für den Stock und alle Bienen auf **`null`**, damit .NET nicht versucht, den Code zu serialisieren, auf den diese Delegates zeigen: auf den Code des Formulars selbst. Zeigen Sie einen Speichern-Dialog an und serialisieren Sie dann das `Welt`-Objekt und die Anzahl an Frames, die ausgeführt wurden. Vergessen Sie nicht, `MeldungSenden()` wieder anzubinden, wenn Sie fertig sind.

2. **Lassen Sie den Öffnen-Button die Welt aus einer Datei serialisieren.** Kümmern Sie sich genau wie im Speichern-Button um den Timer, öffnen Sie einen Öffnen-Dialog und deserialisieren Sie die Welt und die Anzahl an Frames aus der ausgewählten Datei. Dann können Sie die `Melder`-Delegates wieder verbinden und den Timer neu starten (falls das erforderlich ist).

3. **Vergessen Sie das Exception-Handling nicht!** Prüfen Sie, ob die Welt intakt ist, wenn es ein Problem beim Lesen oder Schreiben der Datei gibt. Überlegen Sie, ob Sie eine lesbare Fehlermeldung anzeigen lassen wollen, wenn etwas schiefläuft.



LÖSUNGEN ZU DEN ÜBUNGEN

Sie sollten die Speichern- und Öffnen-Buttons funktionsfähig machen.

Vergessen Sie diese zusätzlichen using-Anweisungen nicht.

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

Sie müssen die Klassen Welt, Stock, Blume und Biene serialisierbar machen. Bei der Serialisierung von Welt findet .NET Referenzen auf Stock-, Blume- und Biene-Objekte und serialisiert auch diese.

```
[Serializable]
public class Welt {
[Serializable]
public class Stock {
[Serializable]
public class Blume {
[Serializable]
public class Biene {
```

Hier ist der Code für den Speichern-Button.

```
private void saveToolStripButton_Click(object sender, EventArgs e) {
    bool aktiviert = timer1.Enabled;
    if (aktiviert)
        timer1.Stop();
    welt.Stock.Melder = null;
    foreach (Biene biene in welt.Bienen)
        biene.Melder = null;
```

Setzen wir die Melder-Delegates nicht auf null, dienen sie bei der Serialisierung der Welt als Referenzen auf Form1. Der BinaryFormatter versucht dann, unser Formular zu serialisieren.

```
    SaveFileDialog speichernDialog = new SaveFileDialog();
    speichernDialog.Filter = "Simulator-Dateien (*.bienen)|*.bienen";
    speichernDialog.CheckPathExists = true;
    speichernDialog.Title = "Datei zum Speichern der aktuellen Simulation wählen";
    if (speichernDialog.ShowDialog() == DialogResult.OK) {
        try {
            BinaryFormatter bf = new BinaryFormatter();
            using (Stream ausgabe = File.OpenWrite(speichernDialog.FileName)) {
                bf.Serialize(ausgabe, welt);
                bf.Serialize(ausgabe, gelaufeneFrames);
            }
        }
        catch (Exception ex) {
            MessageBox.Show("Konnte Simulator-Datei nicht speichern\r\n" + ex.Message,
              "Bienen-Simulator-Fehler", MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
    }
    welt.Stock.Melder = new Biene.BienenMeldung(MeldungSenden);
    foreach (Biene biene in welt.Bienen)
        biene.Melder = new Biene.BienenMeldung(MeldungSenden);
    if (aktiviert)
        timer1.Start();
}
```

Als Dateinamenserweiterung für die Simulator-Speicherdateien haben wir >>.bienen<< gewählt.

Hier wird die Welt in eine Datei geschrieben.

Denken Sie daran, dass bei der Serialisierung der Welt alle Referenzen serialisiert werden ... alle Bienen, alle Blumen und der Stock.

Nachdem wir die Datei gespeichert haben, müssen wir die Melder-Delegates wieder anbinden und dann den Timer erneut starten (wenn wir ihn angehalten haben).

Hier ist der Code für den Öffnen-Button.

```
private void openToolStripButton_Click(object sender, EventArgs e) {
    Welt aktuelleWelt = welt;
    int aktuelleFrames = gelaufeneFrames;

    bool aktiviert = timer1.Enabled;
    if (aktiviert)
        timer1.Stop();

    OpenFileDialog öffnenDialog = new OpenFileDialog();
    öffnenDialog.Filter = "Simulator-Dateien (*.bienen)|*.bienen";
    öffnenDialog.CheckPathExists = true;
    öffnenDialog.CheckFileExists = true;
    öffnenDialog.Title = "Datei zum Laden der Simulation wählen";
    if (öffnenDialog.ShowDialog() == DialogResult.OK) {
        try {
            BinaryFormatter bf = new BinaryFormatter();
            using (Stream eingabe = File.OpenRead(öffnenDialog.FileName)) {
                welt = (Welt)bf.Deserialize(eingabe);
                gelaufeneFrames = (int)bf.Deserialize(eingabe);
            }
        }
        catch (Exception ex) {
      MessageBox.Show("Konnte Simulator-Datei nicht lesen\r\n" + ex.Message,
      "Bienen-Simulator-Fehler", MessageBoxButtons.OK, MessageBoxIcon.Error);
            welt = aktuelleWelt;
            gelaufeneFrames = aktuelleFrames;
        }
    }

    welt.Stock.Melder = new Biene.BienenMeldung(MeldungSenden);
    foreach (Biene biene in welt.Bienen)
        biene.Melder = new Biene.BienenMeldung(MeldungSenden);
    if (aktiviert)
        timer1.Start();
}
```

Bevor wir die Datei öffnen und aus ihr lesen, speichern wir eine Referenz auf die aktuelle Welt und die in ihr gelaufenen Frames. Gibt es ein Problem, können wir zu diesen zurückkehren und den Simulator weiterlaufen lassen.

Richten Sie den Datei öffnen-Dialog ein und blenden Sie ihn ein.

using sorgt dafür, dass der Stream geschlossen wird.

Hier deserialisieren wir die Welt und die Anzahl von Frames aus der Datei.

Löst die Dateioperation eine Exception aus, stellen wir die aktuelle Welt und die aktuell gelaufenen Frames wieder her.

Wurde alles geladen, müssen wir die Delegates wieder anknüpfen und den Timer neu starten.

Sie müssen Ihren Simulator ans Laufen bringen, bevor Sie zum nächsten Kapitel weitergehen. Eine funktionierende Version können Sie unter http://examples.oreilly.de/german_examples/hfcsharpger herunterladen.

13 Steuerelemente und Grafik

Machen Sie es ansehnlich



Manchmal müssen Sie die Grafik in die eigenen Hände nehmen.

Lange haben wir uns darauf verlassen, dass sich Steuerelemente um alles Sichtbare in unseren Anwendungen kümmern. Aber manchmal reicht das nicht – beispielsweise wenn Sie ein **Bild animieren** möchten. Und wenn Sie einmal mit Animationen beginnen, werden Sie bald **eigene Steuerelemente** für Ihre .NET-Programme **erstellen**, denen Sie vielleicht etwas **Double-Buffering** hinzufügen, oder sogar **direkt auf Ihr Formular malen**. Alles beginnt mit dem **Graphics**-Objekt, **Bitmaps** und dem Willen, sich mit dem grafischen Status quo nicht zufriedenzugeben.

Sie haben die ganze Zeit Steuerelemente für die Interaktion mit Ihren Programmen verwendet

`TextBox`, `PictureBox`, `Label` ... mittlerweile wissen Sie ganz gut, was Sie mit diesen Steuerelementen aus der Toolbox der IDE machen können. Aber was wissen Sie ***wirklich*** über sie? An Steuerelementen ist mehr dran, als lediglich Symbole auf Formulare zu ziehen.



Sie können eigene Steuerelemente erstellen.
Die Steuerelemente in der Toolbox sind sehr nützlich, um Formulare und Anwendungen aufzubauen, aber sie haben nichts Magisches. Es sind einfach Klassen, genau wie die Klassen, die Sie selbst geschrieben haben. Und C# macht es Ihnen leicht, eigene Steuerelemente zu erstellen – Sie müssen nur die richtige Basisklasse erweitern.



Eigene Steuerelemente werden ebenfalls in der Toolbox angezeigt.
Auch die Toolbox der IDE birgt kein Geheimnis. Sie sucht einfach in den Klassen Ihres Projekts und den eingebauten .NET-Klassen nach Steuerelementen. Findet sie ein Steuerelement – eine Klasse, die die erforderliche Schnittstelle implementiert –, zeigt sie ein Symbol für die Klasse. Schreiben Sie eigene Steuerelemente, werden auch diese in der Toolbox angezeigt.

Erstellen Sie eine Klasse, die von einer beliebigen der bestehenden Steuerelement-Klassen erbt, wird diese automatisch in der Toolbox angezeigt – selbst wenn sie keinerlei zusätzlichen Code enthält.

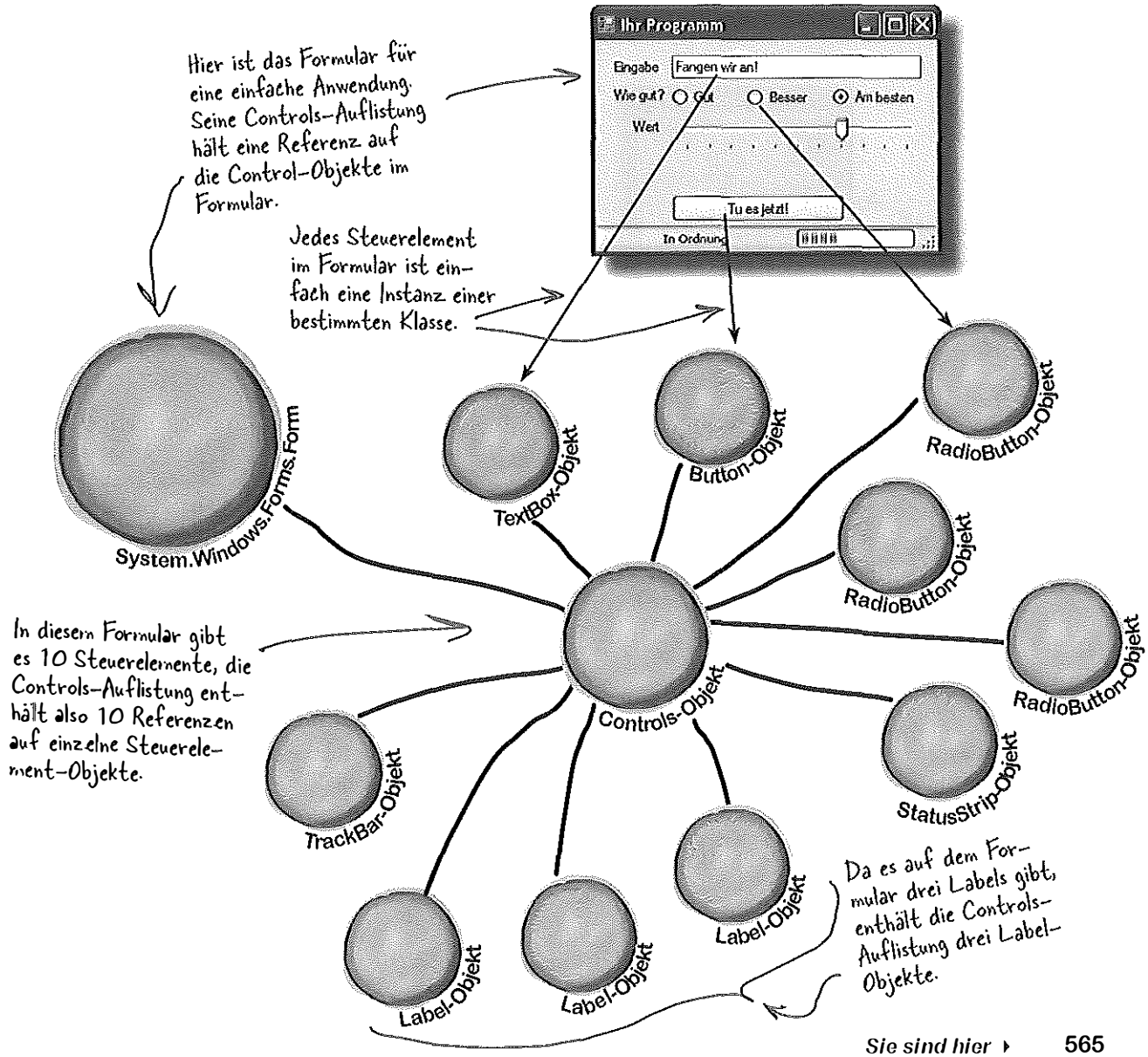


Sie können Code schreiben, der, während das Programm läuft, neue Steuerelemente in das Formular einfügt oder alte aus ihm entfernt.
Bloß weil Sie ein Formular im Form-Designer der IDE gestalten, heißt das nicht, dass es immer so bleiben muss. Sie haben bereits einige `PictureBox`-Steuerelemente herumbewegt (beispielsweise als Sie das Hunderennen erstellt haben). Aber Sie können Steuerelemente auch hinzufügen oder entfernen. Wenn Sie in der IDE ein Programm erstellen, macht auch sie eigentlich nichts anderes: Die IDE schreibt den Code, der dem Formular Steuerelemente hinzufügt. Und das bedeutet, dass Sie ähnlichen Code schreiben und zu beliebiger Zeit ausführen können.

Formular-Steuerelemente sind auch nur Objekte

Sie wissen bereits, wie wichtig **Steuerelemente** für Ihre Formulare sind. Seit Kapitel 1 haben Sie Buttons, Textfelder, PictureBoxen, CheckBoxen, GroupBoxen und Labels eingesetzt. Und es stellt sich heraus, dass diese Steuerelemente auch nur Objekte sind, genau wie alles andere, mit dem Sie gearbeitet haben.

Ein Steuerelement ist ein Objekt – genau wie jedes andere Objekt –, es weiß zusätzlich nur, wie es sich selbst zeichnen muss. Das `Form`-Objekt hält seine Steuerelemente in einer besonderen Auflistung namens **`Controls`** nach, die Sie verwenden können, um in Ihrem eigenen Code Steuerelemente hinzuzufügen oder Steuerelemente daraus zu entfernen.

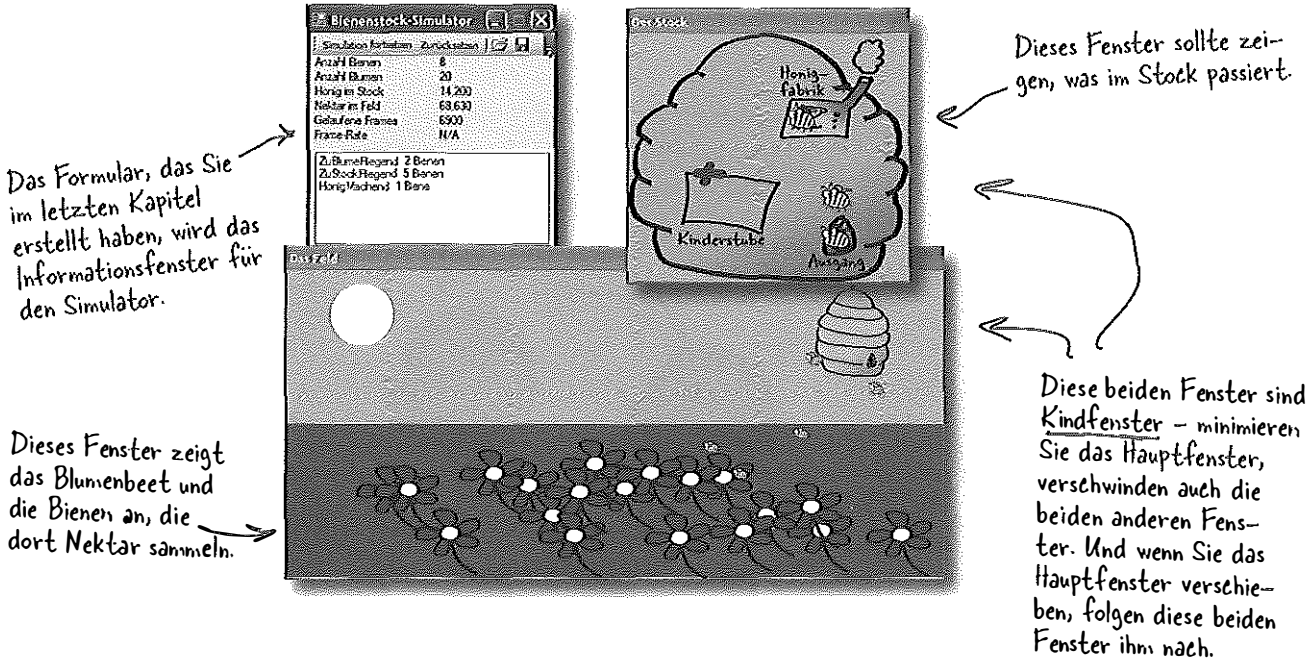


Den Bienenstock-Simulator mit Steuerelementen animieren

Sie haben einen wunderbaren Simulator erstellt, der aber noch nicht sonderlich ansehnlich ist. Es ist Zeit, eine richtig beeindruckende Visualisierung zu erstellen, die die Bienen in Aktion zeigt. Sie werden einen Renderer erstellen, der den Bienenstock animiert … und Steuerelemente sind der Schlüssel dazu.

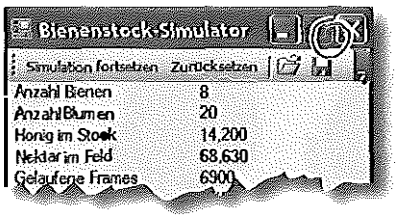
1 Die Benutzerschnittstelle zeigt Ihnen alles, was passiert.

Ihr Simulator hat drei separate Fenster. Das Hauptfenster mit den informativen Statistiken, das Daten zur aktuellen Simulation und die Meldungen der Bienen anzeigt, haben Sie bereits. Jetzt werden Sie ein Fenster erstellen, das anzeigt, was im Stock passiert, und ein weiteres Fenster, das das Blumenbeet zeigt, in dem die Bienen Nektar sammeln.



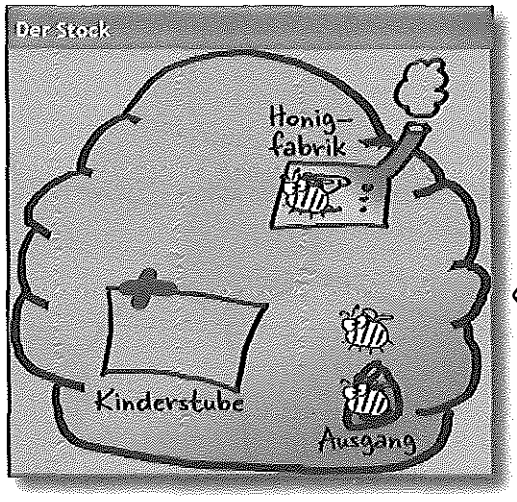
2 Wir fügen dem Statistik-Fenster einen Drucken-Button hinzu.

Das Statistik-Fenster hat bereits einen Speichern- und einen Öffnen-Button. Nachdem wir die Grafiken hinzugefügt haben, erhält die Werkzeugleiste einen Button, über den eine Informationsseite zu den Geschehnissen ausgedruckt werden kann.



3 Das Stock-Fenster zeigt, was im Stock passiert.

Sie müssen die durch die Welt fliegenden Bienen animieren. Manchmal sind sie im Stock. Wenn ja, sind sie in diesem Fenster zu sehen.



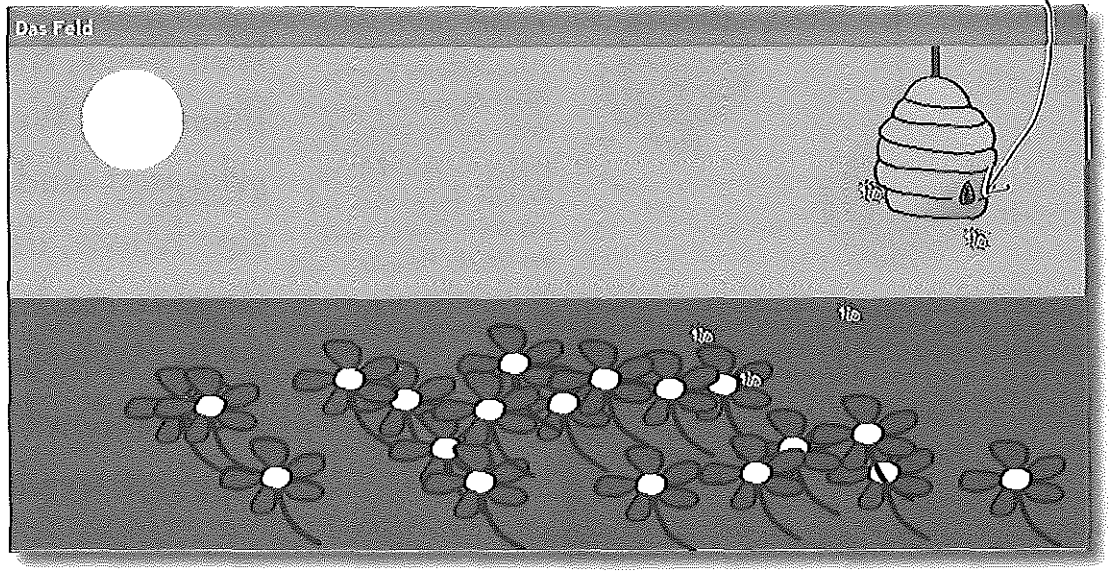
Der Stock enthält drei wichtige Orte. Die Bienen werden in der Kinderstube geboren, müssen zum Ausgang fliegen, um den Stock zu verlassen und bei den Blumen Nektar zu sammeln, und in die Honigfabrik gehen, um Honig zu machen, wenn sie in den Stock zurückkehren.

Der Ausgang des Bienenstocks ist auf dem Stock-Formular, der Eingang auf dem Feld-Formular. (Deswegen geben wir beide im orte-Dictionary des Stocks an.)

4 Im Feld-Fenster sammeln die Bienen Nektar.

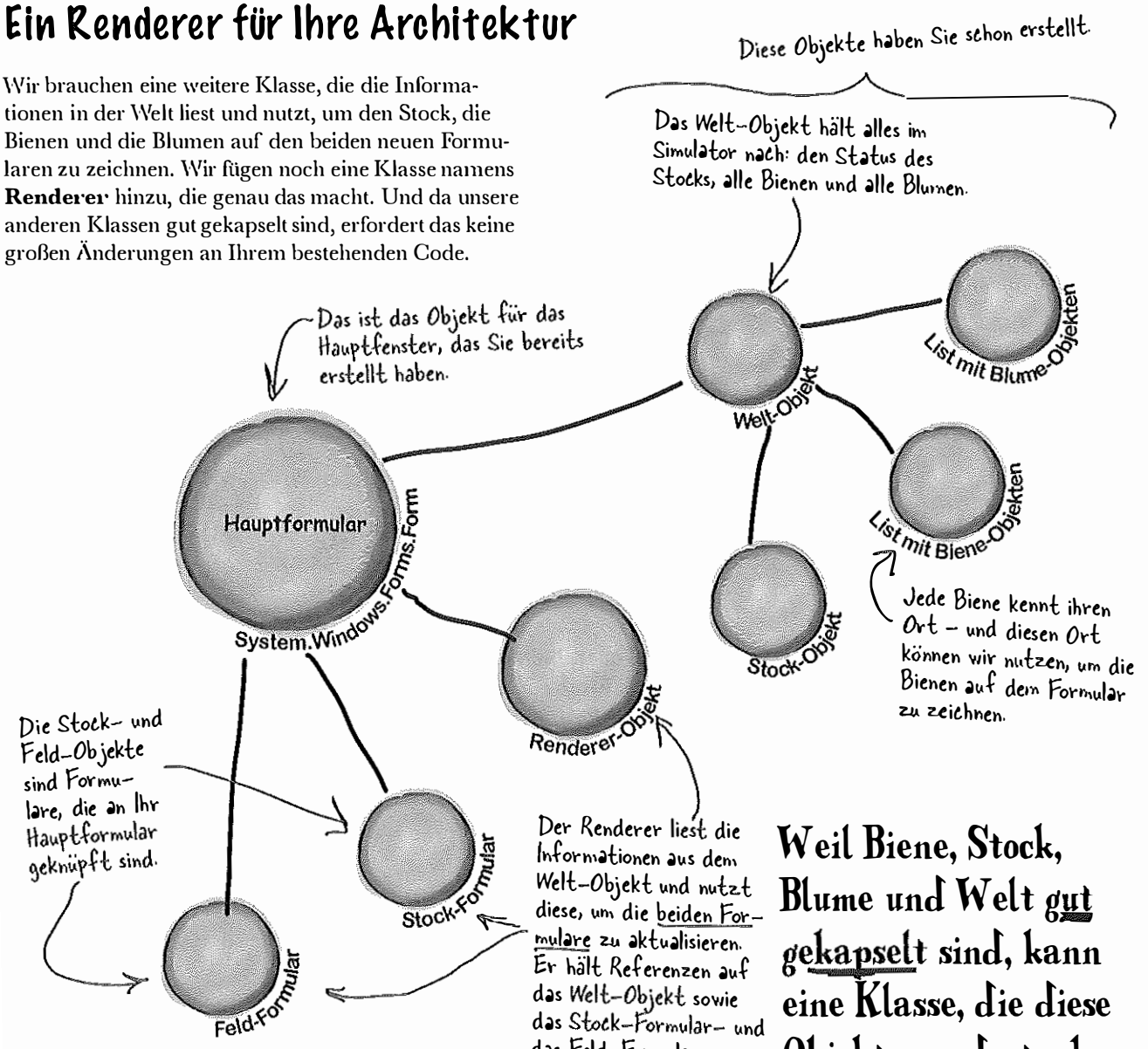
Bienen haben eine wichtige Aufgabe: in Blumen Nektar sammeln und diesen in den Stock zurückbringen, um Honig zu machen. Dann essen sie den Honig, damit sie genug Energie haben, um wieder hinauszufliegen und weiteren Nektar zu sammeln.

Hier ist der Eingang zum Stock. Fliegen Bienen dort hinein, verschwinden sie aus dem Feld-Formular und erscheinen in der Nähe des Ausgangs im Stock-Formular.



Ein Renderer für Ihre Architektur

Wir brauchen eine weitere Klasse, die die Informationen in der Welt liest und nutzt, um den Stock, die Bienen und die Blumen auf den beiden neuen Formularen zu zeichnen. Wir fügen noch eine Klasse namens **Renderer** hinzu, die genau das macht. Und da unsere anderen Klassen gut gekapselt sind, erfordert das keine großen Änderungen an Ihrem bestehenden Code.

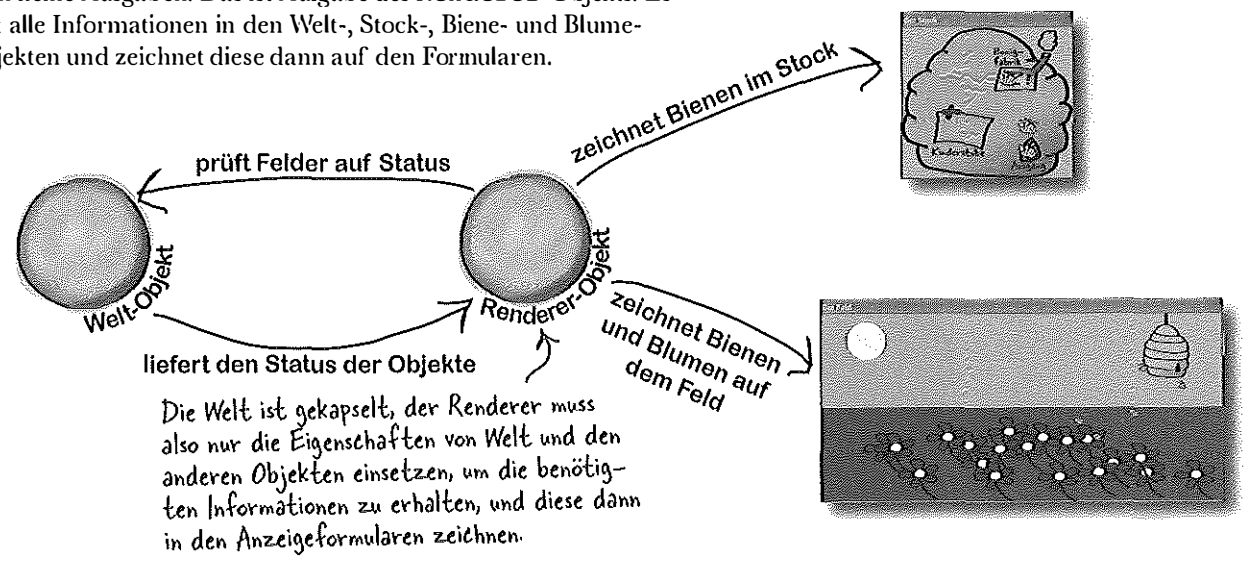


ren-dern, Verb.
Daten grafisch wiedergeben. *Wir fütterten den Rechner mit neuen Daten für den Kurvenverlauf und ließen ihn das Modell neu* ***rendern***.

Weil Biene, Stock, Blume und Welt gut gekapselt sind, kann eine Klasse, die diese Objekte rendert, ohne große Änderungen am bestehenden Code hinzugefügt werden

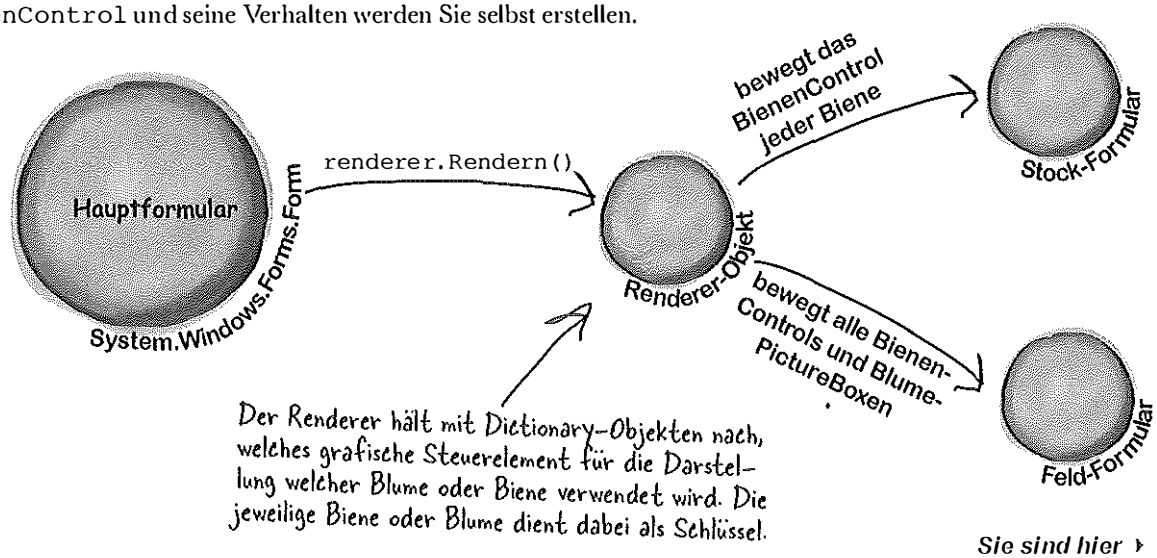
Der Renderer zeichnet alles in der Welt auf den beiden Formularen

Das `Welt`-Objekt hält alles in der Simulation nach: den Stock, die Bienen und die Blumen. Aber selbst zeichnet es nichts und erzeugt auch keine Ausgaben. Das ist Aufgabe des `Renderer`-Objekts. Es liest alle Informationen in den Welt-, Stock-, Biene- und Blume-Objekten und zeichnet diese dann auf den Formularen.



Der Simulator zeichnet die Welt nach jedem Frame

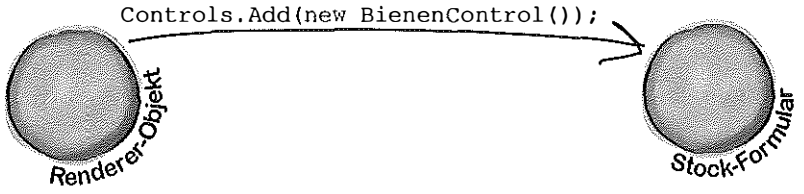
Nachdem das Hauptformular die `Laufen()`-Methode der Welt aufgerufen hat, sollte es die `Rendern()`-Methode des Renderers aufrufen, um die Anzeigefenster neu zu zeichnen. Beispielsweise wird für jede Blume eine `PictureBox` verwendet. Aber bei den Bienen möchten wir einen Schritt weitergehen und ein animiertes Steuerelement erstellen. Dieses neue Steuerelement namens `BienenControl` und seine Verhalten werden Sie selbst erstellen.



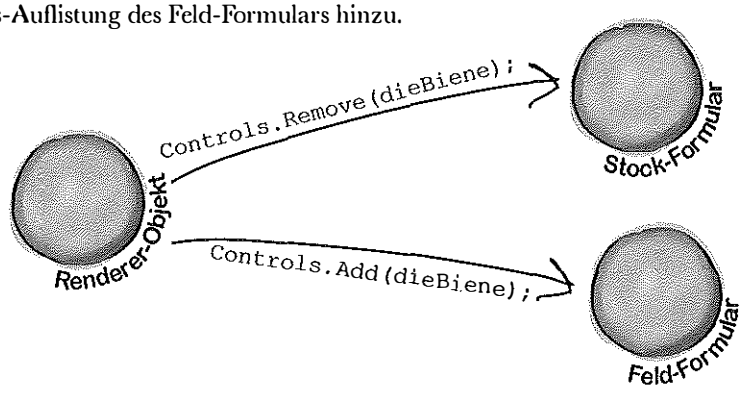
Controls sind für grafisch angezeigte Elemente gut geeignet

Wird dem Stock eine neue Biene hinzugefügt, soll unser Simulator dem Stock-Formular ein neues `BienenControl` hinzufügen und dessen Ort ändern, während die Biene durch die Welt schwirrt. Fliegt die Biene aus dem Stock heraus, muss der Simulator das Steuerelement aus dem Stock-Formular entfernen und dem Feld-Formular hinzufügen. Fliegt sie mit Massen von Nektar in den Stock zurück, muss das Steuerelement aus dem Feld-Formular entfernt und wieder dem Stock-Formular hinzugefügt werden. Dabei soll das animierte Bienenbild auch noch mit den Flügeln schwirren ... und mit Steuerelementen können Sie all das tatsächlich leicht erreichen.

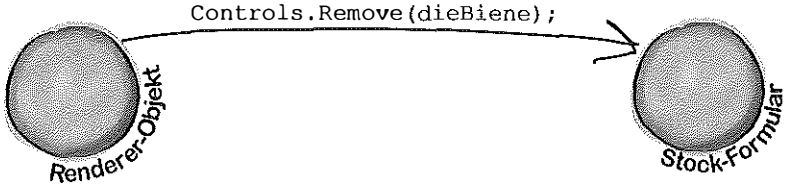
1. Die Welt fügt eine neue Biene hinzu, und der Renderer erzeugt ein neues `BienenControl` und fügt es der `Controls`-Auflistung des Stock-Formulars hinzu.



2. Fliegt die Biene aus dem Stock heraus aufs Feld, entfernt der Renderer das `BienenControl` aus der `Controls`-Auflistung des Stocks und fügt es der `Controls`-Auflistung des Feld-Formulars hinzu.



3. Eine Biene geht in Rente, wenn sie untätig und zu alt ist. Stellt der Renderer beim Prüfen der Bienen-Liste der Welt fest, dass eine Biene nicht mehr vorhanden ist, entfernt er das Steuerelement aus dem Stock-Formular.



Spitzen Sie Ihren Bleistift

Versuchen Sie herauszufinden, was die einzelnen Codeschnipsel machen. Gehen Sie davon aus, dass sich die Schnipsel in einem Formular befinden, und schreiben Sie auf, was Sie denken.

```
this.Controls.Add(new Button());
```

```
Form2 kindfenster = new Form2();
kindfenster.BackgroundImage =
          Properties.Resources.Mosaic;
kindfenster.BackgroundImageLayout =
          ImageLayout.Tile;
kindfenster.Show();
```

Enthält Ihr Formular eine ListBox, können Sie die Methode AddRange() verwenden, um ihr eine Liste mit Elementen hinzuzufügen.

```
Label einLabel = new Label();
einLabel.Text = "Welche Tiere mögen Sie?";
einLabel.Location = new Point(10, 10);
ListBox eineListe = new ListBox();
eineListe.Items.AddRange( new object[]
   { "Katze", "Hund", "Fisch", "Keine" } );
eineListe.Location = new Point(10, 40);
Controls.Add(einLabel);
Controls.Add(eineListe);
```

Sie sollen nicht die einzelnen Zeilen kommentieren, sondern zusammenfassen, was in den einzelnen Codeblöcken passiert.

```
Label zuEntfernendesControl = null;
foreach (Control control in Controls) {
    if (control is Label
          && control.Text == "Bobby")
        zuEntfernendesControl = control as
                    Label;
}
Controls.Remove(zuEntfernendesControl);
zuEntfernendesControl.Dispose();
```

Bonusfrage: Warum haben wir Ihrer Meinung nach Controls.Remove() nicht in die foreach-Schleife gepackt?

Wenn Sie möchten, können Sie es ausprobieren und aufschreiben, warum Sie wohl das Ergebnis erhalten haben, das .NET Ihnen lieferte.

Spitzen Sie Ihren Bleistift
Lösung

Versuchen Sie herauszufinden, was die einzelnen Codeschnipsel machen. Gehen Sie davon aus, dass sich die Schnipsel in einem Formular befinden, und schreiben Sie auf, was Sie denken.

```
this.Controls.Add(new Button());
```

Erstellt einen neuen Button und fügt ihn dem Formular hinzu. Er erhält die Default-Werte (z.B. ist die Eigenschaft Text leer).

```
Form2 kindfenster = new Form2();
kindfenster.BackgroundImage =
          Properties.Resources.Mosaic;
kindfenster.BackgroundImageLayout =
          ImageLayout.Tile;
kindfenster.Show();
```

Die Anwendung hat ein zweites Formular namens Form2. Dieses wird hier erstellt. Dann wird sein Hintergrundbild auf die Ressourcengrafik >>Mosaic<< und der Darstellungsmodus für den Hintergrund auf gekachelt, nicht gestreckt gesetzt. Schließlich wird es dem Benutzer angezeigt.

```
Label einLabel = new Label();
einLabel.Text = "Welche Tiere mögen Sie?";
einLabel.Location = new Point(10, 10);
ListBox eineListe = new ListBox();
eineListe.Items.AddRange( new object[]
       { "Katze", "Hund", "Fisch", "Keine" } );
eineListe.Location = new Point(10, 40);
Controls.Add(einLabel);
Controls.Add(eineListe);
```

Erstellt ein neues Label, setzt seinen Text und verschiebt es an eine neue Position. Erstellt dann eine neue ListBox, der vier Elemente hinzugefügt werden, und positioniert sie unter dem Label. Label und ListBox werden dem Formular hinzugefügt, damit beide sofort angezeigt werden.

```
Label zuEntfernendesControl = null;
foreach (Control control in Controls) {
    if (control is Label
          && control.Text == "Bobby")
        zuEntfernendesControl = control as
                    Label;
}
Controls.Remove(zuEntfernendesControl);
zuEntfernendesControl.Dispose();
```

Die Schleife durchläuft alle Steuerelemente im Formular, bis sie ein Label mit dem Text >>Bobby<< findet. Findet sie eins, entfernt sie es aus dem Formular.

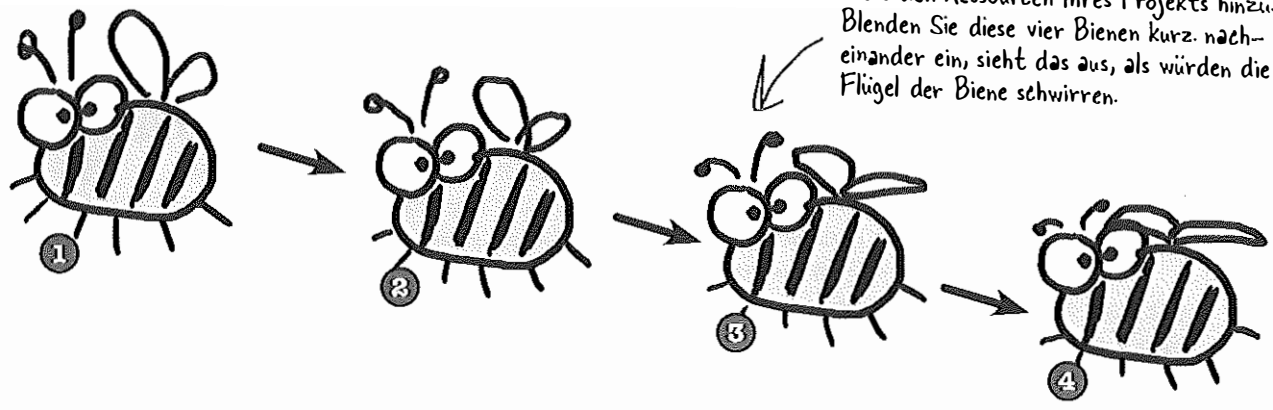
Versuchen Sie das, löst .NET eine Exception aus. Die Auflistung muss intakt bleiben, da .NET sonst die Position verliert und unvorhersagbare Ergebnisse liefert. Deswegen könnte man hier stattdessen eine for-Schleife verwenden.

Bonusfrage: Warum haben wir Ihrer Meinung nach Controls.Remove() nicht in die foreach-Schleife gepackt?

Die Auflistung Controls kann (wie jede andere Auflistung) nicht innerhalb einer foreach-Schleife verändert werden, die über sie iteriert.

Ihr erstes animiertes Steuerelement erstellen

Sie müssen **ein eigenes Steuerelement erstellen**, das ein animiertes Bienenbild zeichnet. Falls Sie noch nie eine Animation erstellt haben: Das ist nicht so schwer, wie es klingt. Sie zeichnen nacheinander eine Folge von Bildern und erzeugen damit die Illusion von Bewegung. Glücklicherweise erleichtert die Art, wie C# und .NET mit Ressourcen umgehen, die Umsetzung von Animationen.

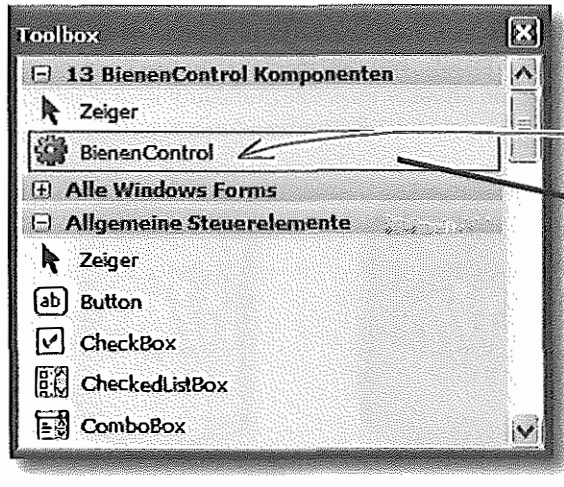


Haben Sie die vier Bilder der Bienen-Animation von der Von Kopf bis Fuß-Seite heruntergeladen (Bee animation 1.png bis Bee animation 4.png), fügen Sie diese den Ressourcen Ihres Projekts hinzu. Blenden Sie diese vier Bienen kurz nacheinander ein, sieht das aus, als würden die Flügel der Biene schwirren.

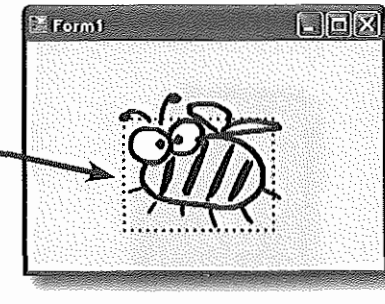
Wir möchten ein Control in der Toolbox

Erstellen Sie das `BienenControl` richtig, erscheint es als Steuerelement, das Sie aus der Toolbox auf Ihr Formular ziehen können. Es sieht genau so aus wie eine `PictureBox` mit dem Bild einer Biene, hat aber animierte, flatternde Flügel.

Laden Sie die Grafiken für dieses Kapitel von der Von Kopf bis Fuß-Website herunter: http://examples.oreilly.de/german_examples/hfcsharpger



Wenn wir die richtigen Klassen erweitern, kümmert sich .NET darum, dass unser Steuerelement in der Toolbox der IDE angezeigt wird.



Es ist wie eine PictureBox, aber das Bild ist bereits gesetzt, und es gibt eine Animation, die wir erstellen werden. Können Sie sich denken, welche Klasse BienenControl erweitert?

BienenControl ist WIE eine PictureBox ... beginnen wir also damit, PictureBox zu ERWEITERN

Da jedes Steuerelement in der Toolbox einfach ein Objekt ist, kann man leicht neue Steuerelemente erstellen. Sie müssen Ihrem Projekt einfach nur eine Klasse hinzufügen, die ein bestehendes Steuerelement erweitert, und ihr die Verhalten geben, die das neue Steuerelement haben soll.

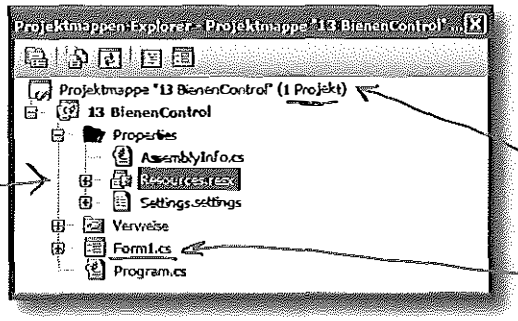
Wir möchten ein Steuerelement – nennen wir es **BienenControl** –, das ein animiertes Bild einer Biene zeigt, deren Flügel flattern. Aber wir werden mit einem Steuerelement beginnen, das ein *nicht*-animiertes Bild zeigt, und dann einfach die Animation hinzufügen. Wir beginnen also mit einer `PictureBox`, der wir dann Code zum Zeichnen einer animierten Biene hinzufügen.



1. **Erstellen Sie ein neues Projekt** und fügen Sie den Projektressourcen die vier Teile der Animation hinzu, genau wie Sie es mit dem Logo der Papierfabrik Objekthausen vor langer Zeit in Kapitel 1 gemacht haben. Aber statt sie den *Formular*ressourcen hinzuzufügen, fügen Sie sie den *Projekt*ressourcen hinzu.

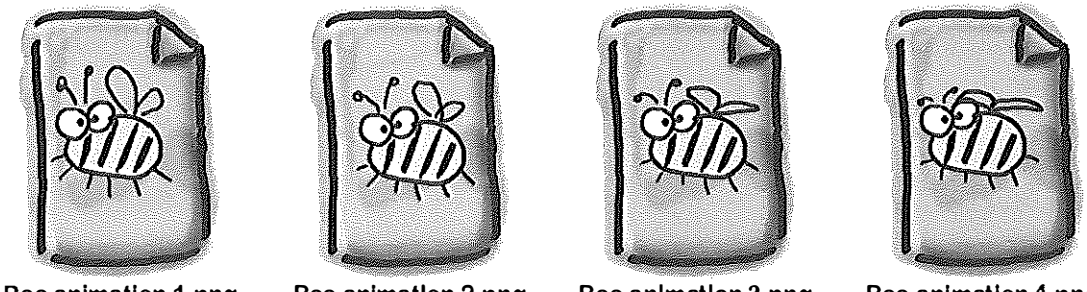
In Kapitel 1 haben wir die Logo-Grafik der Ressourcendatei des Formulars hinzugefügt. Diesmal fügen wir die Ressourcen der globalen Ressourcenliste des Projekts hinzu. Damit machen wir sie für alle Klassen in unserem Projekt verfügbar (über die Auflistung Properties.Resources).

Nehmen Sie sich etwas Zeit und blättern Sie zur Auffrischung zu Kapitel 1 zurück.



Das erscheint unter dem Projekt, nicht unter einem bestimmten Formular.

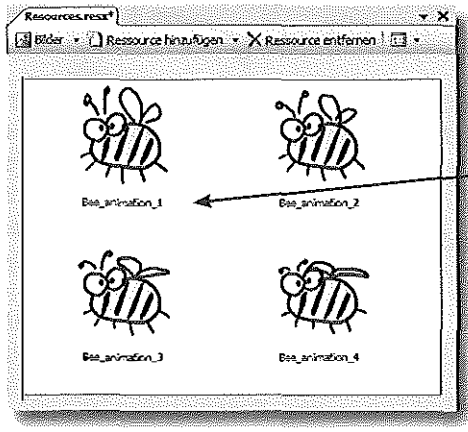
2. Wir haben eine vierteilige Bienenanimation erstellt, die Sie in Ihre Ressourcen importieren können. Sie können sie von **http://examples.oreilly.de/german_examples/hfcsharpger** herunterladen. Gehen Sie dann zur Seite Ressourcen, wählen Sie »Bilder« im ersten Aufklappmenü oben im Fenster und wählen Sie »Vorhandene Datei...« im Aufklappmenü »Ressourcen hinzufügen«.



Importieren Sie diese Bilder in Ihre Projektressourcen.

5 Fügen Sie der Ressourcendatei eines Projekts Bilder oder andere Ressourcen hinzu, können Sie auf diese über `Properties.Resources` zugreifen. Geben Sie einfach auf einer Zeile Ihres Codes `Properties.Resources.` ein – IntelliSense öffnet dann eine Aufklappliste, die alle Bilder zeigt, die Sie importiert haben.

Beachten Sie, den Punkt am Ende ... er sagt der IDE, dass sie die Eigenschaften und Methoden der Klasse einblenden soll, die Sie eingegeben haben.




```
pictureBox1.Image =
    Properties.Resources.Bee_animation_1;
```

Das setzt die Grafik, die für das Bild einer bestimmten PictureBox verwendet wird (und für unser Anfangsbild).

Diese Bilder werden als öffentliche Eigenschaften der Klasse Properties.Resources gespeichert.

>>using System.Windows.Forms<< müssen Sie für die PictureBox und den Timer hinzufügen.

4 **Fügen Sie jetzt das BienenControl hinzu!** Verwenden Sie diesen Code für die Klasse BienenControl:

```
public class BienenControl : PictureBox {
  private Timer animationTimer = new Timer();
  public BienenControl() {
    animationTimer.Tick += new EventHandler(animationTimer_Tick);
    animationTimer.Interval = 150;
    animationTimer.Start();
    BackColor = System.Drawing.Color.Transparent;
    BackgroundImageLayout = ImageLayout.Stretch;
  }

  private int zelle = 0;
  void animationTimer_Tick(object sender, EventArgs e) {
    zelle++;
    switch (zelle) {
      case 1: BackgroundImage = Properties.Resources.Bee_animation_1; break;
      case 2: BackgroundImage = Properties.Resources.Bee_animation_2; break;
      case 3: BackgroundImage = Properties.Resources.Bee_animation_3; break;
      case 4: BackgroundImage = Properties.Resources.Bee_animation_4; break;
      case 5: BackgroundImage = Properties.Resources.Bee_animation_3; break;
      default: BackgroundImage = Properties.Resources.Bee_animation_2;
      zelle = 0; break;
    }
  }
}
```

Hier initialisieren Sie den Timer, indem Sie ihn instantiieren, die Eigenschaft Interval setzen und dann den Tick-Event-Handler hinzufügen.

Sind wir zurück bei Frame 1, setzen wir zelle wieder auf 0.

Wird das Tick-Event des Timers abgesetzt, wird **zelle** inkrementiert. Die switch-Anweisung weist der Image-Eigenschaft (die von PictureBox geerbt wird) auf Basis des Werts von zelle das erforderliche Bild zu.

Wenn Sie den Code eines Steuerelements ändern, müssen Sie Ihr Programm neu erstellen, damit die Änderungen im Designer sichtbar werden.

Erstellen Sie das Programm dann neu. Werfen Sie im Designer einen Blick in die Toolbox. Das BienenControl ist da! Ziehen Sie es auf Ihr Formular – Sie erhalten eine **animierte** Biene!

Fügen Sie einem Formular das BienenControl über einen Button hinzu

Einem Formular können Sie leicht ein Steuerelement geben – fügen Sie es einfach der Controls-Liste hinzu. Und genauso leicht können Sie es aus dem Formular entfernen, indem Sie es aus Controls löschen. Aber Steuerelemente implementieren `IDisposable`, achten Sie also darauf, dass Sie auf **Ihren Steuerelementen immer dispose() aufrufen,** wenn Sie sie entfernen.

Tun Sie das!

1. **Entfernen Sie das BienenControl und fügen Sie einen Button hinzu.**
Gehen Sie zum Form-Designer und **löschen Sie das BienenControl aus dem Formular**. Fügen Sie dann einen Button hinzu. Wir werden den Button ein BienenControl hinzufügen und wieder entfernen lassen.

2. **Fügen Sie einen Button hinzu, der das BienenControl hinzufügt und entfernt.**
Hier ist der Event-Handler für diesen Button:

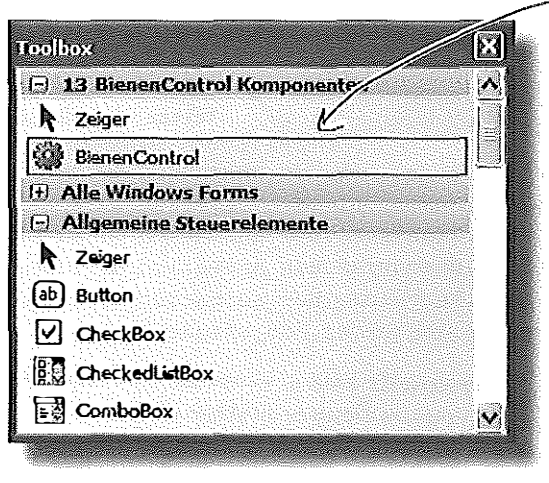
Sie können einen Objektinitialisierer nutzen, um die BienenControl-Eigenschaften zu setzen, nachdem es instantiiert wurde.

Wenn Sie Controls ein Steuerelement hinzufügen, erscheint es sofort auf dem Formular.

```
BienenControl control = null;
private void button1_Click(object sender, EventArgs e) {
    if (control == null) {
        control = new BienenControl() { Location = new Point(100, 100) };
        Controls.Add(control);
    } else {
        Controls.Remove(control);
        control.Dispose();
        control = null;
    }
}
```

Um das Steuerelement zu entfernen, müssen Sie es nur aus der Controls-Auflistung entfernen. Aber denken Sie daran, darauf dispose() aufzurufen – da es sonst weiterhin Ressourcen verbraucht!

Führen Sie das Programm aus, wird dem Formular beim ersten Klick auf den Button ein neues BienenControl hinzugefügt. Klicken Sie wieder darauf, wird es gelöscht. Der Code nutzt das private Feld `control` als Referenz auf das Steuerelement. (Er setzt die Referenz auf null, wenn auf dem Formular kein Steuerelement mehr ist.)



In der Toolbox können Sie ein eigenes Steuerelement hinzufügen, indem Sie einfach eine Klasse erstellen, die Control erweitert.

Hinter den Kulissen

Jedes visuelle Steuerelement in der Toolbox erbt von `System.Windows.Forms.Control`. Diese Klasse hat einige Member, die mittlerweile ziemlich vertraut sein sollten: Visible, Width, Height, Text, Location, BackColor, BackgroundImage ... all diese Eigenschaften sehen Sie im Eigenschaften-Fenster der IDE.

Ihre Steuerelemente müssen dispose() anbieten

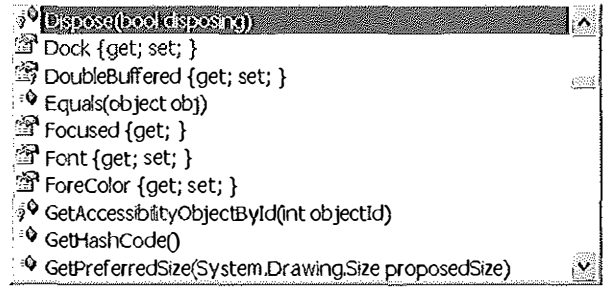
Mit dem BienenControl gibt es ein Problem. Steuerelemente müssen entsorgt werden. Aber das BienenControl erstellt eine neue Instanz von Timer: Das ist ein Steuerelement, das in der Toolbox angezeigt wird ... aber es wird nie entsorgt! Ein Problem! Glücklicherweise lässt es sich leicht beheben – überschreiben Sie einfach die Methode Dispose().

Die Klasse Control implementiert IDisposable. Sie müssen also sicherstellen, dass alle Steuerelemente, die Sie verwenden, entsorgt werden.

3 Überschreiben Sie Dispose() und entsorgen Sie den Timer.

Da BienenControl ein Steuerelement erweitert, muss es eine Dispose()-Methode haben. Wir können diese also einfach überschreiben und erweitern, um unseren Timer zu entsorgen. Gehen Sie dazu in den Code und geben Sie `override` ein.

```
public class BienenControl : PictureBox {
    override
```



Geben Sie innerhalb einer Klasse >>override<< ein, blendet die IDE ein IntelliSense-Fenster mit allen Methoden ein, die Sie überschreiben können. Wählen Sie Dispose(), wird diese Methode für Sie eingefügt.

Wenn Sie auf Dispose() klicken, fügt die IDE eine Methode mit einem Aufruf von base.Dispose() ein:

```
protected override void Dispose(bool disposing) {
    base.Dispose(disposing);
}
```

4 Fügen Sie den Code hinzu, der den Timer entsorgt.

Die von der IDE hinzugefügte Dispose()-Methode müssen wir jetzt nur noch so erweitern, dass sie die BienenControl-spezifischen Aufräumarbeiten verrichten kann. Dazu brauchen wir nur eine Zeile, die auf dem Timer, der seinerseits ein Steuerelement ist und IDispose implementiert, dispose() aufruft:

```
protected override void Dispose(bool disposing) {
    animationTimer.Dispose();
    base.Dispose(disposing);
}
```

Jetzt entsorgt das BienenControl den Timer in seiner eigenen Dispose()-Methode. Es räumt hinter sich auf!

Jetzt entsorgt das BienenControl den erstellten Timer. Die IDE hat einen Aufruf an die Dispose()-Methode der Basisklasse PictureBox eingefügt – der ist erforderlich, damit die PictureBox sich selbst entsorgen kann.

Ihre Steuerelemente müssen alle Steuerelemente (oder entsorgbaren Objekte) entsorgen, die sie selbst erstellen.

Mit UserControl können Steuerelemente leicht erstellt werden

Es gibt einen leichteren Weg, eigene Toolbox-Steuerelemente zu erstellen. Statt eine eigene Klasse zu erzeugen, die ein vorhandenes Steuerelement erweitert, können Sie auch einfach **die IDE verwenden, um Ihrem Projekt ein UserControl hinzuzufügen.** Mit einem UserControl arbeiten Sie genau wie mit einem Formular. Sie können aus der Toolbox andere Steuerelemente darauf ziehen – es wird der gewöhnliche Form-Designer der IDE verwendet. Und wie bei einem Formular können Sie ebenso einfach seine Events nutzen. Bauen wir das BienenControl also erneut, aber diesmal über ein UserControl.



1. Klicken Sie in der IDE mit rechts auf BienenControl.cs und nennen Sie die Datei in AlteBiene.cs um. Die IDE **blendet Folgendes in einem Ja/Nein-Fenster ein**: *»Sie benennen eine Datei um. Möchten Sie auch alle Verweise auf das Codeelement 'BienenControl' in diesem Projekt umbenennen?«* – es fragt Sie also, ob Sie die Klasse BienenControl in die Klasse AlteBiene umbenennen und dann den gesamten Code in Ihrem Projekt so ändern wollen, dass er diese Änderung wiedergibt. **Klicken Sie auf den Button »Ja«**, um Ihr BienenControl in AlteBiene umzutaufen.

2. Klicken Sie im Projektmappen-Explorer mit der rechten Maustaste auf das Projekt und wählen Sie »Hinzufügen→ Benutzersteuerelement...«. Lassen Sie die IDE **ein Benutzersteuerelement namens BienenControl hinzufügen**. Die IDE öffnet das neue Steuerelement im Form-Designer.

Nutzen Sie die Methode animationTimer_Tick() (Sie müssen sie umbenennen) und das zelle-Feld des alten Bienen-Steuerelements – aber kopieren Sie nichts anderes.

3. Ziehen Sie das Timer-Steuerelement auf das neue Benutzersteuerelement. Es wird wie bei einem Formular unten im Designer angezeigt. Setzen Sie im Fenster Eigenschaften `Interval` auf 150 und `Enabled` auf true. Klicken Sie dann doppelt darauf – die IDE fügt den Tick-Event-Handler hinzu. Nutzen Sie nun den Tick-Event-Handler, den Sie bereits im ersten Biene-Steuerelement verwendet haben.

4. Aktualisieren Sie jetzt den BienenControl-Konstruktor:

```
public BienenControl() {
    InitializeComponent();
    BackColor = System.Drawing.Color.Transparent;
    BackgroundImageLayout = ImageLayout.Stretch;
}
```

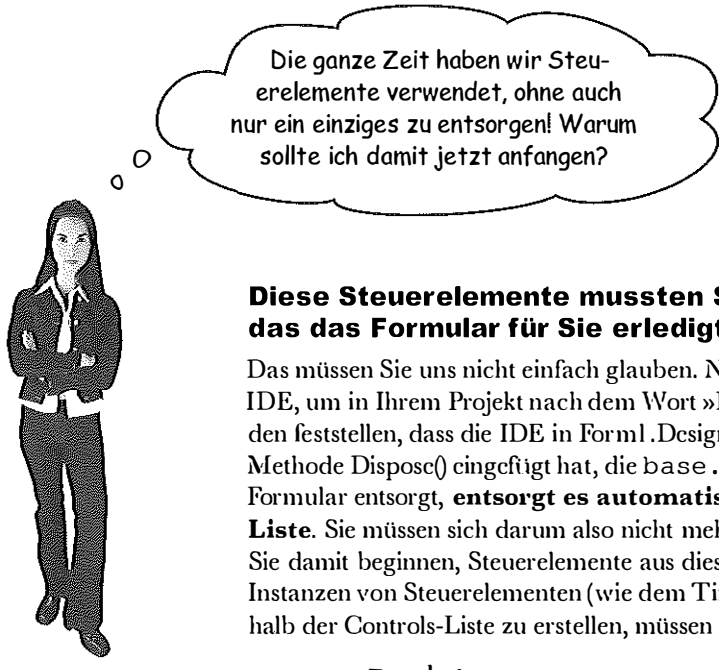
Statt über Code können Sie das auch über das Eigenschaften-Fenster der IDE tun.

5. Kehren Sie zum Button-Event-Handler des Formulars zurück. Als Sie das erste BienenControl in AlteBiene umbenannt haben, wurde auch der Formularcode korrigiert. Ändern Sie also seine beiden Zeilen wieder in BienenControl zurück, damit statt der PictureBox das neue UserControl verwendet wird:

```
BienenControl control = null;
private void button1_Click(...) {
    if (control == null) {
        control = new BienenControl() { ...
```

Führen Sie das Programm jetzt aus – es sollte genau so funktionieren wie zuvor. Der Button fügt dem Formular ein UserControl-basiertes BienenControl hinzu oder entfernt es.

Über ein UserControl können der Toolbox sehr leicht Steuerelemente hinzugefügt werden. Sie bearbeiten es wie ein Formular – ziehen Sie Steuerelemente darauf und verwenden Sie seine Events wie die eines Formulars.



Diese Steuerelemente mussten Sie nicht entsorgen, weil das das Formular für Sie erledigt hat.

Das müssen Sie uns nicht einfach glauben. Nutzen Sie die Suchfunktion der IDE, um in Ihrem Projekt nach dem Wort »Dispose« zu suchen. Sie werden feststellen, dass die IDE in Form1.Designer.cs eine Überschreibung der Methode Dispose() eingefügt hat, die `base.Dispose()` aufruft. Wird das Formular entsorgt, **entsorgt es automatisch alles in seiner Controls-Liste**. Sie müssen sich darum also nicht mehr selbst kümmern. Aber sobald Sie damit beginnen, Steuerelemente aus dieser Liste zu entfernen oder neue Instanzen von Steuerelementen (wie dem Timer im BienenControl) außerhalb der Controls-Liste zu erstellen, müssen Sie diese selbst entsorgen.

Es gibt keine Dummen Fragen

F: Warum funktioniert der Formularcode für das PictureBox-basierte BienenControl genauso gut mit dem UserControl-basierten BienenControl?

A: Weil es den Code nicht interessiert, wie das BienenControl-Objekt implementiert ist. Ihn interessiert nur, dass er das Objekt der Controls-Eigenschaft des Formulars hinzufügen kann.

F: Ich habe im Projektmappen-Explorer auf meine alte AlteBiene-Klasse doppelt geklickt und erhielt eine Nachricht, die mir sagt, dass meiner Klasse Komponenten hinzugefügt werden. Was bedeutet das?

A: Wenn Sie ein Steuerelement erstellen, indem Sie Ihrem Projekt eine Klasse hinzufügen, die PictureBox oder ein anderes Steuerelement erweitert, führt die IDE im Hintergrund einige Operationen durch. Unter anderem lässt sie Sie mit **Komponenten**, den nicht sichtbaren Steuerelementen wie Timer und OpenFileDialog, arbeiten, die im Bereich unterhalb des Formulars erscheinen, wenn Sie mit ihnen arbeiten.

Probieren Sie es aus – erstellen Sie eine leere Klasse, die von PictureBox erbt. Erstellen Sie Ihr Projekt dann neu und klicken Sie in der IDE doppelt darauf. Sie sehen dann folgende Nachricht:

Um einer Klasse Komponenten hinzuzufügen, ziehen Sie diese aus der Toolbox und legen Sie die Eigenschaften im Dialogfeld Eigenschaften fest.

Ziehen Sie einen OpenFileDialog aus der Toolbox auf Ihre neue Klasse. Er erscheint als Symbol. Sie können darauf klicken und seine Eigenschaften setzen. Setzen Sie ein paar. Kehren Sie zum Code der Klasse zurück. Sehen Sie sich den Konstruktor an – die IDE hat Code für die Instantiierung des OpenFileDialog-Objekts und zum Setzen seiner Eigenschaften hinzugefügt.

F: Als ich die Eigenschaften von OpenFileDialog änderte, habe ich eine Fehlermeldung erhalten, die besagte, dass ich das Projekt neu erstellen müsse, damit die Änderungen im Designer sichtbar werden. Warum erhalte ich diesen Fehler?

A: Weil der Designer Ihr Steuerelement ausführt, ihm aber nicht die neueste Version des Steuerelements vorliegt.

Erinnern Sie sich daran, dass die Flügel der Biene nach der Erstellung des ersten BienenControls schon flatterten, als Sie es aus der Toolbox auf das Formular zogen? Das Programm lief noch nicht, aber der von Ihnen geschriebene Code wurde schon ausgeführt. Der Timer setzte sein Tick-Event ab, und Ihr Event-Handler änderte das Bild. Das kann die IDE nur machen, wenn der Code kompiliert ist und irgendwo im Speicher läuft. Deswegen erinnert Sie die IDE daran, dass Sie Ihren Code aktualisieren müssen, damit sie das Steuerelement ordentlich anzeigen kann.

Der Renderer nutzt Ihr BienenControl, um animierte Bienen auf Ihre Formulare zu zeichnen

Nachdem Sie die Klasse BienenControl und die zwei Formulare haben, brauchen Sie nur noch ein Mittel, um die Bienen zu positionieren, von einem Formular zum anderen zu bewegen und die Bienen nachzuhalten. Außerdem müssen Sie die Blumen auf dem Feld-Formular positionieren. Das allerdings ist recht einfach, weil diese sich nicht bewegen. Den gesamten Code dafür können wir in eine neue Klasse, `Renderer`, stecken. Diese Klasse macht Folgendes.

Die Formulare für Stock und Feld müssen an das Statistik-Formular >>geknüpft<< sein – das sorgt z.B. dafür, dass Stock und Feld minimiert werden, wenn die Statistiken minimiert werden. Sie erreichen es, indem Sie Windows sagen, dass das Statistik-Formular deren Eigentümer ist.

1 **Das Statistikformular zum Elternformular machen.**

Um den Bienenstock-Simulator grafisch auszubauen, müssen Sie dem Projekt zunächst zwei Formulare hinzufügen. Das erste wird den Namen StockFormular haben (es zeigt den Stock von innen) und das zweite den Namen FeldFormular (es zeigt das Feld mit den Blumen). Dann werden Sie dem Konstruktor des Hauptformulars Zeilen hinzufügen, um die beiden Kindformulare anzuzeigen und Windows mitzuteilen, dass das Statistikformular ihr **Eigentümer** ist:

```
public Form1() {
    // Weiterer Code im Form1-Konstruktor.
    stockForm.Show(this);
    feldForm.Show(this);
```

Formular-Objekte haben eine Show()-Methode. Möchten Sie ein anderes Formular zum Eigentümer eines Formulars machen, müssen Sie Show() nur eine Referenz auf dieses Formular übergeben.

2 **Dem Renderer Referenzen auf die Welt und die Kindformulare geben.**

Am Anfang der Klasse `Renderer` finden Sie ein paar wichtige Felder. Die Klasse muss den Ort aller Bienen und Blumen kennen, braucht also eine Referenz auf die `Welt`. Und Sie brauchen für beide Formulare die Möglichkeit, Steuerelemente hinzuzufügen, zu bewegen und zu entfernen. Dazu brauchen Sie Referenzen auf diese beiden Formulare:

```
public class Renderer {
    private Welt welt;
    private StockFormular stockForm;
    private FeldFormular feldForm;
```

Beginnen Sie die Klasse Renderer mit diesen Zeilen. Diese Klasse werden wir im Verlauf des Kapitels erweitern.

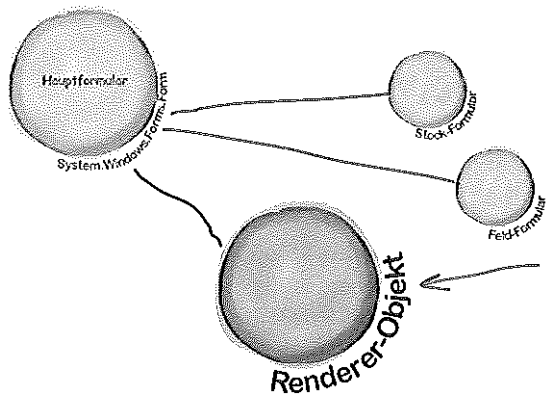
3 **Mit Dictionaries hält der Renderer die Steuerelemente nach.**

`Welt` hält Bienen und Blumen mit einer `List<Biene>` und einer `List<Blume>` fest. Der Renderer muss alle `Biene`- und `Blume`-Objekte einsehen können und ermitteln, welchen `BienenControl`- und `PictureBox`-Objekten sie entsprechen – und muss, wenn er kein passendes Steuerelement finden kann, ein neues erstellen. Damit haben wir das perfekte Einsatzgebiet für Dictionaries. Wir brauchen also zwei weitere private Felder in `Renderer`:

```
private Dictionary<Blume, PictureBox> blumenBuch =
    new Dictionary<Blume, PictureBox>();
private Dictionary<Biene, BienenControl> bienenBuch =
    new Dictionary<Biene, BienenControl>();
```

Diese Dictionaries sind 1:1-Abbildungen von Blumen oder Bienen auf die ihnen entsprechenden Steuerelemente.

In diesen beiden Dictionaries kann der Renderer genau ein Steuerelement für jede Biene oder Blume in der Welt speichern.



Der Renderer wirkt auf die Formulare sowie die Objekte ein, die Sie im letzten Kapitel für den Simulator erstellt haben.

4 **Bienen und Blumen kennen ihren Ort bereits.**

Es gibt einen Grund dafür, dass wir für die Orte von Bienen und Blumen `Point` verwendet haben. Haben wir ein `Biene`-Objekt, können wir leicht sein `BienenControl` suchen und seinen Ort setzen.

```
bienenControl = bienenBuch[biene];
bienenControl.Location = biene.Ort;
```

Für jede Blume oder Biene können wir das passende Steuerelement nachschlagen. Dann setzen wir den Ort des Steuerelements so, dass er dem des Objekts entspricht.

6 **Hat eine Biene kein Steuerelement, muss der Renderer dem Stock eins hinzufügen.**

Der Renderer kann leicht herausfinden, ob es für eine Biene oder Blume ein Steuerelement gibt. Liefert die `ContainsKey()`-Methode des Dictionary für ein bestimmtes `Biene`-Objekt `false`, bedeutet dies, dass es auf keinem Formular ein Steuerelement für diese Biene gibt. `Renderer` muss also ein `BienenControl` erstellen und dieses dem Dictionary und dem Formular hinzufügen (außerdem ruft er die `BringToFront()`-Methode des Steuerelements auf, um sicherzustellen, dass das Steuerelement nicht hinter den `PictureBoxen` für die Blumen versteckt wird):

```
if (!bienenBuch.ContainsKey(biene)) {
    bienenControl = new BienenControl() { Width = 40, Height = 40 };
    bienenBuch.Add(biene, bienenControl);
    stockForm.Controls.Add(bienenControl);
    bienenControl.BringToFront();
} else
   bienenControl = bienenBuch[biene];
```

ContainsKey() sagt uns, ob die Biene im Dictionary vorhanden ist. Falls nicht, müssen wir die Biene und das entsprechende Steuerelement hinzufügen.

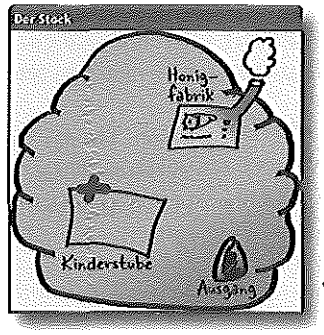
BringToFront() sichert, dass die Biene »über« allen Blumen auf dem Feld-Formular und über dem Hintergrund des Stock-Formulars erscheint

Erinnern Sie sich, dass man in einem Dictionary alles als Schlüssel verwenden kann? Dieses nutzt ein Biene-Objekt als Schlüssel. Der Renderer muss wissen, welches BienenControl auf dem Formular z welcher Biene gehört. Schlägt er das Biene-Objekt im Dictionary nach, liefert dieses das entsprechende Steuerelement. Das kann der Renderer dann bewegen.

Fügen Sie dem Projekt die neuen Formulare hinzu

Sie brauchen Formulare für die Bienen. **Nehmen Sie Ihr Bienenstock-Simulator-Projekt** und **fügen** Sie ihm mit »Hinzufügen→ Vorhandenes Element...« **das neue BienenControl-Benutzersteuerelement hinzu**. Erstellen Sie dann zwei weitere Formulare, indem Sie im Projektmappen-Explorer **auf das Projekt rechtsklicken** und im Menü Hinzufügen »Windows Form...« wählen. Geben Sie den Dateien die Namen StockFormular.cs und FeldFormular.cs, setzt die IDE ihre Name-Eigenschaft automatisch auf `StockFormular` bzw. `FeldFormular`. Sie wissen bereits, dass Formulare einfach nur Objekte sind, `StockFormular` und `FeldFormular` sind also lediglich zwei weitere Klassen.

Das ist eine PictureBox, deren BackgroundImage auf die Außenansicht des Stocks gesetzt ist. Wenn Sie die Stockbilder in den Resource Designer laden, werden sie in der Ressourcenliste angezeigt, wenn Sie im Fenster Eigenschaften auf den »...«-Button neben BackgroundImage klicken.



Achten Sie darauf, dass Sie die Größe beider Formulare so anpassen, dass sie diesen Abbildungen entsprechen.

Das Feld

Sie müssen die Bilder für die Innen- und die Außenansicht des Stocks – »Stock (inside).png« und »Stock (outide).png« – in Ihre Ressourcen laden. Fügen Sie diese beiden Formulare hinzu. Setzen Sie bei beiden Formularen FormBorderStyle auf FixedSingle (damit der Benutzer die Größe nicht verändern kann), ControlBox auf false (um die Minimieren- und Maximieren-Buttons zu entfernen) und StartPosition auf Manual (damit die Eigenschaft Location gesetzt werden kann).

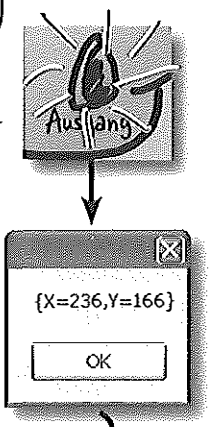
Setzen Sie die BackgroundImage-Eigenschaft des Formulars auf das Innenbild des Stocks und die Eigenschaft BackgroundImageLayout auf Stretch.

Die Orte ermitteln

Klicken Sie im Eigenschaften-Fenster auf das Blitzsymbol, um das Events-Fenster zu öffnen, scrollen Sie zur Zeile MouseClick und klicken Sie doppelt darauf, damit die IDE den Event-Handler hinzufügt.

Sie müssen ermitteln, wo sich der Stock auf dem FeldFormular befindet. Nutzen Sie das Fenster Eigenschaften, um einen Event-Handler für das MouseClick-Event hinzuzufügen, und gehen Sie diesen Code ein:

```
private void FeldFormular_MouseClick(object sender, MouseEventArgs e) {
  MessageBox.Show(e.Location.ToString());
}
```



Jetzt können Sie das Formular anzeigen und auf den Ort des Stocks klicken. Das liefert Ihnen die Koordinaten des Stocks auf dem Formular.

Fügen Sie dem StockFormular den gleichen Event-Handler hinzu und führen Sie es aus. Verschaffen Sie sich dann die Koordinaten des Ausgangs, der Kinderstube und der Honigfabrik, indem Sie darauf klicken. Mithilfe dieser Koordinaten können Sie die `OrteInitialisieren()`-Methode aktualisieren, die Sie im letzten Kapitel für die Klasse `Stock` geschrieben haben:

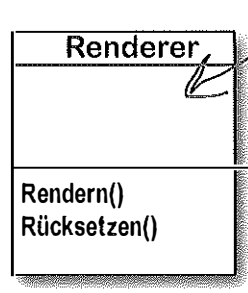
```
private void OrteInitialisieren()
{
    orte = new Dictionary<string, Point>();
    orte.Add("Eingang", new Point(600, 100));
    orte.Add("Kinderstube", new Point(95, 174));
    orte.Add("HonigFabrik", new Point(157, 98));
    orte.Add("Ausgang", new Point(194, 213));
}
```

Diese Koordinaten haben wir ermittelt, sieht Ihr Formular anders aus, können diese bei Ihnen anders sein

Entfernen Sie die Event-Handler, wenn Sie fertig sind ... Sie brauchten sie nur, um die Orte auf den Formularen zu erhalten.

Den Renderer erstellen

Hier ist die vollständige `Renderer`-Klasse. Das Hauptformular ruft die `Rendern()`-Methode unmittelbar nach `Welt.Laufen()` auf, um Bienen und Blumen auf dem Formular zu zeichnen. Sie müssen sicherstellen, dass die Blumengrafik (`Flower.png`) in das Projekt geladen ist – genau wie die Grafiken für die animierte Biene.



Alle Felder im Renderer sind privat, weil keine anderen Klassen seine Eigenschaften aktualisieren müssen. Die Klasse ist vollständig gekapselt. Die Welt ruft Rendern() auf, damit die Welt auf den Formularen gezeichnet wird, und Rücksetzen(), um die Steuerelemente vom Formular zu entfernen, wenn es zurückgesetzt werden muss.

```
public class Renderer {
    private Welt welt;
    private StockFormular stockForm;
    private FeldFormular feldForm;

    private Dictionary<Blume, PictureBox> blumenBuch =
        new Dictionary<Blume, PictureBox>();
    private List<Blume> toteBlumen = new List<Blume>();

    private Dictionary<Biene, BienenControl> bienenBuch =
        new Dictionary<Biene, BienenControl>();
    private List<Biene> bienenInRente = new List<Biene>();

    public Renderer(Welt welt, StockFormular stockForm, FeldFormular feldForm) {
        this.welt = welt;
        this.stockForm = stockForm;
        this.feldForm = feldForm;
    }

    public void Rendern() {
        bienenZeichnen();
        blumenZeichnen();
        ToteBlumenUndBienenEntfernen();
    }

    public void Rücksetzen() {
        foreach (PictureBox blume in blumenBuch.Values) {
            feldForm.Controls.Remove(blume);
            blume.Dispose();
        }
        foreach (BienenControl biene in bienenBuch.Values) {
            stockForm.Controls.Remove(biene);
            feldForm.Controls.Remove(biene);
            biene.Dispose();
        }
        blumenBuch.Clear();
        bienenBuch.Clear();
    }
```

Der Renderer hält Referenzen auf die Welt und die beiden Formulare, auf denen Bienen und Blumen gezeichnet werden.

Die Welt nutzt Biene- und Blume-Objekte, um alle Bienen und Blumen in der Welt nachzuhalten. Die Formulare nutzen eine PictureBox, um die Blumen anzuzeigen, und ein BienenControl, um die Bienen anzuzeigen. Der Renderer nutzt die Dictionaries, um Bienen und Blumen mit ihrem BienenControl oder ihrer PictureBox zu verbinden.

Der Timer auf dem Hauptformular, der die Animation steuert, ruft die Methode Rendern() auf, die die Bienen und die Blumen aktualisiert, und bereinigt dann die Dictionaries.

Stirbt eine Blume oder geht eine Biene in Rente, werden die Listen toteBlumen und bienenInRente eingesetzt, um die Dictionaries zu bereinigen.

Wird der Simulator zurückgesetzt, wird die Methode Rücksetzen() aufgerufen, um alle Steuerelemente von den beiden Formularen zu entfernen. Sie sucht in den beiden Dictionaries alle Steuerelemente, entfernt sie aus den Formularen und ruft auf allen Dispose() auf. Dann leert sie die beiden Dictionaries.

Zum Zeichnen der Blumen brauchen wir zwei foreach-Schleifen. Die erste sucht neue Blumen und fügt ihre PictureBoxen hinzu. Die zweite sucht nach toten Blumen und entfernt ihre PictureBoxen.

```
private void blumenZeichnen() {
    foreach (Blume blume in welt.Blumen)
        if (!blumenBuch.ContainsKey(blume)) {
            PictureBox blumenControl = new PictureBox() {
                Width = 45,
                Height = 55,
                Image = Properties.Resources.Flower,
                SizeMode = PictureBoxSizeMode.StretchImage,
                Location = blume.Ort
            };
            blumenBuch.Add(blume, blumenControl);
            feldForm.Controls.Add(blumenControl);
        }

    foreach (Blume blume in blumenBuch.Keys) {
        if (!welt.Blumen.Contains(blume)) {
            PictureBox zuEntfernendeBlume = blumenBuch[blume];
            feldForm.Controls.Remove(zuEntfernendeBlume);
            zuEntfernendeBlume.Dispose();
            toteBlumen.Add(blume);
        }
    }
}

private void bienenZeichnen() {
    BienenControl bienenControl;
    foreach (Biene biene in welt.Bienen) {
        bienenControl = BienenControlSuchen(biene);
        if (biene.InStock) {
            if (feldForm.Controls.Contains(bienenControl))
                BieneVonFeldZuStock(bienenControl);
        } else if (stockForm.Controls.Contains(bienenControl))
            BieneVonStockZuFeld(bienenControl, biene);
        bienenControl.Location = biene.Ort;
    }

    foreach (Biene biene in bienenBuch.Keys) {
        if (!welt.Bienen.Contains(biene)) {
            bienenControl = bienenBuch[biene];
            if (feldForm.Controls.Contains(bienenControl))
                feldForm.Controls.Remove(bienenControl);
            if (stockForm.Controls.Contains(bienenControl))
                stockForm.Controls.Remove(bienenControl);
            bienenControl.Dispose();
            bienenInRente.Add(biene);
        }
    }
}
```

Die erste foreach-Schleife nutzt das Dictionary blumenBuch, um zu prüfen, ob es für alle Blumen ein Steuerelement auf dem Formular gibt. Gibt es keins, wird mit einem Objektinitialisierer eine PictureBox erstellt, dem Formular hinzugefügt und dann im Dictionary blumenBuch gespeichert.

blumenZeichnen() nutzt die Ort-Eigenschaft des Blume-Objekts, um den Ort der PictureBox auf dem Formular zu setzen.

Die zweite foreach-Schleife sucht im Dictionary blumenBuch nach PictureBoxen, die sich nicht mehr im Formular befinden, und entfernt diese.

Nachdem die PictureBox entfernt wurde, wird ihre Dispose()-Methode aufgerufen. Dann wird das Blume-Objekt toteBlumen hinzugefügt. Es wird später entfernt.

bienenZeichnen() nutzt ebenfalls zwei foreach-Schleifen und macht im Grunde das Gleiche wie blumenZeichnen(). Aber die Methode ist komplexer. Deswegen lagern wir Teile ihres Verhaltens in separate Methoden aus, damit der Code verständlicher bleibt.

bienenZeichnen() prüft, ob eine Biene im Stock ist, ihr Steuerelement aber auf dem Feld-Formular oder umgekehrt. Zwei zusätzliche Methoden werden verwendet, um das BienenControl von einem der Formulare zum anderen zu bewegen.

Wurde das BienenControl entfernt, muss darauf Dispose() aufgerufen werden – das wird für uns auch den Timer entsorgen.

Die zweite foreach-Schleife funktioniert wie die in blumenZeichnen(), muss das BienenControl aber vom richtigen Formular entfernen.

Denken Sie daran, am Anfang der Klassendatei für Renderer `using System.Drawing` und `using System.Windows.Forms` anzugeben.

BienenControlSuchen() sucht eine Biene im Dictionary bienenBuch und liefert sie zurück. Wird die Biene nicht gefunden, wird ein neues BienenControl erstellt und dem Stock-Formular hinzugefügt (da Bienen dort geboren werden).

```
    private BienenControl BienenControlSuchen(Biene biene) {
        BienenControl bienenControl;
        if (!bienenBuch.ContainsKey(biene)) {
            bienenControl = new BienenControl() { Width = 40, Height = 40 };
            bienenBuch.Add(biene, bienenControl);
            stockForm.Controls.Add(bienenControl);
            bienenControl.BringToFront();
        }
        else
            bienenControl = bienenBuch[biene];
        return bienenControl;
    }
```

Vergessen Sie nicht: ! heißt NICHT!

BieneVonStockZuFeld() nimmt ein BienenControl aus der Controls-Liste des Stock-Formulars und fügt es der Controls-Liste des Feld-Formulars hinzu.

```
    private void BieneVonStockZuFeld(BienenControl bienenControl) {
        stockForm.Controls.Remove(bienenControl);
        bienenControl.Size = new Size(20, 20);
        feldForm.Controls.Add(bienenControl);
        bienenControl.BringToFront();
    }
```

Die Bienen auf dem Feld-Formular sind kleiner als die im Stock-Formular. Diese Methode muss also die Size-Eigenschaft des BienenControls anpassen.

```
    private void BieneVonFeldZuStock(BienenControl bienenControl) {
        feldForm.Controls.Remove(bienenControl);
        bienenControl.Size = new Size(40, 40);
        stockForm.Controls.Add(bienenControl);
        bienenControl.BringToFront();
    }
```

BieneVonFeldZuStock() bewegt ein BienenControl zum Stock-Formular zurück. Dazu muss es wieder größer gemacht werden.

```
    private void ToteBlumenUndBienenEntfernen() {
        foreach (Biene biene in bienenInRente)
            bienenBuch.Remove(biene);
        bienenInRente.Clear();
        foreach (Blume blume in toteBlumen)
            blumenBuch.Remove(blume);
        toteBlumen.Clear();
    }
}
```

Stellen bienenZeichnen() und blumenZeichnen() fest, dass eine Blume oder Biene nicht mehr in der Welt ist, fügen sie sie den Listen toteBlumen und bienenInRente hinzu, damit sie am Ende des Frames entfernt werden.

Nachdem alle Steuerelemente bewegt wurden, ruft der Renderer diese Methode auf, um tote Blumen und Bienen in Rente aus den beiden Dictionaries zu entfernen.

Das Hauptformular mit den beiden neuen Formularen verbinden

Dass wir jetzt einen Renderer haben, ist schön, aber bislang gibt es noch keine Formulare, auf die er zeichnen kann. Das können wir beheben, indem wir zur Klasse für das Hauptformular zurückkehren (die wahrscheinlich `Form1` heißt) und einige Codeänderungen durchführen.

```
public partial class Form1 : Form {
  StockFormular stockForm = new StockFormular();
  FeldFormular feldForm = new FeldFormular();
  Renderer renderer;

  // Die restlichen Felder.

  public Form1() {
    InitializeComponent();

    KindformulareVerschieben();
    stockForm.Show(this);
    feldForm.Show(this);
    SimulatorZurücksetzen();

    // Der restliche Code bleibt gleich.
  }

  private void KindformulareVerschieben() {
    stockForm.Location = new Point(Location.X + Width + 10, Location.Y);
    feldForm.Location = new Point(Location.X,
      Location.Y + Math.Max(Height, stockForm.Height) + 10);
  }

  public void FrameAusführen(object sender, EventArgs e) {
    gelaufeneFrames++;
    welt.Laufen(zufall);
    renderer.Rendern();
    // Der vorherige Code.
  }

  private void Form1_Move(object sender, EventArgs e) {
    KindformulareVerschieben();
  }
```

Wird das Hauptformular geladen, erstellt es jeweils eine Instanz der beiden anderen Formulare. Zunächst sind es nur Objekte auf dem Heap – angezeigt werden sie erst, wenn ihre Show()-Methoden aufgerufen werden.

Verschieben Sie den Code zur Instantiierung der Welt in die Methode SimulatorZurücksetzen().

Das Formular übergibt eine Referenz auf sich selbst an Form.Show(), damit es zum Elternformular wird.

Der Konstruktor des Hauptformulars richtet die beiden Kindformulare ein und zeigt sie an. Dann ruft er SimulatorZurücksetzen() auf. Dort wird der Renderer instantiiert.

Alles andere, das sich im Konstruktor befand, wird in die Methode SimulatorZurücksetzen() verschoben.

Dieser Code verschiebt die beiden Formulare so, dass das Stock-Formular neben dem Statistikformular ist und das Feld-Formular unter beiden.

Diese Zeile in FrameAusführen führt dazu, dass die Grafiken jedes Mal aktualisiert werden, wenn die Laufen()-Methode der Welt aufgerufen wurde.

Das Move-Event wird abgesetzt, wenn das Hauptformular verschoben wird. Der Aufruf von KindformulareVerschieben() sichert, dass die Kindformulare mit dem Hauptformular verschoben werden.

Nutzen Sie den Events-Button im Eigenschaften-Fenster, um diesen Event-Handler hinzuzufügen.

Achten Sie darauf, dass Sie die StartPosition-Eigenschaften der Formulare auf Manual gesetzt haben, sonst funktioniert KindformulareVerschieben() nicht.

Hier erstellen wir die neuen Instanzen der Klassen Welt und Renderer. Das setzt den Simulator zurück.

```
private void SimulatorZurücksetzen() {
  gelaufeneFrames = 0;
  welt = new Welt(new Biene.BienenMeldung(MeldungSenden));
  renderer = new Renderer(welt, stockForm, feldForm);
}

private void rücksetzen_Click(object sender, EventArgs e) {
    renderer.Rücksetzen();
    SimulatorZurücksetzen();
    if (!timer1.Enabled)
        toolStrip1.Items[0].Text = "Simulation starten";
}

private void openToolStripButton_Click(object sender, EventArgs e) {
  // Der restliche Code für diesen Button bleibt gleich.

  renderer.Rücksetzen();
  renderer = new Renderer(welt, stockForm, feldForm);
}
}
```

Der Zurücksetzen-Button muss Rücksetzen() aufrufen, um alle BienenControls und PictureBoxen zu entfernen und dann den Simulator zurückzusetzen.

Schließlich müssen Sie dem Öffnen-Button auf dem ToolStrip Code hinzufügen, der die Methode Rücksetzen() nutzt, um Bienen und Blumen aus der Controls-Liste des Formulars zu entfernen, und dann einen neuen Renderer erzeugen, der die neu erzeugte Welt verwendet

Es gibt keine Dummen Fragen

F: Ich verstehe, dass Sie das Formular mit der Methode `Show()` anzeigen. Aber was es bewirkt, wenn wir `this` als Parameter übergeben, ist mir noch nicht klar.

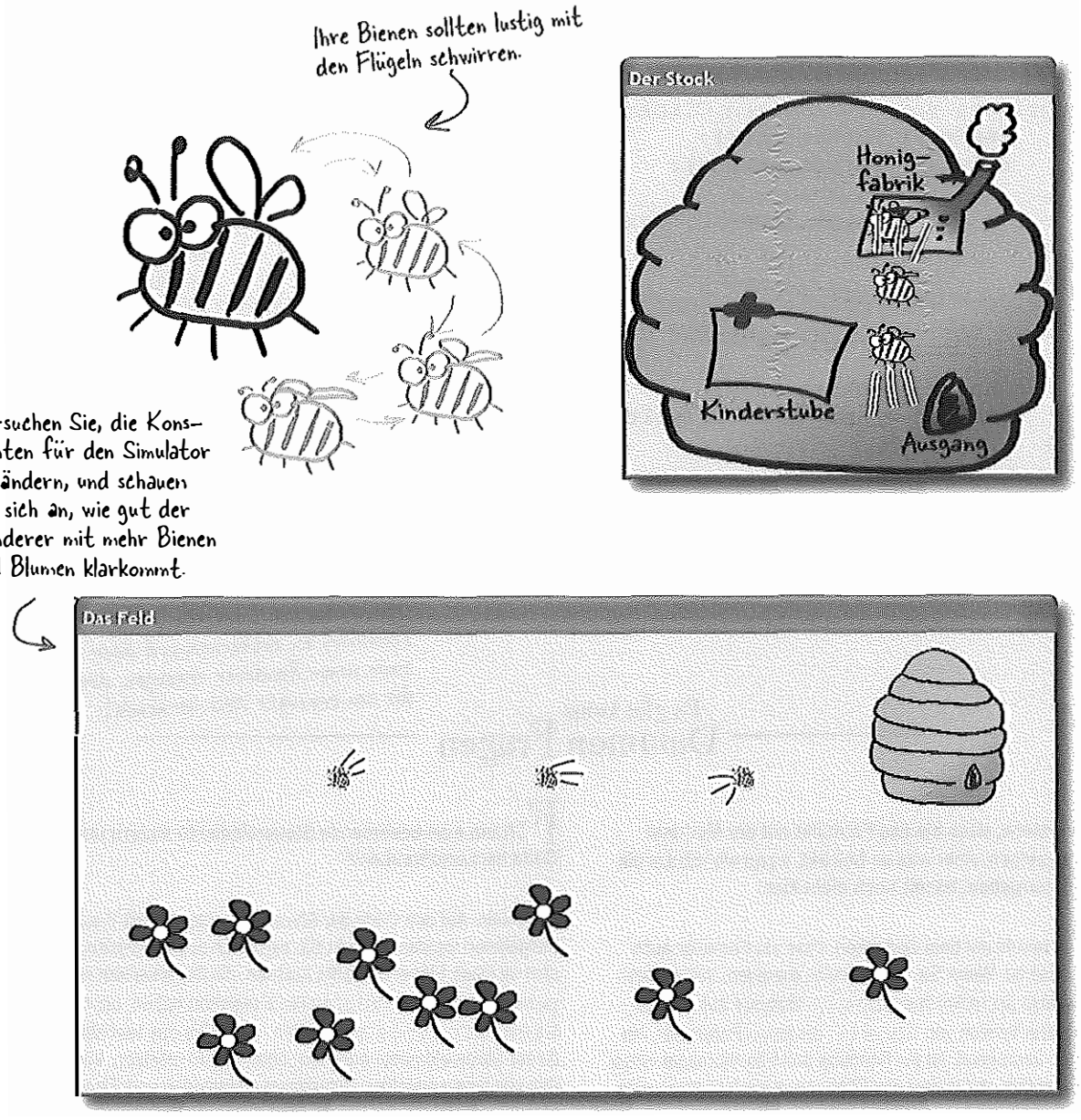
A: All das beruht auf dem Gedanken, dass ein Formular auch nur eine Klasse ist. Wenn Sie ein Formular anzeigen, instantiieren Sie diese Klasse und rufen ihre `Show()`-Methode auf. Es gibt eine überladene Version von `Show()`, die einen Parameter, ein Elternfenster, übernimmt. Ist ein Formular das Elternformular eines anderen Formulars, veranlasst das Windows, zwischen ihnen eine besondere Beziehung herzustellen – wird das Elternfenster minimiert, werden z.B. automatisch auch alle Kindfenster minimiert.

F: Kann man bestehende Steuerelemente ändern und an ihrem Code herumschrauben?

A: Nein. Auf den Code der Steuerelemente, die mit Visual Studio ausgeliefert werden, können Sie eigentlich nicht zugreifen. Allerdings sind all diese Steuerelemente Klassen, die Sie erweitern können, genau so, wie Sie `PictureBox` erweitert haben, um das `BienenControl` zu erstellen. Wenn Sie das Verhalten eines dieser Steuerelemente erweitern oder ändern möchten, können Sie den von Ihnen erstellten Unterklassen Methoden und Eigenschaften hinzufügen, die die der Basisklasse manipulieren.

Testlauf ... nein, Testflug

Kompilieren Sie Ihren Code, verarzten Sie alle Fehler, die Ihnen gemeldet werden, und fahren Sie den Simulator hoch.



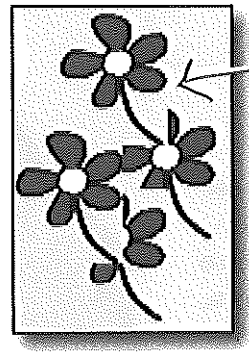
Sieht toll aus, aber ein paar kleine Haken gibt's noch

Sehen Sie sich die Bienen genau an, die um den Stock und die Blumen herumschwirren, werden Ihnen einige Probleme an der Art auffallen, wie sie gezeichnet werden. Erinnern Sie sich, dass Sie die `BackColor`-Eigenschaft aller `BienenControls` auf `Color.Transparent` gesetzt haben? Unglücklicherweise reicht das nicht, um zu verhindern, dass der Simulator einige Probleme aufweist, die für grafische Programme recht typisch sind.

1. **Es gibt ernsthafte Leistungsprobleme.**
Ist Ihnen aufgefallen, dass der gesamte Simulator ausgebremst wird, wenn alle Bienen im Stock sind? Falls nicht, fügen Sie probehalber einmal mehr Bienen hinzu, indem Sie die Konstanten in der Klasse `Stock` heraufsetzen. Achten Sie auf die Frame-Rate – sind mehr Bienen da, beginnt diese merklich zu sinken.

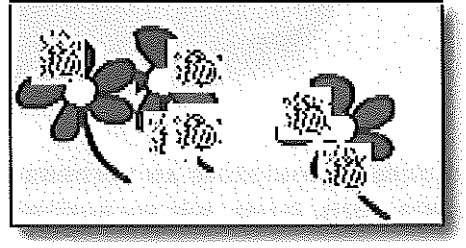
2. **Der Hintergrund der Blumen ist nicht richtig transparent.**
Es gibt ein weiteres, vollständig ***unabhängiges*** Problem. Als wir die Grafikdateien für die Blumen gespeichert haben, haben wir ihnen einen transparenten Hintergrund gegeben. Das hat zwar sichergestellt, dass der Hintergrund der Blumen dem Hintergrund des Formulars entspricht, sieht aber nicht so gut aus, wenn sich Blumen überschneiden.

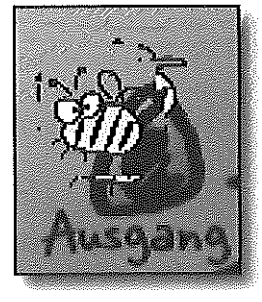
Setzen Sie die Hintergrundfarbe einer PictureBox auf Transparent, werden alle transparenten Pixel im Bild so gezeichnet, dass sie dem Hintergrund des Formulars entsprechen ... aber das ist nicht immer das Richtige.



Überschneiden sich zwei PictureBoxen, zeichnet C# die transparenten Pixel so, dass sie dem Formular entsprechen, nicht dem anderen Steuerelement, das darunter liegt. Das führt zu seltsamen rechteckigen »Ausschnitten«, wenn sich zwei Blumen überschneiden.

3. **Auch die Hintergründe der Bienen sind nicht transparent.**
Es stellt sich heraus, dass `Color.Transparent` einige ernsthafte Einschränkungen hat. Schweben Bienen über Blumen, tritt die gleiche »Aussschnitt«-Störung auf. Beim Stock-Formular funktioniert die Transparenz etwas besser. Das Hintergrundbild des Formulars scheint durch die transparenten Bereiche der Bienengrafiken hindurch. Aber wenn sich Bienen überschneiden, treten die gleichen Probleme auf. Und wenn Sie genau beobachten, wie die Bienen durch den Stock fliegen, werden Sie einige Störungen sehen, die dazu führen, dass das Bild der Bienen verzerrt wird, während sie sich bewegen.

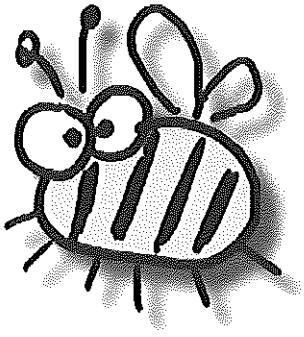




Schauen wir uns zunächst die Leistungsprobleme an

Die von Ihnen heruntergeladenen Biene-Grafiken sind groß, richtig groß. Öffnen Sie eine davon in der Windows Bild- und Faxanzeige und schauen Sie selbst. Das bedeutet, dass die PictureBox sie immer schrumpfen muss, wenn sie das Bild ändert. Das Skalieren nimmt Zeit in Anspruch, und das führt dazu, dass Bienen sich viel langsamer bewegen, wenn viele Bienen im Stock sind, weil die Grafik mit der Innenansicht des Stocks RIESIG ist. Und da wir den Hintergrund für das BienenControl transparent gemacht haben, muss doppelte Arbeit verrichtet werden: Erst muss das Bienenbild geschrumpft werden, und dann muss ein Teil des Formularhintergrunds verkleinert werden, damit dieser im transparenten Bereich hinter der Biene gezeichnet werden kann.

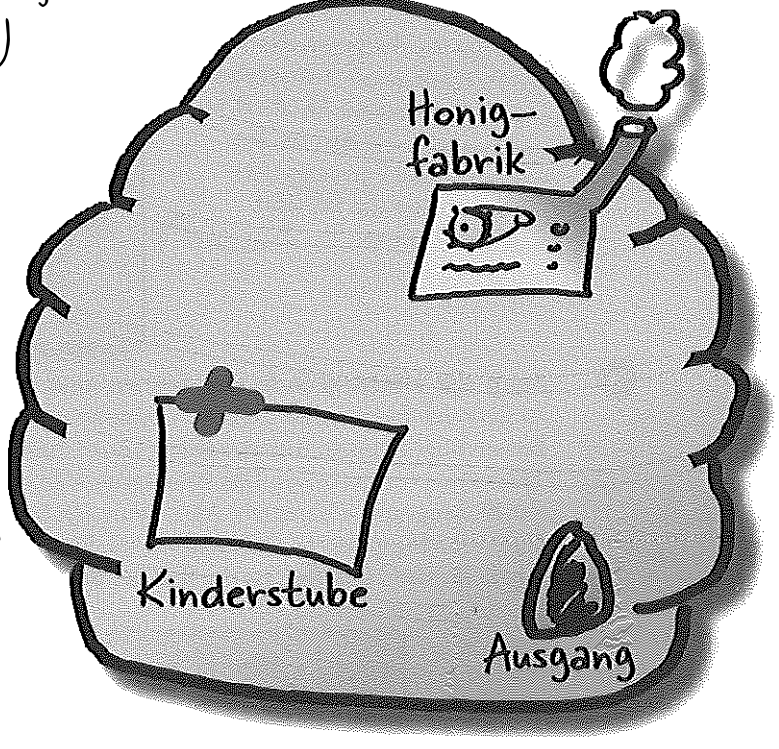
Bee animation 1.png



Die Bilder für die Biene sind ziemlich groß. Die PictureBox muss sie jedes Mal verkleinern, wenn ein neuer Frame der Animation angezeigt wird.

Die Grafikdateien für die Bienen sind echt GROSS. Wenn die PictureBox das Bild in jedem Frame der Animation verkleinern muss, nimmt das eine Menge Zeit in Anspruch ...

Hive (Inside).png



Das Bild für die Stock-Innenansicht ist riesig. Fliegt eine Biene davor, muss ihre PictureBox es auf die Größe des Steuerelements skalieren. Das ist erforderlich, um einen Teil des Bilds zu zeigen, wo es durch den transparenten Hintergrund des Bienenbilds hindurchscheint.

... um die Leistung unseres Simulators zu verbessern, müssen wir also nur alle Grafiken verkleinern, *bevor* wir versuchen, sie anzuzeigen.

Um die Grafikleistung zu verbessern, müssen wir dem Renderer also nur eine Methode hinzufügen, die alle Bilder auf eine andere Größe skaliert. Dann können wir **alle Bilder einmal skalieren, wenn sie geladen werden**, und nur die skalierte Form im BienenControl und für den Hintergrund des Stock-Formulars nehmen.

Tun Sie das!

1 Fügen Sie dem Renderer die Methode BildSkalieren hinzu.

Alle Grafiken in Ihrem Projekt (wie `Properties.Resources.Flower`) sind als `Bitmap`-Objekte gespeichert. Hier ist eine statische Methode, die die Bitmaps skaliert – fügen Sie sie der Klasse Renderer hinzu:

```
public static Bitmap BildSkalieren(Bitmap grafik, int breite, int höhe) {
    Bitmap skalierteGrafik = new Bitmap(breite, höhe);
    using (Graphics graphics = Graphics.FromImage(skalierteGrafik)) {
        graphics.DrawImage(grafik, 0, 0, breite, höhe);
    }
    return skalierteGrafik;
}
```

Was dieses Graphics-Objekt ist und wie diese Methode funktioniert, sehen wir uns auf den folgenden Seiten genauer an.

2 Fügen Sie dem BienenControl diese ZellenSkalieren-Methode hinzu.

Das BienenControl kann seine eigenen Bitmap-Objekte speichern – ein Array mit vieren davon. Hier ist eine Methode, die dieses Array füllt und alle Bitmaps so skaliert, dass sie die richtige Größe haben.

```
private Bitmap[] zellen = new Bitmap[4];
private void ZellenSkalieren() {
  zellen[0] = Renderer.BildSkalieren(Properties.Resources.Bee_animation_1, Width, Height);
  zellen[1] = Renderer.BildSkalieren(Properties.Resources.Bee_animation_2, Width, Height);
  zellen[2] = Renderer.BildSkalieren(Properties.Resources.Bee_animation_3, Width, Height);
  zellen[3] = Renderer.BildSkalieren(Properties.Resources.Bee_animation_4, Width, Height);
}
```

3 Lassen Sie die switch-Anweisung das Array zellen und nicht die Ressourcen verwenden.

Der Tick-Event-Handler des BienenControl hat eine switch-Anweisung, die BackgroundImage setzt:

```
BackgroundImage = Properties.Resources.Bee_animation_1;
```

Ersetzen Sie `Properties.Resources.Bee_animation_1` mit `zellen[0]`. Ersetzen Sie die anderen **case**-Zeilen so, dass Fall 2 `zellen[1]` nutzt, Fall 3 `zellen[2]` usw. Lassen Sie den Default-Fall `zellen[1]` nutzen. Dann werden nur die skalierten Bilder angezeigt.

4 Fügen Sie dem BienenControl Aufrufe an ZellenSkalieren() hinzu.

Sie brauchen zwei Aufrufe von ZellenSkalieren(). **Fügen Sie** den ersten unten in den Konstruktoreode ein. Kehren Sie dann in den Designer zurück, indem Sie im Fenster Eigenschaften auf das BienenControl doppelklicken. Gehen Sie zur Events-Seite, scrollen Sie zu Resize und doppelklicken Sie darauf, um einen **Resize-Event-Handler hinzuzufügen**. Lassen Sie diesen ebenfalls ZellenSkalieren() aufrufen – dann werden die Animationsgrafiken skaliert, wenn die Größe des Formulars geändert wird.

5 Setzen Sie den Formularhintergrund manuell.

Gehen Sie ins Fenster Eigenschaften und setzen Sie den Hintergrund des Formulars auf `(none)`. Gehen Sie dann zum Konstruktor und setzen Sie das Bild auf eine korrekt skalierte Version.

```
public partial class StockFormular: Form {
    public StockFormular() {
        InitializeComponent();
        BackgroundImage = Renderer.BildSkalieren(
                Properties.Resources.Hive__inside_,
                ClientRectangle.Width, ClientRectangle.Height);
    }
}
```

Das Formular hat die Eigenschaft `ClientRectangle`, die ein `Rectangle` mit den Ausmaßen des Anzeigebereichs enthält.

Führen Sie jetzt den Simulator aus – er ist viel schneller!

Bitmaps skalieren Sie über ein Graphics-Objekt

Formulare und Steuerelemente haben die Methode CreateGraphics(), die ein neues Graphics-Objekt liefert. Dazu werden wir Ihnen gleich mehr erklären.

Sehen wir uns die gerade hinzugefügte Methode `BildSkalieren()` genauer an. Zunächst erstellt sie ein neues `Bitmap`-Objekt, das die Größe hat, auf die die Grafik skaliert wird. Dann nutzt sie `Graphics.FromImage()`, um **ein neues Graphics-Objekt zu erstellen**. Über die `DrawImage()`-Methode des Graphics-Objekts wird das Bild auf die `Bitmap` gezeichnet. Beachten Sie die an `DrawImage()` übergebenen Parameter `breite` und `höhe` – über sie geben Sie die Größe an, auf die das Bild skaliert werden soll. Schließlich wird die neu erzeugte `Bitmap` zurückgeliefert, damit sie als Formularhintergrund oder als eine der vier Animationszellen verwendet werden kann.

Sie übergeben der Methode eine Bitmap und die neue Breite und Höhe, auf die die Bitmap skaliert werden soll.

```
public static Bitmap BildSkalieren(Bitmap grafik, int breite, int höhe) {
    Bitmap skalierteGrafik = new Bitmap(breite, höhe);
    using (Graphics graphics = Graphics.FromImage(skalierteGrafik)) {
        graphics.DrawImage(grafik, 0, 0, breite, höhe);
    }
    return skalierteGrafik;
}
```

Die Methode FromImage() liefert ein neues Graphics-Objekt, über das Sie ein Bild in eine Bitmap zeichnen können. Nutzen Sie die IntelliSense-Funktion der IDE, um sich die Methoden der Klasse Graphics anzusehen. Ein Aufruf von DrawImage() kopiert die Bitmap grafik an den Ort (0, 0) und skaliert sie auf die Parameter `breite` und `höhe`.

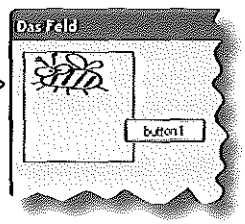
Schauen wir uns das Skalieren im Einsatz an

Tun Sie das nur vorübergehend. Löschen Sie den Code und den Button, wenn Sie fertig sind.

Ziehen Sie einen Button auf das Feld-Formular und fügen Sie diesen Code ein. Er erstellt ein neues PictureBox-Steuerelement mit 100 x 100 Pixeln und setzt seinen Rahmen auf eine schwarze Linie, damit Sie seine Größe erkennen können. Dann nutzt er `BildSkalieren()`, um eine Bienengrafik zu erstellen, die auf 80 x 40 Pixel gequetscht wurde, und weist diese der Eigenschaft `Image` hinzu. Die PictureBox wird dem Formular hinzugefügt, die Biene wird angezeigt.

```
private void button1_Click(object sender, EventArgs e)
{
    PictureBox bienenGrafik = new PictureBox();
    bienenGrafik.Location = new Point(10, 10);
    bienenGrafik.Size = new Size(100, 100);
    bienenGrafik.BorderStyle = BorderStyle.FixedSingle;
    bienenGrafik.Image = Renderer.BildSkalieren(
        Properties.Resources.Bee_animation_1, 80, 40);
    Controls.Add(bienenGrafik);
}
```

Hier sehen Sie das Skalieren von Grafiken in Aktion – die gequetschte Bienengrafik ist viel kleiner als die PictureBox. BildSkalieren() hat sie geschrumpft.



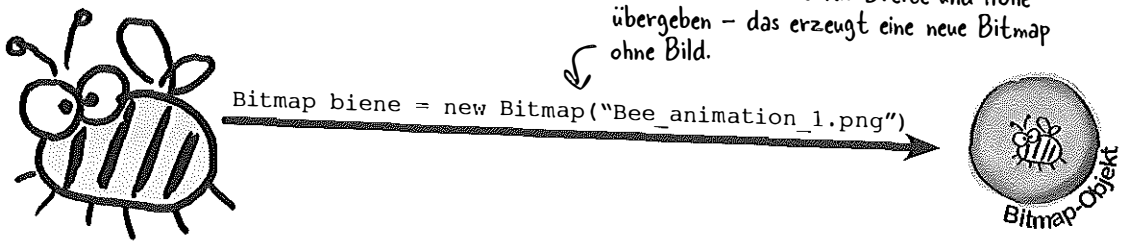
BildSkalieren() erzeugt ein Graphics-Objekt, um auf ein unsichtbares Bitmap-Objekt zu zeichnen. Diese Bitmap wird zurückgeliefert, damit sie im Formular oder einer PictureBox angezeigt werden kann.

Ihre Grafikressourcen werden als Bitmap-Objekt gespeichert

Was passiert mit Grafikdateien, wenn sie in Ihre Projektressourcen importiert werden? Sie wissen bereits, dass Sie darauf über `Properties.Resources` zugreifen können. Aber was genau macht Ihr Programm damit, nachdem sie importiert wurden?

.NET wandelt Ihre Grafik in ein neues **`Bitmap`**-Objekt um:

Die Klasse Bitmap hat verschiedene überladene Konstruktoren. Dieser lädt eine Grafikdatei von der Festplatte. Sie können ihr auch int-Werte für Breite und Höhe übergeben – das erzeugt eine neue Bitmap ohne Bild.



Bee_animation_1.png

Dann werden die Bitmaps auf dem Bildschirm gezeichnet

Befinden sich Ihre Grafiken in `Bitmap`-Objekten, zeichnet Ihr Formular sie mit einem Aufruf wie diesem auf dem Bildschirm.

Dieser Aufruf bewirkt, dass ein Graphics-Objekt auf dem Formular gezeichnet wird. Er nutzt eine using-Anweisung, um sicherzustellen, dass das Graphics-Objekt entsorgt wird.

```
using (Graphics g = CreateGraphics()) {
    g.DrawImage(myBitmap, 30, 30, 150, 150);
}
```

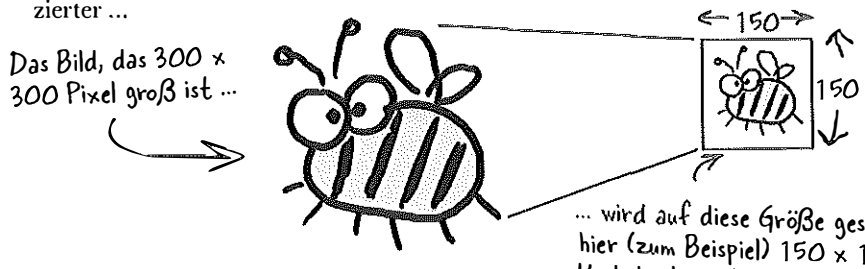
DrawImage() nimmt eine Bitmap, das zu zeichnende Bild ...

... eine X/Y-Koordinate für den Anfangspunkt ...

... und eine Größe, 150 x 150 Pixel.

Je größer sie sind ...

Haben Sie die letzten beiden Parameter für `DrawImage()` bemerkt? Was ist, wenn das Bild in einer `Bitmap` steckt, die 175 x 175 Pixel groß ist? Die Grafikbibliothek muss das Bild dann so skalieren, dass es in 150 x 150 Pixel passt. Und wenn die `Bitmap` ein Bild enthält, das 1.500 x 2.025 Pixel groß ist? Dann wird das Skalieren noch komplizierter ...



Das Skalieren von Bildern benötigt sehr viel Systemressourcen! Tun Sie das einmal, ist das keine große Angelegenheit. Aber wenn Sie es in JEDEM FRAME tun, bremst das Ihr Programm. Wir haben Ihnen für die Bienen und den Stock SEHR GROSSE Bilder gegeben. Wenn der Renderer die Bienen bewegt (insbesondere vor der Innenansicht des Stocks), muss er sie immer wieder skalieren. Das hat zu den Leistungsproblemen geführt!

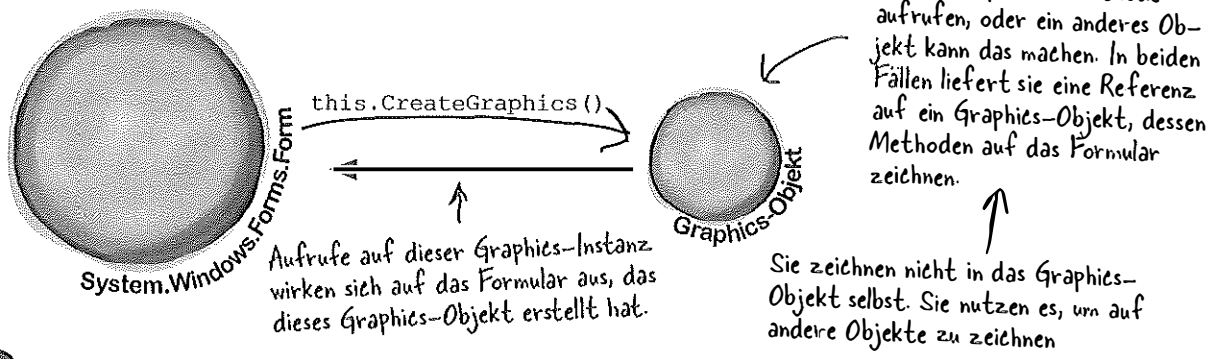
Nutzen Sie System.Drawing, um Ihre Grafiken SELBST zu STEUERN

Die Klasse Graphics ist Teil des Namensraums System.Drawing. Das .NET Framework bietet einige ziemlich mächtige Grafikwerkzeuge, die viel mehr können als das einfache PictureBox-Steuerelement aus der Toolbox. Sie können Figuren zeichnen, Schriften verwenden und die verschiedensten komplizierten Grafikoperationen durchführen ... und all das beginnt mit dem Graphics-Objekt. Jedes Mal, wenn Sie die Grafiken oder Bilder eines Objekts erweitern oder verändern, erzeugen Sie ein Graphics-Objekt, das **mit dem Objekt verknüpft ist, auf das Sie zeichnen möchten**, und nutzen dann die Methoden dieses Graphics-Objekts, um auf Ihr Ziel zu zeichnen.

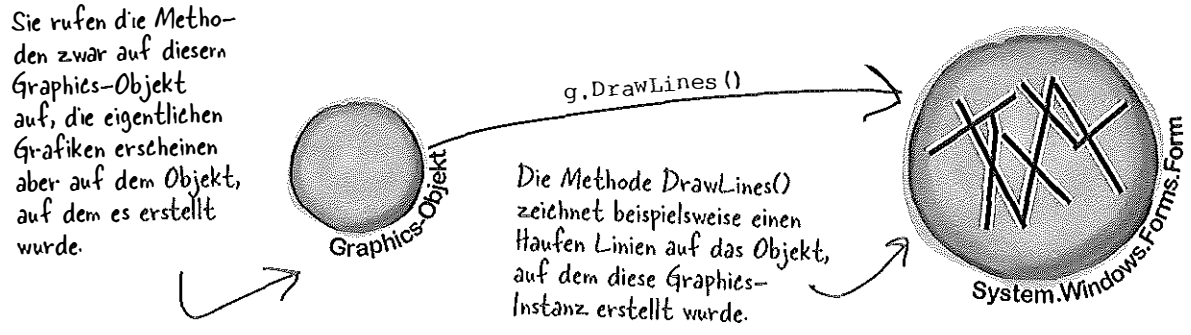
System.Drawing

Die Grafikmethoden im Namensraum System.Drawing werden gelegentlich als GDI+ bezeichnet. Das steht für Graphics Device Interface. Zeichnen Sie mit GDI+ Grafiken, beginnen Sie mit einem Graphics-Objekt, das mit einer Bitmap, einem Formular, einem Steuerelement oder einem anderen Objekt verknüpft ist, das Sie mit dem Methoden des Graphics-Objekts zeichnen möchten.

1 **Beginnen Sie mit dem Objekt, auf das Sie zeichnen möchten.**
Nehmen Sie beispielsweise ein Formular. Rufen Sie die `CreateGraphics()`-Methode des Formulars auf, liefert es eine `Graphics`-Instanz, die dafür eingerichtet ist, auf das Formular selbst zu zeichnen.



2 **Zeichnen Sie mit den Methoden des Graphics-Objekts auf Ihr Objekt.**
`Graphics`-Objekte bieten Methoden, über die Sie auf das Objekt zeichnen können, auf dem sie erstellt wurden. Rufen Sie Methoden auf dem `Graphics`-Objekt auf, um Linien, Kreise, Rechtecke, Text und Bilder zu zeichnen, erscheinen sie auf dem Formular.



Ein GDI+-Schnellkurs

Es gibt alle möglichen Figuren und Bilder, die Sie zeichnen können, wenn Sie ein `Graphics`-Objekt erstellt haben. Sie müssen nur seine Methoden aufrufen, es zeichnet dann direkt auf das Objekt, auf dem es erstellt wurde.

Wenn Sie diese Methoden verwenden möchten, müssen Sie darauf achten, dass Sie am Anfang Ihrer Klassendatei eine `using System.Drawing;`-Zeile haben. Fügen Sie Ihrem Projekt ein Formular hinzu, fügt die IDE sie der Formularklasse automatisch hinzu.

1. Zunächst müssen Sie sich immer ein Graphics-Objekt beschaffen. Nutzen Sie dazu die `CreateGraphics()`-Methode eines Formulars oder lassen Sie sich ein `Graphics`-Objekt als Parameter übergeben. Denken Sie daran, Graphics implementiert die Schnittstelle `IDisposable()`. Verwenden Sie also eine `using`-Anweisung, wenn Sie eins erstellen:

```
using (Graphics g = this.CreateGraphics()) {
```

Denken Sie daran, dass Sie über diese Instanz in das Objekt zeichnen, auf dem sie erstellt wurde.

2. Wenn Sie eine Linie zeichnen möchten, rufen Sie `DrawLine()` mit einem Start- und einem Endpunkt auf, die jeweils durch X/Y-Koordinaten angegeben werden:

```
g.DrawLine(Pens.Blue, 30, 10, 100, 45);
```

Die Startkoordinaten ...

... und die Endkoordinaten.

Oder Sie können zwei `Points`-Objekte verwenden:

```
g.DrawLine(Pens.Blue, new Point(30, 45), new Point(100, 10));
```

3. Dieser Code zeichnet ein gefülltes, schiefergraues Rechteck und gibt ihm dann einen himmelblauen Rahmen. Die Dimensionen werden über ein `Rectangle` definiert – hier befindet sich die obere linke Ecke bei (150, 15). Breite und Höhe sind 140 bzw. 90 Pixel.

```
g.FillRectangle(Brushes.SlateGray, new Rectangle(150, 15, 140, 90));
g.DrawRectangle(Pens.SkyBlue, new Rectangle(150, 15, 140, 90));
```

Sie können viele verschiedene Farben verwenden – geben Sie einfach »Color«, »Pens« oder »Brushes« und einen Punkt ein. IntelliSense zeigt sie dann an.

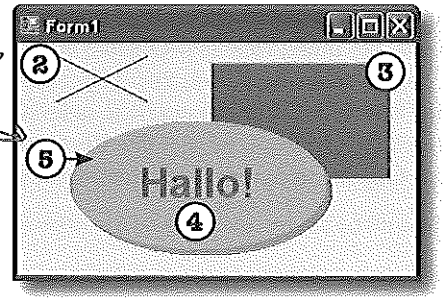
4. Mit `DrawCircle()` oder `FillCircle()` können Sie Kreise oder Ellipsen zeichnen. Auch hier wird die Größe der Figur über ein `Rectangle` angegeben. Dieser Code zeichnet zwei etwas verschobene Ellipsen und simuliert so einen Schatteneffekt:

```
g.FillEllipse(Brushes.DarkGray, new Rectangle(45, 65, 200, 100));
g.FillEllipse(Brushes.Silver, new Rectangle(40, 60, 200, 100));
```

5. Mit `DrawString()` zeichnen Sie Text in einer beliebigen Farbe und einer beliebigen Schrift. Dazu brauchen Sie ein `Font`-Objekt. `Font` implementiert `IDisposable`. Nutzen Sie also auch hier eine `using`-Anweisung:

```
using (Font arial24Bold = new Font("Arial", 24, FontStyle.Bold)) {
   g.DrawString("Hallo!", arial24Bold, Brushes.Red, 90, 90);
}
```

Werden diese Anweisungen nacheinander ausgeführt, erhalten Sie dieses Formular. Die Zahlen darauf entsprechen den Nummern oben. Die Koordinate (0, 0) ist oben links.



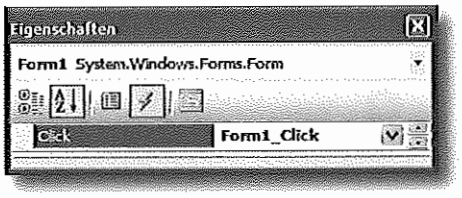
Schritt 1 sehen Sie in dieser Grafik nicht, da darin das eigentliche Graphics erstellt wurde.

Mit Graphics ein Bild auf ein Formular zeichnen

Erstellen wir eine neue Windows-Anwendung, die auf ein Formular zeichnet, **wenn Sie darauf klicken.**

Zeichnen Sie das

1 Beginnen Sie damit, dass Sie dem Formular ein Click-Event hinzufügen.
Gehen Sie zur **Events-Seite des Eigenschaften-Fensters** (indem Sie auf den Blitz klicken), scrollen Sie herunter zum Click-Event und doppelklicken Sie darauf.



Beginnen Sie den Event-Handler mit einer `using`-Zeile, in der das Graphics-Objekt erstellt wird. Wenn Sie mit GDI+ arbeiten, nutzen Sie viele Objekte, die `IDisposable` implementieren. Entsorgen Sie diese nicht, saugen sie nach und nach die Ressourcen Ihres Computers auf, bis Sie das Programm beenden. Sie werden also eine Menge `using`-Anweisung verwenden müssen:

Das ist die erste Zeile in Ihrem Form1_Click()-Event-Handler. Wir geben Ihnen alle Zeilen für den Event-Handler – Sie müssen sie nur noch zusammensetzen.

```
using (Graphics g = CreateGraphics()) {
```

2 Achten Sie auf die Reihenfolge, in der Sie Dinge auf Ihr Formular zeichnen.
Für diese Grafik möchten wir einen himmelblauen Hintergrund haben. Wir zeichnen also zunächst ein großes blaues Rechteck – alles, was Sie danach zeichnen, wird **darüber** gezeichnet. Dazu nutzen Sie die `ClientRectangle`-Eigenschaft des Formulars. Das ist ein `Rectangle`, das die Grenzen des Inhaltsbereichs des Formulars definiert. `Rectangles` sind sehr nützlich – ein Rechteck können Sie erstellen, indem Sie einen `Point` für die linke obere Ecke, eine Breite und eine Höhe angeben. Tun Sie das, werden die Eigenschaften `Top`, `Left`, `Right` und `Bottom` automatisch für Sie berechnet. Und es hat **nützliche Methoden wie `Contains()`, die true liefert, wenn sich ein angegebener Point im Rectangle befindet.**

Das wird sich später im Buch noch als praktisch erweisen! Haben Sie eine Idee, was Sie mit Contains() tun werden?

```
g.FillRectangle(Brushes.SkyBlue, ClientRectangle);
```

3 Die Biene und die Blume zeichnen.
Sie wissen bereits, wie die Methode `DrawImage()` funktioniert.

```
g.DrawImage(Properties.Resources.Bee_animation_1, 50, 20, 75, 75);
g.DrawImage(Properties.Resources.Flower, 10, 130, 100, 150);
```

Mit Pen-Objekten zeichnen Sie Linien. Diese haben eine Breite. Möchten Sie eine gefüllte Figur oder Text zeichnen, brauchen Sie ein Brush-Objekt.

4 Holen Sie sich einen Pen zum Zeichnen.
Wenn Sie eine Linie zeichnen, nutzen Sie ein Pen-Objekt, um ihre Farbe und Stärke festzulegen. Es gibt eine eingebaute `Pens`-Klasse, die Ihnen viele Pen-Objekte gibt (`Pens.Red` ist beispielsweise ein dünner roter Stift). Aber mit dem Konstruktor der Klasse `Pen` können Sie eigene Pen-Objekte erstellen. Er nimmt ein `Brush`-Objekt und eine Strichstärke (das ist ein float, achten Sie also auf das `F` am Ende). Mit Brush-Objekten zeichnen Sie ausgefüllte Grafiken (wie gefüllte Rechtecke oder Ellipsen). Es gibt eine Klasse `Brushes`, die Ihnen Brush-Objekte in unterschiedlichen Farben liefert.

```
using (Pen schwarzerStift = new Pen(Brushes.Black, 3.0F)) {
```

Das kommt in die using-Anweisung, die den Pen erstellt.

5 Zeichnen Sie einen Pfeil, der auf die Blume weist.

Es gibt einige Graphics-Methoden, die ein Array mit Point-Objekten annehmen und diese mit einer Folge von Linien oder Kurven verbinden. Für die Pfeilspitze werden wir `DrawLines()` verwenden, für den Schaft `DrawCurve()`. Es gibt noch weitere Methoden, die auch Point-Arrays übernehmen (wie `DrawPolygon()`, die eine geschlossene Figur zeichnet, und `FillPolygon()`, die sie ausfüllt).

```
g.DrawLines(schwarzerStift, new Point[] {
   new Point(130, 110), new Point(120, 160), new Point(155, 163)});
g.DrawCurve(schwarzerStift, new Point[] {
   new Point(120, 160), new Point(175, 120), new Point(215, 70) });
}
```

Hier endet der using-Block – wir brauchen schwarzerStift nicht mehr. Das Objekt kann also entsorgt werden.

Übergeben Sie DrawCurve() ein Array mit Point-Objekten, wird eine glatte Kurve gezeichnet, die sie der Reihe nach verbindet.

6 Holen Sie sich ein Font-Objekt, um Text zu zeichnen.

Wenn Sie Text zeichnen, müssen Sie zunächst ein `Font`-Objekt erstellen. Auch hier sollten Sie wieder eine using-Anweisung verwenden. `Font` implementiert ebenfalls `IDisposable`. Das Erstellen eines Font-Objekts ist einfach. Es gibt mehrere überladene Konstruktoren – der einfachste nimmt einen Font-Namen, eine Font-Größe und einen `FontStyle`-Wert.

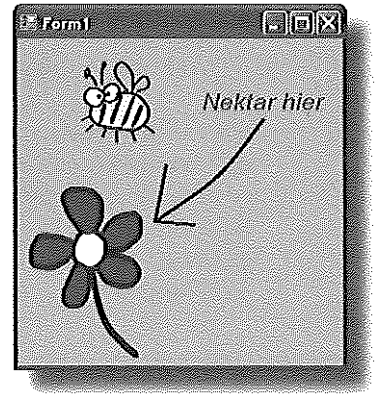
```
using (Font font = new Font("Arial", 16, FontStyle.Italic)) {
```

7 Fügen Sie den Text »Nektar hier« hinzu.

Nachdem Sie einen Font haben, können Sie ermitteln, wie Sie den String positionieren müssen, indem Sie messen, wie groß er ist, wenn er gezeichnet wird. Die Methode `MeasureString()` liefert ein `SizeF`-Objekt, das seine Größe angibt. (`SizeF` ist nur eine `float`-Version von `Size` – beide geben einfach eine Breite und eine Höhe an.) Da wir wissen, wo der Pfeil endet, nutzen wir die String-Maße, um den Text genau über dem Pfeil zu zentrieren.

```
      SizeF size = g.MeasureString("Nektar hier", font);
      g.DrawString("Nektar hier", font, Brushes.Red, new Point(
            215 - (int)size.Width / 2, 70 - (int)size.Height));
   }
}
```

Achten Sie darauf, dass Sie beide using-Blöcke schließen.



Ein Rectangle können Sie über ein Point- und ein Size-Objekt (oder Breite und Höhe) erstellen. Mit Contains() können Sie feststellen, ob ein Point innerhalb des Rechtecks liegt.

Spitzen Sie Ihren Bleistift

1. Bei der Arbeit mit Graphics-Objekten müssen Sie sich Ihr Formular häufig als ein Raster mit x/y-Koordinaten vorstellen. Hier ist der Code, mit dem Sie das Raster unten erstellen. Sie sollen die fehlenden Teile ergänzen.

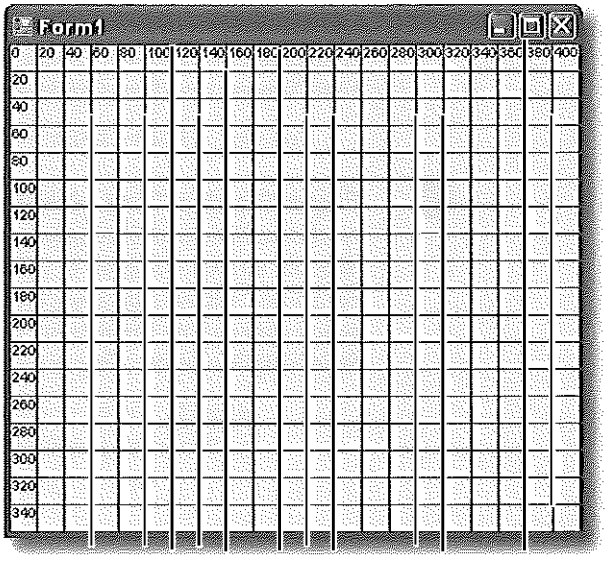
```
using (Graphics g = this.CreateGraphics())
using (Font f = new Font("Arial", 6, FontStyle.Regular)) {
  for (int x = 0; x < this.Width; x += 20) {

     ..........

  } ..........
  for (int y = 0; y < this.Height; y += 20) {

     ..........

  } ..........
}
```



2. Können Sie sich vorstellen, was passiert, wenn der Code unten ausgeführt wird? Zeichnen Sie die Ausgabe auf das Formular und nutzen Sie das gerade gezeichnete Raster, um die einzelnen Punkte zu lokalisieren.

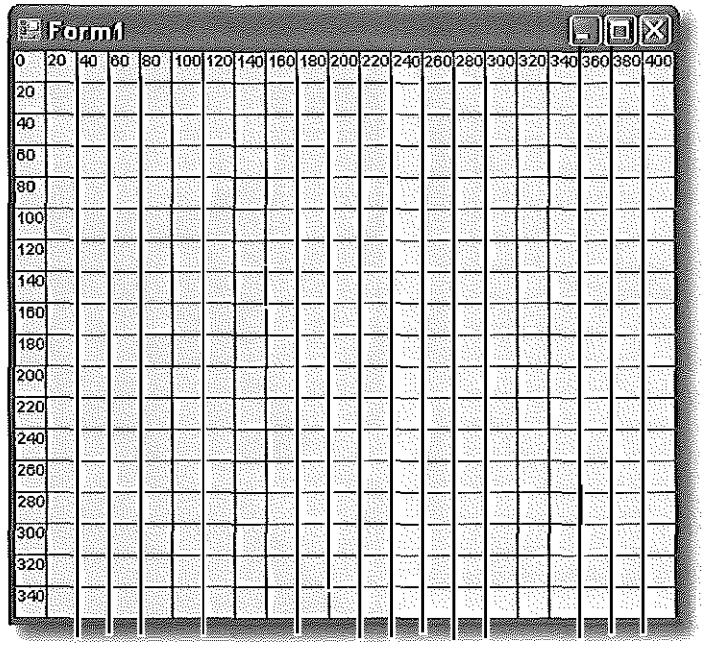
```
using (Pen pen =
         new Pen(Brushes.Black, 3.0F)) {
   g.DrawCurve(pen, new Point[] {
      new Point(80, 60),
      new Point(200,40),
      new Point(180, 60),
      new Point(300,40),
   });
   g.DrawCurve(pen, new Point[] {
      new Point(300,180), new Point(180, 200),
      new Point(200,180), new Point(80, 200),
   });
   g.DrawLine(pen, 300, 40, 300, 180);
   g.DrawLine(pen, 80, 60, 80, 200);
   g.DrawEllipse(pen, 40, 40, 20, 20);
   g.DrawRectangle(pen, 40, 60, 20, 300);
   g.DrawLine(pen, 60, 60, 80, 60);
   g.DrawLine(pen, 60, 200, 80, 200);
}
```

FillPolygon(), DrawLines() und ein paar weitere Graphics-Methoden haben Überladungen, die ein Array mit Point-Objekten übernehmen, die Eckpunkte einer Folge verbundener Linien definieren.

3. Hier ist noch etwas Graphics-Code, der ungleichmäßige Figuren zeichnet. Ermitteln Sie, was gezeichnet wird, auf Basis des Rasters, das Sie unter dem Code sehen.

```
g.FillPolygon(Brushes.Black, new Point[] {
        new Point(60,40), new Point(140,80), new Point(200,40),
        new Point(300,80), new Point(380,60), new Point(340,140),
        new Point(320,180), new Point(380,240), new Point(320,300),
        new Point(340,340), new Point(240,320), new Point(180,340),
        new Point(20,320), new Point(60, 280), new Point(100, 240),
        new Point(40, 220), new Point(80,160),
    });

using (Font fett = new Font("Times New Roman", 24, FontStyle.Italic)) {
    g.DrawString("Buh!", fett, Brushes.White, new Point(80, 80));
    g.DrawString("Buh!", fett, Brushes.White, new Point(120, 120));
    g.DrawString("Buh!", fett, Brushes.White, new Point(160, 160));
    g.DrawString("Buh!", fett, Brushes.White, new Point(200, 200));
    g.DrawString("Buh!", fett, Brushes.White, new Point(240, 240));
}
```



Spitzen Sie Ihren Bleistift
Lösung

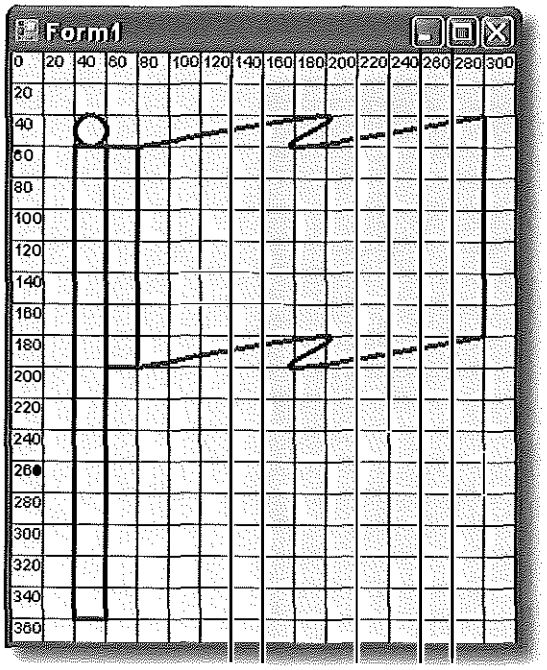
Sie sollten den fehlenden Code zum Zeichnen eines Rasters ergänzen und einzeichnen, was zwei Codefragmente in das Raster zeichnen.

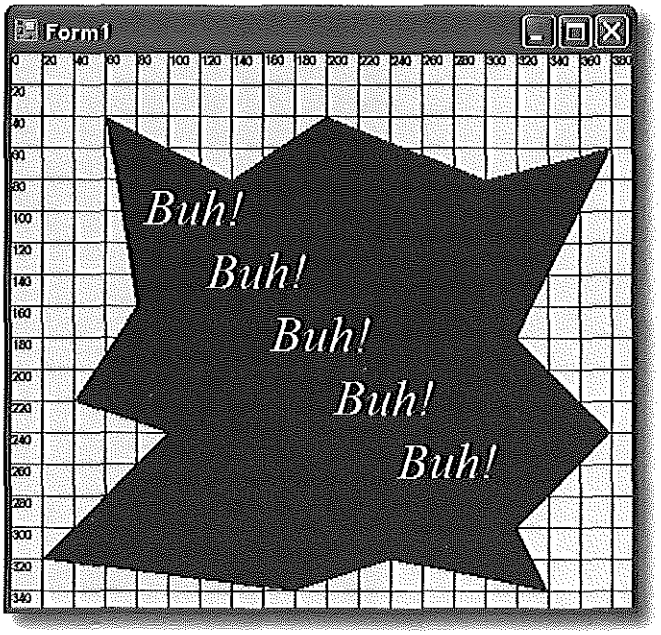
```
using (Graphics g = this.CreateGraphics())
using (Font f = new Font("Arial", 6, FontStyle.Regular)) {
    for (int x = 0; x < this.Width; x += 20) {
        g.DrawLine(Pens.Black, x, 0, x, this.Height);
        g.DrawString(x.ToString(), f, Brushes.Black, x, 0);
    }
    for (int y = 0; y < this.Height; y += 20) {
        g.DrawLine(Pens.Black, 0, y, this.Width, y);
        g.DrawString(y.ToString(), f, Brushes.Black, 0, y);
    }
}
```

Erst zeichnen wir die vertikalen Linien und die Zahlen auf der y-Achse. Es gibt alle 20 Pixel eine vertikale Linie.

Wir haben using-Anweisungen verwendet, um sicherzustellen, dass das Graphics- und das Font-Objekt entsorgt werden, wenn das Formular gezeichnet ist.

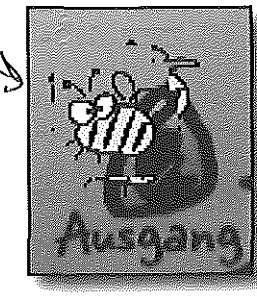
Dann haben wir die horizontalen Linien und die Zahlen auf der x-Achse gezeichnet. Um eine horizontale Linie zu zeichnen, wählen Sie einen y-Wert und zeichnen eine Linie von (0, y) auf der linken Seite des Formulars zu (this.Width, y) auf der rechten Seite des Formulars.





Graphics kann unser Transparenzproblem lösen

Die Bienen wurden so gezeichnet, dass sie seltsam aussahen, wenn sie einander überschnitten.



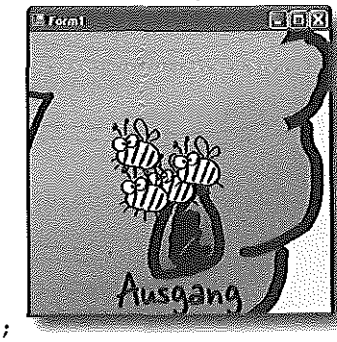
Wahrscheinlich haben Sie schon erkannt, dass `DrawImage()` der Schlüssel für die Reparatur des Renderer-Problems ist, das dazu führte, dass die Boxen um die Bienen zu Ausschneidungen führen, wenn sich Bienen und Blumen überschneiden. Gehen wir also jetzt dieses Problem an!



1. Fügen Sie eine `BieneZeichnen()`-Methode ein, die eine Biene auf ein Graphics-Objekt zeichnet. Sie nutzt die überladene `DrawImage()`-Version, die ein Rectangle annimmt, das sagt, wo das Bild gezeichnet wird und wie groß es ist.

```
public void BieneZeichnen(Graphics g, Rectangle rect) {
  g.DrawImage(Properties.Resources.Bee_animation_1, rect);
}
```

Viel besser – klicken Sie auf das Formular, überschneiden sich die Bienen.



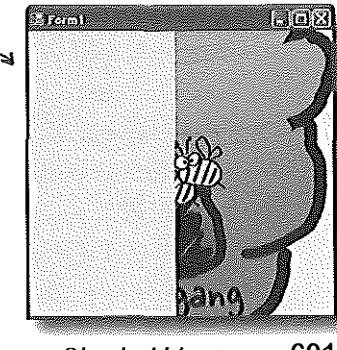
2. Hier ist der neue **Click-Event-Handler für das Formular**. Achten Sie genau darauf, wie er funktioniert – er zeichnet den Stock so, dass seine linke obere Ecke etwas außerhalb des Formulars am Ort `(-Width, -Height)` liegt und er doppelt so breit und hoch wie das Formular ist – auch wenn das Formular vergrößert wird, wird er also immer noch ordentlich gezeichnet. Dann werden mit `BieneZeichnen()` vier Bienen gezeichnet.

```
private void Form1_Click(object sender, EventArgs e) {
    using (Graphics g = CreateGraphics()) {
        g.DrawImage(Properties.Resources.Hive__inside_,
            -Width, -Height, Width * 2, Height * 2);
        Size size = new Size(Width / 5, Height / 5);
        BieneZeichnen(g, new Rectangle(
                new Point(Width / 2 - 50, Height / 2 - 40), size));
        BieneZeichnen(g, new Rectangle(
                new Point(Width / 2 - 20, Height / 2 - 60), size));
        BieneZeichnen(g, new Rectangle(
                new Point(Width / 2 - 80, Height / 2 - 30), size));
        BieneZeichnen(g, new Rectangle(
                new Point(Width / 2 - 90, Height / 2 - 80), size));
    }
}
```

Erst zeichnen wir den Stockhintergrund so, dass wir nur einen kleinen Teil davon sehen. Dann werden die vier Bienen so gezeichnet, dass sie sich überschneiden – machen sie das nicht, sollten Sie das Formular vergrößern und erneut auf den Button klicken.

... aber es gibt ein Problem

Schauen Sie, was passiert, wenn Sie das Formular über den Bildschirmrand schieben! Mist!



3. Führen Sie das Programm aus, klicken Sie auf das Formular und schauen Sie sich die Bienen an! Aber etwas ist immer noch falsch. Schieben Sie das Formular über die Seite des Bildschirms hinaus und ziehen Sie es dann wieder zurück. **Das Bild verschwindet!** Gehen Sie jetzt zu dem »Nektar hier«-Programm, das Sie vor ein paar Seiten geschrieben haben – **es hat das gleiche Problem!**

Was, glauben Sie, ist passiert?

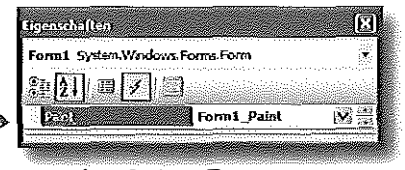
Paint macht Ihre Zeichnungen dauerhaft

Das Paint-Event von Formularen und Steuerelementen liefert ein Graphics-Objekt, das seine Zeichnungen automatisch neu zeichnet.

Was bringen Zeichnungen, die vom Formular verschwinden, sobald es verdeckt wird? Überhaupt nichts. Glücklicherweise können Sie mit einem einfachen Mittel sicherstellen, dass Ihre Zeichnungen auf dem Formular bleiben: **Schreiben Sie einfach einen Paint-Event-Handler**. Ihr Formular setzt ein Paint-Event ab, wenn es sich selbst neu zeichnen muss – beispielsweise nachdem es den Bildschirm verlassen hatte. Eine der Eigenschaften seines `PaintEventArgs`-Parameters ist ein Graphics namens `Graphics`. Alles, was Sie darauf zeichnen, bleibt bestehen.

1. **Fügen Sie einen Paint-Event-Handler hinzu.**
Klicken Sie auf der Event-Seite des Fensters Eigenschaften doppelt auf »Paint«, um einen Paint-Event-Handler einzufügen. Das Paint-Event wird abgesetzt, wenn das Bild auf Ihrem Formular »beschädigt« wird. Erneuern Sie Ihre Zeichnungen dann, sorgt das dafür, dass sie erhalten bleiben.

Ein Doppelklick auf Paint fügt einen Paint-Event-Handler hinzu. Alles, was Sie über die Graphics-Eigenschaft seines PaintEventArgs zeichnen, bleibt auf Ihrem Formular erhalten.



2. **Nutzen Sie das Graphics-Objekt aus den EventArgs des Paint-Events.**
Statt mit einer using-Anweisung lassen Sie Ihren Event-Handler folgendermaßen beginnen:

```
private void Form1_Paint(object sender, PaintEventArgs e) {
    Graphics g = e.Graphics;
```

Sie **müssen** keine **`using`**-Anweisung verwenden – da Sie das Objekt nicht erstellt haben, **müssen Sie es auch nicht entsorgen**.

3. **Kopieren Sie den Code, der die überlappenden Bienen und den Stock zeichnet.**
Fügen Sie die neue BiencZeichnen()-Methode von der vorigen Seite in das neue Benutzersteuerelement ein. Kopieren Sie dann den Code aus dem Click-Event und das neue Paint-Event – **ohne die erste Zeile mit der using-Anweisung**, weil Sie das Graphics-Objekt g bereits haben. (Achten Sie auch darauf, dass Sie die korrespondierende schließende Klammer entfernen.) Führen Sie das Programm jetzt aus. **Der Inhalt bleibt erhalten!**

Machen Sie Gleiches mit der »Nektar hier«-Zeichnung, damit auch diese erhalten bleibt

Formulare und Steuerelemente zeichnen sich permanent neu

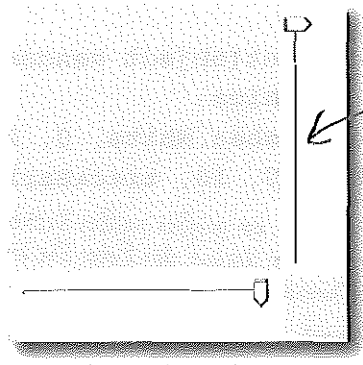
Es mag nicht so scheinen, aber Ihre Formulare müssen sich permanent neu zeichnen. Ist auf einem Formular ein Steuerelement, zeigt es eine Grafik an – Labels zeigen Text, Buttons das Bild eines Buttons, Checkboxen kleine Kästchen mit Kreuz. Sie arbeiten damit als Steuerelemente, die Sie herumziehen können, aber eigentlich zeichnet jedes Steuerelement sein eigenes Bild. Wird ein Teil eines Formulars verdeckt und dann wieder sichtbar, ist der zwischenzeitlich verdeckte Teil des Formulars ungültig, d.h., er zeigt nicht mehr das Bild, das er zeigen sollte. Dann sendet .NET dem Formular eine Nachricht, die ihm sagt, dass es sich neu zeichnen muss. Das Formular setzt das Paint-Event ab, wenn es nicht mehr intakt ist und neu gezeichnet werden muss. Über die Methode `Invalidate()` können Sie ein Formular oder Benutzersteuerelement zwingen, sich selbst neu zu zeichnen.



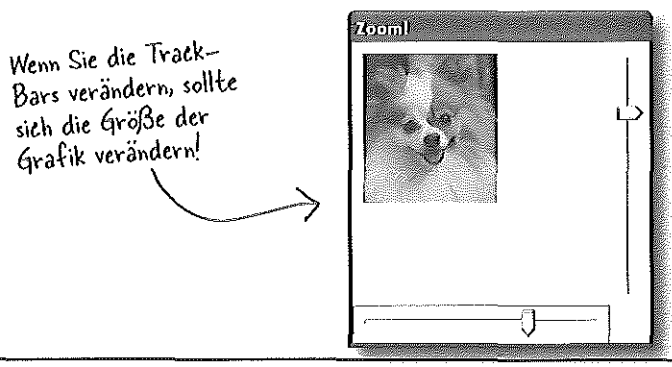
Schauen Sie, ob Sie das, was Sie über Formulare und Benutzersteuerelemente wissen, kombinieren – und gleichzeitig etwas mehr Erfahrung im Umgang mit `Bitmap`-Objekten und der Methode `DrawImage()` sammeln – können, indem wir ein Benutzersteuerelement erstellen, das über TrackBars ein Bild zoomt.

4 Fügen Sie einem neuen Benutzersteuerelement ein TrackBar-Steuerelement hinzu.
Erstellen Sie ein neues Windows-Anwendung-Projekt. **Fügen Sie ihm ein Benutzersteuerelement** namens Zoomer **hinzu** und setzen Sie seine `Size`-Eigenschaft auf (300, 300). Ziehen Sie TrackBar-Steuerelemente aus der Toolbox. Ziehen Sie `trackBar1` an den unteren Rand des Steuerelements. Ziehen Sie dann `trackBar2` auf die rechte Seite und setzen Sie seine `Orientation`-Eigenschaft auf `Vertical`. Bei beiden sollte `Minimum` auf 1, `Maximum` auf 175, `Value` auf 175 und `TickStyle` auf `None` gesetzt sein. Setzen Sie bei beiden TrackBars die Hintergrundfarbe auf Weiß. Klicken Sie dann doppelt auf beide, um für beide einen `Scroll`-Event-Handler einzufügen. Lassen Sie beide Event-Handler die `Invalidate()`-Methode des Formulars aufrufen.

Ein UserControl hat ein Paint-Event, das genau so funktioniert wie jenes, das Sie gerade im Formular verwendet haben. Nutzen Sie einfach den `PaintEventArgs`-Parameter `e`. Er hat die Eigenschaft `Graphic` und alles, was Sie über dieses Graphics-Objekt zeichnen, wird auf jede Instanz des UserControl gezeichnet, die Sie aus der Toolbox ziehen.



3 Laden Sie ein Bild in ein Bitmap-Objekt und zeichen Sie es in das Steuerelement.
Fügen Sie Ihrem Benutzersteuerelement Zoomer ein privates `Bitmap`-Feld namens `foto` hinzu. Nutzen Sie beim Erstellen der `Bitmap` den Konstruktor, um eine Bilddatei zu laden – wir haben einen süßen Hund verwendet. Fügen Sie dem Steuerelement dann einen `Paint`-Event-Handler hinzu. Dieser sollte ein Graphics-Objekt erstellen, über das auf das Steuerelement gezeichnet wird. Der Event-Handler sollte ein gefülltes weißes Rechteck zeichnen, das das gesamte Steuerelement füllt. Dann soll er `DrawImage()` nutzen, um den Inhalt des Felds `foto` so in das Steuerelement zu zeichnen, dass die linke obere Ecke sich beim Punkt (10, 10) befindet, seine Breite gleich `trackBar1.Value` und seine Höhe gleich `trackBar2.Value` ist. Ziehen Sie Ihr Steuerelement anschließend auf ein Formular – passen Sie dessen Größe so an, dass sich die beiden TrackBars an den Rändern befinden.



Wird der Zeiger einer der TrackBars bewegt, wird die Invalidate()-Methode des UserControl aufgerufen. Das bewirkt, dass das UserControl sein Paint-Event absetzt und die Größe des Fotos angepasst wird. Denken Sie daran, dass Sie das Graphics-Objekt nicht entsorgen müssen, da Sie es nicht selbst erstellt haben – es wurde Ihnen per `PaintEventArgs` übergeben. Sie brauchen also keine `using`-Anweisung. Zeichnen Sie im Paint-Event-Handler einfach das Bild.



Lösungen zu den Übungen

Sammeln Sie etwas Erfahrung im Umgang mit `Bitmap`-Objekten und der `DrawImage()`-Methode, indem Sie ein Formular aufbauen, das sie nutzt, um ein Foto aus einer Datei zu laden und es zu zoomen.

```
public partial class Zoomer : UserControl {

  Bitmap foto = new Bitmap(@"c:\Graphics\fluffy_dog.jpg");

  public Zoomer() {
     InitializeComponent();
  }

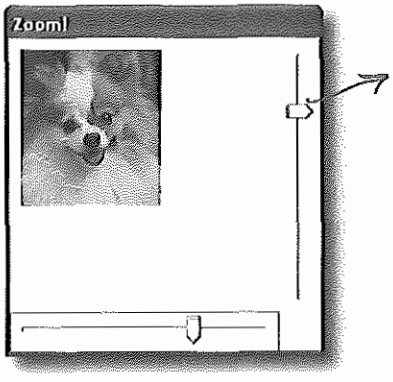
  private void Zoomer_Paint(object sender, PaintEventArgs e) {
      Graphics g = e.Graphics;
      g.FillRectangle(Brushes.White, 0, 0, Width, Height);
      g.DrawImage(foto, 10, 10, trackBar1.Value, trackBar2.Value);
    }
  }

  private void trackBar1_Scroll(object sender, EventArgs e) {
     Invalidate();
  }
  private void trackBar2_Scroll(object sender, EventArgs e) {
     Invalidate();
  }
}
```

Dieser Bitmap-Konstruktor lädt ein Bild aus einer Datei. Es gibt weitere überladene Konstruktoren, z.B. einem über den Sie eine Breite und eine Höhe angeben – er erstellt eine leere Bitmap.

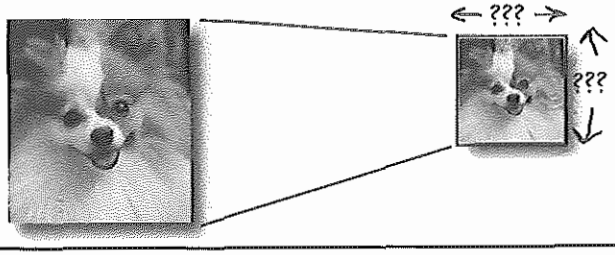
Erst zeichnen wir ein großes weißes Rechteck, das das ganze Formular ausfüllt. Dann zeichnen wir darauf das Foto. Die letzten beiden Parameter geben die Größe des zu zeichnenden Bilds an – trackBar1 setzt die Breite, trackBar2 die Höhe.

Wird der Schieber eines der TrackBar-Steuerelemente verschoben, wird das Scroll-Event abgesetzt. Indem wir die Event-Handler die `Invalidate()`-Methode des Steuerelements aufrufen lassen, bewirken wir, dass sich das Formular neu zeichnet … tut es das, zeichnet es eine neue Kopie des Bilds mit einer anderen Größe.



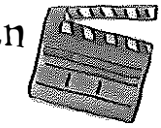
Jede Bewegung hier bewirkt eine Veränderung der Bildgröße durch DrawImage().

```
g.DrawImage(myBitmap, 30, 30, 150, 150);
```



Das Neuzeichnen von Formularen und Steuerelementen aus der Nähe betrachtet

Hinter den Kulissen



Erinnern Sie sich, dass Sie die grafische Darstellung erst richtig in die Hand nehmen, wenn Sie beginnen, mit Graphics-Objekten zu arbeiten. Es ist, als würden Sie .NET sagen: »Ich weiß, was ich mache. Ich kann die Verantwortung dafür übernehmen.« Vielleicht möchten Sie beispielsweise verhindern, dass das Formular neu gezeichnet wird, wenn es minimiert oder maximiert wird ... oder Sie möchten es ***häufiger*** neu zeichnen. Wissen Sie einmal, was im Hintergrund mit Ihren Formularen oder Steuerelementen passiert, können Sie das Neuzeichnen selbst in die Hand nehmen:

1. **Formulare haben ein Paint-Event, das die Grafiken auf dem Formular neu zeichnet.**
Gehen Sie zur Event-Liste eines Formulars und suchen Sie das **`Paint`**-Event. Muss sich das Formular neu zeichnen, wird dieses Event abgesetzt. Alle Formulare und Steuerelemente nutzen das `Paint`-Event intern, um zu entscheiden, wann sie neu gezeichnet werden müssen. Aber wer setzt das Event ab? Es wird von einer Methode aufgerufen, die **`OnPaint`** heißt. Diese Methode erben Formulare und Benutzersteuerelemente von der Klasse Control. Nehmen Sie ein Formular und überschreiben Sie OnPaint:

Machen Sie das genau so wie früher bei Dispose().

Überschreiben Sie OnPaint und fügen Sie diese Zeile hinzu.

```
protected override void OnPaint(PaintEventArgs e) {
    Console.WriteLine("OnPaint {0} {1}", DateTime.Now, e.ClipRectangle);
    base.OnPaint(e);
}
```

Ziehen Sie das Formular herum – ziehen Sie es halb vom Bildschirm, minimieren Sie es und verstecken Sie es hinter anderen Fenstern. Achten Sie genau auf die Ausgabe, die geschrieben wird. Sie sehen, dass OnPaint jedes Mal ein Paint-Event absetzt, wenn ein Teil davon **ungültig** wird und neu gezeichnet werden muss. Schauen Sie sich das ClipReetangle genau an: Sie werden bemerken, dass es ein Rechteck ist, das den Teil des Formulars beschreibt, der neu gezeichnet werden muss. Er wird an das `PaintEventArgs`-Argument des Events übergeben, weil es die Leistung verbessern kann, wenn nur der ungültige Teil neu gezeichnet wird.

2. **Invalidate() steuert, wann WAS neu gezeichnet wird.**
.NET löst das Paint-Event aus, wenn etwas auf dem Formular gestört, verdeckt oder vom Bildschirm geschoben und dann wieder angezeigt wird. Es ruft `Invalidate()` auf und übergibt der Methodc ein `Rectangle`. Das `Rectangle` sagt `Invalidate()`, welcher Teil des Formulars neu gezeichnet werden muss ... welcher Teil des Formulars ungültig ist. Dann ruft .NET OnPaint auf, um Ihr Formular aufzufordern, das `Paint`-Event abzusetzen und alles im ungültigen Bereich neu zu zeichnen.

Invalidate() sagt im Wesentlichen, dass ein Teil des Formulars ungültig sein könnte. Dieser wird dann neu gezeichnet.

3. **Update() gibt der Invalidate-Anforderung höchste Priorität.**
Ihr Formular erhält permanent Meldungen. Das gleiche System, das ihm sagt, dass es verdeckt wurde, und OnPaint aufruft, muss auch noch alle möglichen anderen Meldungen senden. Schauen Sie selbst: Geben Sie `override` ein und scrollen Sie durch die Methoden, die mit »On« beginnen – jede davon ist eine Nachricht, auf die Ihr Formular reagiert. `Update()` setzt die Invalidate-Nachricht an den Anfang der Liste.

Rufen Sie sie selbst auf, sagen Sie .NET, dass das gesamte Control ungültig ist und neu gezeichnet werden muss. Sie können ein eigenes ClipReetangle übergeben – das wird an das PaintEventArgs des Paint-Events weitergereicht.

4. **Refresh() ist Invalidate() plus Update()**
Formulare und Steuerelemente bieten Ihnen eine Abkürzung. Sie haben eine Refresh()-Methode, die erst Invalidate() aufruft, um den gesamten Clientbereich ungültig zu machen, und dann Updatc(), um sicherzustellen, dass diese Mcldung an den Anfang der Liste gerückt wird.

Es gibt keine Dummen Fragen

F: Ich denke trotzdem, dass es besser wäre, die Größe der Grafiken in einem Programm wie Paint oder Photoshop anzupassen. Warum kann ich das nicht tun?

A: Das können Sie. Falls Sie die Bilder steuern können, mit denen Sie in Ihren Anwendungen arbeiten, und wenn diese immer die gleiche Größe behalten. Aber meist ist das nicht der Fall. Häufig erhalten Sie Grafiken von einer anderen Quelle, z.B. aus dem Internet oder von einem Kollegen. Oder Sie beziehen ein Bild von einer Quelle, die Sie nur lesen können, und müssen es dann innerhalb der Anwendung anpassen.

F: Aber es wäre besser, wenn ich es außerhalb von .NET anpassen könnte, stimmt's?

A: Wenn Sie sicher sind, dass Sie es nie in einer größeren Größe benötigen, könnte das richtig sein. Aber wenn Ihr Programm das Bild in unterschiedlichen Größen anzeigen muss, müssen Sie es irgendwann doch anpassen. Und soll ihr Bild jemals größer angezeigt werden, als Ihre angepasste Version ist, haben Sie ein richtiges Problem. Bilder lassen sich viel leichter verkleinern als vergrößern.In der Regel ist es besser, wenn man Grafiken über Programme anpassen kann und nicht auf ein externes Programm angewiesen ist oder von Rahmenbedingungen wie nur lesbaren Dateien eingeschränkt wird.

F: Dass CreateGraphics() ein Graphics-Objekt zum Zeichnen auf einem Formular abruft, habe ich verstanden, aber was macht dieser FromImage()-Aufruf in der Methode BildSkalieren()?

A: `FromImage()` ruft das `Graphics`-Objekt für ein `Bitmap`-Objekt ab. Und genau wie `CreateGraphics()` bei einem Aufruf auf einem Formular ein `Graphics`-Objekt liefert, mit dem man auf dieses Formular zeichnen kann, so ruft `FromImage()` ein `Graphics`-Objekt auf der `Bitmap` ab, auf die Methode aufgerufen wurde.

F: Graphics-Objekte werden also nicht nur verwendet, um auf Formulare zu zeichnen?

A: Eigentlich kann man mit einem Graphics-Objekt auf alles zeichnen, d.h. auf das, was Ihnen ein Graphics-Objekt gibt. Die Bitmap gibt Ihnen ein Graphics-Objekt, mit dem Sie auf ein unsichtbares Bild zeichnen, das Sie später verwenden können. Darüber hinaus finden Sie Graphics-Objekte bei viel mehr als nur bei Formularen. Ziehen Sie einen Button auf ein Formular, gehen Sie dann in Ihren Code, geben Sie den Namen des Buttons und dann einen Punkt ein. Werfen Sie einen Blick in das IntelliSense-Fenster, das angezeigt wird – es enthält eine CreateGraphics()-Methode, die ein Graphics-Objekt liefert. Alles, was Sie darauf zeichnen, erscheint auf dem Button! Das Gleiche gilt für Labels, PictureBoxen, StatusStrips ... fast alle Toolbox-Steuerelemente bieten ein Graphics-Objekt.

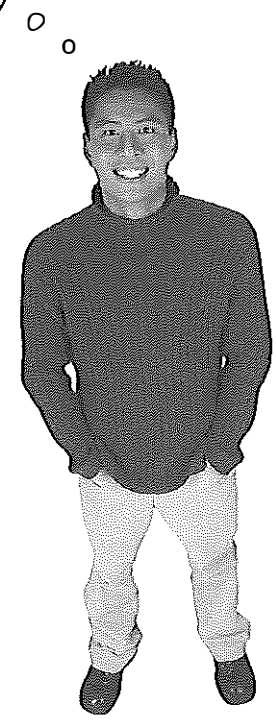
F: Moment. Ich dachte, `using` würde man nur mit Streams verwenden. Warum nutze ich `using` mit Graphics-Objekten?

A: Das `using`-Schlüsselwort ist bei der Arbeit mit Streams praktisch. Aber eigentlich können Sie es mit jeder Klasse verwenden, die die Schnittstelle `IDisposable` implementiert. Implementiert eine Klasse `IDisposable`, sollten Sie ihre `Dispose()`-Methode auf jeder Instanz aufrufen, die Sie selbst erstellt haben, wenn Sie sie nicht mehr benötigen. Bei Streams sichert `Dispose()`, dass alle geöffneten Dateien geschlossen werden. `Graphics`-, `Pen`- und `Brush`-Objekte sind ebenfalls entsorgbar. Erstellen Sie eins davon, beansprucht es eine kleine Menge Speicher und andere Ressourcen und gibt diese nicht immer sofort zurück. Wenn Sie nur einmal etwas zeichnen, wird Ihnen kaum ein Unterschied auffallen. Aber meist wird Ihr Graphics-Code immer wieder aufgerufen – wie bei einem `Paint`-Event-Handler, der bei einem sehr beschäftigten Formular in einer Sekunde mehrfach aufgerufen werden könnte. Deswegen sollten Sie auf all ihren Graphics-bezogenen Objekten immer `Dispose()` aufrufen. Dass Sie das tun, können Sie am leichtesten sicherstellen, indem Sie eine `using`-Zeile verwenden und das Entsorgen von .NET erledigen lassen. Für jedes mit `using` erstellte Objekt wird am Ende des auf `using` folgendes Blocks automatisch `Dispose()` aufgerufen. Das sichert, dass Ihr Programm mit der Zeit nicht immer mehr Speicher in Anspruch nimmt, wenn es eine lange Zeit läuft.

F: Sollte ich, wenn ich ein neues Steuerelement erstelle, ein UserControl verwenden, oder ist es besser, eine neue Klasse zu erstellen, die eins der Toolbox-Steuerelemente erweitert?

A: Das ist davon abhängig, was Ihr neues Steuerelement tun soll. Wenn Sie ein Steuerelement erstellen, das große Ähnlichkeit mit einem hat, das sich bereits in der Toolbox befindet, ist es wahrscheinlich am einfachsten, wenn sie jenes Steuerelement erweitern. Aber meist nutzen Programmierer Benutzersteuerelemente, wenn sie in C# neue Steuerelemente erstellen wollen. Ein praktischer Vorteil von Benutzersteuerelementen ist, dass Sie **Toolbox-Steuerelemente darauf ziehen können**. Sie funktionieren ähnlich wie eine `GroupBox` oder ein anderes Container-Steuerelement – Sie können einen Button oder eine Checkbox auf Ihr Benutzersteuerelement ziehen und mit diesem arbeiten wie mit den Steuerelementen auf einem Formular. Der Form-Designer der IDE wird zu einem mächtigen Werkzeug, das Sie beim Entwurf von Benutzersteuerelementen unterstützt.

Mir ist in meinem Zoomer-Steuerelement ein Flackern aufgefallen. Da Sie die ganze Zeit davon reden, dass wir die grafische Darstellung selbst in die Hand nehmen sollen, nehme ich mal an, dass wir auch dagegen etwas tun können! Aber warum passiert das?



Auch nach der Größenanpassung braucht das Formular noch Zeit, um ein Bild in ein Formular zu zeichnen.

Nehmen Sie an, Sie haben die Größe aller Bilder im Simulator angepasst. Auch dann braucht es immer noch Zeit, bis all diese Bienen und Blumen und der Stock gezeichnet sind. Und jetzt zeichnen wir direkt in das Graphics-Objekt auf dem Formular. Fängt Ihr Auge die Arbeitsspuren des Renderers ein, nehmen Sie das als ein leichtes Flackern wahr.

Das Problem ist, dass eine Menge Zeichenoperationen erfolgen und es deswegen wahrscheinlich ist, dass selbst nach einer Größenanpassung ein Flackern auftritt. Auf dieses Problem stoßen Sie auch bei vielen Amateur-Computerspielen ... das menschliche Auge nimmt das Ende eines Render-Zyklus auf und empfindet es als leichtes Flackern auf dem Bildschirm.



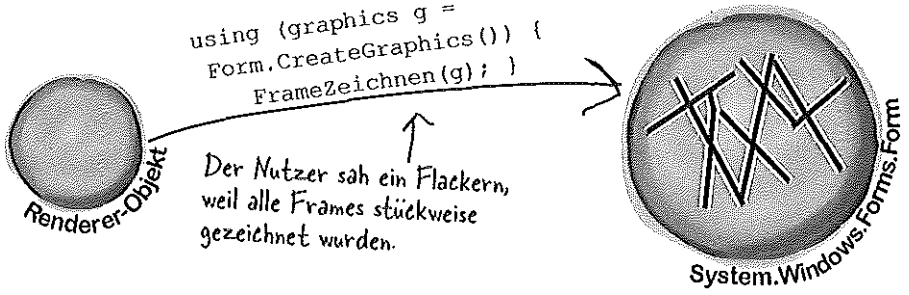
KOPF-NUSS

Wie könnte man dieses Flackern loswerden? Führt es zu einem Flackern, wenn auf ein Formular viele Dinge gezeichnet werden, wie könnten Sie das Flackern verhindern, wenn Sie viele Bilder zeichnen müssen?

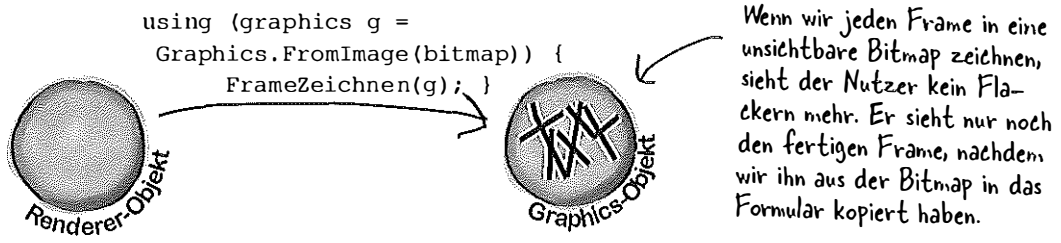
Double-Buffering macht Ihre Animation glatter

Kehren Sie in Ihren Bildvergrößerer zurück und spielen Sie mit den Trackbars herum. Sehen Sie, dass es stark flackert, wenn Sie die Schieber bewegen? Das liegt daran, dass der `Paint`-Event-Handler bei jeder winzigen Bewegung des Schiebers erst das weiße Rechteck und dann das Bild zeichnen muss. Diesen pro Sekunde mehrfach erfolgenden Wechsel vom weißen Rechteck zum Bild nehmen Ihre Augen als Flackern wahr. Das ist störend ... und lässt sich mit einer Technik vermeiden, die als **Double-Buffering** bezeichnet wird. Das bedeutet, dass jeder Frame oder jede Zelle der Animation erst in eine unsichtbare Bitmap (einen »Buffer«, Puffer) gezeichnet und diese erst angezeigt wird, wenn sie vollständig gezeichnet wurde. So würde das mit einer `Bitmap` funktionieren:

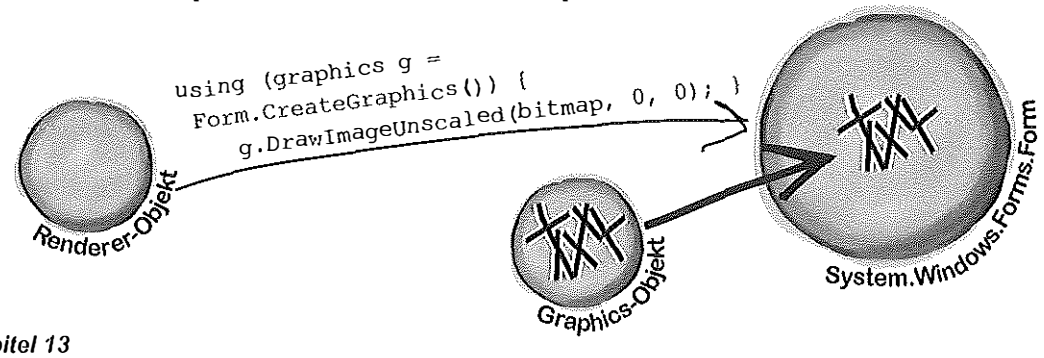
1. Hier ist ein typisches Programm, das über das Graphics-Objekt etwas auf ein Formular zeichnet.



2. Um ein Double-Buffering einzurichten, fügen wir dem Programm ein `Bitmap`-Objekt hinzu, das als Puffer dient. Jedes Mal, wenn Ihr Formular oder Steuerelement neu gezeichnet werden muss, zeichnen Sie nicht direkt auf das Formular, sondern in den Puffer.



3. Nachdem der Frame auf das unsichtbare `Bitmap`-Objekt gezeichnet wurde, können wir `DrawImageUnscaled()` verwenden, um das Objekt in das `Graphics`-Objekt des Formulars zu kopieren. Es wird alles auf einmal kopiert, und das verhindert das Flackern.



Double-Buffering ist in Formulare und Steuerelemente eingebaut

Mit `Bitmaps` können Sie Double-Buffering selbst aufbauen, aber C# und .NET erleichtern Ihnen das Leben mit einer eingebauten Unterstützung für Double-Buffering. **Sie müssen einfach die Eigenschaft DoubleBuffered auf true setzen.** Probieren Sie das bei Ihrem Zoomer-UserControl – gehen Sie ins Eigenschaften-Fenster, setzen Sie DoubleBuffered auf true, und schon endet das Flackern bei Ihrem Steuerelement! Kehren Sie jetzt **zum BienenControl zurück** und machen Sie dort das Gleiche. Das repariert zwar nicht alle grafischen Probleme – das machen wir gleich –, aber es *macht* einen Unterschied.

Nutzen Sie das Paint-Event für Ihre Grafiken, können Sie Double-Bufferung mit einer einfachen Eigenschaftsänderung anschalten.

Jetzt sind Sie bereit, die grafischen Probleme im Simulator zu beheben!

Überarbeiten Sie den Bienenstock-Simulator

In der nächsten Übung werden Sie den Bienenstock-Simulator vollständig überarbeiten. Wahrschcinlich sollten Sie ein völlig neues Projekt erstellen und »Hinzufügen→ Vorhandenes Element...« einsetzen, um die aktuellen Dateien darin zu übernehmen, damit Sie eine Kopie des aktuellen Simulators behalten. (Vergessen Sie nicht, den Namensraum an den des neuen Projekts anzupassen.)

Sie werden Folgendes tun:

1. **Beginnen Sie damit, dass Sie das BienenControl entfernen.**
   In Stock und Feld wird es überhaupt keine Steuerelemente mehr geben. Keine BienenControls, keine PictureBoxen, nichts. Die Bienen, Blumen und der Stock werden über GDI+-Grafikoperationen gezeichnet. Klicken Sie im Projektmappen-Explorer mit der rechten Maustaste auf BienenControl.cs (und AlteBiene.cs) und wählen Sie Löschen – die Dateien werden aus dem Projekt entfernt und dauerhaft gelöscht.

2. **Sie brauchen einen Timer für das Schwirren der Bienenflügel.**
   Die Bienenflügel flattern viel langsamer als die Frame-Rate des Simulators. Sie brauchen also einen zweiten, langsameren Timer. Das wird Sie nicht überraschen, da das BienenControl einen eigenen Timer hatte, der genau das machte.

3. **Die große Aufgabe: den Renderer umarbeiten.**
   Sie müssen den aktuellen Renderer wegschmeißen, weil er alles über Steuerelemente macht. Auch die Dictionaries zum Nachschlagen brauchen wir nicht mehr, weil es keine PictureBoxen oder BienenControls mehr gibt, die gesucht werden müssen. Stattdessen erhält er zwei wichtige Methoden: StockZeichnen(g), die ein Stock-Formular auf ein Graphics-Objekt zeichnet, und FeldZeichnen(g), die ein Feld-Formular zeichnet.

4. **Zum Abschluss müssen Sie den neuen Renderer einbinden.**
   Die Formulare für Stock und Feld brauchen Paint-Event-Handler. Beide rufen die FeldZeichnen(g)- oder StockZeichnen(g)-Methoden des Renderers auf. Die beiden Timer – der eine sagt dem Simulator, dass er den nächsten Frame zeichnen muss, und der andere lässt die Flügel der Bienen flattern – rufen die Invalidate()-Methoden der Formulare auf, damit sich diese neu zeichnen. Wenn sie das tun, zeichnen ihre Paint-Event-Handler den Frame.

Fangen wir an!



Es ist an der Zeit, dass wir uns um die grafischen Störungen am Bienenstock-Simulator kümmern. Nutzen Sie Graphics und Double-Buffering, um das Aussehen des Simulators zu verbessern.

1 **Ändern Sie die FrameAusführen()-Methode des Hauptformulars.**
Sie müssen den Aufruf von `Renderer.Rendern()` entfernen und zwei `Invalidate()`-Aufrufe einfügen.

```
public void FrameAusführen(object sender, EventArgs e) {
    gelaufeneFrames++;
    welt.Laufen(zufall);
    ende = DateTime.Now;
    TimeSpan frameDauer = ende - start;
    start = ende;
    StatistikAktualisieren(frameDauer);
    stockForm.Invalidate();
    feldForm.Invalidate();
}
```

Den Aufruf renderer.Rendern() müssen Sie entfernen, weil es diese Methode nicht mehr gibt.

Solange Sie die Welt aktuell halten und beide Formulare eine Referenz auf das Renderer-Objekt haben, müssen Sie nur ihre `Invalidate()`-Methoden aufrufen, um sie zu animieren. Ihre `Paint`-Event-Handler kümmern sich um den Rest.

2 **Geben Sie dem Hauptformular einen zweiten Timer für das Schwirren der Flügel.**
Ziehen Sie einen neuen Timer auf das Hauptformular, setzen Sie `Interval` auf 150ms und `Enabled` auf true. Klicken Sie dann doppelt darauf, um einen Handler für sein Event zu generieren:

```
private void timer2_Tick(object sender, EventArgs e) {
    renderer.BienenAnimieren();
}
```

Fügen Sie dem Renderer dann diese `BienenAnimieren()`-Methode hinzu, um die Flügel der Bienen zum Schwirren zu bringen:

```
private int zelle = 0;
private int frame = 0;
public void BienenAnimieren() {
    frame++;
    if (frame >= 6)
    frame = 0;
    switch (frame) {
        case 0: zelle = 0; break;
        case 1: zelle = 1; break;
        case 2: zelle = 2; break;
        case 3: zelle = 3; break;
        case 4: zelle = 2; break;
        case 5: zelle = 1; break;
        default: zelle = 0; break;
    }
    stockForm.Invalidate();
    feldForm.Invalidate();
}
```

Der Grundgedanke ist hier, dass ein Feld namens zelle gesetzt wird, das Sie verwenden können, wenn Sie im Renderer die Bienen zeichnen. Achten Sie darauf, dass Sie im Stock-Formular immer BienenBilderGroß[zelle] und im Feld-Formular immer BienenBilderKlein[zelle] aufrufen. Der Timer ruft regelmäßig die Methode BienenAnimieren() auf, die das Feld zelle ändert und damit bewirkt, dass die Bienenflügel schwirren.

3 **Richten Sie die Formulare für Stock und Feld für eine Animation mit Double-Buffering ein.** Entfernen Sie aus dem Stock-Formular-Konstruktor den Code, der das Hintergrundbild setzt. Entfernen Sie aus beiden Formularen die Steuerelemente, **setzen Sie ihre `DoubleBuffered`-Eigenschaften auf `true`** und fügen Sie beiden einen Paint-Event-Handler hinzu. Hier ist der Handler für den Stock – der für das Feld-Formular ist identisch, ruft aber `renderer.FeldZeichnen()` statt `renderer.StockZeichnen()` auf.

```
private void StockFormular_Paint(object sender, PaintEventArgs e) {
    renderer.StockZeichnen(e.Graphics);
}
```

Denken Sie an das Double-Buffering, da die Formulare sonst flackern.

4 **Die Formulare für Stock und Feld brauchen beide eine öffentliche `renderer`-Eigenschaft.** Fügen Sie den Stock- und Feld-Formularen ein öffentliches Feld hinzu:

```
public Renderer renderer;
```

`Renderer` werden an zwei Orten erstellt: im Öffnen-Button (unter einem Aufruf von `renderer.Rücksetzen()`) und in der Methode SimulatorZurücksetzen(). Entfernen Sie alle Aufrufe von `renderer.Rücksetzen()` und ersetzen Sie die `new Renderer`-Anweisungen mit dem Aufruf folgender `RendererErzeugen()`-Methode:

```
private void RendererErzeugen() {
    renderer = new Renderer(welt, stockForm, feldForm);
    stockForm.renderer = renderer;
    feldForm.renderer = renderer;
}
```

Im Renderer brauchen Sie keine Rücksetzen()-Methode mehr. Sie entfernte nur die Steuerelemente aus den Formularen, die es jetzt gar nicht mehr gibt.

6 **Tauschen Sie im Renderer den steuerelementbasierten Code gegen Graphics-Code aus.** Das müssen Sie im Renderer ändern:

- ★ Entfernen Sie die Dictionaries. Es gibt ja keine Steuerelemente mehr. Und da Sie gerade dabei sind, auch `BienenControl`, `Rendern()`, `bienenZeichnen()` und `blumenZeichnen()`.
- ★ Fügen Sie `Bitmap`-Felder namens `ImStock`, `AufFeld` und `Blume` zur Speicherung der Bilder hinzu. Erstellen Sie dann zwei `Bitmap[]`-Arrays namens `BienenBilderGroß` und `BienenBilderKlein`. Beide enthalten vier Bienenbilder – das eine große (40 x 40), das andere kleine (20 x 20). Erstellen Sie die Methode `BilderInitialisieren()`, die die Größe der Ressourcen ändert und sie in den Feldern speichert. Rufen Sie diese Methode aus dem Konstruktor auf.
- ★ Fügen Sie die Methode `StockZeichnen()` hinzu, die als Parameter ein `Graphics`-Objekt erwartet und darauf den Stock zeichnet. Zeichnen Sie erst ein himmelblaues Rechteck, dann mit `DrawImageUnscaled()` nacheinander die Innenansicht des Stocks und die Bienen, die im Stock sind.
- ★ Fügen Sie dann die Methode `FeldZeichnen()` ein. Sie sollte in der oberen Hälfte des Formulars ein himmelblaues Rechteck zeichnen, in der unteren ein grünes. Dabei können Ihnen zwei Form-Eigenschaften helfen: ClientSize und ClientRectangle sagen Ihnen, wie groß der Inhaltsbereich ist, die halbe Höhe ermitteln Sie also mit `ClientSize.Height / 2`. Zeichnen Sie dann mit `FillEllipse()` eine gelbe Sonne in den Himmel, mit `DrawLine()` einen Ast, an dem der Stock hängt, und mit `DrawImageUnscaled()` das Bild mit der Außenansicht des Stocks. Zeichnen Sie dann die Bienen (mit dem kleinen Bild) – zeichnen Sie diese zum Schluss, damit sie vor den Blumen sind.
- ★ Denken Sie beim Zeichnen der Bienen daran, dass `BienenAnimieren()` das Feld `zelle` setzt.



LÖSUNGEN ZU DEN ÜBUNGEN

Es ist an der Zeit, dass wir uns um die grafischen Störungen am Bienenstock-Simulator kümmern. Nutzen Sie Graphics und Double-Buffering, um das Aussehen des Simulators zu verbessern.

Hier ist die vollständige Klasse Renderer einschließlich der Methode BienenAnimieren(), die wir Ihnen gegeben haben. Achten Sie darauf, dass Sie alle Änderungen an den drei Formularen vornehmen – insbesondere dass Sie die Paint-Event-Handler in Stock- und Feld-Formular einfügen. Diese Event-Handler rufen die StockZeichnen()- und FeldZeichnen()-Methoden des Renderers auf, die die gesamte Animation durchführen.

```
using System.Drawing;

public class Renderer {
    private Welt welt;
    private StockFormular stockForm;
    private FeldFormular feldForm;

    public Renderer(Welt dieWelt, StockFormular stockForm, FeldFormular feldForm) {
        this.welt = dieWelt;
        this.stockForm = stockForm;
        this.feldForm = feldForm;
        BilderInitialisieren();
    }

    public static Bitmap BildSkalieren(Image bild, int Width, int Height) {
        Bitmap bitmap = new Bitmap(Width, Height);
        using (Graphics graphics = Graphics.FromImage(bitmap)) {
            graphics.DrawImage(bild, 0, 0, Width, Height);
        }
        return bitmap;
    }

    Bitmap ImStock;
    Bitmap AufFeld;
    Bitmap Blume;
    Bitmap[] BienenBilderKlein;
    Bitmap[] BienenBilderGroß;
    private void BilderInitialisieren() {
        AufFeld = BildSkalieren(Properties.Resources.Hive_outside_, 85, 100);
        Blume = BildSkalieren(Properties.Resources.Flower, 75, 75);
        ImStock = BildSkalieren(Properties.Resources.Hive_inside_,
            stockForm.ClientRectangle.Width, stockForm.ClientRectangle.Height);
        BienenBilderGroß = new Bitmap[4];
        BienenBilderGroß[0] = BildSkalieren(Properties.Resources.Bee_animation_1, 40, 40);
        BienenBilderGroß[1] = BildSkalieren(Properties.Resources.Bee_animation_2, 40, 40);
        BienenBilderGroß[2] = BildSkalieren(Properties.Resources.Bee_animation_3, 40, 40);
        BienenBilderGroß[3] = BildSkalieren(Properties.Resources.Bee_animation_4, 40, 40);
        BienenBilderKlein = new Bitmap[4];
        BienenBilderKlein[0] = BildSkalieren(Properties.Resources.Bee_animation_1, 20, 20);
        BienenBilderKlein[1] = BildSkalieren(Properties.Resources.Bee_animation_2, 20, 20);
        BienenBilderKlein[2] = BildSkalieren(Properties.Resources.Bee_animation_3, 20, 20);
        BienenBilderKlein[3] = BildSkalieren(Properties.Resources.Bee_animation_4, 20, 20);
    }
```

Die Methode BilderInitialisieren() passt die Größe aller Bildressourcen an und speichert sie in Bitmap-Feldern im Renderer-Objekt. Dann können die Methoden StockZeichnen() und FeldZeichnen() die Bilder unskaliert zeichnen, indem sie die DrawImageUnscaled()-Methode des Graphics-Objekts des Formulars verwenden.

```
    public void StockZeichnen(Graphics g) {
        g.FillRectangle(Brushes.SkyBlue, stockForm.ClientRectangle);
        g.DrawImageUnscaled(ImStock, 0, 0);
        foreach (Biene biene in welt.Bienen) {
            if (biene.InStock)
                g.DrawImageUnscaled(BienenBilderGroß[zelle],
                                    biene.Ort.X, biene.Ort.Y);
        }
    }

    public void FeldZeichnen(Graphics g) {
        using (Pen braunerStift = new Pen(Color.Brown, 6.0F)) {
            g.FillRectangle(Brushes.SkyBlue, 0, 0,
                    feldForm.ClientSize.Width, feldForm.ClientSize.Height / 2);
            g.FillEllipse(Brushes.Yellow, new RectangleF(50, 15, 70, 70));
            g.FillRectangle(Brushes.Green, 0, feldForm.ClientSize.Height / 2,
                    feldForm.ClientSize.Width, feldForm.ClientSize.Height / 2);
            g.DrawLine(braunerStift, new Point(643, 0), new Point(643, 30));
            g.DrawImageUnscaled(AufFeld, 600, 20);
            foreach (Blume blume in welt.Blumen) {
                g.DrawImageUnscaled(Blume, blume.Ort.X, blume.Ort.Y);
            }
            foreach (Biene biene in welt.Bienen) {
                if (!biene.InStock)
                    g.DrawImageUnscaled(BienenBilderKlein[zelle],
                                        biene.Ort.X, biene.Ort.Y);
            }
        }
    }

    private int zelle = 0;
    private int frame = 0;
    public void BienenAnimieren()
    {
        frame++;
        if (frame >= 6)
            frame = 0;
        switch (frame) {
            case 0: zelle = 0; break;
            case 1: zelle = 1; break;
            case 2: zelle = 2; break;
            case 3: zelle = 3; break;
            case 4: zelle = 2; break;
            case 5: zelle = 1; break;
            default: zelle = 0; break;
        }
        stockForm.Invalidate();
        feldForm.Invalidate();
    }
}
```

Die ClientSize-Eigenschaft des Formulars ist ein Rectangle, das den Inhaltsbereich des Formulars angibt.

Die Methode FeldZeichnen() sucht die Bienen und Blumen, die sich in der Welt befinden, und verwendet ihre Orte, um das Feld zu zeichnen. Erst zeichnet sie den Himmel und den Boden, dann die Sonne und den Bienenstock. Danach zeichnet sie die Blumen und die Bienen. Es ist wichtig, dass alles in der richtigen Reihenfolge gezeichnet wird – würden wir die Bienen vor den Blumen zeichnen, sähe es aus, als würden die Bienen hinter den Blumen fliegen.

Hier ist die BienenAnimieren()-Methode, die wir Ihnen in der Übung gezeigt haben. Sie durchläuft die Animationen über das Feld frame – erst zeigt sie zelle 0, dann zelle 1, dann 2, dann 3, dann wieder 2 und dann wieder 1. So sieht das Schwirren der Bienenflügel glatt aus.

Mit einem Graphics-Objekt und einem Event-Handler drucken

Die `Graphics`-Methoden, mit denen Sie auf Ihre Formulare gezeichnet haben, **werden auch für das Drucken verwendet**. Mit den .NET-Objekten fürs Drucken aus `System.Drawing.Printing` können Sie Ihren Anwendungen leicht eine Druckfunktion und eine Druckvorschau spendieren. Dazu müssen Sie nur **ein `PrintDocument`-Objekt erstellen**. Es hat ein Event namens `PrintPage`, das Sie genau so verwenden können wie das `Tick`-Event eines Timers. Rufen Sie dann die `Print()`-Methode von `PrintDocument` auf, wird das Dokument ausgegeben. Und denken Sie daran, dass die IDE es Ihnen besonders leicht macht, den Event-Handler hinzuzufügen. Das geht so:



1. **Ziehen Sie auf das Formular einer neuen Windows-Anwendung** einen Button. Fügen Sie dem Formularcode **`using System.Drawing.Printing;`** hinzu. Doppelklicken Sie auf den Button, um den Event-Handler zu generieren. Schauen Sie, was passiert, wenn Sie **`+=`** eingeben:

```
private void button1_Click(object sender, EventArgs e) {
    PrintDocument dokument = new PrintDocument();
    dokument.PrintPage +=
```

`new PrintPageEventHandler(dokument_PrintPage);` (Zum Einfügen TAB-Taste drücken)

2. Drücken Sie Tab, füllt die IDE den Rest der Zeile automatisch ein. Genau so haben Sie in Kapitel 11 die Event-Handler hinzugefügt:

```
private void button1_Click(object sender, EventArgs e) {
    PrintDocument dokument = new PrintDocument();
    dokument.PrintPage += new PrintPageEventHandler(dokument_PrintPage);
```

Zum Generieren des Handlers "dokument_PrintPage" in dieser Klasse die TAB-Taste drücken

3. Drücken Sie erneut Tab, fügt die IDE dem Formular eine Event-Handler-Methode hinzu.

```
void dokument_PrintPage(object sender, PrintPageEventArgs e) {
    throw new NotImplementedException();
}
```

Jetzt können Sie hier BELIEBIGEN Graphics-Code angeben – ersetzen Sie einfach die throw-Zeile und nutzen Sie zum Zeichnen e.Graphics.

Der `PrintPageEventArgs`-Parameter `e` hat die Eigenschaft Graphics. Ersetzen Sie die `throw`-Anweisung durch Code, der die Zeichenmethoden des `e.Graphics`-Objekts aufruft.

4. Schließen Sie jetzt den `button1_Click`-Event-Handler ab, indem Sie **`dokument.Print()`** aufrufen. Wird diese Methode aufgerufen, erzeugt das `PrintDocument`-Objekt ein `Graphics`-Objekt und setzt dann das `PrintPage`-Event mit dem `Graphics`-Objekt als Parameter ab. Alles, was dieser Event-Handler auf das `Graphics`-Objekt zeichnet, wird an den Drucker gesendet.

```
private void button1_Click(object sender, EventArgs e) {
    PrintDocument dokument = new PrintDocument();
    dokument.PrintPage += new PrintPageEventHandler(dokument_PrintPage);
    dokument.Print();
}
```

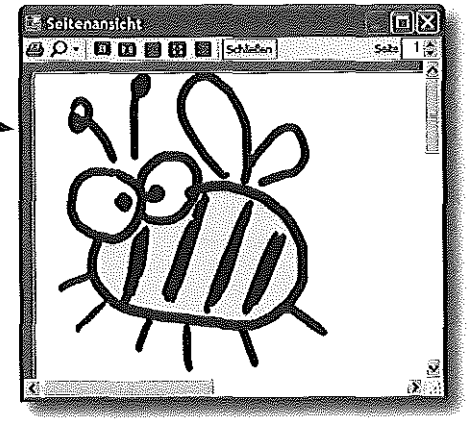
PrintDocument nutzt den Drucken-Dialog und die Druckvorschau

Das Hinzufügen einer Druckvorschau oder eines Drucken-Dialogs hat große Ähnlichkeit mit dem Hinzufügen eines Speichern- oder Öffnen-Dialogs. Sie müssen nur ein `PrintDialog`- oder ein `PrintPreviewDialog`-Objekt erstellen, seine `Document`-Eigenschaft auf Ihr `Document`-Objekt setzen und dann die `Show()`-Methode des Dialogs aufrufen. Der Dialog kümmert sich darum, dass das Dokument an den Drucker geschickt wird – die `Print()`-Methode selbst muss nicht aufgerufen werden. Fügen wir das dem in Schritt 1 erzeugten Button hinzu:

Haben Sie ein PrintDocument und einen Event-Handler zum Drucken der Seite, können Sie leicht eine Druckvorschau einblenden, indem Sie ein PrintPreviewDialog-Objekt erstellen.

```
private void button1_Click(object sender, EventArgs e) {
    PrintDocument dokument = new PrintDocument();
    dokument.PrintPage += new PrintPageEventHandler(dokument_PrintPage);
    PrintPreviewDialog vorschau = new PrintPreviewDialog();
    vorschau.Document = dokument;
    vorschau.ShowDialog(this);
}

void dokument_PrintPage(object sender,
                        PrintPageEventArgs e) {
    BieneZeichnen(e.Graphics, new Rectangle(0, 0, 300, 300));
}
```



Mehrseitige Dokumente drucken Sie mit e.HasMorePages

Müssen Sie mehrere Seiten ausgeben, müssen Sie nur Ihren `PrintPage`-Event-Handler `e.HasMorePages` auf `true` setzen lassen. Das sagt dem `Document`, dass noch eine weitere Seite gedruckt werden muss. Es ruft den Event-Handler immer wieder auf, einmal für jede Seite, solange der Event-Handler `e.HasMorePages` auf `true` setzt. Ändern Sie den Event-Handler für Ihr Dokument so, dass zwei Seiten gedruckt werden.

```
bool ersteSeite = true;
void dokument_PrintPage(object sender, PrintPageEventArgs e) {
    BieneZeichnen(e.Graphics, new Rectangle(0, 0, 300, 300));
    using (Font font = new Font("Arial", 36, FontStyle.Bold)) {
        if (ersteSeite) {
            e.Graphics.DrawString("Erste Seite", font, Brushes.Black, 0, 0);
            e.HasMorePages = true;
            ersteSeite = false;
        } else {
            e.Graphics.DrawString("Zweite Seite", font, Brushes.Black, 0, 0);
            ersteSeite = true;
        }
    }
}
```

Setzen Sie e.HasMorePages auf true, ruft das Document den Event-Handler erneut auf, damit die nächste Seite ausgegeben wird.

Führen Sie das Programm jetzt erneut aus und schauen Sie, ob die Druckvorschau zwei Seiten anzeigt.



Schreiben Sie den Code für den Drucken-Button im Simulator so, dass er eine Druckvorschau einblendet, die Bienenstatistiken und Bilder von Stock und Feld enthält.

1 Lassen Sie den Button eine Druckvorschau einblenden.

Fügen Sie einen Event-Handler für das Click-Event des Buttons ein, der den Simulator anhält, eine Druckvorschau öffnet und den Simulator wieder fortsetzt, wenn er fertig ist. (Ist der Simulator angehalten, wenn auf den Button geklickt wird, sollte er auch nach Anzeige der Vorschau angehalten bleiben.)

2 Erzeugen Sie den PrintPage-Event-Handler für das Dokument.

Er sollte eine Seite erzeugen, die genau wie die auf der gegenüberliegenden Seite abgebildete aussieht. Wir geben Ihnen den Anfang:

```
private void dokument_PrintPage(object sender, PrintPageEventArgs e) {
    Graphics g = e.Graphics;
    Size stringGröße;
    using (Font arial24bold = new Font("Arial", 24, FontStyle.Bold)) {
        stringGröße = Size.Ceiling(
            g.MeasureString("Bienen-Simulator", arial24bold));
        g.FillEllipse(Brushes.Gray,
            new Rectangle(e.MarginBounds.X + 2, e.MarginBounds.Y + 2,
            stringGröße.Width + 30, stringGröße.Height + 30));
        g.FillEllipse(Brushes.Black,
            new Rectangle(e.MarginBounds.X, e.MarginBounds.Y,
            stringGröße.Width + 30, stringGröße.Height + 30));
        g.DrawString("Bienen-Simulator", arial24bold,
            Brushes.Gray, e.MarginBounds.X + 17, e.MarginBounds.Y + 17);
        g.DrawString("Bienen-Simulator", arial24bold,
            Brushes.White, e.MarginBounds.X + 15, e.MarginBounds.Y + 15);
    }
    int tabX = e.MarginBounds.X + (int)stringGröße.Width + 50;
    int tabBreite = e.MarginBounds.X + e.MarginBounds.Width - tabX - 20;
    int spalte1X = tabX + 2;
    int spalte2X = tabX + (tabBreite / 2) + 5;
    int tabY = e.MarginBounds.Y;
    // Ihre Aufgabe: Füllen Sie den Rest der Methode ein, um das zu drucken.
```

Das Oval mit dem Text darin haben wir erzeugt, indem wir die Methode MeasureString() verwendet haben, die ein Size-Objekt liefert, das die Größe des Strings liefert. Wir haben das Oval und den Text doppelt gezeichnet, um einen Schatteneffekt zu erzeugen.

Das brauchen Sie, um die Tabelle zu erstellen.

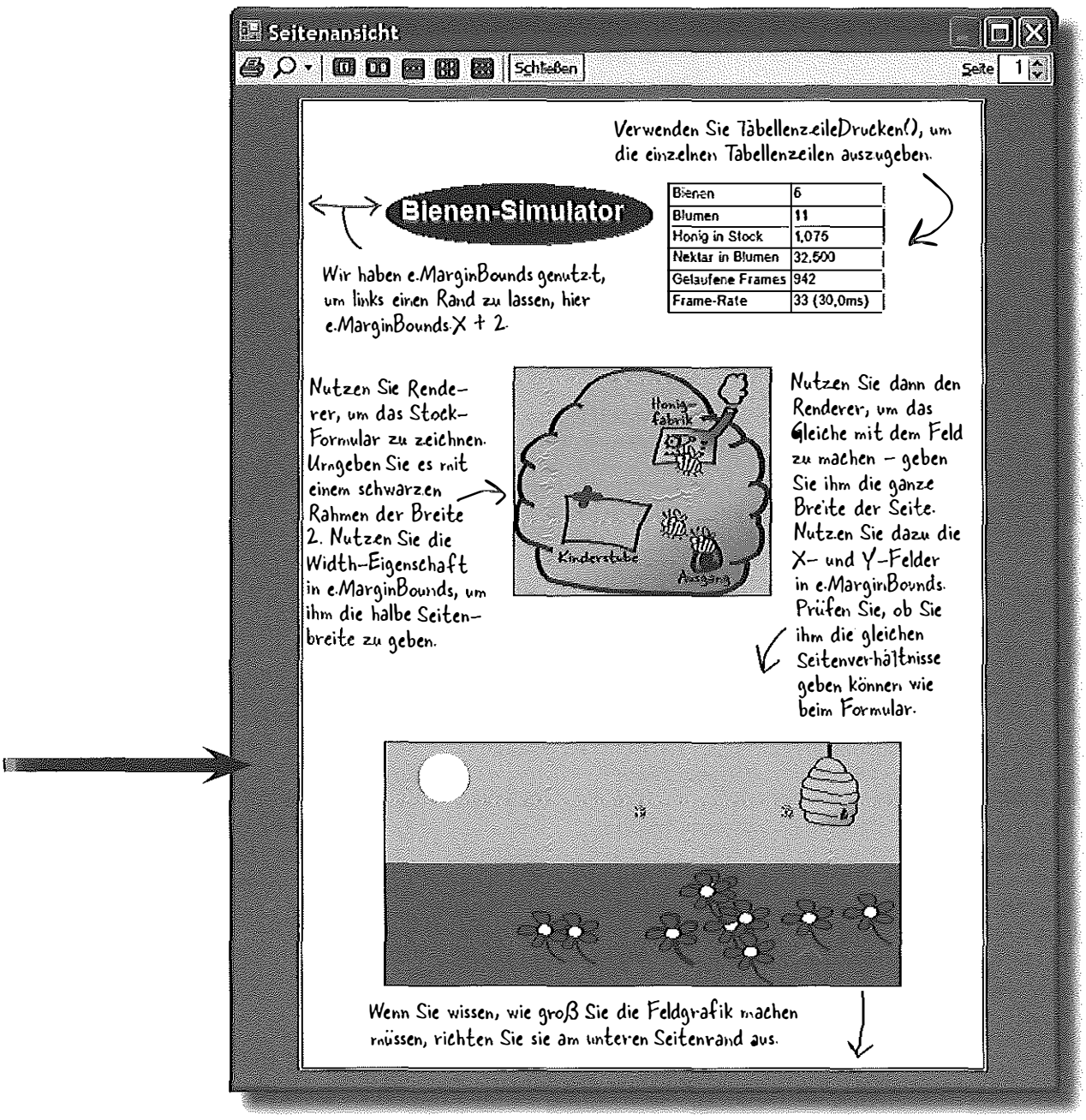
3 Diese TabellenzeileDrucken()-Methode wird sich als praktisch erweisen.

Diese Methode sollte Ihnen helfen, oben auf der Seite die Statistiktabelle zu erstellen.

```
private int TabellenzeileDrucken(Graphics g, int tabX,
                int tabBreite, int spalte1X, int spalte2X,
                int tabY, string ersteSpalte, string zweiteSpalte) {
  Font arial12 = new Font("Arial", 12);
  Size stringGröße = Size.Ceiling(g.MeasureString(ersteSpalte, arial12));
  tabY += 2;
  g.DrawString(ersteSpalte, arial12, Brushes.Black,
                spalte1X, tabY);
  g.DrawString(zweiteSpalte, arial12, Brushes.Black,
                spalte2X, tabY);
  tabY += (int)stringGröße.Height + 2;
  g.DrawLine(Pens.Black, tabX, tabY, tabX + tabBreite, tabY);
  arial12.Dispose();
  return tabY;
}
```

Bei jedem Aufruf von TabellenzeileDrucken() wird tabY um die Höhe der gedruckten Zeile erhöht und der neue Wert zurückgeliefert.

Lesen Sie die Erläuterungen aufmerksam. Das ist etwas komplexer – nehmen Sie sich Zeit!



Ein Tipp: Um die Höhe der Formulare zu berechnen, müssen Sie das Verhältnis von Höhe und Breite ermitteln und mit der gewünschten Breite multiplizieren. Den oberen Rand des Felds berechnen Sie, indem Sie seine Höhe vom unteren Seitenabstand abziehen: (e.MarginBounds.Y + e.MarginBounds.Height – feldHöhe).



LÖSUNGEN ZU DEN ÜBUNGEN

Schreiben Sie den Code für den Drucken-Button im Simulator so, dass er eine Druckvorschau einblendet, die Bienenstatistiken und Bilder von Stock und Feld enthält.

Hier ist der Event-Handler für das PrintPage-Event von Document. Er kommt ins Formular.

```
using System.Drawing.Printing;

private void dokument_PrintPage(object sender, PrintPageEventArgs e) {
   Graphics g = e.Graphics;

   Size stringGröße;
   using (Font arial24bold = new Font("Arial", 24, FontStyle.Bold)) {
       stringGröße = Size.Ceiling(
             g.MeasureString("Bienen-Simulator", arial24bold));
       g.FillEllipse(Brushes.Gray,
           new Rectangle(e.MarginBounds.X + 2, e.MarginBounds.Y + 2,
           stringGröße.Width + 30, stringGröße.Height + 30));
       g.FillEllipse(Brushes.Black,
           new Rectangle(e.MarginBounds.X, e.MarginBounds.Y,
           stringGröße.Width + 30, stringGröße.Height + 30));
       g.DrawString("Bienen-Simulator", arial24bold,
           Brushes.Gray, e.MarginBounds.X + 17, e.MarginBounds.Y + 17);
       g.DrawString("Bienen-Simulator", arial24bold,
           Brushes.White, e.MarginBounds.X + 15, e.MarginBounds.Y + 15);
   }

   int tabX = e.MarginBounds.X + (int)stringGröße.Width + 50;
   int tabBreite = e.MarginBounds.X + e.MarginBounds.Width - tabX - 20;
   int spalte1X = tabX + 2;
   int spalte2X = tabX + (tabBreite / 2) + 20;
   int tabY = e.MarginBounds.Y;

   tabY = TabellenzeileDrucken(g, tabX, tabBreite, spalte1X,
        spalte2X, tabY, "Bienen", Bienen.Text);
   tabY = TabellenzeileDrucken(g, tabX, tabBreite, spalte1X,
        spalte2X, tabY, "Blumen", Blumen.Text);
   tabY = TabellenzeileDrucken(g, tabX, tabBreite, spalte1X,
         spalte2X, tabY, "Honig in Stock", HonigInStock.Text);
   tabY = TabellenzeileDrucken(g, tabX, tabBreite, spalte1X,
        spalte2X, tabY, "Nektar in Blumen", NektarInBlumen.Text);
   tabY = TabellenzeileDrucken(g, tabX, tabBreite, spalte1X,
        spalte2X, tabY, "Gelaufene Frames", GelaufeneFrames.Text);
   tabY = TabellenzeileDrucken(g, tabX, tabBreite, spalte1X,
        spalte2X, tabY, "Frame-Rate", FrameRate.Text);

   g.DrawRectangle(Pens.Black, tabX, e.MarginBounds.Y,
        tabBreite, tabY - e.MarginBounds.Y);
   g.DrawLine(Pens.Black, spalte2X, e.MarginBounds.Y,
        spalte2X, tabY);
```

Diesen Teil haben wir Ihnen bereits gegeben. Er zeichnet das Oval und setzt die Variablen, die Sie verwenden werden, um die Tabelle mit den Bienenstatistiken zu schreiben.

Haben Sie herausgefunden, wie die Methode TabellenzeileDrucken() funktioniert? Sie müssen sie nur einmal pro Zeile aufrufen, damit sie den Text druckt, den sie in den beiden Spalten haben möchten. Der Trick ist, dass Sie den tabY-Wert für die nächste Zeile zurückliefert.

Vergessen Sie das Rechteck um die Tabelle und den Strich zwischen den Spalten nicht.

```
        using (Pen blackPen = new Pen(Brushes.Black, 2))
        using (Bitmap stockBitmap = new Bitmap(stockForm.ClientSize.Width,
                                               stockForm.ClientSize.Height))
        using (Bitmap feldBitmap = new Bitmap(feldForm.ClientSize.Width,
                                              feldForm.ClientSize.Height))
        {
            using (Graphics stockGraphics = Graphics.FromImage(stockBitmap))
            {
                renderer.StockZeichnen(stockGraphics);
            }
            int stockBreite = e.MarginBounds.Width / 2;
            float ratio = (float)stockBitmap.Height / (float)stockBitmap.Width;
            int stockHöhe = (int)(stockBreite * ratio);
            int stockX = e.MarginBounds.X + (e.MarginBounds.Width - stockBreite) / 2;
            int stockY = e.MarginBounds.Height / 3;
            g.DrawImage(stockBitmap, stockX, stockY, stockBreite, stockHöhe);
            g.DrawRectangle(blackPen, stockX, stockY, stockBreite, stockHöhe);

            using (Graphics feldGraphics = Graphics.FromImage(feldBitmap))
            {
                renderer.FeldZeichnen(feldGraphics);
            }
            int feldBreite = e.MarginBounds.Width;
            ratio = (float)feldBitmap.Height / (float)feldBitmap.Width;
            int feldHöhe = (int)(feldBreite * ratio);
            int feldX = e.MarginBounds.X;
            int feldY = e.MarginBounds.Y + e.MarginBounds.Height - feldHöhe;
            g.DrawImage(feldBitmap, feldX, feldY, feldBreite, feldHöhe);
            g.DrawRectangle(blackPen, feldX, feldY, feldBreite, feldHöhe);
        }
    }


    private void printToolStripButton_Click(object sender, EventArgs e) {
        bool timerAngehalten = false;
        if (timer1.Enabled) {
            timer1.Stop();
            timerAngehalten = true;
        }
        PrintPreviewDialog vorschau = new PrintPreviewDialog();
        PrintDocument dokument = new PrintDocument();
        vorschau.Document = dokument;
        dokument.PrintPage += new PrintPageEventHandler(dokument_PrintPage);
        vorschau.ShowDialog(this);
        if (timerAngehalten)
            timer1.Start();
    }
```

Der Stift für die Formularrahmen muss schwarz und 2 Pixel breit sein.

Die Bitmaps müssen die gleiche Größe wie der Inhaltsbereich des Formulars haben.

Wir verwenden einen großen using-Block, da all diese Objekte entsorgt werden müssen.

StockZeichnen() braucht ein Graphics zum Zeichnen. Dieser Code erstellt eine leere Bitmap und übergibt sie an StockZeichnen().

e.MarginBounds.Width ist die Breite des Druckbereichs der Seite. So breit soll das Bild des Felds sein.

Das berechnet die Bildhöhe auf Basis des Seitenverhältnisses des Formulars.

Der Code für den Drucken-Button hält den Simulator an (wenn er läuft), erzeugt das PrintDocument, verbindet es mit dem PrintPage-Event-Handler und zeigt den Dialog an.

Das setzt den Simulator wieder in Gang, wenn er angehalten werden musste.

Es gibt noch so viel zu tun ...

Sie haben einen hübschen kleinen Simulator erstellt. Warum sollten Sie ihn jetzt verkommen lassen? Es gibt noch so viele Dinge, die Sie eigenständig tun könnten. Hier sind ein paar Vorschläge – schauen Sie, ob Sie einige davon implementieren können.

Fügen Sie ein Schaltpult hinzu

Wandeln Sie die Konstanten in den Klassen Welt und Stock in Eigenschaften um. Fügen Sie dann ein neues Formular für ein Schaltpult mit Schiebern hinzu, über die sie gesteuert werden können.

Fügen Sie Feinde hinzu

Lassen Sie den Stock von Feinden angreifen. Je mehr Blumen es gibt, umso mehr Feinde zieht der Stock an. Fügen Sie Wachbienen und Wartungsbienen hinzu, die den Stock verteidigen bzw. reparieren. Diese Bienen brauchen extraviel Honig.

Gute Simulationen bieten viele Wechselwirkungen und lassen den Benutzer entscheiden, welche davon den Fortschritt des Stocks beeinflussen.

Machen Sie den Stock erweiterbar

Wenn der Stock genug Honig hat, wird er größer. In einen größeren Stock passen mehr Bienen, er braucht aber mehr Honig und zieht mehr Feinde an. Wenn Feinde zu viel Schaden verursachen, wird der Stock wieder kleiner.

Fügen Sie eine Königin hinzu, die Eier legt

Die Eier brauchen Brutpflegebienen, die sich um sie kümmern. Mehr Honig im Stock bewirkt, dass die Königin mehr Eier legt, für die dann wieder mehr Arbeiterinnen erforderlich sind, die mehr Honig brauchen.

Animieren Sie den Hintergrund des Feld-Formulars

Animieren Sie den Hintergrund des Feld-Formulars. Lassen Sie die Sonne über den Himmel wandern. Lassen Sie es bei Nacht dunkel werden und zeichnen Sie Sterne und einen Mond. Führen Sie Perspektive ein – machen Sie die Bienen kleiner, je weiter sie sich im Feld vom Stock entfernen.

Nutzen Sie Ihre Vorstellungskraft!

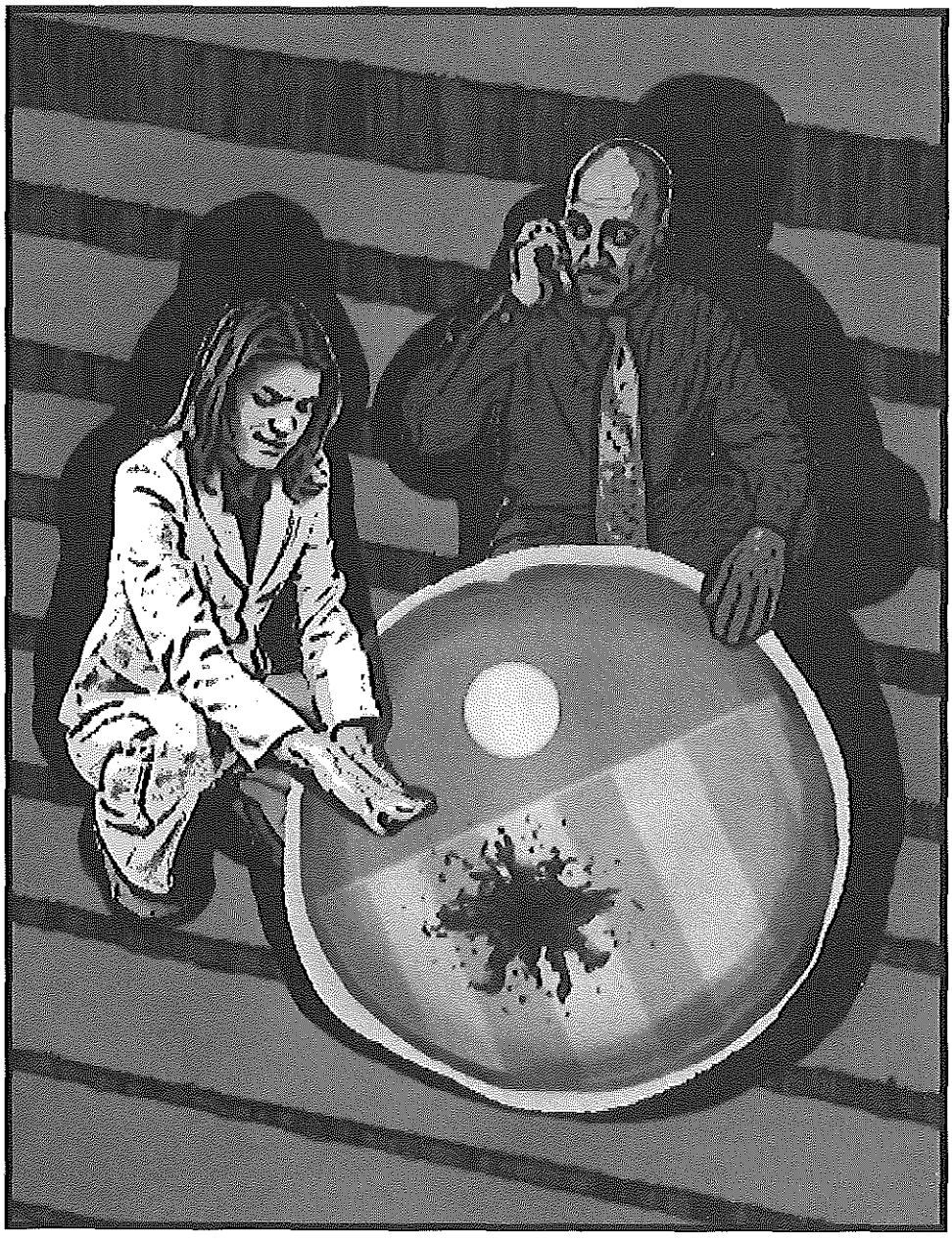
Fallen Ihnen weitere Möglichkeiten ein, die Simulation interessanter und interaktiver zu machen?

Haben Sie eine coole Version des Simulators entwickelt? Zeigen Sie sie vor – veröffentlichen Sie den Code Ihres Projekts im Head First C#-Forum unter www.headfirstlabs.com/books/hfcsharp/.

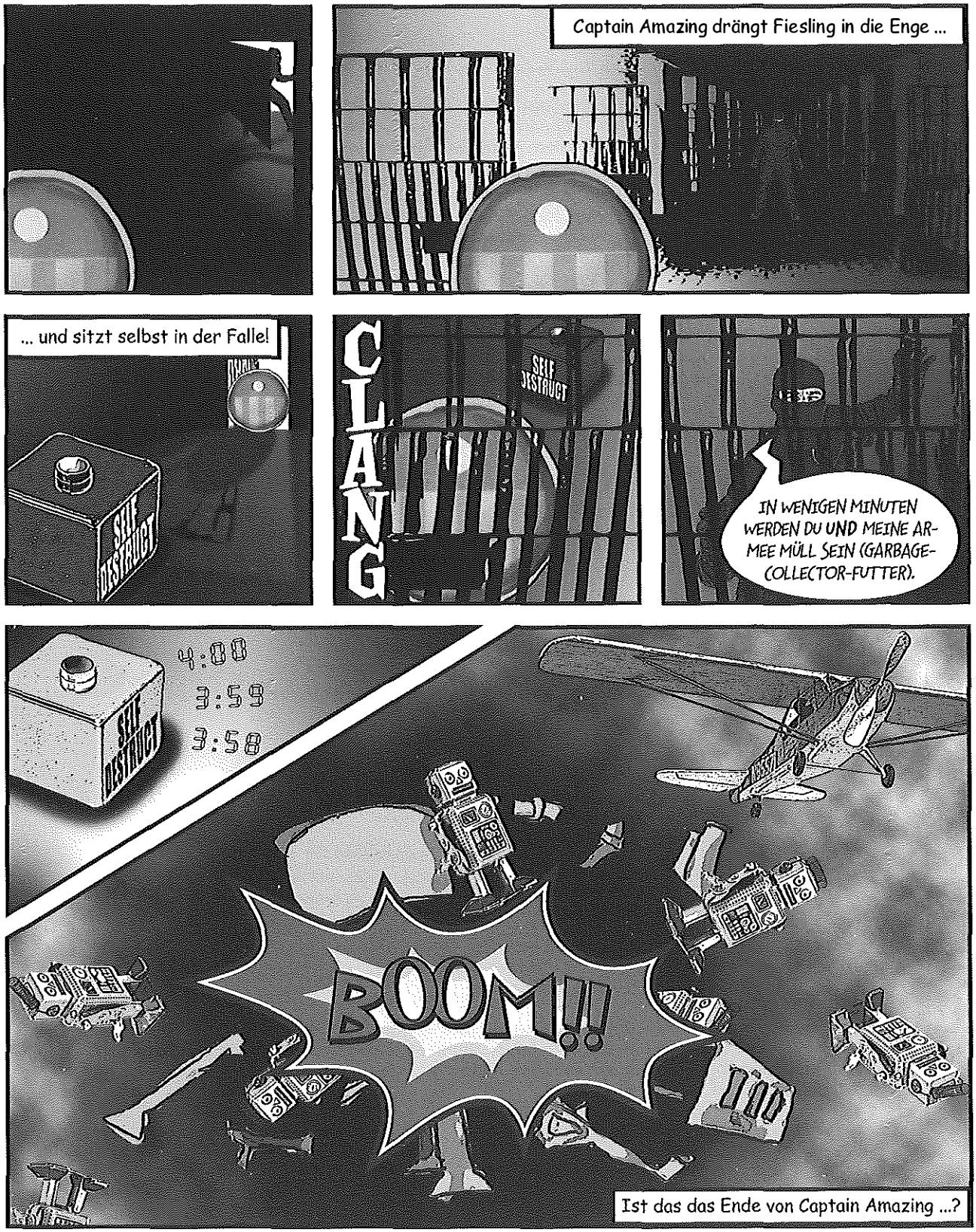
CAPTAIN AMAZING

DER TOD DES OBJEKTS

Von Kopf bis Fuß	
2,98 €	Kapitel 14




Captain Amazing, das Super-Objekt von Objekthausen, verfolgt seinen Erzfeind ...
DU SITZT IN DER FALLE, FIESLING.
ZU SPÄT! GENAU IN DIESEM MOMENT SAMMELT SICH IN DER FABRIK UNTER UNS MEINE KLON-ARMEE ...
... BEREIT, DIE STRASSEN VON OBJEKTHAUSEN IN ANGST UND SCHRECKEN ZU VERSETZEN!
Welcome to
Objectville
Home of Polymorphism
POW!!
ICH WERDE DIE KLON-REFERENZEN NACHEINANDER ÜBERWÄLTIGEN.


Captain Amazing drängt Fiesling in die Enge ...
... und sitzt selbst in der Falle!
SELF DESTRUCT
CLANG
IN WENIGEN MINUTEN WERDEN DU UND MEINE ARMEE MÜLL SEIN (GARBAGE-COLLECTOR-FUTTER).
4:00
3:59
3:58
BOOM!!!
Ist das das Ende von Captain Amazing ...?

Spitzen Sie Ihren Bleistift

Unten ist der Code, der das letzte Gefecht zwischen Captain Amazing und dem Fiesling (seine Klon-Armee nicht zu vergessen) nachstellt. Sie sollen aufzeichnen, was im Speicher passiert, wenn die Klasse LetztesGefecht instantiiert wird.

Sie können davon ausgehen, dass Klone über einen Objektinitialisierer gesetzt wurde.

```
public class LetztesGefecht {
  public KlonFabrik Fabrik = new KlonFabrik();
  public List<Klon> Klone = new List<Klon>();
  public FieslingFluchtFlugzeug fluchtFlugzeug;

  public LetztesGefecht() {
    public Bösewicht fiesling = new Bösewicht(this);
    using (SuperHeld captainAmazing = new SuperHeld()) {
      Fabrik.PersonenInFabrik.Add(captainAmazing);
      Fabrik.PersonenInFabrik.Add(fiesling); (1)
      captainAmazing.Denken("Ich werde die Klon-Referenzen
                             nacheinander überwältigen.");
      captainAmazing.KloneIdentifizieren(Klone);
      captainAmazing.KloneEntfernen(Klone);
      fiesling.Denken("In wenigen Minuten werden du und meine Armee Müll sein");
      fiesling.Denken(" (Garbage-Collector-Futter)");
      fluchtFlugzeug = new FieslingFluchtFlugzeug(fiesling); (2)
      fiesling.CaptainAmazingHereinlegen(Fabrik);
      MessageBox.Show("Der Fiesling entkam");
    }
  } (3)
}
[Serializable]
public class SuperHeld : IDisposable {
      private List<Klon> zuEntfernendeKlone = new List<Klon>();
      public void KloneIdentifizieren(List<Klon> klone) {
          foreach (Klon klon in klone)
              zuEntfernendeKlone.Add(klon);
      }
      public void KloneEntfernen(List<Klon> klone) {
          foreach (Klon klon in zuEntfernendeKlone)
              klone.Remove(klon);
        ...
      }
}
public class Bösewicht {
  private LetztesGefecht letztesGefecht;
  public Bösewicht(LetztesGefecht letztesGefecht) {
    this.letztesGefecht = letztesGefecht;
  }
  public void CaptainAmazingHereinlegen(KlonFabrik fabrik) {
    fabrik.SelbstZerstörung.Tick += new EventHandler(SelbstZerstörung_Tick);
    fabrik.SelbstZerstörung.Interval = 600;
    fabrik.SelbstZerstörung.Start();
  }
  private void SelbstZerstörung_Tick(object sender, EventArgs e) {
    letztesGefecht.fabrik = null;
  }
}
```

Die Beschreibung dessen, was im Fabrik-Objekt passiert, haben wir schon für Sie erledigt.

Zeichnen Sie auf, was hier passiert, wenn das FieslingFluchtFlugzeug-Objekt instantiiert wird.

Zeichnen Sie auf, wie der Heap genau eine Sekunde nach der Ausführung des LetztesGefecht-Konstruktors aussieht.

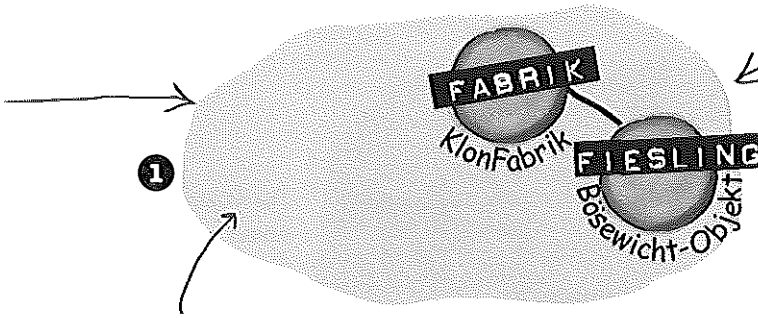
Hier ist noch mehr Code (einschließlich der Dispose()-Methode), den wir Ihnen nicht zeigen, weil Sie ihn nicht benötigen, um diese Fragen zu beantworten.

```
public class FieslingFluchtFlugzeug {
  public Bösewicht Pilotensitz;
  public FieslingFluchtFlugzeug(Bösewicht flüchtling) {
    Pilotensitz = flüchtling;
  }
}

public class KlonFabrik {
  public Timer SelbstZerstörung = new Timer();
  public List<object> PersonenInFabrik = new List<object>();
  ...
}
```

Es gibt eine Klon-Klasse, die wir Ihnen in diesem Code ebenfalls nicht zeigen. Sie benötigen sie zur Beantwortung der Fragen nicht.

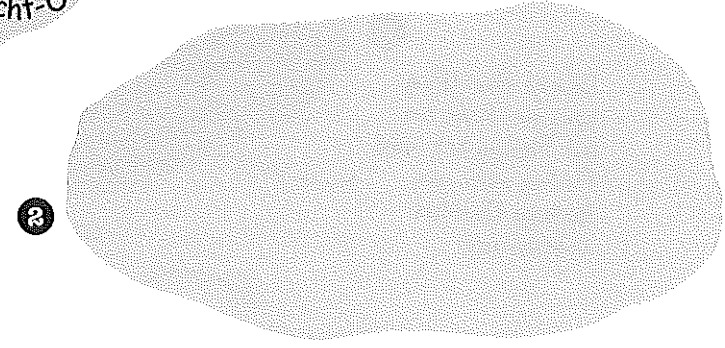
Achten Sie darauf, dass Sie Ihre Objekte beschriften, um die Referenzen anzugeben, die darauf zeigen.

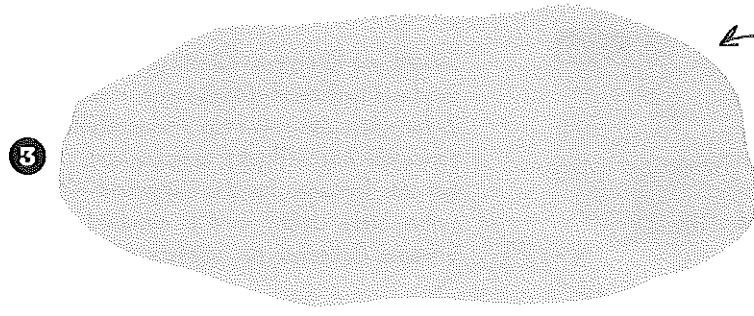


Die erste Frage haben wir für Sie beantwortet. Achten Sie darauf, dass Sie Verbindungslinien einzeichnen, die die Architektur zeigen – wir haben eine Linie von der Klon-Fabrik zum Bösewicht-Objekt gezogen, weil die Fabrik eine Referenz darauf hat (über das Feld PersonenInFabrik).

Wir haben Platz gelassen, da für diese Phase noch mehr gezeichnet werden muss.

Sie können die Klon- und List-Objekte außer acht lassen und sich in diesen Zeichnungen auf die wichtigsten Objekte konzentrieren.





Sie sollen einzeichnen, was zu diesen beiden Zeitpunkten im Speicher passiert.

An welchem Punkt des Codes stirbt Captain Amazing gemäß Ihren Diagrammen?

.....

.....

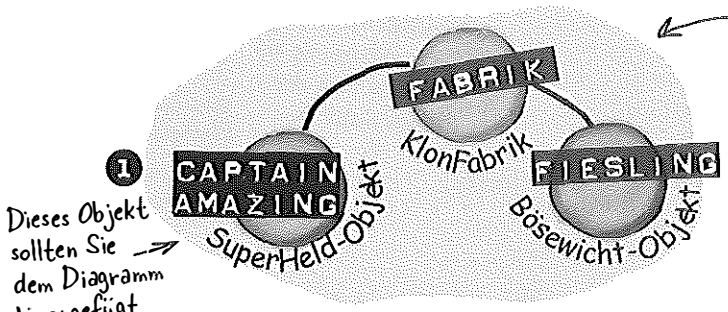
.....

.....

Halten Sie auch das in Ihrem Diagramm fest.

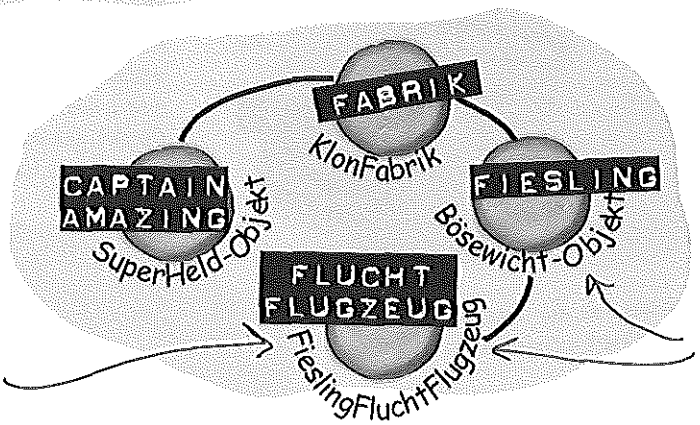
Spitzen Sie Ihren Bleistift
Lösung

Zeichnen Sie auf, was im Programm LetztesGefecht im Speicher passiert.



Die Referenz captainAmazing zeigt auf ein SuperHeld-Objekt, die Referenz fiesling auf ein Bösewicht-Objekt, und die PersonenInFabrik-Liste von KlonFabrik erhält eine Referenz auf beide.

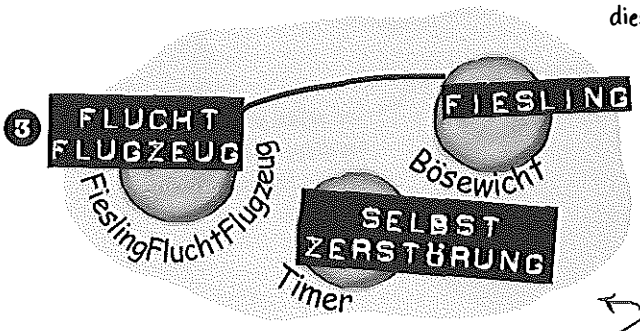
1 Dieses Objekt sollten Sie dem Diagramm hinzugefügt haben.



8 Solange fluchtFlugzeug eine Referenz auf Fiesling hat, wird Fiesling nicht vom Garbage Collector eingesammelt.

Die fluchtFlugzeug-Referenz zeigt jetzt auf die neue FieslingFluchtFlugzeug-Instanz und ihr Pilotensitz-Feld auf das Bösewicht-Objekt.

Wird selbstZerstörung abgesetzt, wird die Referenz fabrik auf null gesetzt und kann vom Garbage Collector eingesammelt werden. Deswegen ist sie in dieser Zeichnung verschwunden.



3

Sobald die Referenz fabrik verschwindet, nimmt sie das KlonFabrik-Objekt mit – und das sorgt dafür, dass das vom Feld PersonenInFabrik referenzierte List-Objekt verschwindet ... und das war das Einzige, das das SuperHeld-Objekt am Leben hielt. Jetzt wird es zerstört, wenn der Garbage Collector das nächste Mal läuft.

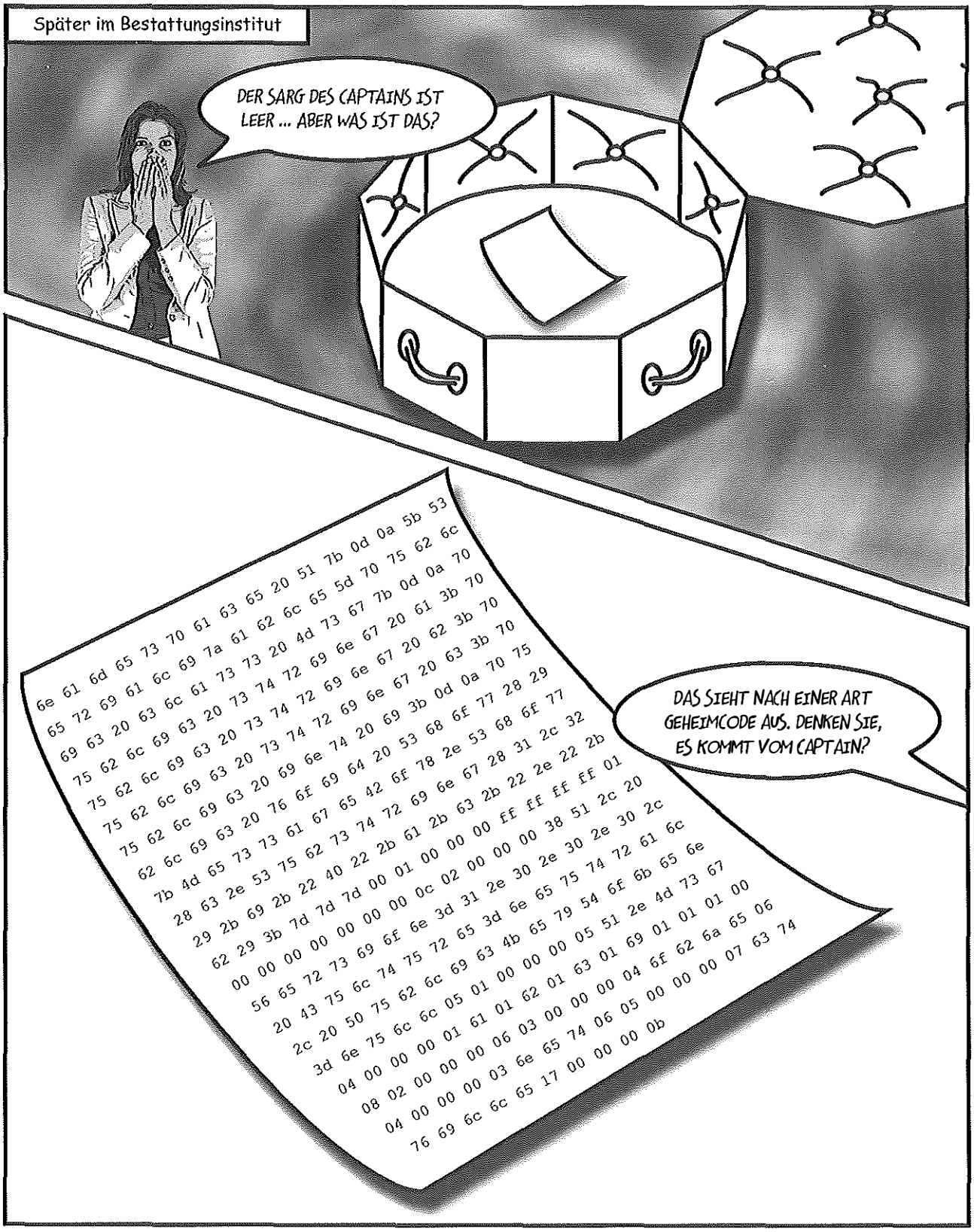
An welchem Punkt des Codes stirbt Captain Amazing gemäß Ihren Diagrammen?

```
void SelbstZerstörung_Tick(object sender, EventArgs e) {
    letztesGefecht.fabrik = null;
}
```

Eine Sekunde nach dem Konstruktor von LetztesGefecht ist der Held verschwunden.

Nachdem die SuperHeld-Instanz nicht mehr von der KlonFabrik referenziert wurde, wurde sie auch für die Garbage Collection vorgemerkt.

Sobald letztesGefecht.Fabrik auf null gesetzt ist, kann es vom Garbage Collector eingesammelt werden. Und die Referenz auf den Captain gleich mit!


Später im Bestattungsinstitut
DER SARG DES CAPTAINS IST LEER ... ABER WAS IST DAS?
6e 61 6d 65 73 70 61 63 65 20 51 7b 0d 0a 5b 53
65 72 69 61 6c 69 7a 61 62 6c 65 5d 70 75 62 6c
69 63 20 63 6c 61 73 73 20 4d 73 67 7b 0d 0a 70
75 62 6c 69 63 20 73 74 72 69 6e 67 20 61 3b 70
75 62 6c 69 63 20 73 74 72 69 6e 67 20 62 3b 70
75 62 6c 69 63 20 73 74 72 69 6e 67 20 63 3b 70
75 62 6c 69 63 20 69 6e 74 20 69 3b 0d 0a 70 75
75 62 6c 69 63 20 76 6f 69 64 20 53 68 6f 77 28
62 6c 69 63 20 73 74 61 74 69 63 20 76 6f 69 64
7b 4d 65 73 73 61 67 65 42 6f 78 2e 53 68 6f 77
28 63 2e 53 75 62 73 74 72 69 6e 67 28 31 2c 32
29 2b 69 2b 22 40 22 2b 61 2b 63 2b 22 2e 22 2b
62 29 3b 7d 7d 00 01 00 00 00 ff ff ff ff 01
00 00 00 00 00 00 0c 02 00 00 00 38 51 2c 20
56 65 72 73 69 6f 6e 3d 31 2e 30 2e 30 2c 20
20 43 75 6c 74 75 72 65 3d 6e 65 75 74 72 61 6c
2c 20 50 75 62 6c 69 63 4b 65 79 54 6f 6b 65 6e
3d 6e 75 6c 6c 05 01 00 00 00 05 51 2e 4d 73 67
04 00 00 00 01 61 01 62 01 63 01 69 01 01 01 00
08 02 00 00 00 06 03 00 00 00 04 6f 62 6a 65 06
04 00 00 00 03 6e 65 74 06 05 00 00 00 07 63 74
76 69 6c 65 17 00 00 00 0b
DAS SIEHT NACH EINER ART GEHEIMCODE AUS. DENKEN SIE, ES KOMMT VOM CAPTAIN?

Ihre letzte Chance, etwas zu TUN ... der Destruktor Ihres Objekts

Manchmal müssen Sie sicherstellen, dass etwas passiert, ***bevor*** Ihr Objekt vom Garbage Collector eingesammelt wird. Vielleicht möchten Sie Verbindungen oder Ressourcen freigeben ... oder der Welt eine kodierte Nachricht senden.

Eine besondere Methode Ihres Objekts, der **Destruktor**, ermöglicht Ihnen, Code zu schreiben, der immer ausgeführt wird, wenn Ihr Objekt zerstört wird. Betrachten Sie ihn als den persönlichen `finally`-Block des Objekts: Er wird unter allen Umständen ausgeführt.

Hier ist ein Beispiel für den Destruktor der Klasse `Klon`:

```
[Serializable]
public class Klon {
  string Ort;
  int KlonID;

  public Klon (int klonID, string ort) {
    this.KlonID = klonID;
    this.Ort = ort;
  }

  public void OrtMitteilen(string ort, int klonID){
     Console.WriteLine("Meine Identifikationsnummer ist {0} und ich" +
                       " bin hier: {1}.", klonID, ort);
  }

  public void ChaosVerursachen() {...}

  ~Klon() {
    TellOrt(this.Ort, this.KlonID);
    Console.WriteLine ("{0} wurde zerstört", KlonID);
  }
}
```

Hier ist der Konstruktor. Es scheint, als würden die Felder KlonID und Ort immer gefüllt, wenn ein Klon erzeugt wird.

Das ~-Zeichen (»Tilde«) sagt, dass der Code in diesem Block ausgeführt wird, wenn das Objekt vom Garbage Collector eingesammelt wird.

Das ist der Destruktor. Er sendet dem Bösewicht eine Nachricht, die ihm die ID und den Ort des scheidenden Klons mitteilt. Aber er wird nur ausgeführt, wenn das Objekt vom Garbage Collector eingesammelt wird.

Destruktoren schreiben Sie genau so wie Konstruktoren, aber statt eines Zugriffsmodifizierers stellen Sie dem Klassennamen ~ voran. Das sagt .NET, dass der Code im Destruktorblock ausgeführt werden soll, wenn das Objekt vom Garbage Collector eingesammelt wird.

Destruktoren können keine Parameter haben, da .NET dieses Objekt aufruft und nicht irgendein anderes.



Destruktoren und Finalisierer sind das Gleiche.

Manchmal sprechen Leute vom Finalisierer eines Objekts, manchmal von seinem Destruktor. Beide Begriffe bezeichnen eine Methode, die ausgeführt wird, wenn ein Objekt vom Garbage Collector eingesammelt wird. Wir werden in der Regel »Destruktor« verwenden.

Wann GENAU läuft ein Destruktor?

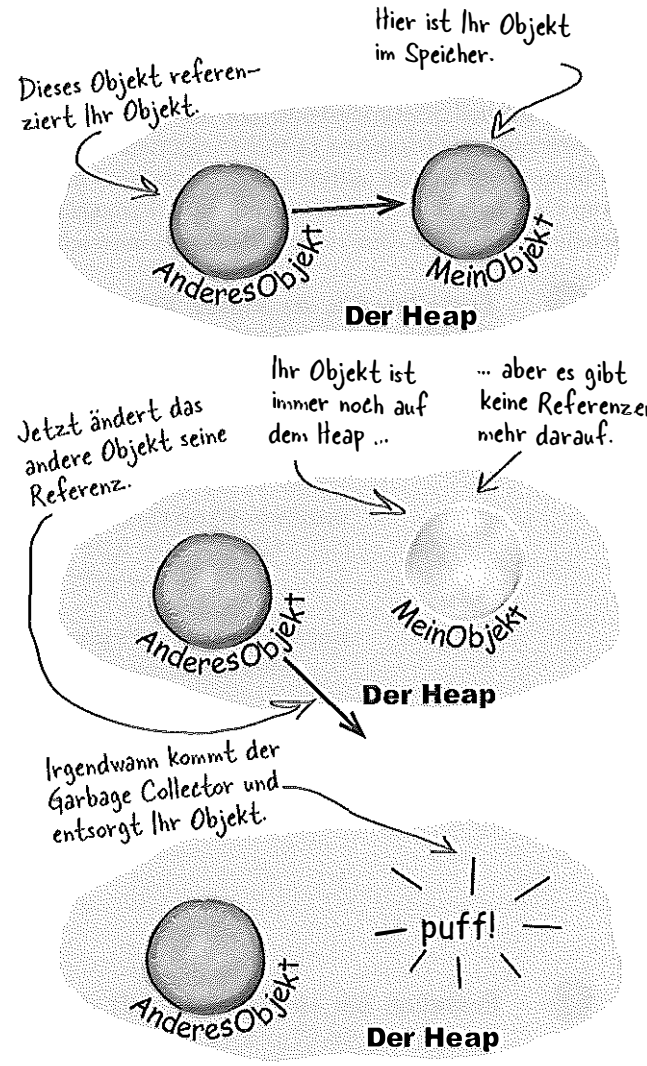
Der Destruktor, den Sie für Ihr Objekt erstellen, läuft, wenn das Objekt vom Garbage Collector eingesammelt wird. Und der Garbage Collector tritt auf den Plan, nachdem ***alle*** Referenzen auf Ihr Objekt verschwunden sind. Aber die Garbage Collection tritt nicht immer *sofort* ein, wenn die Referenzen verschwunden sind.

Nehmen Sie an, Sie haben ein Objekt und eine Referenz auf ein Objekt. .NET schickt den Garbage Collector an die Arbeit, und dieser prüft Ihr Objekt. Da es Referenzen auf das Objekt gibt, übergeht er es und geht weiter. Ihr Objekt bleibt im Speicher bestehen.

Dann passiert etwas. Das letzte Objekt, das eine Referenz auf *Ihr* Objekt enthält, verschwindet. Jetzt sitzt Ihr Objekt im Speicher, aber keine Referenz zeigt mehr darauf. Es kann nicht darauf zugegriffen werden. Im Grunde ist das **Objekt tot**.

Aber eine Sache fehlt noch. Die ***Garbage Collection wird von .NET gesteuert,*** nicht von Ihren Objekten. Wird der Garbage Collector in den nächsten paar Sekunden oder sogar Minuten nicht an die Arbeit geschickt, bleibt Ihr Objekt weiter im Speicher. Es kann nicht verwendet werden, wurde aber auch noch nicht eingesammelt. **Und der Destruktor Ihres Objekts wurde noch nicht ausgeführt.**

Schließlich schickt .NET den Garbage Collector aus. Diesmal ist Ihr Objekt tot. Es wird also entsorgt, und dann wird auch endlich der Destruktor ausgeführt ... vielleicht mehrere Minuten nachdem die letzte Referenz auf das Objekt gelöscht oder geändert wurde.



Sie können .NET VORSCHLAGEN, dass es Zeit ist, den Müll einzusammeln

.NET lässt Sie ***vorschlagen***, dass eine Garbage Collection jetzt eine gute Idee wäre. **Sie werden diese Methode selten verwenden, weil die Garbage Collection auf viele Bedingungen in der CLR eingestellt ist und *ihr Aufruf eigentlich keinguter Gedanke ist.*** Aber die Garbage Collection selbst aufzurufen hilft zu verstehen, wie ein Destruktor funktioniert. Um das zu tun, rufen Sie die Methode `GC.Collect()`.

```
public void KloneEntfernen(
                    List<Klon> klone) {
  foreach (Klon klon
                    in zuEntfernendeKlone)
    klone.Remove(klon);
  GC.Collect();
}
```

Geben Sie aber acht. Diese Methode ***zwingt*** .NET nicht, sofort an die Garbage Collection zu gehen. Sie sagt nur: »Führe sobald wie möglich eine Garbage Collection durch.«

Dispose() arbeitet mit `using`, Destruktoren mit der Garbage Collection

`Dispose()` wird ausgeführt, wenn ein Objekt, das in einer `using`-Anweisung erstellt wurde, auf null gesetzt wird oder alle Referenzen verliert. Nutzen Sie keine `using`-Anweisung, wird `Dispose()` nicht aufgerufen, wenn die Referenz auf null gesetzt wird – Sie müssen sie direkt aufrufen. Der Destruktor eines Objekts wird ausgeführt, wenn es vom Garbage Collector eingesammelt wird. Erzeugen wir ein paar Objekte, um uns anzuschauen, worin sich diese beiden Methoden unterscheiden:

Tun Sie das!

1 Erstellen Sie die Klasse Klon und lassen Sie sie IDisposable implementieren. Die Klasse sollte eine automatische `int`-Eigenschaft namens ID haben. Sie hat einen Konstruktor, eine `Dispose()`-Methode und einen Destruktor:

```
public class Klon : IDisposable {
    public int ID { get; private set; }

    public Klon(int ID) {
        this.ID = ID;
    }

    public void Dispose() {
        MessageBox.Show("Ich wurde entsorgt!",
                        "Klon " + ID + " sagt ...");
    }

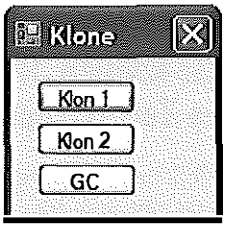
    ~Klon() {
        MessageBox.Show("Aaargh! Ich bin erledigt!",
                        "Klon " + ID + " sagt ...");
    }
}
```

Da die Klasse IDisposable implementiert, muss sie die Methode Dispose() haben.

Hier ist der Destruktor. Er wird ausgeführt, wenn das Objekt vom Garbage Collector vernichtet wird.

2 Erstellen Sie ein Formular mit drei Buttons. Erzeugen Sie im `Click`-Handler des ersten Buttons in einer `using`-Anweisung ein `Klon`-Objekt. Hier ist der erste Teil des Codes für den Button.

Hier ist das zu erstellende Formular.




```
private void klon1_Click(object sender, EventArgs e) {
    using (Klon klon1 = new Klon(1)) {
        // Nichts tun!
    }
}
```

Diese Methode erzeugt einen neuen Klon und tötet ihn dann sofort, indem sie die einzige Referenz löscht.

Da wir klon1 in einer using-Anweisung erstellt haben, wird seine Dispose()-Methode aufgerufen.

Sobald der using-Block fertig ist und die Dispose()-Methode aufgerufen wurde, gibt es keine Referenz darauf mehr. Der Klon wird für die Garbage Collection vorgemerkt.

3 Die anderen beiden Buttons implementieren.

Erzeugen Sie im `Click`-Handler des zweiten Buttons einen zweiten Klon und setzen Sie ihn manuell auf `null`:

```
private void klon2_Click(object sender, EventArgs e) {
    Klon klon2 = new Klon(2);
    klon2 = null;
}
```

Da hier keine using-Anweisung verwendet wird, wird Dispose() nicht ausgeführt, der Destruktor schon.

Rufen Sie aus dem dritten Button `GC.Collect()` auf, um eine Garbage Collection vorzuschlagen.

```
private void gc_Click(object sender, EventArgs e) {
    GC.Collect();
}
```

Das schlägt vor, den Garbage Collector auszuführen.

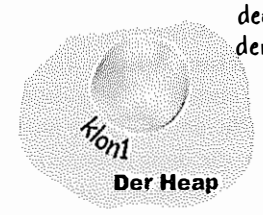
Denken Sie daran, dass das normalerweise keine gute Idee ist. Aber hier ist es in Ordnung, weil es ein gutes Mittel ist zu lernen, wie die Garbage Collection abläuft.

4 Führen Sie das Programm aus und spielen Sie mit Dispose() und Destruktoren.

Klicken Sie auf den ersten Button, und sehen Sie sich die Meldung an: `Dispose()` wird erst ausgeführt.



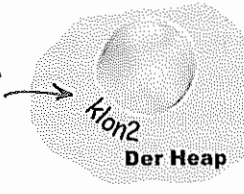
Obwohl das klon1-Objekt auf null gesetzt und seine Dispose-Methode ausgeführt wurde, ist es immer noch auf dem Heap und wartet auf den Garbage Collector.



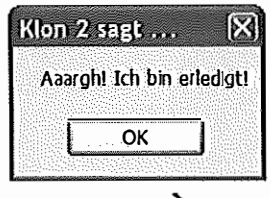
In den meisten Fällen sehen Sie die Garbage Collection-Nachricht ***nicht***. Auf Ihr Objekt zeigt zwar keine Referenz mehr, aber der Garbage Collector ist noch nicht gelaufen.

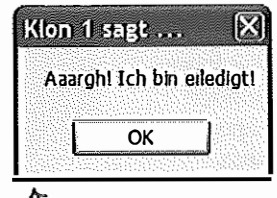
Klicken Sie jetzt auf den zweiten Button ... nichts passiert, stimmt's? Das liegt daran, dass wir keine `using`-Anweisung verwendet haben und `Dispose()` nicht aufgeführt wird. Und bis der Garbage Collector läuft, sehen wir keine Nachrichten aus den Destruktoren.

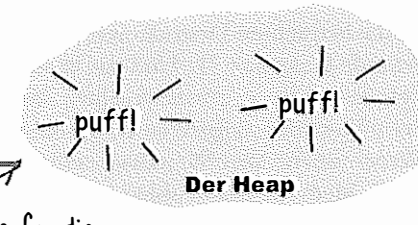
Jetzt ist auch klon2 auf dem Heap, aber es zeigt keine Referenz darauf.



Klicken Sie jetzt auf den dritten Button, um eine Garbage Collection vorzuschlagen. Sie sollten die Nachrichten aus den Destruktoren für `klon1` und `klon2` sehen.







Wird GC.Collect() ausgeführt, laufen die Destruktoren beider Objekte.

Spielen Sie mit dem Programm. Klicken Sie auf den Klon1-Button, den Klon2-Button und dann auf den GC-Button. Machen Sie das mehrfach. Manchmal wird klon1 zuerst eingesammelt und manchmal klon2. Gelegentlich wird der Garbage Collector sogar ohne den Aufruf von GC.Collect() ausgeführt.

Destruktoren können von der Stabilität abhängen

Beim Schreiben von Destruktoren können Sie sich nicht darauf verlassen, dass sie zu einem bestimmten Zeitpunkt ausgeführt werden. Selbst wenn Sie `GC.Collect()` aufrufen – was Sie vermeiden sollten, wenn Sie nicht einen sehr guten Grund dafür haben –, ***schlagen*** Sie nur vor, dass der Garbage Collector ausgeführt wird. Es gibt keine Garantie dafür, dass das sofort passiert. Und wenn es passiert, haben Sie keine Möglichkeit, in Erfahrung zu bringen, in welcher Reihenfolge Ihre Objekte eingesammelt werden.

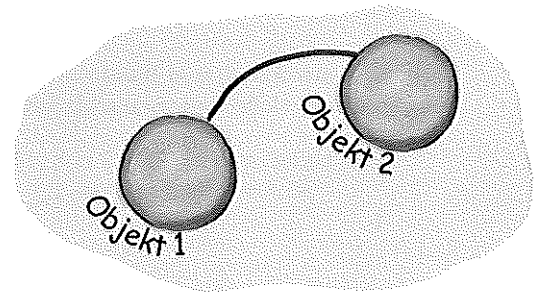
Was heißt das für die Praxis? Na, denken Sie mal darüber nach, was passiert, wenn Sie zwei Objekte haben, die einander referenzieren. Wird Objekt 1 zuerst eingesammelt, zeigt die Referenz von Objekt 2 auf ein Objekt, das es nicht mehr gibt. Aber wird Objekt 2 zuerst eingesammelt, ist die Referenz von Objekt 1 ungültig. Das bedeutet, dass ***Sie sich im Destruktor Ihres Objekts nicht auf Referenzen verlassen können***. Es wäre also ziemlich unklug, in einem Destruktor etwas zu machen, das davon abhängig ist, dass bestimmte Referenzen noch gültig sind.

Serialisierung ist ein gutes Beispiel für etwas, das Sie **in einem Destruktor nicht machen sollten**. Hält Ihr Objekt einen Haufen Referenzen auf andere Objekte, ist die Serialisierung davon abhängig, dass ***alle*** diese Objekte noch im Speicher sind ... und alle Objekte, die diese referenzieren und so weiter. Versuchen Sie, etwas zu serialisieren, wenn die Garbage Collection durchgeführt wird, kann das dazu führen, dass wichtige Teile Ihres Objekts **fehlen**, weil einige Objekte eingesammelt wurden, ***bevor*** der Destruktor ausgeführt wurde.

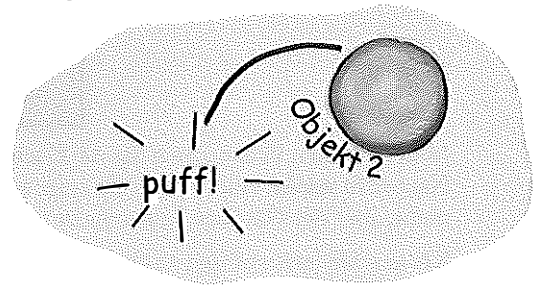
Glücklicherweise bietet uns C# ein sehr gute Lösung: `IDisposable`. Alles, was Ihre Kerndaten modifizieren könnte oder davon abhängig ist, dass andere Objekte noch immer im Speicher sind, sollte in der Methode `Dispose()` erfolgen, nicht in einem Destruktor.

Manche Leute betrachten Destruktoren als Sicherungssystem für die Methode `Dispose()`. Und das ist sinnvoll – bei Ihrem Klon-Objekt haben Sie ja gerade gesehen, dass die Methode nicht notwendigerweise aufgerufen wird, wenn Sie IDisposable implementieren. Aber Sie müssen aufpassen – wenn Ihre Dispose()-Methode davon abhängig ist, dass sich andere Objekte noch auf dem Heap befinden, kann es zu Problemen führen, wenn Sie aus Ihrem Destruktor Dispose() aufrufen. Das umgehen Sie am besten, indem Sie **immer eine using-Anweisung verwenden**, wenn Sie eine Klasse instantiieren, die IDispoable implementiert.

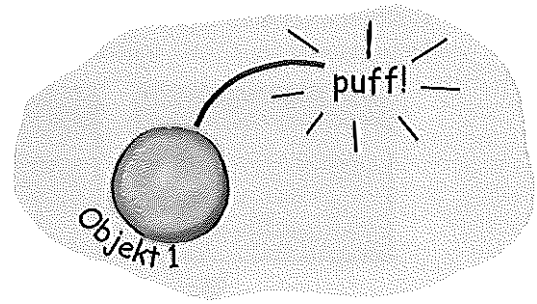
Nehmen wir an, Sie haben zwei Objekte, die einander referenzieren.



Werden beide zur gleichen Zeit für die Garbage Collection markiert, könnte Objekt 1 zuerst verschwinden ...



... oder Objekt 2 könnte vor Objekt 1 verschwinden. Sie können nicht wissen, in welcher Reihenfolge ...



... und deswegen kann sich der Destruktor eines Objekts nicht darauf verlassen, dass ein anderes Objekt noch auf dem Heap ist.

Lassen Sie ein Objekt sich selbst in Dispose() serialisieren

Hat man den Unterschied zwischen Dispose() und einem Destruktor einmal verstanden, ist es leicht, Klassen zu schreiben, deren Objekte sich selbst automatisch serialisieren, wenn sie entsorgt werden.



1 **Machen Sie die Klasse Klon (von Seite 630) serialisierbar.**
Stellen Sie der Klassendeklaration einfach das Attribut [Serializable] voran, damit wir sie in einer Datei speichern können.

```
[Serializable]
public class Klon : IDisposable
```

2 **Verändern Sie Dispose() so, dass das Objekt in eine Datei serialisiert wird.**
Nutzen wir einen `BinaryFormatter`, um `Klon` in `Dispose()` in eine Datei serialisieren zu lassen:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

// Bestehender Code.

public void Dispose() {
  string dateiname = "C:\\Temp\\Klon.dat";
  string verzname = "C:\\Temp\\";
  if (File.Exists(dateiname) == false) {
    Directory.CreateDirectory(verzname);
  }
  BinaryFormatter bf = new BinaryFormatter();
  using (Stream ausgabe = File.OpenWrite(dateiname)) {
    bf.Serialize(ausgabe, this);
  }
  MessageBox.Show("Hier ist das" + this.ID +
                  " muss.. serialisieren.. Objekt.");
}
```

Sie brauchen ein paar weitere using-Anweisungen, um die I/O-Klassen zu verwenden.

Das ist eine ganz gewöhnliche Serialisierung, obwohl wir die Datei fest vorgeben, da Dispose() keine Parameter übernehmen kann.

Eine Klasse wie diese könnten Sie so einrichten, dass sie im Konstruktor den Namen einer Datei übernimmt, in die sie sich schreiben soll.

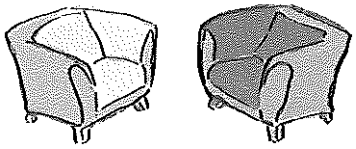
3 **Führen Sie die Anwendung aus.**
Sie werden das gleiche Verhalten sehen wie vor ein paar Seiten ... aber bevor `klon1` vom Garbage Collector eingesammelt wird, wird das Objekt in eine Datei serialisiert. Werfen Sie einen Blick in die Datei, sehen Sie eine binäre Darstellung des Objekts.



KOPF-NUSS

Wie, glauben Sie, sah der Rest des SuperHeld-Codes aus? Einen Teil haben wir Ihnen auf Seite 624 gezeigt. Könnten Sie jetzt den Rest schreiben?

Kamingespräche



Heute Abend: **Die Methode Dispose() und ein Destruktor fechten aus, wer wichtiger ist.**

Dispose:

Ehrlich gesagt, bin ich etwas überrascht, dass ich heute Abend eingeladen wurde. Ich dachte, in der Programmiererwelt sei man sich mittlerweile einig. Ist doch klar, dass ich viel wertvoller bin als du. Du bringst doch wirklich nicht viel. Du kannst dich nicht einmal serialisieren oder wichtige Daten ändern und so. Ziemlich instabil, oder?

Die Schnittstelle gibt es gerade, **weil** ich so wichtig bin. Ich bin sogar die einzige Methode darin!

Okay, stimmt schon, stimmt schon. Programmierer müssen wissen, dass sie mich brauchen, und müssen mich entweder direkt aufrufen oder eine using-Anweisung verwenden. Aber sie können sich immer darauf verlassen, zu welchem Zeitpunkt ich ausgeführt werde, und können mich einsetzen, um die Arbeiten durchzuführen, die erforderlich sind, um hinter ihren Objekten den Mist wegzukehren. Ich bin mächtig, zuverlässig und leicht zu verwenden. Ich bin eine dreifache Versuchung. Und du? Keiner weiß, wann du ausgeführt wirst oder in welchem Zustand sich die Anwendung befindet, wenn du dich endlich entschließt, in Aktion zu treten.

Du kannst eigentlich also nichts, was ich nicht auch kann. Aber du meinst, du wärst 'ne große Sache, weil du läufst, wenn die Garbage Collection erfolgt.

Destruktor:

Wie bitte? Sie müssen wohl immer über das Ziel hinausschießen! Ich sei schwach ... na gut. Ich wollte damit ja nicht anfangen. Aber wenn wir uns schon auf das Niveau begeben: Ich bin wenigstens nicht von einer Schnittstelle abhängig. Ohne `IDisposable` sind auch Sie nur eine ganz gewöhnliche nutzlose Methode.

Gut, gut ... reden Sie sich das ruhig ein. Und was passiert, wenn einer bei der Instantiierung eines Objekts die `using`-Anweisung vergisst? Dann werden Sie einfach ignoriert.

Handles nutzt Ihr Programm, wenn es .NET und die CLR übergeht und direkt mit Windows interagiert. Da .NET darüber nicht informiert ist, kann es diese für Sie auch nicht bereinigen.

Aber ohne mich gibt es keine Möglichkeit, etwas im allerletzten Augenblick auszuführen, unmittelbar bevor ein Objekt vom Garbage Collector eingesammelt wird. Ich kann Netzwerkressourcen freigeben, Fenster-Handles, Streams, alles, was für das restliche Programm zu einem Problem werden könnte, wenn es nicht ordentlich bereinigt wird. Ich kann sichern, dass Objekte gnädiger darauf reagieren, dass sie zu Müll werden. Darüber rümpft man die Nase nicht.

Das ziehe ich Ihrer Affèrei jederzeit vor, Freundchen.



Es gibt keine Dummen Fragen

F: Kann ein Destruktor alle Felder und Methoden eines Objekts nutzen?

A: Klar. Einem Destruktor können Sie zwar keine Parameter übergeben, aber auf die Felder des Objekts können Sie zugreifen – aufpassen müssen Sie nur, wenn diese Felder Referenzen auf andere Objekte sind. Diese könnten bereits eingesammelt sein. Methoden können Sie auf dem Objekt, das zerstört wird, natürlich auch aufrufen (solange diese Methoden nicht von anderen Objekten abhängig sind).

F: Was passiert, wenn in einem Destruktor eine Exception ausgelöst wird?

A: Gute Frage. In einem Destruktor kann man `try/catch` verwenden. Probieren Sie es aus. Erzeugen Sie in einem try-Block der gerade geschriebenen `Klon`-Klasse eine DevideByZeroException. Fangen Sie sie ab und zeigen Sie eine Meldung an, die sagt: »Es wurde eine Exception abgefangen.« Führen Sie das Programm jetzt aus und klicken Sie auf den ersten und dann auf den GC-Button. Sie sehen nun die Exception-Meldung und die alte Zerstört-Meldung. (Natürlich ist es normalerweise ein **sehr schlechter Gedanke**, in Destruktoren von Objekten, die mehr als Spielzeug sind, Meldungsfenster zu öffnen ... und es kann ja auch passieren, dass diese Meldungen nie angezeigt werden.)

F: Wie oft läuft der Garbage Collector automatisch?

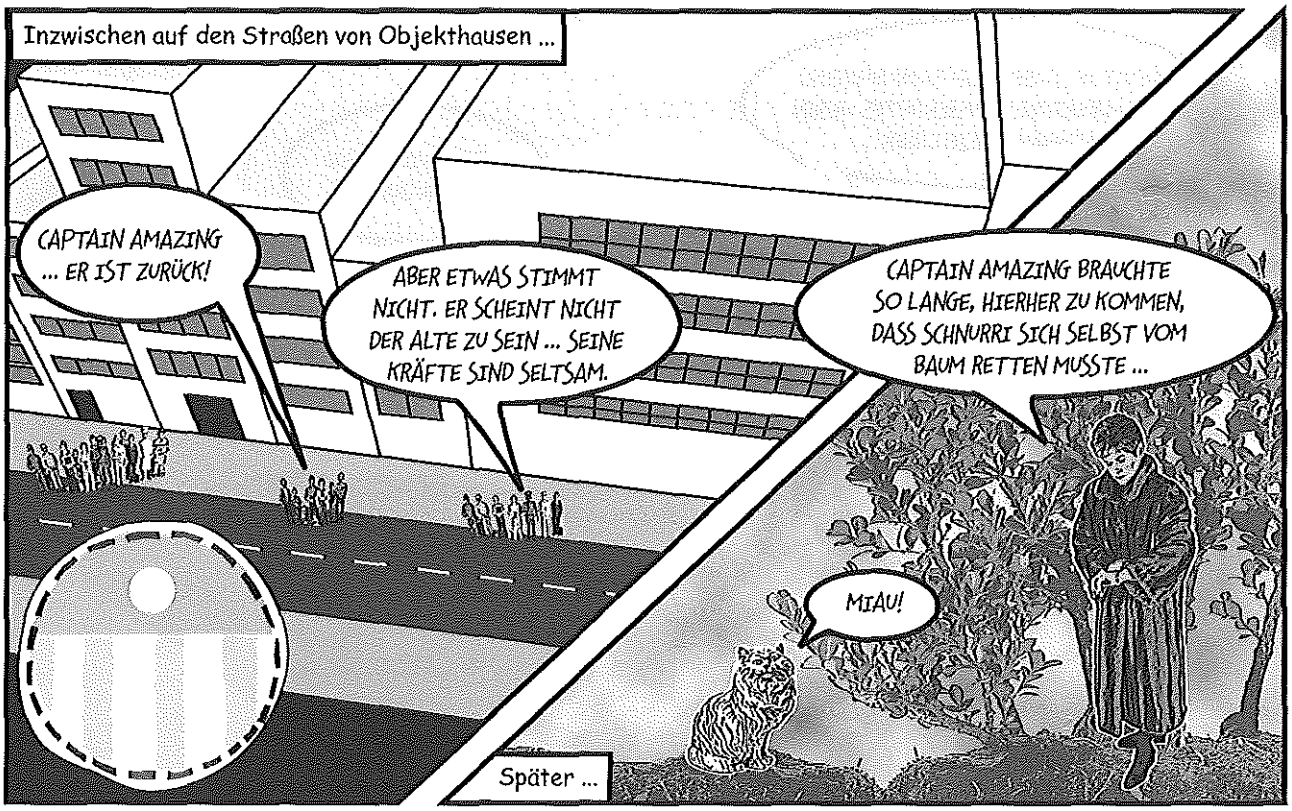
A: Auf diese Frage gibt es keine vernünftige Antwort. Er läuft in keinem leicht vorhersagbaren Rhythmus, und Sie haben keine genaue Kontrolle darüber. Sie können sicher sein, dass er ausgeführt wird, wenn das Programm beendet wird. Wann er ausgeführt wird, können Sie mit GC.Collect() beeinflussen ... und auch dann bleibt das Timing ein Problem.

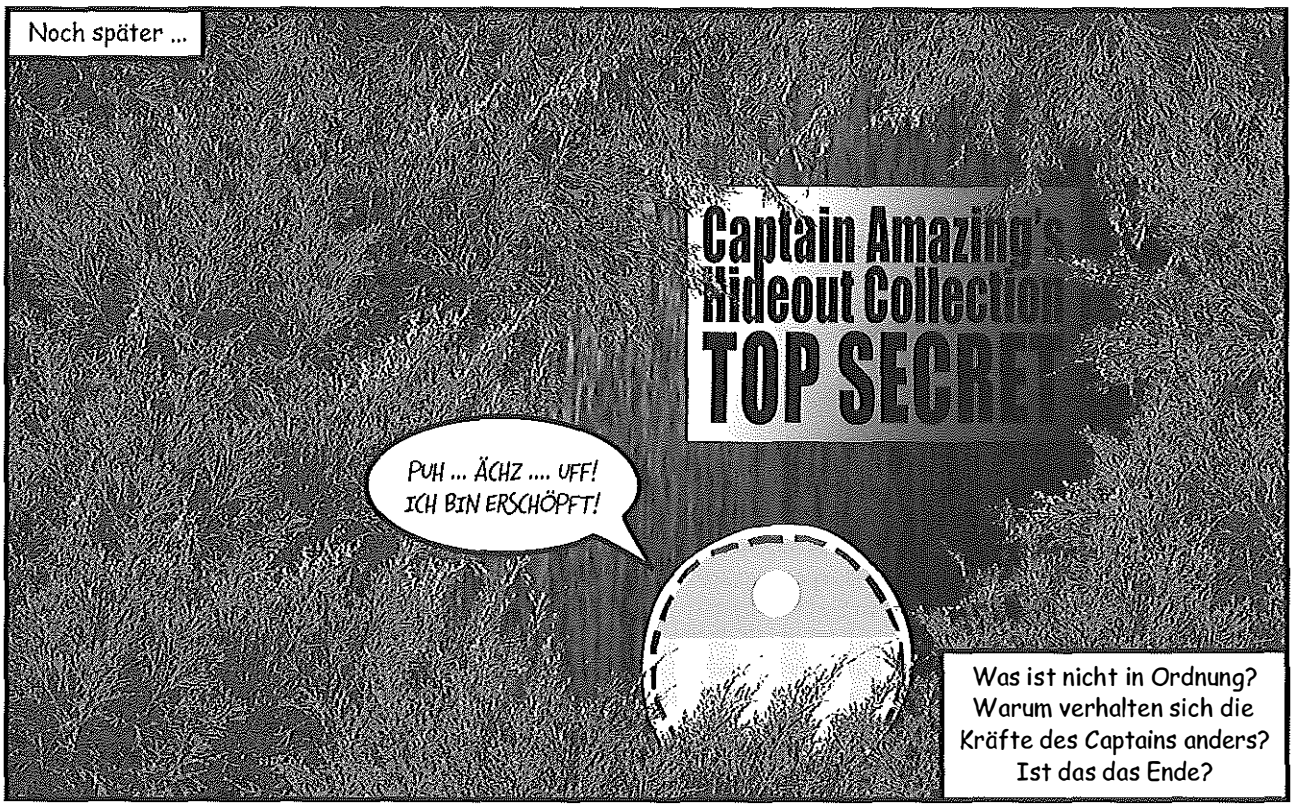
F: Wie schnell führt .NET die Garbage Collection aus, nachdem ich GC.Collect() aufgerufen habe?

A: Wenn Sie `GC.Collect()` ausführen, sagen Sie .NET, dass es die Garbage Collection sobald wie möglich ausführen soll. ***In der Regel*** erfolgt das, sobald .NET mit seiner aktuellen Arbeit fertig ist. Das heißt, dass es recht bald geschehen wird. Wann genau, können Sie aber nicht steuern.

F: Wenn etwas unbedingt ausgeführt werden muss, stecke ich es in einen Destruktor, oder?

A: Es kann passieren, dass Destruktoren nicht ausgeführt werden. Destruktoren können unterdrückt werden, wenn die Garbage Collection erfolgt. Oder der Prozess könnte vollständig beendet werden. Aber normalerweise sollten Destruktoren ausgeführt werden.


Inzwischen auf den Straßen von Objekthausen ...
CAPTAIN AMAZING ... ER IST ZURÜCK!
ABER ETWAS STIMMT NICHT. ER SCHEINT NICHT DER ALTE ZU SEIN ... SEINE KRÄFTE SIND SELTSAM.
CAPTAIN AMAZING BRAUCHTE SO LANGE, HIERHER ZU KOMMEN, DASS SCHNURRI SICH SELBST VOM BAUM RETTEN MUSSTE ...
MIAU!
Später ...


Noch später ...
Captain Amazing's Hideout Collection
TOP SECRET
PUH ... ÄCHZ .... UFF! ICH BIN ERSCHÖPFT!
Was ist nicht in Ordnung? Warum verhalten sich die Kräfte des Captains anders? Ist das das Ende?

Ein Struct *sieht aus* wie ein Objekt ...

Einer der Typen in .NET, über den wir noch nicht gesprochen haben, ist `struct`. `struct` ist eine Kurzform für **Struktur**, und `structs` sehen Objekten sehr ähnlich. Genau wie Objekte haben sie Felder und Eigenschaften. Und Sie können sie sogar an Methoden übergeben, die einen Parameter des Typs object erwarten:

Structs können Schnittstellen implementieren, aber keine anderen Klassen erweitern.

```
public struct FastEinSuperHeld : IDisposable {
    public int SuperKraft;
    public int SuperDing { get; private set; }

    public void BösewichtEliminieren(Bösewicht bösewicht)
    {
        Console.WriteLine("Gib auf, " + bösewicht.Name +
            ", mach dem Wahnsinn ein Ende!");
        if (bösewicht.Ergeben)
            bösewicht.InGefängnisGehen();
        else
            bösewicht.Töten();
    }

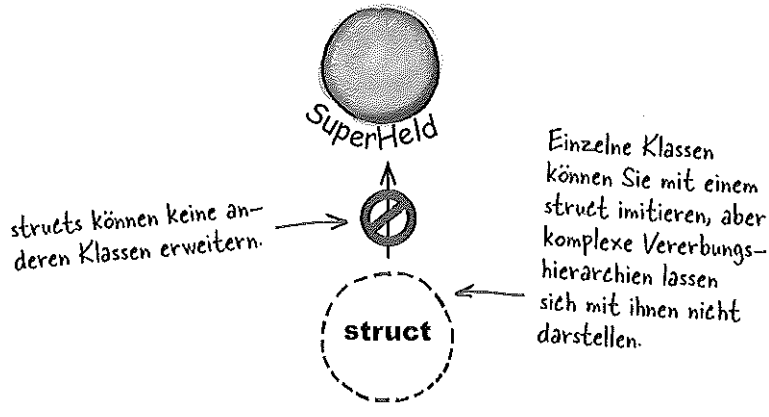
    public void Dispose() { ... }
}
```

Ein struct kann Eigenschaften und Felder haben ...

... und Methoden definieren.

... ist aber *nicht* auf dem Heap

Aber `structs` ***sind keine*** Objekte auf dem Heap. Sie können Methoden und Felder haben, aber keine Destruktoren. Außerdem können sie nicht von Klassen oder anderen `structs` erben oder selbst von Klassen oder `structs` erweitert werden.



Die Macht von Klassen liegt in ihrer Fähigkeit, Verhalten aus dem wahren Leben durch Vererbung und Polymorphie zu imitieren.

Structs sind am besten zur Speicherung von Daten geeignet, sind aber nicht so nützlich wie Klassen, wenn es um die Repräsentation von Verhalten geht.

Werte werden kopiert, Referenzen werden zugewiesen

Sie haben bereits eine Vorstellung davon, wie sich Typen unterscheiden. Auf der einen Seite haben Sie **Werttypen** wie `int`, `bool` und `string`. Auf der anderen Seite haben Sie **Objekttypen** wie `List`, `Stream` und `Exception`. Und diese funktionieren nicht auf gleiche Weise, oder?

Hier ist eine kurze Auffrischung zu Werttypen vs. Referenztypen.

Weisen Sie einer Werttypvariablen über das Gleichheitszeichen den Wert einer anderen Variablen zu, wird **eine Kopie des Werts erstellt**. Anschließend sind die beiden Variablen nicht mehr miteinander verbunden. Aber wenn Sie das Gleichheitszeichen mit Referenzen nutzen, **lassen Sie beide Referenzen auf das gleiche Objekt zeigen**.

Variablendeklarationen und -zuweisungen funktionieren bei Wert- und Referenztypen gleich.

```
int wieViele = 25;
bool furchteinflößend = true;
List<float> temperaturen = new List<float>();
Exception ex = new Exception("Berechnung unmöglich");
```

int und bool speichern Werttypen, List und Exception sind Objekttypen.

Die Initialisierung erfolgt bei allen auf die gleiche grundlegende Weise.

Bei der Zuweisung von Werten schleichen sich allerdings Unterschiede ein. Werttypen werden grundsätzlich kopiert. Hier ist ein Beispiel:

```
int fünfzehnMehr = wieViele;
fünfzehnMehr += 15;
Console.WriteLine("wieViele ist {0}, fünfzehnMehr ist {1}",
                  wieViele, fünfzehnMehr);
```

Diese Zeile kopiert den in der Variablen wieViele gespeicherten Wert in die Variable fünfzehnMehr und fügt ihm 15 hinzu.

Eine Änderung der Variablen fünfzehnMehr hat keine Auswirkungen auf wieViele und umgekehrt.

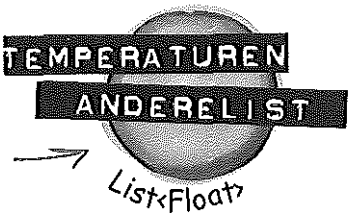
Die Ausgabe hier zeigt, dass `fünfzehnMehr` und `wieViele` ***nicht*** verbunden sind:

```
wieViele ist 25, fünfzehnMehr ist 40
```

Bei Objektzuweisungen werden aber keine Werte, sondern Referenzen zugewiesen:

```
temperaturen.Add(56.5F);
temperaturen.Add(27.4F);
List<float> andereList = temperaturen;
andereList.Add(62.9F);
```

Das lässt die Referenz andereList auf dasselbe Objekt zeigen wie die Referenz temperaturen.



Beide Referenzen zeigen auf dasselbe Objekt.

Eine Änderung der `List` bedeutet also, dass beide Referenzen die Aktualisierung sehen ... da beide auf das gleiche `List`-Objekt zeigen.

```
Console.WriteLine("temperaturen hat {0}, differentlist {1} Elemente",
                  temperaturen.Count(), andereList.Count());
```

Die Ausgabe hier zeigt, dass `andereList` und `temperaturen` tatsächlich auf ***dasselbe*** Objekt zeigen:

```
temperaturen hat 3, andereList 3 Elemente
```

Als Sie andereList.Add() aufriefen, wurde dem Objekt ein neuer Wert zugewiesen, auf das andereList und temperaturen zeigen.

Structs sind Werttypen, Objekte sind Referenztypen

Wenn Sie ein `struct` erstellen, erzeugen Sie einen **Werttyp**. Das bedeutet, dass Sie, wenn Sie eine `struct`-Variable über das Gleichheitszeichen einer anderen gleichsetzen, in der neuen Variablen eine neue *Kopie* des Structs erzeugen. Obwohl ein `struct` wie eine Klasse *aussieht*, verhält es sich nicht wie eine.



1 Erstellen Sie ein Struct namens Hund.

Hier ist ein einfaches Struct, das Hunde repräsentiert. Es sieht wie eine Klasse aus, ist aber keine.

```
public struct Hund {
  public string Name;
  public string Rasse;

  public Hund(string name, string rasse) {
    this.Name = name;
    this.Rasse = rasse;
  }

  public void Reden() {
      Console.WriteLine("Ich heiße {0} und bin ein {1}.", Name, Rasse);
  }
}
```

Ja, das ist keine ordentliche Kapselung. Aber so brauchen wir das, um dieses Problem zu erläutern.

2 Erstellen Sie eine Klasse namens Hundeartig.

Erstellen Sie eine Kopie des Structs `Hund` und ersetzen Sie `struct` durch `class` und `Hund` durch `Hundeartig`. Jetzt haben Sie eine Klasse `Hundeartig`, mit der Sie spielen können, die fast genau dem Struct `Hund` entspricht.

3 Fügen Sie einen Button hinzu, der einige Kopien von Hunden und Hundeartigen erstellt.

Hier ist der Code:

```
Hundeartig fido = new Hundeartig("Fido", "Mops");
Hundeartig tim = fido;
tim.Name = "Tim";
tim.Rasse = "Beagle";
fido.Reden();

Hund hans = new Hund("Hans", "Pudel");
Hund grete = hans;
grete.Name = "Grete";
grete.Rasse = "Pitbull";
hans.Reden();
```



4 Bevor Sie auf diesen Button klicken ...

Was wird auf die Konsole geschrieben, wenn Sie diesen Code ausführen?

.....

.....

Spitzen Sie Ihren Bleistift Lösung

Was wird auf die Konsole geschrieben?

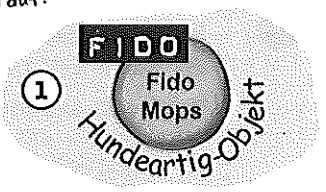
Ich heiße Tim und bin ein Beagle.

Ich heiße Hans und bin ein Pudel.

Folgendes ist passiert ...

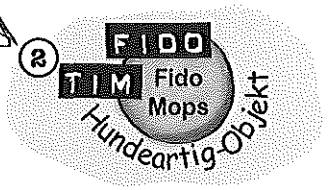
Die Referenzen `tim` und `fido` zeigen beide auf dasselbe Objekt, beide ändern also dieselben Felder und rufen dieselbe `Reden()`-Methode auf. Aber Structs funktionieren anders. Als Sie `grete` erstellt haben, haben Sie eine neue Kopie der Daten in `hans` erstellt. Diese beiden Structs sind vollkommen unabhängig voneinander.

Ein Hundeartig-Objekt wurde erstellt, und die Referenz fido zeigt darauf.

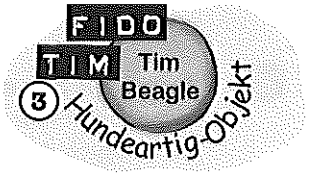



```
Hundeartig fido = new Hundeartig("Fido", "Mops"); (1)
Hundeartig tim = fido; (2)
tim.Name = "Tim";
tim.Rasse = "Beagle";
fido.Reden(); (3)
```

Die neue Referenzvariable tim wurde erstellt, aber dem Heap wird kein neues Objekt hinzugefügt – die Referenz tim zeigt auf dasselbe Objekt wie fido.

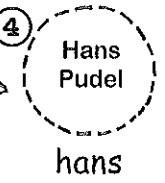


Da fido und tim auf dasselbe Objekt zeigen, rufen fido.Reden() und tim.Reden() die gleiche Methode auf und erzeugen beide dieselbe Ausgabe mit »Tim« und »Beagle«.

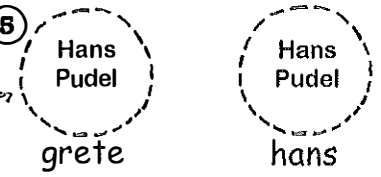


```
Hund hans = new Hund("Hans", "Pudel") (4)
Hund grete = hans (5)
grete.Name = "Grete";
grete.Rasse = "Pitbull";
hans.Reden(); (6)
```

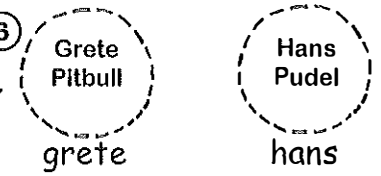
Erstellen Sie eine Instanz eines Structs, sieht diese einem Objekt sehr ähnlich – Sie haben eine Variable, über die Sie auf seine Felder und Methoden zugreifen können.



Das ist der entscheidende Unterschied. Als Sie die Variable grete hinzufügten, haben Sie ein neues Struct erzeugt, obwohl Sie das Schlüsselwort new gar nicht verwendet haben.



Da Sie eine neue Kopie der Daten erstellt hatten, blieb hans unverändert, als Sie die Felder von grete änderten.



Setzen Sie ein Struct einem anderen gleich, erzeugen Sie eine KOPIE der Daten des Structs. Das liegt daran, dass struct ein WERTTYP ist.

Stack vs. Heap: mehr zum Speicher

Hinter den Kulissen



Wie sich `structs` und Objekte unterscheiden, lässt sich leicht verstehen – eine Kopie eines Structs können Sie erstellen, indem Sie einfach das Gleichheitszeichen verwenden. Das geht bei Objekten nicht. Aber was passiert da hinter den Kulissen? Der Grund dafür, dass sich `structs` wie andere Werttypen verhalten, ist, dass **Werttypen nicht auf dem Heap gespeichert werden**. Die .NET CLR speichert Ihre Daten im Speicher an zwei Orten. Sie wissen bereits, dass sich Objekte auf dem **Heap** befinden. Daneben hält sie noch einen weiteren Speicherbereich, der als **Stack** (Stapel) bezeichnet wird. In ihm werden alle lokalen Variablen gespeichert, die Sie in Ihren Methoden deklarieren oder ihnen als Parameter übergehen. Den Stack können Sie sich als einen Haufen Fächer vorstellen, in die Sie Werte packen können. Wird eine Methode aufgerufen, fügt die CLR oben auf dem Stack weitere Fächer hinzu. Ist die Methode fertig, werden ihre Fächer entfernt.

Rufen Sie sich ins Gedächtnis zurück, dass die CLR aktiv den Speicher verwaltet, während Ihr Programm läuft, und sich um den Heap und die Garbage Collection kümmert.

Obwohl Sie eine object-Variable auf ein Struct setzen können, unterscheiden sich Structs und Objekte.

Der Code

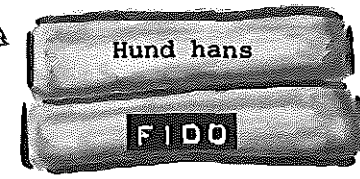
Code wie diesen könnten Sie in einem Programm sehen.

So sieht der Stack aus, nachdem diese Anweisungen ausgeführt wurden.

Der Stack

Hier befinden sich Structs und lokale Variablen.

```
Hundeartig fido = new Hundeartig("Fido", "Mops");
Hund hans = new Hund("Hans", "Pudel");
```



```
Hundeartig fido = new Hundeartig("Fido", "Mops");
Hund hans = new Hund("Hans", "Pudel");
Hund grete = hans;
```

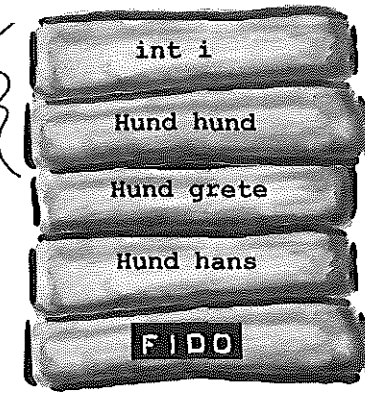
Erstellen Sie eine Variable für ein Struct – oder einen beliebigen anderen Werttyp –, wird auf den Stack ein neues »Fach« gestapelt. Dieses Fach ist eine Kopie des Werts in Ihrer Variablen.



```
Hundeartig fido = new Hundeartig("Fido", "Mops");
Hund hans = new Hund("Hans", "Pudel");
Hund grete = hans;
DreiMalSprechen(hans);

public DreiMalSprechen(Hund hund) {
    int i;
    for (i = 0; i < 5; i++)
        hund.Reden();
}
```

Rufen Sie eine Methode auf, legt die CLR ihre lokalen Variablen oben auf den Stack und nimmt sie wieder herunter, wenn die Methode fertig ist.





Moment. Sagten Sie nicht gerade, dass man object-Variablen auf ein Struct setzen kann? Wie soll das gehen, wenn ein Objekt auf dem Heap, ein Struct aber auf dem Stack ist?

Weisen Sie einer Referenzvariablen einen Werttypen zu, wird er eingepackt.

Gelegentlich müssen Sie eine Methode schreiben, die einen Werttypen ***oder*** einen Referenztypen akzeptieren kann – vielleicht eine Methode, die mit `Hund` oder mit `Hundeartig` arbeiten muss. Befinden Sie sich in dieser Situation, können Sie das Schlüsselwort `object` verwenden:

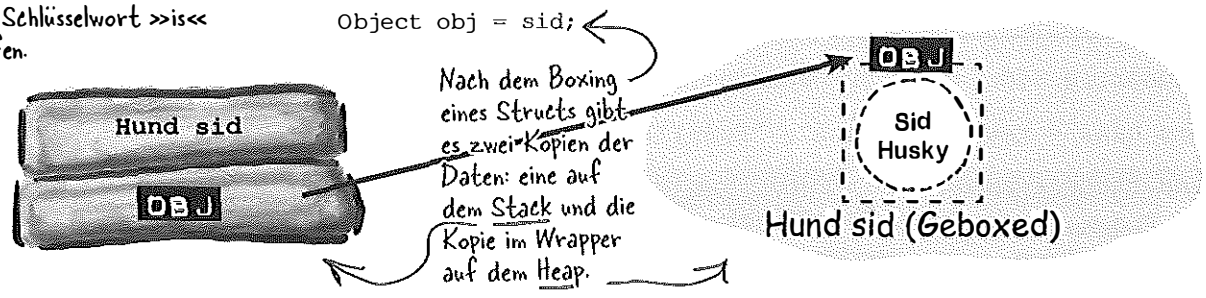
```
public HundOderHundeartigesAusführen(object etwas) { ... }
```

Senden Sie dieser Methode ein Struct, wird das Struct in eine spezielle Objekt-»Verpackung« **eingepackt**, die ihm ein Leben auf dem Heap erlaubt. Diese Verpackung bezeichnet man üblicherweise als **Wrapper** und den Vorgang des Ein-/Auspackens als **Boxing** bzw. **Unboxing**. Solange sich der Wrapper auf dem Heap befindet, können Sie mit dem Struct nicht viel anfangen. Sie müssen es »auspacken«, um mit ihm zu arbeiten. Glücklicherweise passiert das *automatisch*, wenn Sie einen Referenztyp auf einen Werttyp setzen oder einer Methode, die eine object-Referenz erwartet, einen Werttyp übergeben.

Ob ein Objekt ein verpacktes Struct oder ein anderer verpackter Werttyp ist, können Sie ebenfalls mit dem Schlüsselwort »is« prüfen.

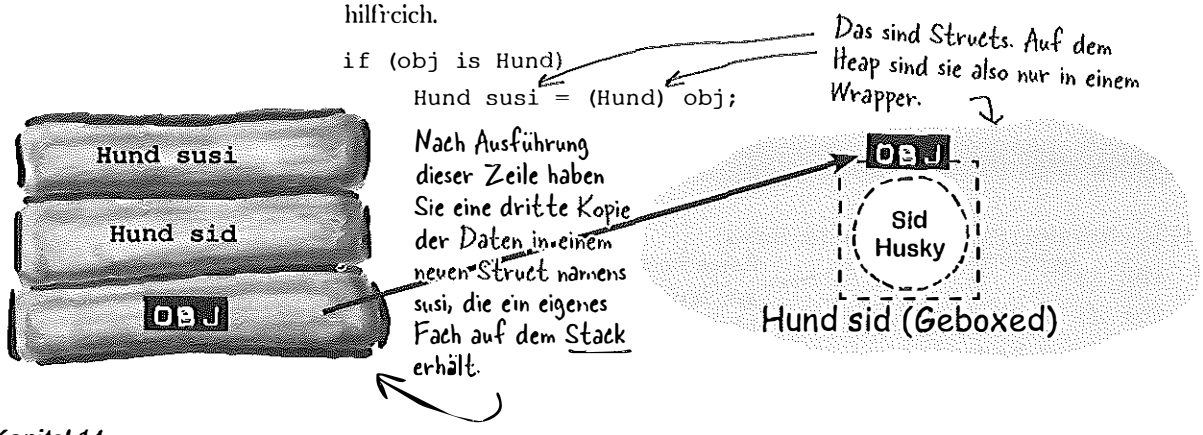
1 So sehen Stack und Heap aus, nachdem Sie eine object-Variable erstellt und ihr einen Hund-Struct-Wert zugewiesen haben.

```
Hund sid = new Hund("Sid", "Husky");
Object obj = sid;
```

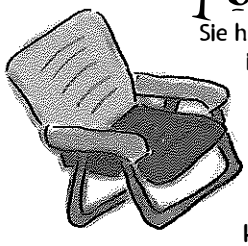


2 Möchten Sie das Objekt auspacken, müssen Sie es einfach auf den benötigten Typ casten. Das Unboxing erfolgt dann automatisch. Das Schlüsselwort `is` ist dabei hilfreich.

```
if (obj is Hund)
    Hund susi = (Hund) obj;
```



Pool-Puzzle



Sie haben die **Aufgabe**, die leeren Zeilen im Code mit den Codeschnipseln aus dem Pool zu füllen. Einzelne Schnipsel **können** mehrfach verwendet werden, und Sie werden nicht alle Schnipsel benötigen. Das **Ziel** ist es, eine Klasse zu erstellen, die sich kompilieren lässt und die gezeigte Ausgabe in die Konsole schreibt, wenn **eine neue Instanz der Klasse Wasserhahn erstellt wird.**

```
public class Wasserhahn {
    public Wasserhahn() {
        Tisch wein = new Tisch();
        Gelenk buch = new Gelenk();
        wein.Set(buch);
        buch.Set(wein);
        wein.Lampe(10);
        buch.Garten.Lampe("Vor");
        buch.Birne *= 2;
        wein.Lampe("Minuten");
        wein.Lampe(buch);
    }
}
```

Ausgabe, wenn Sie ein Wasserhahn-Objekt erstellen:

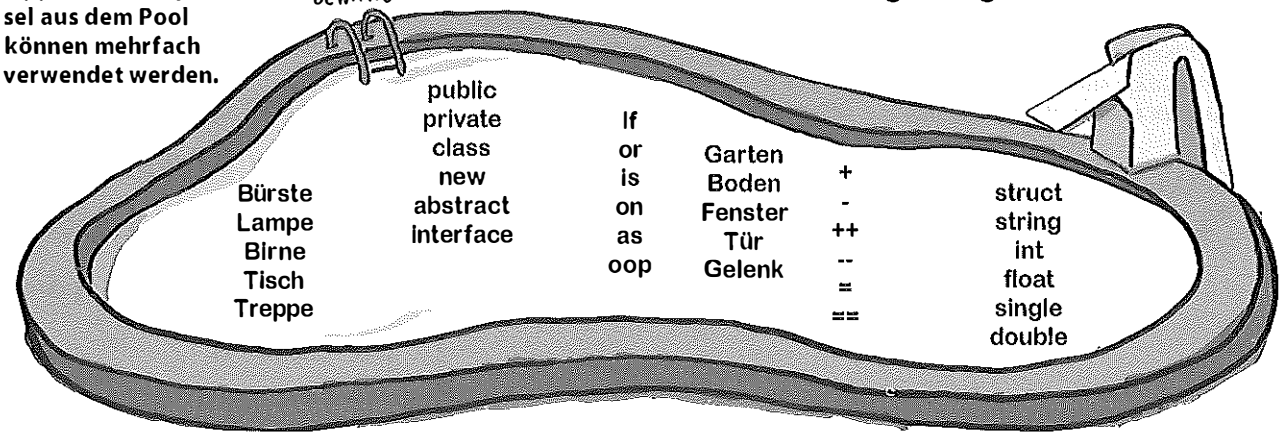
```
Vor 20 Minuten
```

```
public _________ Tisch {
    public string Treppe;
    public Gelenk Boden;
    public void Set(Gelenk b) {
        Boden = b;
    }
    public void Lampe(object öl) {
        if (öl ____ int)
            _________.Birne = (int)öl;
        else if (öl ____ string)
            Treppe = (string)öl;
        else if (öl ____ Gelenk) {
            _________ bier = öl ____ _________;
            Console.WriteLine(bier.Tisch()
             + " " + _________.Birne + " " + Treppe);
        }
    }
}
public _________ Gelenk {
    public int Birne;
    public Tisch Garten;
    public void Set(Tisch a) {
        Garten = a;
    }
    public string Tisch() {
        return _________.Treppe;
    }
}
```

Bonuspunkt: Kreisen Sie die Zeilen ein, in denen das Boxing erfolgt.

Tipp: Alle Schnipsel aus dem Pool können mehrfach verwendet werden.

Hier ist das Ziel ... diese Ausgabe soll bewirkt werden.



Antworten auf Seite 652.

Es gibt keine Dummen Fragen

F: Einen Moment noch. Warum sollte ich mich um den Stack kümmern?

A: Weil Sie Referenztypen und Werttypen besser verstehen, wenn Ihnen der Unterschied zwischen Stack und Heap klar ist. Dass Structs und Klassen anders funktionieren, vergisst man leicht – nutzen Sie mit ihnen ein Gleichheitszeichen, sieht das zunächst sehr gleich aus. Aber wenn Ihnen klar ist, wie .NET und die CLR das im Hintergrund verwalten, verstehen Sie auch, ***warum*** sich Wert- und Referenztypen unterschiedlich verhalten.

F: Und das Boxing? Welche Rolle spielt das für mich?

A: Weil Sie wissen müssen, wann Dinge auf dem Stack landen, und weil Sie wissen müssen, wenn Daten hin- und herkopiert werden. Boxing nimmt zusätzlichen Speicher und mehr Zeit in Anspruch. Wenn Sie das in Ihrem Programm nur ein paarmal machen (oder auch hundertmal), wird Ihnen der Unterschied nicht auffallen. Aber nehmen wir an, dass ein Programm das Gleiche immer wieder macht, millionenmal in der Sekunde. So weit hergeholt ist das gar nicht. Schließlich macht unser Bienenstock-Simulator genau das. Stellen Sie dann fest, dass Ihr Programm immer mehr Speicher benötigt oder immer langsamer läuft, können Sie es eventuell effektiver machen, indem Sie im wiederholten Teil Boxing vermeiden.

F: Ich erhalte eine neue Kopie eines Structs, wenn ich eine Struct-Variable einer anderen zuweise. Aber was bringt mir das?

A: Ein Punkt, bei dem das wirklich hilfreich ist, ist die **Kapselung**. Schauen Sie sich diesen vertrauten Code einer Klasse an, die ihren eigenen Ort kennt:

```
private Point ort;
public Point Ort{
  get { return ort; }
}
```

Wäre `Point` eine Klasse, wäre das eine armselige Kapselung. Es würde keine Rolle spielen, dass `ort` privat ist, weil Sie eine schreibgeschützte Eigenschaft öffentlich gemacht haben, die eine Referenz darauf zurückliefert. Jedes andere Objekt könnte also darauf zugreifen.

Glücklicherweise ist `Point` eigentlich ein Struct. Und das bedeutet, dass die öffentliche Eigenschaft `Ort` eine neue Kopie des Punkts liefert. Das Objekt, das die Kopie verwendet, kann mit ihr machen, was es will – keine dieser Änderungen wirkt sich auf das private Feld `ort` aus.

F: Point ist also ein Struct! Heißt das, dass ich die ganze Zeit auch schon mit *anderen* Structs gearbeitet habe?

A: Ja! Ein Struct, das bei der Arbeit mit Grafiken und Formularen sehr nützlich und verbreitet ist, ist **`Rectangle`**. Es hat einige sehr nützliche Methoden, die sehr praktisch sind, wenn Sie seine Grenzen ermitteln oder prüfen müssen, ob sich Punkte innerhalb oder außerhalb des Rechtecks befinden müssen. Sie müssen nur den Ort und die Größe setzen, und es berechnet automatisch die Eigenschaften Top, Bottom, Left, Right, Width und Height.

Ein weiteres nützliches Struct, das Ihnen begegnen wird, ist **Size**. Sie haben es bereits in Aktion gesehen – nämlich als Sie mit der Methode `MeasureString()` die Ausmaße eines Strings berechnet haben. Auch das ist ein Struct.

F: Wie weiß ich, ob ich ein Struct oder eine Klasse verwenden soll?

A: Meist nutzen Programmierer Klassen. Structs unterliegen einer Menge Einschränkungen, die dazu führen, dass sich komplexere Aufgaben mit ihnen nur schwer lösen lassen. Sie unterstützen keine Vererbung, keine Abstraktion und keine Polymorphie, und Sie wissen bereits, wie wichtig diese Dinge sind, wenn man saubere Programme aufbauen will.

Praktisch sind Structs, wenn Sie eine kleine, beschränkte Menge an Daten haben, mit denen Sie wiederholt arbeiten müssen. Rectangle und Point sind gute Beispiele dafür – Sie machen nicht viel damit, aber was Sie damit machen, machen Sie meist sehr oft. Structs sind häufig klein und haben eine beschränkte Reichweite. Haben Sie eine kleine Gruppe mit ein paar unterschiedlichen Arten von Daten, die Sie in einem Feld speichern oder an eine Methode übergeben wollen, ist das wahrscheinlich ein guter Kandidat für ein Struct.

Structs können die Kapselung verbessern, weil eine schreibgeschützte Eigenschaft, die ein Struct liefert, immer eine neue Kopie liefert.

Spitzen Sie Ihren Bleistift



Diese Methode soll ein Klon-Objekt vernichten, funktioniert aber nicht. Warum nicht?

```
private void KlonNullSetzen(Klon klon) {
    klon = null;
}
```

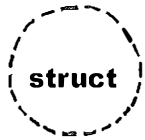
.....

.....

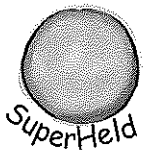
← Ein unangekündigter Test! Antworten gibt es auf Seite 649.

Captain Amazing ... nicht ganz

Nach all dem Gerede über Boxing sollte Ihnen recht klar sein, was mit dem weniger mächtigen und schneller erschöpften Captain Amazing los war. Es war eigentlich gar nicht Captain Amazing, sondern ein verpacktes Struct:



VS.



Das ist ein großer Vorteil von Structs (und anderen Werten) – von ihnen können Sie leicht Kopien erstellen.

1. **Structs können keine Klassen erben.**
   Kein Wunder, dass die Superkräfte des Captains etwas schwach schienen! Er hat keinerlei Verhalten geerbt.

2. **Structs kommen nur in einem Wrapper auf den Heap**
   Das Struct kam erst nach einem Boxing auf den Heap.

1. **Man kann keine neue Kopie eines Objekts erstellen.**
   Weisen Sie einer Referenzvariablen eine andere zu, kopieren Sie eine ***Referenz*** auf ***dasselbe*** Objekt.

3. **Mit Objekt kann man das Schlüsselwort »as« verwenden.**
   Objekte ermöglichen Polymorphie, indem Sie gestatten, dass ein Objekt eins der Objekte vertritt, deren Klasse es erbt.



Erweiterungsmethoden fügen BESTEHENDEN Klassen neue Verhalten hinzu

Manchmal müssen Sie eine Klasse erweitern, von der Sie nicht erben können, beispielsweise eine Klasse, die als `sealed` gekennzeichnet ist (viele .NET-Klassen sind versiegelt und können deswegen nicht erweitert werden). C# gibt Ihnen ein mächtiges Mittel dafür: **Erweiterungsmethoden**. Fügen Sie Ihrem Projekt eine Klasse mit Erweiterungsmethoden hinzu, **fügt** es bestehenden **Klassen neue Methoden hinzu**. Dazu müssen Sie nur eine statische Klasse erstellen und ihr eine statische Methode hinzufügen, die als ersten Parameter eine Instanz der Klasse annimmt, und diesen mit dem Schlüsselwort `this` verbinden.

Nehmen wir an, Sie hätten die Klasse GewöhnlicherMensch:

```
public sealed class GewöhnlicherMensch {
    private int alter;
    private int gewicht;

    public GewöhnlicherMensch(int gewicht){
        this.gewicht = gewicht;
    }

    public void ZurArbeitGehen() { // Zur-Arbeit-Gehen-Code }
    public void RechnungenBezahlen() { // Rechnung-Zahlen-Code }
}
```

Die Klasse GewöhnlicherMensch ist versiegelt. Von ihr können also keine Unterklassen gebildet werden. Aber was ist, wenn wir ihr eine Methode hinzufügen möchten?

Eine Erweiterungsmethode verwenden Sie, indem Sie den ersten Parameter über das Schlüsselwort »this« angeben.

Da wir die Klasse GewöhnlicherMensch erweitern möchten, geben wir als ersten Parameter `this GewöhnlicherMensch` an.

Die Klasse SuperheldSerum fügt GewöhnlicherMensch eine Erweiterungsmethode hinzu:

```
public static class SuperheldSerum {
    public static string MauerBrechen(this GewöhnlicherMensch h, double wandstärke) {
        return ("Habe die Wand der Stärke " + wandstärke + " durchbrochen.");
    }
}
```

Erweiterungsmethoden sind immer statisch und müssen sich in statischen Klassen befinden.

Wird dem Projekt die Klasse SuperheldSerum hinzugefügt, erhält GewöhnlicherMensch die Methode MauerBrechen. Diese kann dann verwendet werden:

```
private void button1_Click(object sender, EventArgs e) {
    GewöhnlicherMensch stefan = new GewöhnlicherMensch(185);
    Console.WriteLine(stefan.MauerBrechen(89.2));
}
```

Erzeugt das Formular eine Instanz von GewöhnlicherMensch, kann es direkt auf die Methode MauerBrechen() zugreifen – wenn es Zugriff auf die Klasse SuperheldSerum hat.

Spitzen Sie Ihren Bleistift

Diese Methode soll ein Klon-Objekt vernichten, funktioniert aber nicht. Warum nicht?

```
private void KlonNullSetzen(Klon klon) {
    klon = null;
}
```

Der Parameter klon ist nur auf dem Stack. Wird er auf null gesetzt, hat das keine Auswirkungen auf den Heap.

Es wird nur der eigene Methodenparameter auf null gesetzt, der lediglich eine der Referenzen auf den Klon ist.

Es gibt keine Dummen Fragen

F: Können Sie mir noch einmal erklären, warum ich die neue Methode der Klasse nicht direkt hinzufügen kann? Warum muss ich stattdessen die Erweiterung verwenden?

A: Wenn Sie die Klasse ändern können, werden Sie das in den meisten Fällen auch tun – Erweiterungsmethoden sind nicht als Ersatz für die Vererbung gedacht. Aber sie können sehr praktisch sein, wenn Sie mit Klassen zu tun haben, die Sie nicht erweitern können. Richtig mächtig werden Erweiterungsmethoden, wenn Sie das Verhalten von etwas ändern müssen, auf das Sie ***normalerweise keinen Zugriff haben***, wie einen Typ oder eine Klasse, die mit dem .NET Framework oder einer anderen Bibliothek ausgeliefert wird.

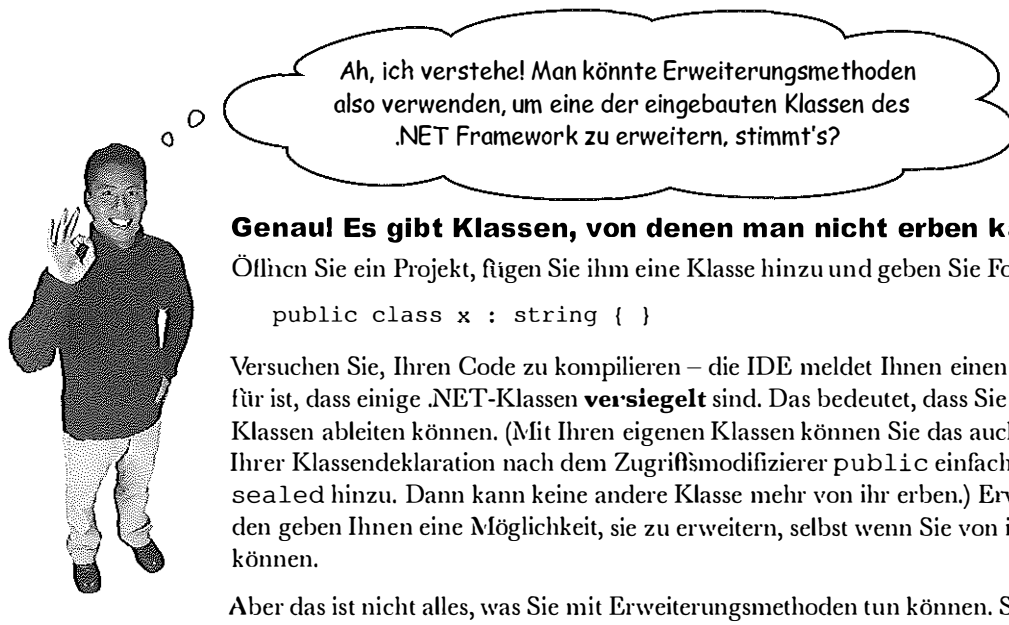
F: Warum soll ich Erweiterungsmethoden überhaupt verwenden? Warum nutzen wir nicht einfach Vererbung?

A: Sicher. Aber dann müssten Sie Ihre eigenen Klassen verwenden, und bestehender Code, der die ursprünglichen, nicht erweiterten Klassen verwendet, müsste sich auch ändern. Mit Erweiterungsmethoden können Sie das Verhalten einer ganzen Gruppe von Objekten ändern und sogar einigen der grundlegendsten Klassen des .NET Framework neue Funktionalitäten geben.

Die Erweiterung von Klassen gibt Ihnen neue Verhalten, verlangt aber, dass Sie die Unterklasse verwenden, wenn Sie dieses neue Verhalten nutzen wollen.

F: Wirkt sich meine Erweiterungsmethode auf alle Instanzen einer Klasse oder nur auf eine bestimmte aus?

A: Sie wirkt sich auf alle Instanzen der Klasse aus, die Sie erweitern. Nachdem Sie eine neue Erweiterungsmethode erstellt haben, wird die neue Methode in der IDE sogar unter den normalen Methoden angezeigt.



Genau! Es gibt Klassen, von denen man nicht erben kann.

Öffnen Sie ein Projekt, fügen Sie ihm eine Klasse hinzu und geben Sie Folgendes ein:

```
public class x : string { }
```

Versuchen Sie, Ihren Code zu kompilieren – die IDE meldet Ihnen einen Fehler. Grund dafür ist, dass einige .NET-Klassen **versiegelt** sind. Das bedeutet, dass Sie von ihnen keine Klassen ableiten können. (Mit Ihren eigenen Klassen können Sie das auch tun! Fügen Sie Ihrer Klassendeklaration nach dem Zugriffsmodifizierer `public` einfach das Schlüsselwort `sealed` hinzu. Dann kann keine andere Klasse mehr von ihr erben.) Erweiterungsmethoden geben Ihnen eine Möglichkeit, sie zu erweitern, selbst wenn Sie von ihr nicht erben können.

Das ist eine andere Sache, die Sie mit Vererbung nicht machen können – Schnittstellen können nicht geerbt werden.

Aber das ist nicht alles, was Sie mit Erweiterungsmethoden tun können. Sie können damit nicht nur Klassen, sondern auch **Schnittstellen** erweitern. Dazu müssen Sie als ersten Parameter der Erweiterungsmethode nach dem Schlüsselwort `this` statt des Klassennamens einen Schnittstellennamen angeben. Tun Sie das, wird die Erweiterungsmethode **jeder Klasse hinzugefügt, die diese Schnittstelle implementiert**. Erinnern Sie sich an den LINQ-Code, den Sie Ihrem Simulator in Kapitel 12 hinzugefügt haben? LINQ wurde vollständig mit Erweiterungsmethoden aufgebaut, die die Klasse `IEnumerable` erweitern. (Eine Menge mehr zu LINQ lernen Sie in Kapitel 15.)

Einen elementaren Typ erweitern: string

Es passiert nicht oft, dass Sie die Möglichkeit haben, das Verhalten eines der grundlegendsten Typen einer Sprache wie Strings zu erweitern. Aber mit Erweiterungsmethoden können Sie genau das tun!

Tun Sie das!

1 Stecken Sie Ihre Erweiterungsmethoden in einen separaten Namensraum.

Sie sollten Ihre Erweiterungsmethoden in einem anderen Namensraum speichern als den Rest Ihres Codes. So vermeiden Sie, dass Sie Schwierigkeiten haben, Ihre Erweiterungen zu finden, wenn Sie sie in anderen Projekten verwenden möchten. Richten Sie auch eine statische Klasse für Ihre Methoden ein.

```
namespace Notfall {
  public static class StringErweiterung {
```

Die Verwendung eines separaten Namensraums hilft der Organisation.

Die Klasse, in der die Erweiterungsmethode definiert wird, muss statisch sein.

2 Erstellen Sie eine statische Erweiterungsmethode und definieren Sie den ersten Parameter mit `this` sowie dem zu erweiternden Typ.

Bei der Deklaration von Erweiterungsmethoden müssen Sie zwei wichtige Dinge beachten: Die Methode muss statisch sein und als ersten Parameter die zu erweiternde Klasse angeben.

```
public static bool IstNotruf (this string s){
```

>>this string<< sagt, dass wir string erweitern.

Auch die Erweiterungsmethode muss statisch sein.

3 Stecken Sie in die Methode Code zur String-Auswertung.

```
public static class StringErweiterung {
  public static bool IstNotruf(this string s){
    if (s.Contains("Hilfe!"))
      return true;
    else
      return false;
  }
}
```

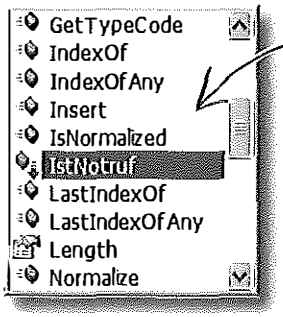
Das prüft den String auf einen bestimmten Wert ... etwas, das sich in der Standardklasse string ganz sicher nicht befindet.

4 Erstellen Sie ein Formular und fügen Sie ihm einen String hinzu.

Fügen Sie Ihrem Code `using Notfall;` hinzu. Wenn Sie einen String nutzen, erhalten Sie dann auch die Erweiterungsmethode. Prüfen Sie das, indem Sie den Namen einer String-Variablen und einen Punkt eingeben:

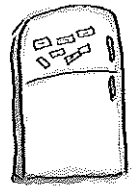
```
string nachricht;
nachricht = "Eine Klonarmee zerstört die Fabrik. Hilfe!";
nachricht.
```

Sobald Sie den Punkt eingeben, blendet die IDE ein IntelliSense-Fenster mit allen Methoden von String ein ... einschließlich Ihrer Erweiterungsmethode.



Kommentieren Sie die using-Zeile aus, verschwindet die Erweiterungsmethode aus dem IntelliSense-Fenster.

Diese Spielerei zeigt Ihnen nur die Syntax von Erweiterungsmethoden. Wie nützlich sie sind, können Sie im nächsten Kapitel sehen. Das dreht sich um LINQ, das vollständig über Erweiterungsmethoden implementiert ist.



Erweiterungs-Magneten

Ordnen Sie die Magneten so, dass diese Ausgabe erzeugt wird:

```
ein Euro erzeugt mehr Euros
```

```
namespace Oben {
    public static class Rand {
        public static void Senden
```

```
namespace Seitlich {
    using Oben;
    public static class Ticker {
```

```
    }
    public static string ZuPreis
```

```
    }
```

```
    }
}
```

```
    }
}
```

```
s.Senden();
```

```
i = 3;
```

```
b.Grün().Senden();.ZuPreis()
```

```
b.Grün().Senden(); = false
```

```
(this bool b) {
```

```
bool b = true;
```

```
if (b == true)
    return "er";
```

```
else
    return " mehr Euros";
```

```
int i = 1;
```

```
Console.Write(s);
```

```
(this string s) {
```

```
(this int n) {
```

```
public static string Grün
```

```
if (n == 1)
    return "ein Euro ";
```

```
b.Grün().Senden();;
```

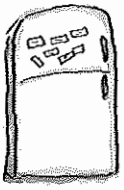
```
Grün().Senden();
```

```
public static void Main() {
```

```
.Senden();
```

```
else
    return "zeugt";
```

```
string s = i.ZuPreis();
```



Erweiterungs-Magneten, Lösung

Sie sollten mithilfe der Magneten diese Ausgabe erzeugen:

```
ein Euro erzeugt mehr Euros
```

Der Namensraum Oben enthält die Erweiterungen. Der Namensraum Seitlich enthält den Einstiegspunkt.

Die Klasse Rand erweitert string mit einer Methode namens Senden(), die einfach einen String auf die Konsole schreibt, und einer Methode namens ZuPreis(), die >>ein Euro<< liefert, wenn der int gleich 1 ist, oder >>mehr Euros<<, wenn er das nicht ist.

Der Einstiegspunkt nutzt die Erweiterungen, die Sie in der Klasse Rand ergänzt haben.

```
namespace Oben {
  public static class Rand {
    public static void Senden (this string s) {
      Console.Write(s);
    }
    public static string ZuPreis (this int n) {
      if (n == 1)
        return "ein Euro ";
      else
        return " mehr Euros";
    }
    public static string Grün (this bool b) {
      if (b == true)
        return "er";
      else
        return "zeugt";
    }
  }
}
```

Die Methode Grün erweitert bool – ist der bool-Wert true, liefert sie den String >>er<<, ist er false, den String >>zeugt<<.

```
namespace Seitlich {
  using Oben;
  public static class Ticker {
    public static void Main() {
      int i = 1;
      string s = i.ZuPreis();
      s.Senden();
      bool b = true;
      b.Grün().Senden();
      b = false;
      b.Grün().Senden();
      i = 3;
      i.ZuPreis().Senden();
    }
  }
}
```





Der GLOBUS

CAPTAIN AMAZING RECYCELT

Der Tod war nicht das Ende!

Von Stefan Stark
GLOBUS-Redakteur

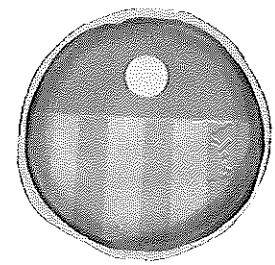
OBJEKTHAUSEN

Captain Amazing deserialisiert sich und startet Comeback

In einer überraschenden Wendung der Dinge ist Captain Amazing nach Objekthausen zurückgekehrt. Letzten Monat wurde entdeckt, dass der Sarg von Captain Amazing leer war und anstelle seines Körpers nur eine seltsame Nachricht enthielt. Eine Analyse der Nachricht hat jetzt die DNA des Captain Amazing-Objekts offenbart – seine letzten Felder und Werte, zuverlässig festgehalten in einem binären Format.

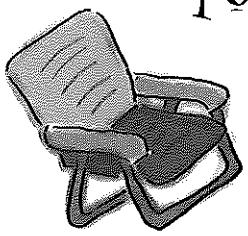
Heute wurden diese Daten wieder zum Leben erweckt. Der Captain ist zurück, aus seiner eigenen Nachricht deserialisiert. Gefragt, wie er einen solchen Plan ersonnen habe, zuckte der Captain nur mit den Schultern und murmelte: »Kapitel 9.« Personen aus dem Umfeld des Captains weigerten sich, diese kryptische Antwort zu kommentieren, bestätigten aber, dass der Captian vor seiner gescheiterten Auseinandersetzung mit dem Fiesling eine Menge Zeit mit Büchern verbracht und Dispose-Methoden und Persistenz studiert hatte. Wir erwarten …

… Fortsetzung auf Seite 5



Captain Amazing ist zurück!

Pool-Puzzle, Lösung von Seite 643



Die Methode Lampe() setzt die verschiedenen Strings und ints. Rufen Sie sie mit einem int auf, wird das Birne-Feld des Objekts, auf das Gelenk zeigt, darauf gesetzt.

Ausgabe, wenn Sie ein Wasserhahn-Objekt erstellen:

```
Vor 20 Minuten
```

```
public class Wasserhahn {
    public Wasserhahn() {
        Tisch wein = new Tisch();
        Gelenk buch = new Gelenk();
        wein.Set(buch);
        buch.Set(wein);
        wein.Lampe(10);
        buch.Garten.Lampe("Vor");
        buch.Birne *= 2;
        wein.Lampe("Minuten");
        wein.Lampe(buch);
    }
}
```

Deswegen muss Tisch ein Struct sein. Wäre es eine Klasse, würde wein auf das gleiche Objekt zeigen wie buch.Garten. Das würde dazu führen, dass dieser Code den String >>Vor<< überschreibt.

Bonuspunkt: Kreisen Sie die Zeilen ein, in denen das Boxing erfolgt.

Da die Methode Lampe() einen object-Parameter erwartet, erfolgt automatisch ein Boxing, wenn ihr ein int oder ein String übergeben wird.

```
public struct Tisch {
    public string Treppe;
    public Gelenk Boden;
    public void Set(Gelenk b) {
        Boden = b;
    }
    public void Lampe(object öl) {
        if (öl is int)
            Boden.Birne = (int)öl;
        else if (öl is string)
            Treppe = (string)öl;
        else if (öl is Gelenk) {
            Gelenk bier = öl as Gelenk;
            Console.WriteLine(bier.Tisch()
             + " " + Boden.Birne + " " + Treppe);
        }
    }
}
```

Übergeben Sie Lampe einen String, wird das Treppe-Feld auf den Wert des Strings gesetzt.

Denken Sie daran, dass das Schlüsselwort as nur mit Klassen, nicht mit Structs funktioniert.

```
public class Gelenk {
    public int Birne;
    public Tisch Garten;
    public void Set(Tisch a) {
        Garten = a;
    }
    public string Tisch() {
        return Garten.Treppe;
    }
}
```

Gelenk und Tisch haben beide eine Set()-Methode. Die Set()-Methode von Gelenk setzt das Tisch-Feld Garten, die Set()-Methode von Set() setzt das Gelenk-Feld Boden.

15 LINQ

Bekommen Sie Ihre Daten in den Griff

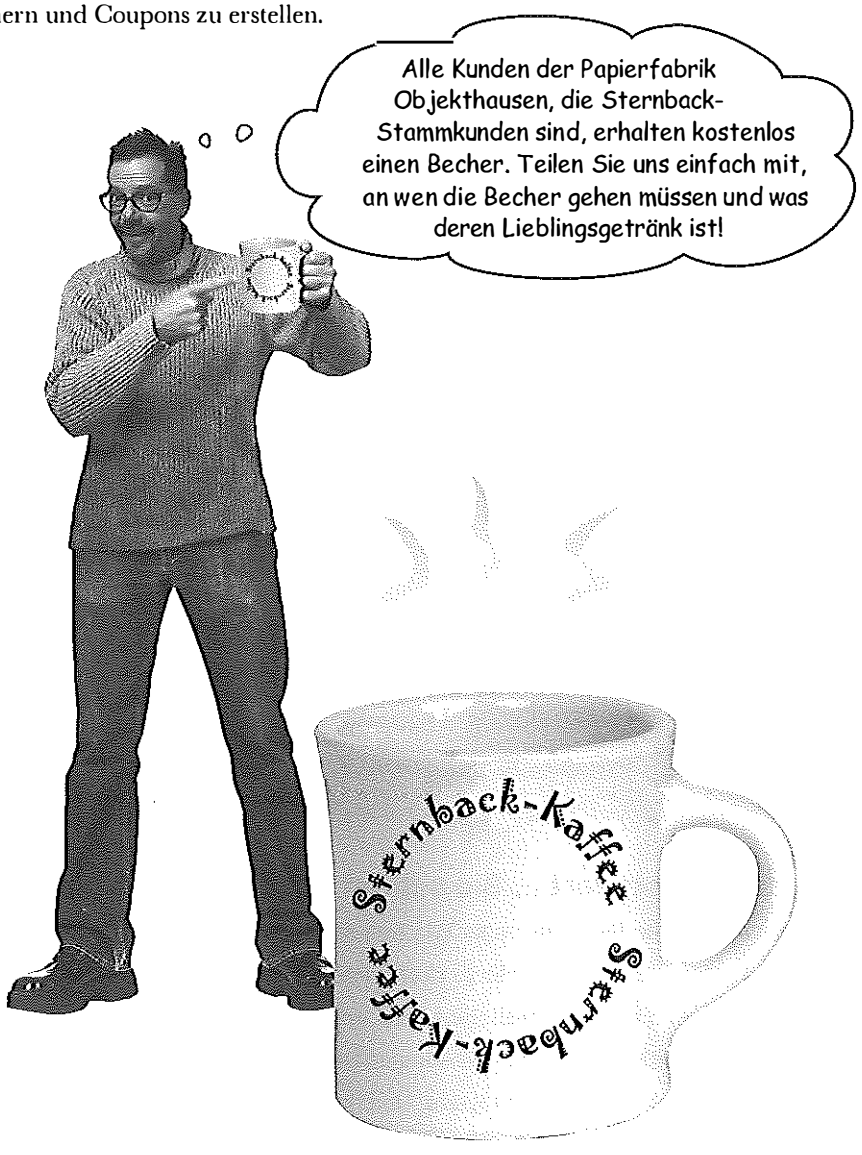


Die Welt ist datengesteuert ... Sie sollten damit zu leben wissen.

Vorbei sind die Zeiten, in denen Sie Tage, sogar Wochen programmieren konnten, ohne sich mit **Massen von Daten** befassen zu müssen. Heute ***dreht sich alles um Daten.*** Und häufig muss man sich sogar mit Daten **aus unterschiedlichen Quellen** herumschlagen ... und in unterschiedlichen Formaten. Datenbanken, XML, Auflistungen aus anderen Programmen ... all das ist heute die Aufgabe eines guten C#-Programmierers. Und mit **LINQ** lässt sich all das bewältigen. Mit LINQ können Sie **Ihre Daten** nicht bloß auf leichte, intuitive Weise **abfragen**, sondern auch **gruppieren** und **Daten von unterschiedlichen Quellen zusammenführen**.

Ein leichtes Projekt ...

Die Papierfabrik Objekthausen möchte eine gemeinsame Werbeaktion mit Sternback-Kaffee starten. Sternback hat ein Programm für Stammkunden, das nachhält, wer wie oft welche Getränke kauft. Die Papierfabrik Objekthausen möchte herausfinden, **welche ihrer Kunden Sternback-Stammkunden sind**, und diesen einen Kaffeebecher und einen Coupon für ihr Lieblingsgetränk zusenden ... Sie haben die Aufgabe, die Daten zusammenzuführen und Kundenlisten für den Versand von Bechern und Coupons zu erstellen.

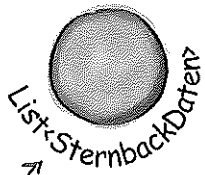


... aber die Daten sind völlig zerstreut

Sternback hat alle Daten in Klassen, die zu einer riesigen `List` zusammengefasst sind. Aber die Daten der Papierfabrik stecken in einer Datenbank (die aus Kapitel 1). Wir möchten alle Sternback-Kunden finden, die mehr als 90 € ausgegeben haben, und diese mit der Kontaktliste der Papierfabrik Objekthausen vergleichen: **Wir suchen nach den Namen der Personen, dem Unternehmen, für das sie arbeiten, und ihrem Sternback-Lieblingsgetränk.**

Die Sternback-Daten stecken in einer List<>

Die Leute von Sternback lieferten ein Programm, das eine Verbindung mit ihrer Website herstellt und alle Daten in eine `List` mit `SternbackDaten`-Objekten zieht.



Hier sind eine Klasse und ein Enum aus dem Sternback-Code.

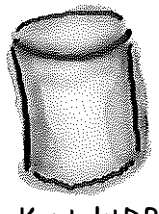
Sie müssen die Liste mit den Sternback-Daten abrufen und die Kunden finden, die denen der Papierfabrik entsprechen.

```
public class SternbackDaten
{
    public string Name { get; set; }
    public Getränk Lieblingsgetränk { get; set; }
    public int GeldAusgegeben { get; set; }
    public int Besuche { get; set; }
}


public enum Getränk {
    StandardKaffee,
    SchokoRockoLatte,
    DreifachEspresso,
    ZitronengrasTee,
    DoppelCappuccino,
    Entkoffeiniert,
    SchokoMacchiato,
    BananaSplitImBecher,
}
```

Die Kundendaten haben Sie bereits

Die Kontaktliste der Papierfabrik Objekthausen haben Sie bereits in Kapitel 1 aufgebaut – sie enthält Teile der Daten, die Sie benötigen.



Die Kundendaten der Papierfabrik stecken in einer Datenbank.



KOPF-NUSS

Wie würden Sie die Daten von Sternback und der Papierfabrik Objekthausen kombinieren, um eine vollständige Kontaktliste zu erhalten?

LINQ kann Daten aus mehreren Quellen ziehen

Im Bienenstock-Simulator haben Sie LINQ eingesetzt, um die Bienen nach Tätigkeit zu gruppieren. Dort haben Sie die Macht von LINQ genutzt, um eine einfache Abfrage zu schreiben, die Daten aus einer Auflistung zieht. Mit den Sternhack-Daten kann LINQ genau so arbeiten wie mit den Bienen. Sie können Abfragen nutzen, um die Kundendaten abzurufen. Vorausgesetzt, die Auflistung implementiert die Schnittstelle `IEnumerable`, können Sie darauf LINQ-Abfragen verwenden.

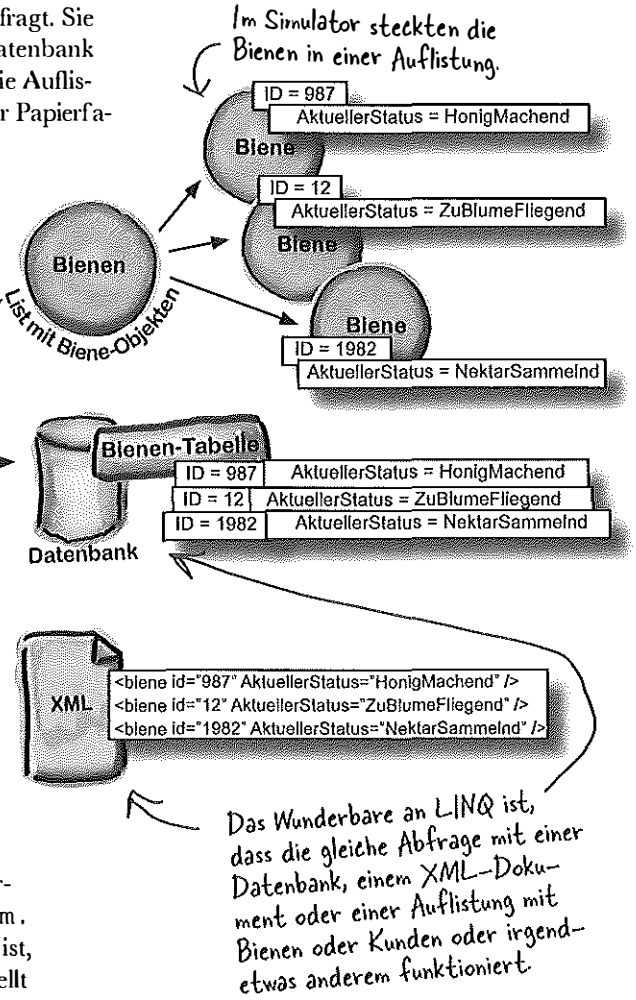
Aber mit LINQ werden noch andere Dinge als Auflistungen abgefragt. Sie können die gleichen Abfragen verwenden, um Daten aus einer Datenbank oder sogar einem XML-Dokument herauszuziehen. Haben wir die Auflistungen einmal im Griff, können wir LINQ auf der Datenbank der Papierfabrik verwenden.

Hier ist die Abfrage, die wir im Bienenstock-Simulator verwendet haben, um die Bienen anhand ihres Zustands zu gruppieren und zu ordnen.

```
var bienenGruppen =
  from biene in welt.Bienen
  group biene by biene.AktuellerStatus
  into bienenGruppe
  orderby bienenGruppe.Key
  select bienenGruppe;
```

LINQ

Wir brauchen eine ähnliche Abfrage, um die Daten aus den Sternback-Kundendaten herauszuziehen, die sich ebenfalls in einer Auflistung befinden.



LINQ funktioniert mit fast jeder Datenquelle, die Sie in .NET verwenden können. Ihr Code muss am Anfang eine `using System.Linq;`-Zeile enthalten, aber das war es auch schon. Noch besser ist, dass die IDE in alle Codedateien, die mit Visual Studio 2008 erstellt werden, automatisch eine LINQ-Referenz einschließt. Wenn Sie Visual Studio 2008 oder höher verwenden, können Sie also gleich mit dem Programmieren beginnen. LINQ steht Ihnen sofort zur Verfügung.

.NET-Auflistungen sind bereits für LINQ eingerichtet

Alle Auflistungstypen in .NET implementieren die Schnittstelle `IEnumerable<Int>`. Geben Sie in der IDE `System.Collections.Generic.IEnumerable` ein, klicken Sie mit der rechten Maustaste auf die Zeile und wählen Sie »Gehe zu Definition«. Sie werden feststellen, dass die Schnittstelle `IEnumerable` die Methode `GetEnumerator()` definiert:

Alle Auflistungen und Arrays implementieren IEnumerable. Viele von ihnen tun das, indem sie die Schnittstelle IEnumerable<T> implementieren, die ihrerseits IEnumerable implementiert.

Das T bedeutet, dass IEnumerable mit jedem beliebigen Typ arbeitet.

```
namespace System.Collections.Generic {
  public interface IEnumerable<T> : IEnumerable {
        // Zusammenfassung:
        //     Gibt einen Enumerator zurück, der eine Auflistung durchläuft.
        //
        // Rückgabewerte:
        //     Ein System.Collections.IEnumerator<T>-Objekt, das zum
        //     Durchlaufen der Auflistung verwendet werden kann.
    IEnumerator<T> GetEnumerator();
  }
}
```

Das ist die einzige Methode in dieser Schnittstelle. Alle Auflistungen implementieren diese Methode. Sie könnten auch eine eigene Auflistung implementieren, die IEnumerable implementiert ... Wenn Sie das tun, können Sie LINQ auch mit Ihrer Auflistung verwenden.

Diese Methode verlangt, dass Auflistungen eine Möglichkeit bieten, die Elemente der Auflistung nacheinander zu durchlaufen. Das ist die einzige Voraussetzung von LINQ. Wenn Sie eine Datenliste elementweise durchlaufen können, kann LINQ die Auflistung abfragen.



Hinter den Kulissen

LINQ nutzt **Erweiterungsmethoden**, über die Sie Daten abfragen, sortieren und aktualisieren können. Prüfen Sie das selbst. Erstellen Sie ein `int`-Array namens `linqtest`, stecken Sie einige Zahlen in das Array und geben Sie dann diese Codezeile ein (keine Sorge, was sie macht, werden Sie gleich lernen):

```
var ergebnis = from i in linqtest where i < 3 select i;
```

Jetzt verstehen Sie, warum die Erweiterungsmethoden aus Kapitel 14 so wichtig sind ... sie ermöglichen .NET (und Ihnen), bestehenden Typen neue Verhalten zu geben.

Kommentieren Sie jetzt `using System.Linq;` am Anfang der von Ihnen erstellten Datei aus. Versuchen Sie, die Projektmappe neu zu erstellen, werden Sie feststellen, dass sich diese Zeile nicht mehr kompilieren lässt. Die Methoden, die Sie aufrufen, wenn Sie LINQ nutzen, sind einfach Erweiterungsmethoden, die verwendet werden, um Arrays zu erweitern.

LINQ vereinfacht Abfragen

Hier ist ein einfaches Beispiel für die LINQ-Syntax. Es wählt in einem `int`-Array alle Zahlen kleiner 37 aus und bringt sie in eine aufsteigende Reihenfolge. Das macht sie mit vier **Klauseln**, die sagen, welche Auflistung abgefragt werden soll, welche Kriterien verwendet werden, welche Member der Auflistung abgefragt und wie die Ergebnisse zurückgeliefert werden sollen.

```
int[] werte = new int[] {0, 12, 44, 36, 92, 54, 13, 8};

var ergebnis = from w in werte
               where w < 37
               orderby w
               select w;

foreach(int i in ergebnis)
  Console.WriteLine(i);
```

Das weist den Werten im abzufragenden Array in der Anfrage den Buchstaben »w« zu. w ist also erst 0, dann 12, dann 44, dann 36 ... usw.

Diese LINQ-Abfrage hat vier Klauseln: die from-Klausel, eine where-Klausel, eine orderby-Klausel und die select-Klausel.

Das sagt, dass alle Werte im Array ausgewählt werden, die kleiner als 37 sind.

Dann werden diese Werte in eine Reihenfolge gebracht (von klein nach groß).

Wenn Sie schon mit SQL gearbeitet haben, erscheint es Ihnen vielleicht seltsam, dass das select am Ende steht, aber so funktioniert das in LINQ eben.

Jetzt können Sie das Ergebnis-Array durchlaufen und die einzelnen LINQ-Ergebnisse ausgeben

Ausgabe:

```
0, 8, 12, 13, 36
```

var

var ist ein Schlüsselwort, das den Compiler anweist, den Typ einer Variablen zur Kompilierzeit zu ermitteln. .NET ermittelt den Typ der lokalen Variablen, die Sie mit LINQ abfragen. Wenn Sie Ihre Projektmappe erstellen, ersetzt der Compiler var mit dem erforderlichen Typ für die Daten, mit denen Sie arbeiten.

Im Beispiel oben wird das var in dieser Zeile

```
var ergebnis = from w in werte
```

durch den Compiler folgendermaßen ersetzt:

```
IEnumerable<int>
```



LINQ ist ein neues Feature von C# 3.0 und Visual Studio 2008.

Wenn Sie eine frühere Version von C# verwenden, sollten Sie sich einen Moment Zeit nehmen und Visual C# 2008 Express Edition herunterladen und installieren. Es wird von Microsoft frei zur Verfügung gestellt und kann neben vorangehenden Versionen installiert werden.

LINQ ist einfach, Ihre Abfragen sind das nicht immer

Tim hat gerade sein Start-up-Unternehmen an einen großen Investor verkauft und möchte Teile seines Gewinns in die teuerste Captain Amazing-Ausgabe investieren, die er finden kann. Aber da sind diese Unmengen an Daten. Wie kann LINQ ihm helfen, seine beiden Sammlungen durchzugehen und herauszufinden, welche Comics am teuersten sind?

1. Tim hat von einer Captain Amazing-Fansite eine Liste aller Captain Amazing-Folgen heruntergeladen. Er hat sie in eine `List<>` mit `Comic`-Objekten gesteckt, die zwei Felder haben: `Name` und `Folge`.

```
public class Comic {
    public string Name { get; set; }
    public int Folge { get; set; }
}
```

Seinen Katalog hat Tim mit Objekt- und Auflistungsinitialisierern erstellt:

```
private static List<Comic> KatalogGenerieren() {
    List<Comic> comics = new List<Comic>();
    comics.Add(new Comic("Johnny America vs. the Pinko", 6));
    comics.Add(new Comic("Rock and Roll (limitierte Auflage)", 19));
    comics.Add(new Comic("Woman's Work", 36));
    comics.Add(new Comic("Hippie Madness (Fehldruck)", 57));
    comics.Add(new Comic("Revenge of the New Wave Freak (beschädigt)", 68));
    comics.Add(new Comic("Black Monday", 74));
    comics.Add(new Comic("Tribal Tattoo Madness", 83));
    comics.Add(new Comic("The Death of an Object", 97));
    return comics;
}
```

Folge 74 von Captain Amazing heißt »Black Monday«.

2. Glücklicherweise gibt es auf Gregors Liste einen florierenden Marktplatz für Captain Amazing-Comics. Er weiß, dass Folge 57, »Hippie Madness«, falsch gedruckt wurde und der Verlag deswegen fast die gesamte Auflage makuliert hat. Auf Gregors Liste hat er kürzlich eine seltene Kopie für 13.525 € gefunden. Nach ein paar Stunden Suchcrei konnte Tim ein `Dictionary<>` aufbauen, das Folgen Preise zuordnete.

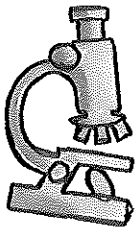
```
private static Dictionary<int, int> PreiseAufbauen() {
    Dictionary<int, int> werte = new Dictionary<int, int>();
    werte.Add(6, 3600);
    werte.Add(19, 500);
    werte.Add(36, 650);
    werte.Add(57, 13525);
    werte.Add(68, 250);
    werte.Add(74, 75);
    werte.Add(83, 25);
    werte.Add(97, 35);
    return werte;
}
```

Folge 57 ist 13.525 € wert.



KOPF-NUSS

Schauen Sie sich die LINQ-Abfrage auf Seite 658 genau an. Was muss Tim in seine Abfrage packen, um die teuersten Folgen zu finden?



Anatomie einer Abfrage

Es ist leicht, Tims Daten mit einer LINQ-Abfrage abzufragen. Die `where`-Klausel sagt LINQ, welche Elemente aus der Auflistung in das Ergebnis eingeschlossen werden sollen. Aber diese Klausel muss kein einfacher Vergleich sein. Sie kann jeden gültigen C#-Ausdruck enthalten – beispielsweise das `values`-Dictionary verwenden, um nur die Comics zurückzuliefern, die mehr als 500 € wert sind. Und die `orderby`-Klausel funktioniert auf gleiche Weise – wir können LINQ anweisen, die Comics nach Wert zu sortieren.

```
List<Comic> comics = KatalogGenerieren();
Dictionary<int, int> werte = PreiseAufbauen();

var amTeuersten =
    from comic in comics
    where werte[comic.Folge] > 500
    orderby werte[comic.Folge] descending
    select comic;

foreach (Comic comic in amTeuersten)
    Console.WriteLine("{0} ist {1:c} wert",
                      comic.Name, werte[comic.Folge]);
```

Die Abfrage nutzt die Daten im Dictionary werte, um Comic-Objekte aus der List comics zu ziehen.

Die erste Klausel ist die `from`-Klausel. Sie weist LINQ an, `comics` abzufragen und den Namen `comic` für die einzelnen Datenelemente aus der Liste zu verwenden.

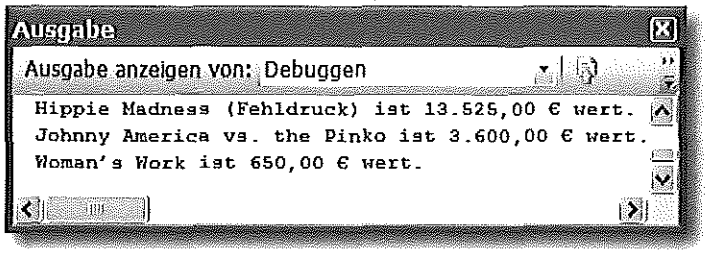
Der Name in der where-Klausel ist beliebig. Wir haben comic gewählt.

Der Name `comic` wurde in der from-Klausel definiert, damit wir ihn in diesen Klauseln verwenden können.

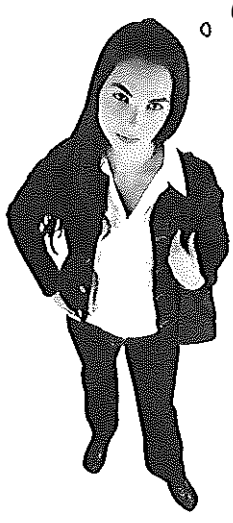
Die where- und orderby-Klauseln können BELIEBIGE C#-Ausdrücke enthalten. Wir können also `werte` verwenden, um nur die Comics zu wählen, die mehr als 500 € wert sind, und sie so liefern lassen, dass die teuersten zuerst kommen.

Fügen Sie der WriteLine-Ausgabe >>{1:c}<< hinzu, wird der zweite Parameter im lokalen Währungsformat ausgegeben.

Das Ergebnis der Abfrage wird in der Auflistung `amTeuersten` gespeichert. Die select-Klausel bestimmt, was in das Ergebnis kommt – da sie `comic` wählt, liefert die Abfrage Comic-Objekte.



Machen Sie sich keine Gedanken, wenn Sie noch nie mit SQL gearbeitet haben – um mit LINQ zu arbeiten, müssen Sie nichts darüber wissen. Aber wenn Sie neugierig sind, sollten Sie sich »SQL von Kopf bis Fuß« ansehen.



LINQ *sieht* vielleicht aus wie SQL, *funktioniert* aber nicht so.

Wenn Sie schon viel mit SQL gearbeitet haben, kann es verführerisch sein, den ganzen LINQ-Kram als intuitiv und trivial abzutun – und damit wären Sie nicht allein: Viele Entwickler machen diesen Fehler. Es stimmt, dass LINQ die Schlüsselwörter `select`, `from`, `where`, `ascending` und `join` verwendet, die aus SQL entlehnt sind. Aber LINQ unterscheidet sich stark von SQL. Wenn Sie versuchen, mit LINQ so zu arbeiten, wie Sie mit SQL arbeiten, führt das zu Code, der **nicht macht, was Sie erwarten**.

Ein wichtiger Unterschied zwischen beiden ist, dass SQL auf ***Tabellen*** operiert, die ganz anders sind als ***Auflistungen***. Führen Sie auf einer Tabelle eine SQL-SELECT-Abfrage aus, können Sie sicher sein, dass diese Tabelle dabei nicht verändert wird. SQL bietet die unterschiedlichsten eingebauten Mechanismen für die Datensicherheit, auf die Sie sich verlassen können. Außerdem sind SQL-Abfragen Gruppenoperationen: Das bedeutet, dass Sie die Zeilen in einer Tabelle nicht in einer vorhersehbaren Abfolge bearbeiten. Eine Auflistung hingegen kann *alles* speichern – Werte, Structs, Objekte, alles – und Auflistungen haben eine festgelegte Ordnung. (Die Zeilen einer Tabelle stehen in keiner bestimmten Reihenfolge, bis Sie eine SQL-Abfrage ausführen lassen, die sie ordnet. Die Elemente in einer List hingegen stehen in einer Reihenfolge.) Und mit LINQ können Sie alle Operationen durchführen, die von den Elementen in der Auflistung unterstützt werden – es kann auf den Objekten in einer Auflistung sogar Methoden aufrufen. Außerdem durchläuft LINQ die Auflistung, was bedeutet, dass es seine Operationen in einer bestimmten Reihenfolge durchführt. Vielleicht erscheint das alles nicht wichtig, aber wenn Sie mit SQL vertraut sind und erwarten, dass sich Ihre LINQ-Abfragen wie SQL verhalten, kann es passieren, dass Ihre LINQ-Abfragen Sie überraschen werden.

Es gibt noch eine Menge weiterer Differenzen zwischen LINQ und SQL, aber diese müssen Sie nicht verstanden haben, um erfolgreich mit LINQ zu arbeiten. Gehen Sie es einfach vorbehaltlos an und erwarten Sie nicht, dass es wie SQL funktioniert.

LINQ ist vielseitig

Mit LINQ können Sie nicht nur ein paar Elemente aus einer Auflistung herausziehen. Sie können die Elemente auch verändern, bevor Sie sie zurückliefern. Und nachdem Sie eine Gruppe von Ergebnisauflistungen generiert haben, gibt LINQ Ihnen einen Haufen Methoden, mit denen Sie arbeiten können. Alles in allem liefert Ihnen LINQ die Werkzeuge, die Sie zur Verwaltung Ihrer Daten benötigen.

Alle Elemente verändern, die von der Abfrage geliefert werden

Dieser Code hängt an das Ende jedes Strings im Array einen String an. Das Array selbst ändert er nicht – er **erzeugt eine neue Auflistung** mit den veränderten Strings.

```
string[] sandwiches = { "Schinken und Käse", "Salami mit Mayo",
                        "Pute und Mangold", "Hähnchenbrust" };
var graubrotSandwiches =
    from sandwich in sandwiches
    select sandwich + " auf Graubrot";

foreach (var sandwich in graubrotSandwiches)
    Console.WriteLine(sandwich);
```

Das hängt » auf Graubrot« an jedes Element im Ergebnis der Abfrage an.

Diese Änderung erfolgt in den von der Abfrage zurückgelieferten Ergebnissen ... nicht an den Elementen in der ursprünglichen Auflistung oder Datenbank.

Beachten Sie, dass an alle zurückgelieferten Elemente » auf Graubrot« angehängt wurde.

Ausgabe:

```
Schinken und Käse auf Graubrot
Salami mit Mayo auf Graubrot
Pute und Mangold auf Graubrot
Hähnchenbrust auf Graubrot
```

Auf Auflistungen Berechnungen durchführen

Erinnern Sie sich, dass wir gesagt hatten, dass LINQ Ihren Auflistungen neue Methoden hinzufügt (und Objekte für den Datenbankzugriff) ... einige davon sind auch für sich sehr mächtig und erfordern nicht einmal eine Abfrage:

```
List<int> listMitZahlen = new List<int>();
int länge = random.Next(50, 150);
for (int i = 0; i < länge; i++)
    listMitZahlen.Add(random.Next(100));

Console.WriteLine("Es gibt {0} Zahlen",
                  listMitZahlen.Count());
Console.WriteLine("Die kleinste ist {0}",
                  listMitZahlen.Min());
Console.WriteLine("Die größte ist {0}",
                  listMitZahlen.Max());
Console.WriteLine("Ihre Summe ist {0}",
                  listMitZahlen.Sum());
Console.WriteLine("Ihr Durchschnitt ist {0:F2}",
                  listMitZahlen.Average());
```

Wenn Sie C# 3.0 verwenden (das Teil von Visual Studio 2008 ist), ist jede Auflistung, die Sie erstellen, mit LINQ-Fertigkeiten ausgestattet.

Keine dieser Methoden ist Teil der .NET-Auflistungsklassen ... sie werden alle von LINQ definiert.

Es sind alles Erweiterungsmethoden, die in der Enumerable-Klasse im Namensraum System.Linq definiert werden.



LINQ-Abfragen, werden erst ausgeführt, wenn Sie auf ihr Ergebnis zugreifen!

Das nennt man »verzögerte Auswertung« – die LINQ-Abfrage tut nichts, bis eine Anweisung ausgeführt wird, die ihr Ergebnis verwendet. Deswegen ist ToList() so wichtig: Es weist LINQ an, die Abfrage sofort auszuwerten.

Ergebnisse in einer neuen Auflistung speichern

Manchmal möchten Sie die Ergebnisse einer LINQ-Abfrage aufbewahren. Dazu können Sie `ToList()` verwenden:

```
var unter50sortiert =
  from number in listMitZahlen
  where number < 50
  orderby number descending
  select number;

List<int> neueList = unter50sortiert.ToList();
```

Diesmal sortieren wir eine List mit Zahlen absteigend.

ToList() wandelt einen LINQ-var in eine object-List um, damit Sie das Ergebnis speichern können. Es gibt auch ToArray() und ToDictionary(), die die entsprechenden Auflistungstypen liefern.

Mit der Methode `Take()` können Sie auch nur eine Untermenge des Ergebnisses abrufen:

```
var ersteFünf = unter50sortiert.Take(5);

List<int> kurzeListe = ersteFünf.ToList();
foreach (int n in kurzeListe)
    Console.WriteLine(n);
```

Take() nimmt die angegebene Zahl von Elementen vom Anfang der Ergebnisse. Stecken Sie diese in einen anderen var, können Sie sie in eine List umwandeln.

Microsofts offizielle 101 LINQ-Beispiele

LINQ kann noch eine Menge mehr. Netterweise bietet Microsoft eine Referenz, die Ihnen weiterhelfen kann.

`http://msdn2.microsoft.com/en-us/vcsharp/aa336746.aspx`

Es gibt keine Dummen Fragen

F: Das sind eine Menge neuer Schlüsselwörter – from, where, orderby, select ... das ist wie eine ganz andere Sprache. Warum sieht es so anders aus als der Rest von C#?

A: Weil es einem anderen Zweck dient. Die C#-Syntax wurde im Wesentlichen zur Ausführung kleiner Operationen und Berechnungen gestaltet. Sie können eine Schleife starten, eine Variable setzen, eine mathematische Operation durchführen oder eine Methode aufrufen ... das sind alles separate Operationen.

LINQ-Abfragen sehen anders aus, weil eine einzelne LINQ-Abfrage in der Regel auf einmal eine Menge Dinge tut. Sehen wir uns eine einfache Abfrage mal genauer an:

```
var kleiner10 =
 from zahl in zahlArray
 where zahl < 10
 select zahl;
```

Das sieht sehr einfach aus – viel steckt da nicht drin, stimmt's? Aber eigentlich ist das ziemlich komplexer Code. Überlegen Sie, was alles passieren muss, damit das Programm in `zahlArray` tatsächlich alle Zahlen kleiner 10 auswählt. Erst muss es das gesamte Array durchlaufen und alle Zahlen mit 10 vergleichen. Dann müssen alle passenden Zahlen gesammelt werden, damit Ihr Code sie nutzen kann. Und deswegen sieht LINQ etwas seltsam aus: weil C# eine Menge Verhalten auf kleinem Raum zusammenpacken muss.

Mit LINQ können Sie komplexe Aufgaben mit wenig Code erledigen.

Punkt für Punkt

- Mit **`from`** geben Sie die abzufragende Auflistung an. Darauf folgen der Name einer Variablen, dann **`in`** und dann der Name der Auflistung (`from wert in werte`).
- **`where`** folgt normalerweise der `from`-Klausel. Dort verwenden Sie eine gewöhnliche C#-Bedingung, um LINQ zu sagen, welche Elemente aus der Auflistung herausgezogen werden sollen (`where wert < 10`).
- Mit **`orderby`** können Sie die Ergebnisse sortieren. Darauf folgen die Sortierkriterien und optional **`descending`**, um die Sortierung umzukehren (`orderby wert descending`).
- Mit **`select`** geben Sie an, was in das Ergebnis kommen soll (`select wert`).
- Mit **`Take`** können Sie die ersten Elemente aus dem Ergebnis herausziehen (`ergebnis.Take(10)`). LINQ gibt Ihnen weitere Methoden für Auflistungen: `Min()`, `Max()`, `Sum()` und `Average()`.
- Mit `select` können Sie alles auswählen – Sie sind nicht auf den Namen beschränkt, den Sie in der `from`-Klausel erzeugt haben. Ein Beispiel: Zieht LINQ eine Menge mit Preisen aus einem `int`-Array und gibt ihnen in der `from`-Klausel den Namen `wert`, können Sie folgendermaßen eine Liste mit Preis-Strings zurückliefern: `select String.Format("{0:c}", wert().)`

Das ist genau wie das {0:x}, das Sie in Kapitel 9 im Hex-Dumper verwendet haben. Es gibt auch {0:d} und {0:D} für kurze und lange Datumswerte und {0:P} oder {0:Pn} für Prozentwerte (mit n Nachkommastellen).

Es gibt keine Dummen Fragen

F: Wie funktioniert die `from`-Klausel?

A: Sie hat Ähnlichkeit mit der ersten Zeile einer `foreach`-Schleife. LINQ-Abfragen kann man sich schlechter vorstellen, weil sie anders als die meisten üblichen C#-Anweisungen nicht nur eine einzige Operation durchführen.

Eine LINQ-Abfrage macht die gleiche Sache für eine Auflistung immer wieder. Die `from`-Klausel macht zwei Dinge: Sie sagt LINQ, auf welcher Auflistung die Abfrage operieren soll, und gibt einen Namen an, der für die einzelnen Elemente der abzufragenden Auflistung verwendet wird.

Wie die `from`-Klausel den neuen Namen für die Elemente in der Auflistung erzeugt, hat große Ähnlichkeit damit, wie es eine `foreach`-Schleife macht. Hier ist die erste Zeile einer `foreach`-Schleife:

```
foreach (int i in werte)
```

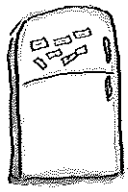
Diese `foreach`-Schleife erzeugt eine temporäre Variable `i`, der sie nacheinander die Elemente in der Auflistung `werte` zuweist. Sehen Sie sich jetzt die `from`-Klausel einer LINQ-Abfrage auf der gleichen Auflistung an:

```
from i in werte
```

Diese Klausel macht so ziemlich das Gleiche. Sie erzeugt eine temporäre Variable namens `i` und weist ihr nacheinander die Elemente in der Auflistung `werte` zu. Die `foreach`-Schleife führt für jedes Element in der Auflistung den gleichen Codeblock aus, während die LINQ-Abfrage in der `where`-Klausel auf alle Elemente in der Auflistung die gleichen Kriterien anwendet, um zu ermitteln, ob es in das Ergebnis eingeschlossen werden soll oder nicht.

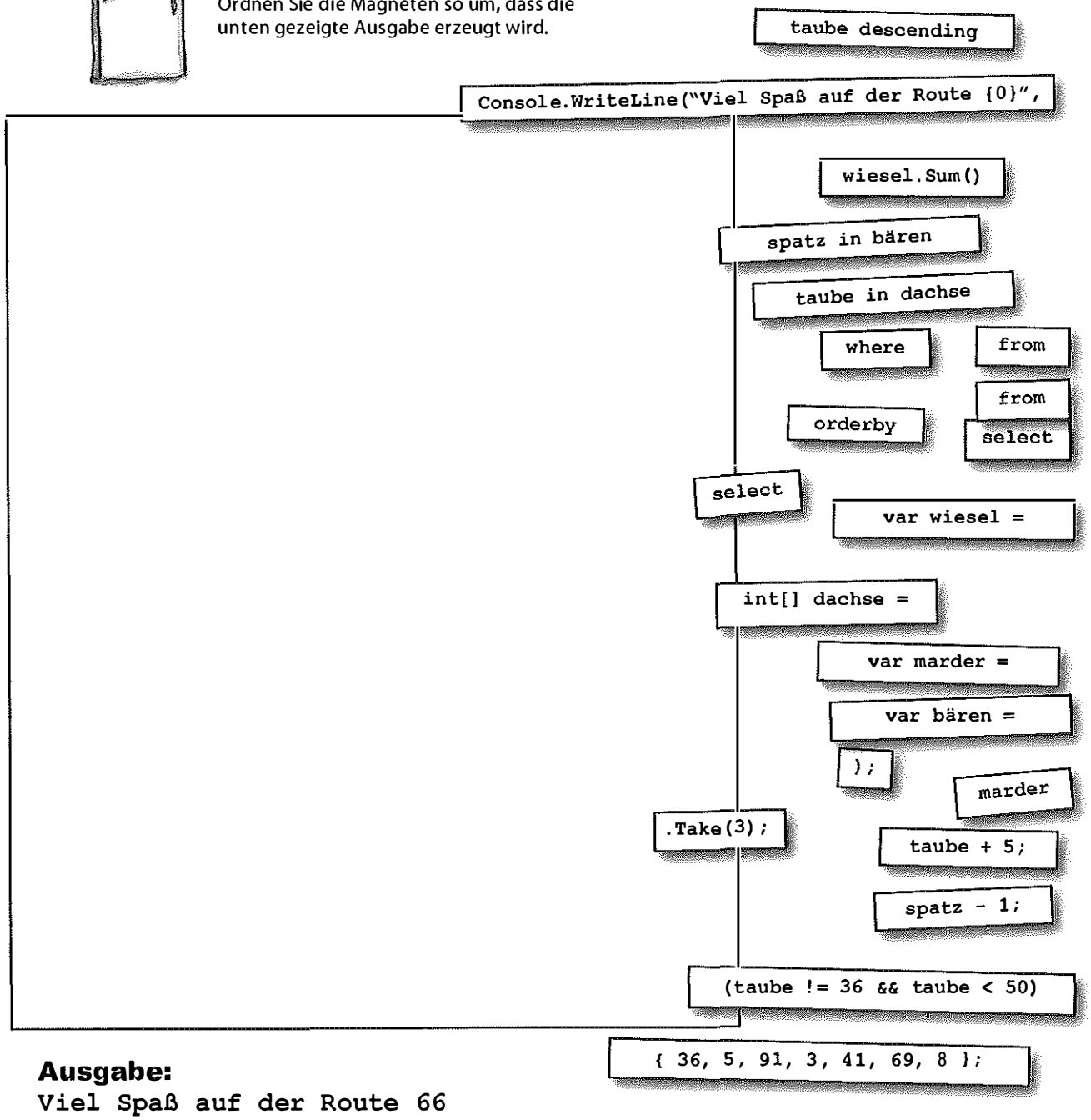
F: Wie entscheidet LINQ, was ins Ergebnis übernommen wird?

A: Dazu dient die `select`-Klausel. Jede LINQ-Abfrage liefert eine Auflistung, und alle Elemente darin haben den gleichen Typ. Sie sagt LINQ genau, was diese Auflistung enthalten soll. Fragen Sie ein Array oder eine List ab, die nur einen einzigen Typ enthält (wie ein Array mit `int`s oder eine `List<string>`) ist klar, was in die `select`-Klausel kommt. Aber was ist, wenn Sie aus einer Liste mit `Comic`-Objekten auswählen? Sollen Sie wie Tim jeweils sämtliche Objekte auswählen? Oder sollten Sie die letzte Zeile der Abfrage so ändern, dass nur `comic.Name` gewählt wird, damit eine Liste mit Strings geliefert wird? Oder Sie können `comic.Folge` wählen und eine Liste mit `int`s liefern lassen.



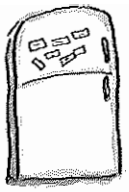
LINQ-Magneten

Ordnen Sie die Magneten so um, dass die unten gezeigte Ausgabe erzeugt wird.



Ausgabe:

```
Viel Spaß auf der Route 66
```



LINQ-Magneten, Lösung

Ordnen Sie die Magneten so um, dass die unten gezeigte Ausgabe erzeugt wird.

LINQ beginnt mit einer Auflistung oder einem Array – hier einem Array mit int-Werten.

```
int[] dachse = { 36, 5, 91, 3, 41, 69, 8 };
```

>>from taube in dachse<< ist in einem Puzzle brauchbar, als LINQ-Abfrage aber unlesbar. >>from dachs in dachse<< ist lesbarer.

```
var marder =
    from taube in dachse
    where (taube != 36 && taube < 50)
    orderby taube descending
    select taube + 5;
```

Nach dieser Anweisung enthält marder vier Zahlen: 46, 13, 10 und 8.

Diese LINQ-Anweisung zieht alle Zahlen, die kleiner 50 und nicht gleich 36 sind, aus dem Array heraus, addiert jeder 5 hinzu, sortiert sie dann absteigend und steckt sie in eine neue Auflistung namens marder.

```
var bären =
    marder.Take(3);
```

Nach dieser Anweisung enthält bären drei Zahlen: 46, 13 und 10.

Hier holen wir die ersten drei Zahlen aus marder und stecken sie in eine neue Auflistung namens bären.

```
var wiesel =
    from spatz in bären
    select spatz - 1;
```

Nach dieser Anweisung enthält wiesel drei Zahlen: 45, 12 und 9.

Diese Anweisung zieht einfach von allen Zahlen in bären 1 ab und steckt sie in wiesel.

```
Console.WriteLine("Viel Spaß auf der Route {0}",
    wiesel.Sum() );
```

Die Zahlen in wiesel ergeben zusammen 66.

45 + 12 + 9 = 66

Ausgabe:

```
Viel Spaß auf der Route 66
```

LINQ kann Ergebnisse zu Gruppen zusammenfassen

Sie wissen bereits, dass Sie LINQ einsetzen können, um Ihre Ergebnisse zu gruppieren. Genau das haben wir schließlich im Bienenstock-Simulator gemacht. Sehen wir uns diese Abfrage genauer an, um herauszufinden, wie sie funktioniert.

```
var bienenGruppen =

    from biene in welt.Bienen

    group biene by biene.AktuellerStatus

    into bienenGruppe

    orderby bienenGruppe.Key

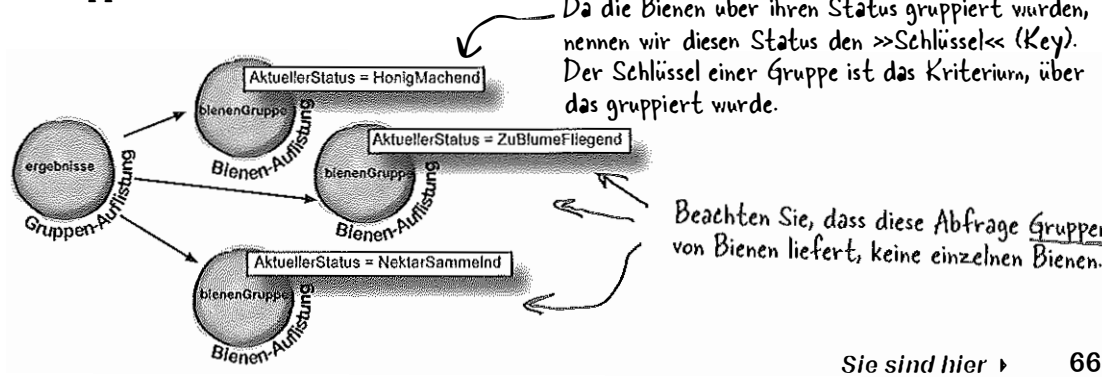
    select bienenGruppe;
```

1 Die Abfrage beginnt wie alle anderen Abfragen, die Sie bisher gesehen haben – indem sie Biene-Objekte aus der `List<Biene>`-Auflistung `welt.Bienen` zieht.

2 Die nächste Zeile der Abfrage enthält ein neues Schlüsselwort: **`group`**. Das sagt, dass die Abfrage ***Gruppen*** von Bienen liefern soll. Dies bedeutet, dass nicht eine einzige Auflistung geliefert wird, sondern eine **Auflistung von Auflistungen**. **`group biene by biene.AktuellerStatus`** sagt LINQ, dass eine Gruppe für jeden `AktuellerStatus`-Wert bei den ausgewählten Bienen zurückgeliefert werden soll. Schließlich müssen wir LINQ einen Namen für die Gruppe geben. Das macht die nächste Zeile: **`into bienenGruppe`** sagt, dass der Name »bienenGruppe« auf die neuen Gruppen verweist.

3 Nachdem wir Gruppen haben, können wir sie manipulieren. Da wir eine Auflistung mit Gruppen liefern, können wir das Schlüsselwort `orderby` verwenden, um die Gruppen in eine Reihenfolge gemäß den AktuellerStatus-Werten zu bringen (Untätig, ZuBlumeFliegen usw.): **`orderby bienenGruppe.Key`** sagt der Abfrage, dass die Auflistung über den Gruppenschlüssel sortiert werden soll. Da wir die Bienen über ihren `AktuellerStatus`-Wert gruppiert haben, wird dieser als Schlüssel verwendet.

4 Jetzt müssen wir nur noch das Schlüsselwort `select` einsetzen, um anzuzeigen, was von der Abfrage zurückgeliefert wird. Da wir Gruppen zurückliefern, wählen wir den Gruppennamen: **`select bienenGruppe;`**



Tims Werte gruppieren

Tim findet eine Menge billiger Comics, einige Comic-Bücher im mittleren Preissegment und ein paar teurere und möchte wissen, welche Optionen er hat, bevor er sich entscheidet, welche Comics er kauft. Die Preise hat er auf Gregors Liste gefunden und mit `PreiseAufbauen()` in ein `Dictionary<int, int>` gesteckt – er nutzt LINQ, um sie in drei Gruppen einzuteilen: eine für billige Comics unter 100 €, eine für Comics zwischen 100 und 1000 € und eine für die teuren, die mehr als 1000 € kosten. Wir erstellen ein `Preisbereich`-Enum, das wir als Schlüssel für die Gruppen verwenden werden, und die Methode PreisAuswerten(), die den Preis auswertet und einen `Preisbereich`-Wert liefert.

1 Jede Gruppe braucht einen Schlüssel - dazu nutzen wir ein Enum.

Der Schlüssel ist das Ding, das alle Mitglieder einer Gruppe gemeinsam haben. Der Schlüssel kann alles sein: ein String, eine Zahl oder auch eine Objektreferenz. Wir werden uns die Preise ansehen, die Tim auf Gregors Liste fand. Jede Gruppe, die die Abfrage liefert, ist eine Auflistung mit Nummern von Folgen und ihr Schlüssel ein `Preisbereich`-Wert. Die Methode `PreisAuswerten()` erwartet als Parameter einen Preis und liefert einen `Preisbereich`-Wert:

```
public enum Preisbereich { billige, mittlere, teure }

public Preisbereich PreisAuswerten(int preis) {
    if (preis < 100) return Preisbereich.billige;
    else if (preis < 1000) return Preisbereich.mittlere;
    else return Preisbereich.teure;
}
```

2 Jetzt können wir die Comics über ihre Preiskategorie gruppieren.

Die LINQ-Abfrage liefert eine **Auflistung mit Auflistungen**. Jede Auflistung im Ergebnis hat eine Key-Eigenschaft, die dem `Preisbereich` entspricht, der von `PreisAuswerten()` zurückgeliefert wird. Sehen Sie sich die `group by`-Klausel genau an – wir ziehen Paare aus dem Dictionary und verwenden für diese den Namen `paar`: `paar.Key` ist die Nummer der Folge, und `paar.Value` ist der Preis von Gregors Liste. Fügen wir `group paar.Key` hinzu, erstellt LINQ Gruppen mit Nummern von Folgen und bündelt diese Gruppen dann auf Basis der Preiskategorie, die von `PreisAuswerten()` geliefert wird:

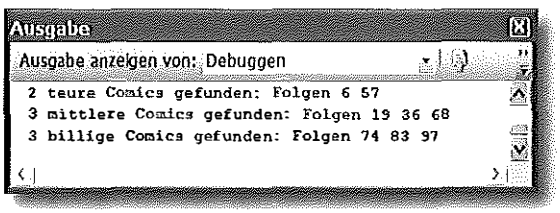
```
Dictionary<int, int> werte = PreiseAufbauen();

var preisGruppen =
    from paar in werte
    group paar.Key by PreisAuswerten(paar.Value)
        into preisGruppe
        orderby preisGruppe.Key descending
        select preisGruppe;

foreach (var gruppe in preisGruppen) {
    Console.Write("{0} {1} Comics gefunden: Folgen", gruppe.Count(), gruppe.Key);
    foreach (var preis in gruppe)
        Console.Write(preis.ToString() + " ");
    Console.WriteLine();
}
```

Die Abfrage ermittelt, in welche Gruppe ein bestimmter Preis fällt, indem sie den Preis an PreisAuswerten() sendet. Diese Methode liefert einen Preisbereich-Wert, der als Schlüssel der Gruppe verwendet wird.

Jede Gruppe ist eine Auflistung, die in der inneren foreach-Schleife gelesen wird, um die Preise herauszuziehen.



Pool-Puzzle



Sie haben die **Aufgabe**, die leeren Zeilen im Code mit den Codeschnipseln aus dem Pool zu füllen. Einzelne Schnipsel können mehrfach verwendet werden, und Sie werden nicht alle Schnipsel benötigen. Das **Ziel** ist es, eine Klasse zu erstellen, die die gezeigte ***Ausgabe*** erzeugt.

Horses enjoy eating carrots, but they love eating apples.

```
public class Vers {
    public string[] Wörter;
    public int Value;
    public Vers(string[] Wörter, int Wert) {
        this.Wörter = Wörter; this.Wert = Wert;
}   }
```

Tipp: LINQ sortiert Strings in alphabetischer Reihenfolge.

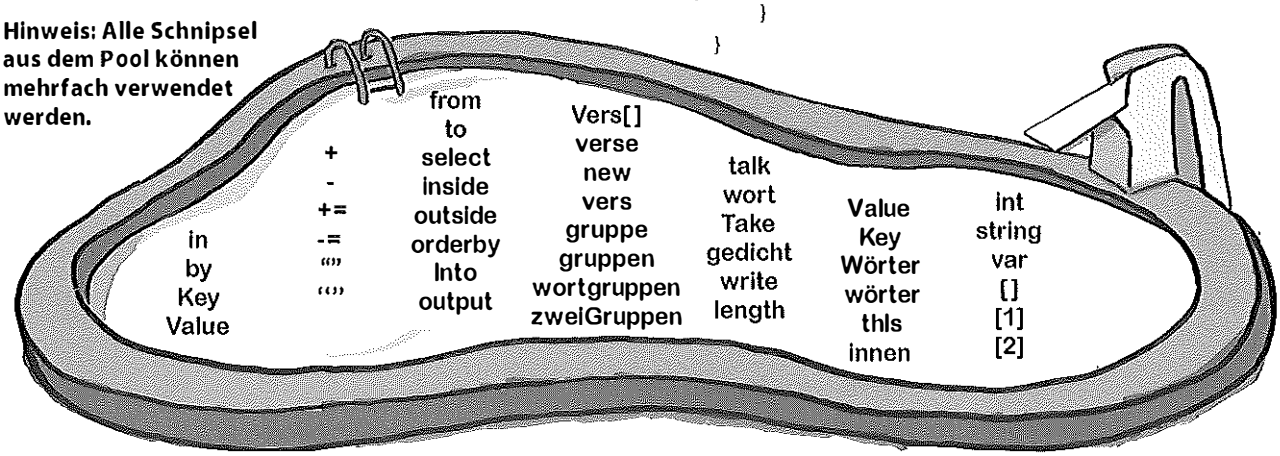
```
Vers[] verse = {
  new Vers( new string [] { "eating", "carrots,",
         "but", "enjoy", "Horses" } , 1),
  new Vers( new string [] { "zebras?", "hay",
        "Cows", "bridge.", "bolted" } , 2),
  new Vers( new string [] { "fork", "dogs!",
        "Engine", "and" }, 3 ) ,
  new Vers( new string [] { "love", "they",
        "apples.", "eating" }, 2 ) ,
  new Vers( new string [] { "whistled.", "Bump"
        }, 1 ) };
```

```
var _______ =
    from _______ in _______
    _______ vers by vers._______
    into wortgruppen
    orderby _________._______
    select _________;

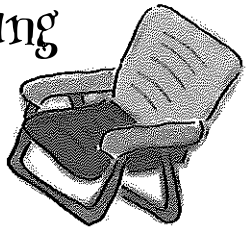
____ __________ = wörter._______(2);

foreach (var gruppe in zweiGruppen)
{
   int i = 0;
   foreach (_______ innen in _______) {
      i++;
      if (i == _______.Key) {
         var gedicht =
            _______ wort in _______._______
            _________ wort descending
            _______ wort + _____;
         foreach (var wort in _______)
            Console.Write(wort);
      }
   }
}
```

Hinweis: Alle Schnipsel aus dem Pool können mehrfach verwendet werden.



Pool-Puzzle, Lösung



```
public class Vers {
    public string[] Wörter;
    public int Wert;
    public Vers(string[] Wörter, int Wert) {
        this.Wörter = Wörter; this.Wert = Wert;
    }
}

Vers[] verse = {
  new Vers( new string [] { "eating", "carrots,", "but", "enjoy", "Horses" } , 1),
  new Vers( new string [] { "zebras?", "hay", "Cows", "bridge.", "bolted" } , 2),
  new Vers( new string [] { "fork", "dogs!", "Engine", "and" }, 3 ) ,
  new Vers( new string [] { "love", "they", "apples.", "eating" }, 2 ) ,
  new Vers( new string [] { "whistled.", "Bump" }, 1 )
};

var wörter =
    from vers in verse
    group vers by vers.Wert
    into wortgruppen
    orderby wortgruppen.Key
    select wortgruppen;
```

Die erste LINQ-Abfrage teilt die Vers-Objekte im verse[]-Array anhand ihres Werts und in aufsteigender Reihenfolge nach dem Wert in Gruppen ein.

```
var zweiGruppen = wörter.Take(2);
```

Die ersten beiden Gruppen sind die Verse mit den Werten 1 und 2.

```
foreach (var gruppen in zweiGruppen)
{
    int i = 0;
    foreach (var innen in gruppe) {
        i++;
        if (i == gruppe.Key) {
            var gedicht =
                from wort in innen.Wörter
                orderby wort descending
                select wort + " ";
            foreach (var wort in gedicht)
                Console.Write(wort);
        }
    }
}
```

Diese Schleife führt eine LINQ-Abfrage auf dem ersten Vers-Objekt in der ersten Gruppe und dem zweiten Vers-Objekt in der zweiten Gruppe aus.

Haben Sie erkannt, dass die beiden Satzfragmente »Horses enjoy eating carrots, but« und »they love eating apples« in absteigender alphabetischer Reihenfolge stehen?

Ausgabe: **Horses enjoy eating carrots, but they love eating apples.**

Zwei Auflistungen in einer Abfrage mit join kombinieren

Tim hat eine ganze Sammlung von Comics erworben, möchte diese nun mit den Preisen vergleichen, die er auf Gregors Liste gefunden hat, und prüfen, ob sich die Preise zum Guten oder Schlechten gewandelt haben. Er hat seine Käufe in einer `Erwerb`-Klasse mit zwei automatischen Eigenschaften festgehalten, `Folge` und `Preis`. Und er hat eine `List<Erwerb>` namens `erwerbe`, die alle Comics enthält, die er gekauft hat. Aber jetzt muss er seine Erwerbe mit den Preisen auf Gregors Liste vergleichen. Wie soll er das tun?

LINQ kommt ihm zu Hilfe! Mit seinem `join`-Schlüsselwort können Sie in einer Abfrage **Daten aus zwei Auflistungen kombinieren**. Das macht es, indem es Elemente in der ersten Auflistung mit Elementen in der zweiten Auflistung vergleicht. (LINQ arbeitet hier sehr effizient – alle Elementpaare vergleicht es nur, wenn es das wirklich muss.) Das führt zu einem Ergebnis, das alle passenden Paare kombiniert.

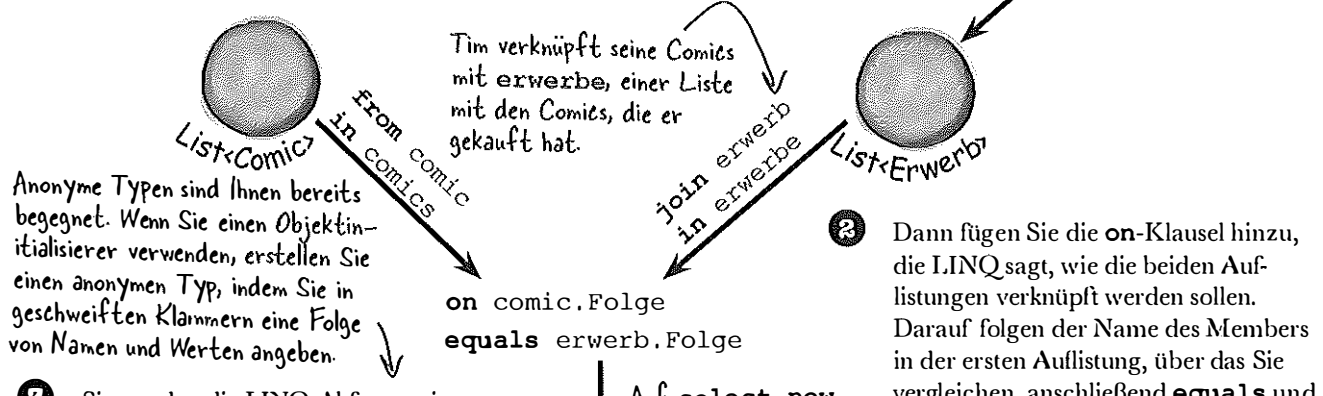
Tim hat seine Daten in einer Auflistung mit Erwerb-Objekten namens `erwerbe`.

```
public class Erwerb {
  public int Folge
             { get; set; }
  public int Preis
             { get; set; }
}
```

1 Beginnen Sie Ihre Abfrage mit der üblichen `from`-Klausel. Aber statt auf sie die Kriterien folgen zu lassen, die verwendet werden, um zu bestimmen, was ins Ergebnis kommt, fügen Sie Folgendes an:

```
join name in Auflistung
```

Die `join`-Klausel sagt LINQ, dass es beide Auflistungen durchlaufen soll, um Paare mit Elementen aus beiden Auflistungen zu bilden. *name* wird dem Element zugewiesen, das in jedem Durchlauf aus der verknüpften Auflistung herausgezogen wird.



2 Dann fügen Sie die **on**-Klausel hinzu, die LINQ sagt, wie die beiden Auflistungen verknüpft werden sollen. Darauf folgen der Name des Members in der ersten Auflistung, über das Sie vergleichen, anschließend **equals** und dann der Name des Members in der zweiten Auflistung, mit dem verglichen werden soll.

3 Sie werden die LINQ-Abfrage wie gewöhnlich mit `where`- und `orderby`-Klauseln fortsetzen. Sie könnten sie mit einer gewöhnlichen `select`-Klausel abschließen, aber meist möchten Sie ein Ergebnis zurückliefern, das ein paar Daten aus der einen und ein paar aus der anderen Auflistung liefert. Dann nutzen Sie **`select new`**, um mithilfe eines **anonymen Typs** eine angepasste Ergebnismenge zu liefern.

Tim hat eine Menge Kohle gespart

Es scheint, als verhandele Tim hart. Er erzeugte eine List mit Erwerb-Objekten, die seine Erwerbe enthielt, und verglich diese mit den Preisen, die er auf Gregors Liste fand.

1 Erst erzeugte Tim die Auflistung für den Join.

Die erste Auflistung hatte Tim schon – er nutzte einfach seine alte `KatalogGenerieren()`-Methode. Er musste also nur noch eine `ErwerbeSuchen()`-Methode schreiben, um eine Liste mit `Erwerb`-Objekten aufzubauen.

```
public List<Erwerb> ErwerbeSuchen() {
    List<Erwerb> erwerbe = new List<Erwerb>() {
        new Erwerb() { Folge = 68, Preis = 225 },
        new Erwerb() { Folge = 19, Preis = 375 },
        new Erwerb() { Folge = 6, Preis = 3600 },
        new Erwerb() { Folge = 57, Preis = 13215 },
        new Erwerb() { Folge = 36, Preis = 660 },
    };
    return erwerbe;
}
```

Für Folge 57 zahlte Tim 13.215 €.

3 Jetzt konnte er den Join durchführen!

Sie haben bereits alle Teile seiner Abfrage gesehen ... hier sind sie zusammengesetzt in einem Stück.

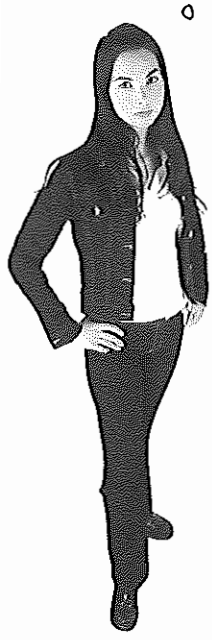
```
List<Comic> comics = KatalogGenerieren();
Dictionary<int, int> werte = PreiseAufbauen();
List<Erwerb> erwerbe = ErwerbeSuchen();
var ergebnisse =
    from comic in comics
    join erwerb in erwerbe
    on comic.Folge equals erwerb.Folge
    orderby comic.Folge ascending
    select new { comic.Name, comic.Folge, erwerb.Preis };
int listenWert = 0;
int ausgabenGesamt = 0;
foreach (var ergebnis in ergebnisse) {
    listenWert += werte[ergebnis.Folge];
    ausgabenGesamt += ergebnis.Preis;
    Console.WriteLine("Folge {0} ({1}) erworben für {2:c}",
            ergebnis.Folge, ergebnis.Name, ergebnis.Preis);
}
Console.WriteLine("{0:c} ausgegeben für Comics im Wert von {1:c}",
            ausgabenGesamt, listenWert);
```

Da Tim eine `join`-Klausel verwendet, vergleicht LINQ alle Elemente in der Auflistung `comics` mit allen Elementen in `erwerbe`, um zu prüfen, wo `comic.Folge` gleich `erwerb.Folge` ist.

Die Klausel `select new` erzeugt eine Ergebnismenge mit Name und Folge des `comic`-Elements und dem Preis des `erwerb`-Elements..

Tim ist sehr glücklich, dass er LINQ beherrscht, weil es ihm zeigt, wie hart er verhandelt hat!

```
Ausgabe
Ausgabe anzeigen von: Debuggen
Folge 6 (Johnny America vs. the Pinko) erworben für 3.600,00 €
Folge 19 (Rock and Roll (limitierte Auflage)) erworben für 375,00 €
Folge 36 (Woman's Work) erworben für 660,00 €
Folge 57 (Hippie Madness (Fehldruck)) erworben für 13.215,00 €
Folge 68 (Revenge of the New Wave Freak (beschädigt)) erworben für 225,00 €
18.075,00 € ausgegeben für Comics im Wert von 18.525,00 €
```



Gut. Jetzt weiß ich also, dass Tim mit seinen Comics herumgespielt hat, indem er LINQ zur Abfrage seiner Auflistungen verwendet hat ... aber was ist mit dem Problem bei der Sternback-Werbung? Ich weiß immer noch nicht, wie LINQ mit Datenbanken arbeitet.

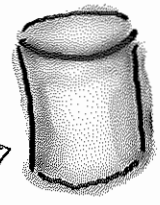
Obwohl LINQ to SQL im Hintergrund ganz anders arbeitet, sieht der Code dafür anderen LINQ-Abfragen sehr ähnlich.

Für Datenbanken nutzt LINQ die gleiche Syntax wie für Auflistungen.

Sie haben schon in Kapitel 1 gesehen, wie leicht .NET die Arbeit mit Datenbanken macht. Die IDE bietet Ihnen wirklich bequeme Mittel, Verbindungen mit Datenbanken herzustellen, ihnen Tabellen hinzuzufügen und die Daten in diesen Tabellen mit Ihren Formularen zu verknüpfen.

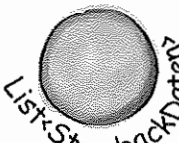
Jetzt können Sie die Datenbank, mit der Sie sich bereits verbunden haben, mit LINQ abfragen. Und nicht nur das: Mit LINQ können Sie die Daten aus Ihrer Datenbank nahtlos mit den Daten in Ihren Objekten verknüpfen.

Und Sie können dazu sogar genau die gleiche Abfragesyntax verwenden ... Sie müssen sich nur Zugriff auf Ihre Datenbank verschaffen, damit Sie darauf eine LINQ-Abfrage ausführen können.



KontaktDB-Datenbank

Die Kontaktdatenbank der Papierfabrik Objekthausen enthält die Adressen.



Das sind die Daten mit den Sternback-Stammkunden.

LINQ

Mit LINQ können wir Daten von mehreren Datenquellen vergleichen und kombinieren sowie eine Auflistung mit Ergebnissen erzeugen.



LINQ mit einer SQL-Datenbank verbinden

LINQ operiert auf Auflistungen, die die Schnittstelle `IEnumerable` implementieren, stimmt's? Es wäre also sinnvoll, dass Sie auf Ihre SQL-Datenbank über ein Objekt zugreifen, das `IEnumerable` implementiert. Und C# macht es leicht, Ihrem Projekt eine entsprechende Klasse hinzuzufügen.

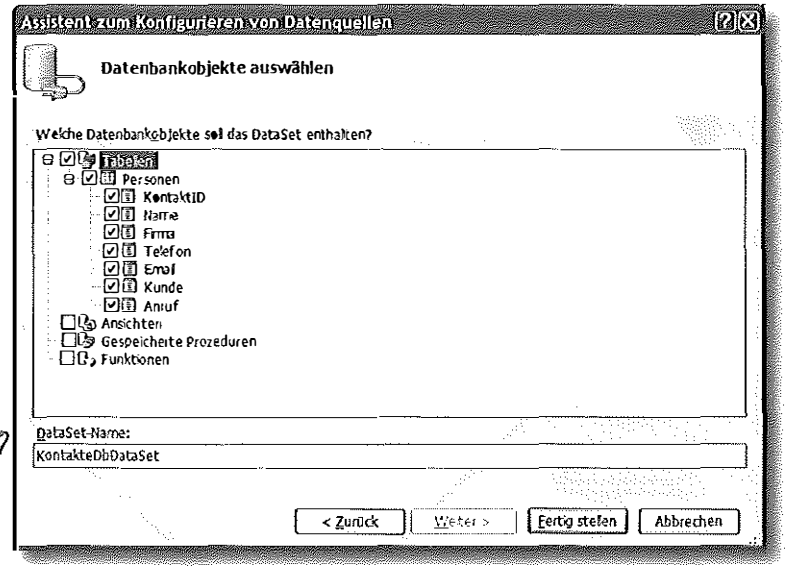
Tun Sie das!

1 Fügen Sie einem neuen Projekt die Kontaktdaten der Papierfabrik hinzu.
In Kapitel 1 haben Sie eine Kontaktdatenbank für die Papierfabrik Objekthausen erstellt und in einer Datei namens `KontaktDB.mdf` gespeichert. Beginnen Sie eine neue Windows-Anwendung, klicken Sie im Projektmappen-Explorer mit der rechten Maustaste auf Ihr Projekt, wählen Sie »Hinzufügen→Bestehendes Element...« und fügen Sie die Datenbank hinzu. Achten Sie darauf, dass Sie in der Filterliste »Dateityp« »Datendateien« auswählen.

2 Die IDE öffnet den Assistenten zum Konfigurieren von Datenquellen.
Wählen Sie die Tabelle Personen, indem Sie die entsprechende Checkbox ankreuzen. Klicken Sie auf Fertig – der Assistent erzeugt ein Dataset namens KontaktDBDataSet und fügt es automatisch Ihrem Projekt hinzu.

Genau diesen Assistenten haben Sie auch in Kapitel 1 verwendet, als Sie die Datenbank erstellten. Er fügt Ihrem Projekt Klassen hinzu, über die Sie direkt auf die Datenbank zugreifen können.

Nehmen Sie sich eine Minute Zeit und blättern Sie zu Kapitel 1 zurück, um sich anzuschauen, wie Sie die Datenbank erstellt haben.



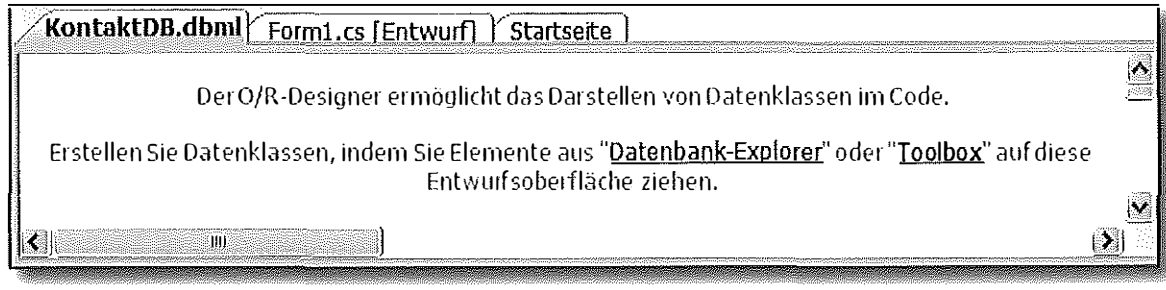
3 Fügen Sie Ihrem Projekt die LINQ to SQL-Klassen hinzu.
Klicken Sie im Projektmappen-Explorer rechts auf das Projekt und wählen Sie »Hinzufügen→Neues Element...«. Es wird die vertraute Liste mit Symbolen angezeigt – wählen Sie **LINQ to SQL-Klassen** und verwenden Sie dafür den Namen `KontaktDB.dbml`.



Fügen Sie Ihrem Projekt die LINQ to SQL-Klassen hinzu, ergänzt die IDE Ihr Projekt automatisch um eine Auflistungsklasse, die Sie mit LINQ verwenden können.

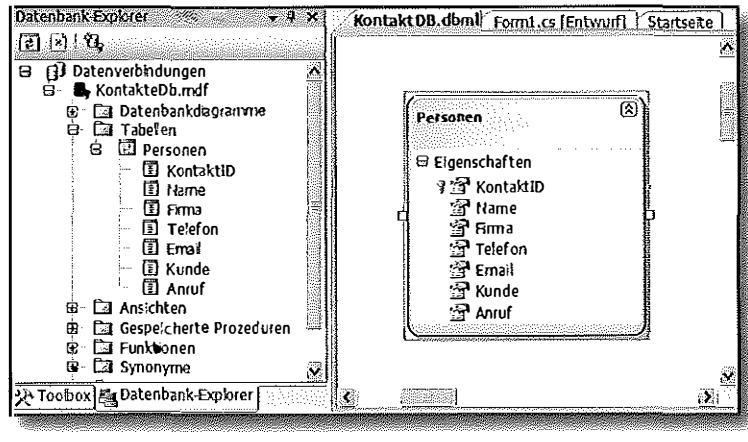
4 Die IDE bietet einen Designer zum Aufbau Ihrer SQL-Auflistung.

Sobald Sie Ihrem Projekt die LINQ to SQL-Klassen hinzufügen, öffnet die IDE ein Fenster namens O/R-Designer. Dieses sieht so aus:



5 Ziehen Sie die Tabelle Personen in den O/R-Designer.

Klicken Sie im Fenster O/R-Designer auf den Link Datenbank-Explorer – die IDE öffnet den Datenbank-Explorer. Expandieren Sie den Knoten Tabellen, klicken Sie auf das Symbol für die Tabelle Personen, ziehen Sie es in den O/R-Designer und speichern Sie das Projekt.



Speichern Sie das Projekt, fügt die IDE ihm automatisch eine Klasse hinzu, die als ein DataContext bezeichnet wird – eine ihrer Eigenschaften ist die mit der Datenbank verbundene Auflistung, die Sie mit LINQ abfragen können.

6 Jetzt können Sie die Daten mit LINQ-Abfragen aus der Datenbank ziehen.

Fügen Sie dem Formular einen Button hinzu – hier ist der Code dafür. Beachten Sie, wie wir **`select new`** verwendet haben, um ein angepasstes Ergebnis zu erzeugen, das nur Name und Unternehmen enthält.

```
KontaktDBDataContext kontext = new KontaktDBDataContext();

var personendaten =
    from person in kontext.Personen
    select new { person.Name, person.Firma };

foreach (var person in personendaten)
    Console.WriteLine("{0} arbeitet bei {1}", person.Name, person.Firma);
```

Sammeln Sie etwas Erfahrung mit **`select new`**. Es zieht nur die Werte der Spalten Name und Unternehmen aus der Datenbank.

Punkt für Punkt

- Die **`group`**-Klausel weist LINQ an, Ergebnisse zu gruppieren – wird sie verwendet, erstellt LINQ eine Auflistung mit jeweils einer Auflistung pro Gruppe.
- Die Elemente einer Gruppe haben einen Member gemeinsam. Das ist der **Schlüssel** der Gruppe. Nutzen Sie das Schlüsselwort **`by`**, um den Schlüssel der Gruppe anzugeben. Jede Gruppenauflistung hat den Member **`Key`**, der den Schlüssel der Gruppe angibt.
- Mit der **`join`**-Klausel können Sie zwei Auflistungen in einer einzigen Abfrage zusammenfassen. Tun Sie das, vergleicht LINQ alle Elemente der ersten Auflistung mit allen Elementen der zweiten und schließt passende Paare in die Ergebnismenge ein.
- Mit der **`on ... equals`**-Klausel sagen Join-Abfragen, wie die Elementpaare verglichen werden sollen.
- Bei einer Join-Abfrage erstreben Sie in der Regel ein Ergebnis, das einige Elemente aus der ersten Auflistung und einige aus der zweiten enthält. Mit **`select new`** können Sie angepasste Ergebnisse aus beiden erstellen.
- LINQ kann eine SQL-Datenbank über **LINQ to SQL-Klassen** abfragen. Da LINQ nur mit Auflistungen funktioniert, die `IEnumerable` implementieren, enthalten diese eine Auflistung, über die Sie auf Tabellen und Abfragen zugreifen können, als wären es Auflistungen.
- Mit dem O/R-Designer der IDE können Sie die Tabellen auswählen, auf die Sie über LINQ zugreifen möchten. Er generiert dann eine DataContext-Klasse für Ihr Projekt, die diese Tabellen abbildet. Wird sie instantiiert, können Sie sie in LINQ abfragen und über ihre Member auf SQL-Tabellen zugreifen.

Es gibt keine Dummen Fragen

F: Können Sie einen Moment Innehalten und noch mal erklären, was ein var ist?

A: Ja, klar. Das Schlüsselwort `var` löst ein verzwicktes Problem, das LINQ mitbringt. Normalerweise ist beim Aufruf einer Methode oder der Ausführung einer Anweisung immer vollkommen klar, mit welchen Typen Sie arbeiten. Liefert Ihre Methode einen `string`, können Sie ihr Ergebnis beispielsweise nur in einer `string`-Variablen speichern.

Aber so einfach ist LINQ nicht. Wenn Sie eine LINQ-Anweisung aufbauen, liefert diese üblicherweise einen Typ, *der an keiner Stelle Ihres Programms definiert ist*. Sie wissen, dass es irgendeine Art Auflistung ist. Aber welche Art Auflistung? Das wissen Sie nicht – weil die Objekte, die in diese Auflistung kommen, vollständig davon abhängig sind, was Sie in Ihre LINQ-Abfrage stecken.

Schauen Sie sich beispielsweise diese Abfrage an:

```
var amTeuersten =
    from comic in comics
    where values[comic.Folge] > 500
    select comic;
```

Was wäre, wenn Sie die letzte Zeile so änderten:

```
    select comic.Folge;
```

Das ist eine absolut zulässige LINQ-Abfrage. Statt einer Auflistung mit `Comic`-Objekten liefert sie eine Auflistung mit Werten. Das stellt für C# ein Problem dar – das sind zwei verschiedene Typen, und wir bräuchten zusätzliche Anweisungen, um diese Typen zu definieren. Stattdessen gibt uns C# das Schlüsselwort `var`, das dem Compiler Folgendes sagt: »Wir wissen, dass das ein zulässiger Typ ist. Aber welcher, können wir jetzt noch nicht sagen. Finde das also bitte selbst heraus und quäl uns nicht damit. Vielen Dank.«

Es gibt keine Dummen Fragen

F: Wie Joins funktionieren, verstehe ich noch nicht so recht.

A: Joins arbeiten mit zwei beliebigen Auflistungen. Angenommen, Sie haben eine Auflistung mit Fußballspielern namens `team` – ihre Elemente sind Objekte mit den Eigenschaften `Name`, `Position` und `Nummer`. Die Spieler, deren Trikots eine Nummer größer 10 haben, könnten wir also mit dieser Abfrage ermitteln:

```
var ergebnisse =
 from spieler in team
 where spieler.Nummer > 10
 select spieler;
```

Nun möchten wir die Trikotgrößen aller Spieler ermitteln und verfügen über eine `trikots`-Auflistung, deren Elemente die Eigenschaften `Nummer` und `Größe` haben. Das ginge mit einem Join problemlos:

```
var ergebnisse =
 from spieler in team
 where spieler.Nummer > 10
 join trikot in trikots
 on spieler.Nummer
 equals trikot.Nummer
 select trikot;
```

F: Moment, diese Abfrage liefert mir nur einen Haufen mit Trikots. Was ist, wenn ich jedem Spieler seine Trikotgröße zuordnen will und mich für die Trikotnummer überhaupt nicht interessiere?

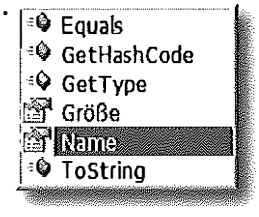
A: Dazu dient `select new`. Damit können Sie einen **anonymen Typ** aufbauen, der nur die Daten enthält, die Sie benötigen. Und Sie können dabei beliebige Member der Elemente aus den per Join verknüpften Auflistungen wählen.

Folgendermaßen können Sie den Namen des Spielers, seine Trikotgröße und sonst nichts auswählen:

```
var ergebnisse =
 from spieler in team
 where spieler.Nummer > 10
 join trikot in trikots
 on spieler.Nummer
 equals trikot.Nummer
 select new {
      spieler.Name,
      trikot.Größe
 };
```

Die IDE ist so clever, dass sie genau herausfinden kann, welche Ergebnisse von Ihrer Abfrage erzeugt werden. Erzeugen Sie eine Schleife, um die Ergebnisse zu durchlaufen, blendet die IDE ein IntelliSense-Fenster ein, sobald Sie den Namen der Variablen eingeben.

```
foreach (var r in
         ergebnisse)
   r.
```



Beachten Sie, dass die Liste Name und Größe enthält. Fügen Sie der `select new`-Klausel weitere Member hinzu, erscheinen auch diese in der Liste. Das liegt daran, dass die IDE dann einen anderen anonymen Typ mit anderen Membern erzeugt.

F: Und das funktioniert nur mit Joins?

A: Nein. `select new` können Sie in jeder LINQ-Abfrage verwenden, wenn Sie eine Ergebnisauflistung erzeugen möchten, die nur bestimmte Elemente enthält.

F: Muss ich immer die LINQ to SQL-Klassen einschließen, wenn Ich mit LINQ eine SQL-Datenbank abfrage? Was ist das überhaupt?

A: Ja, Sie müssen sie erstellen. LINQ arbeitet mit Klassen, die die Schnittstelle `IEnumerable` implementieren. Das macht eine SQL-Datenbank üblicherweise nicht, genauso wenig wie andere Schnittstellen, da sie kein Objekt ist. Wenn LINQ mit SQL – oder einer anderen Datenquelle, die Sie abfragen können – arbeiten soll, benötigen Sie ein Objekt, das mit ihr interagiert und die Schnittstelle `IEnumerable` implementiert.

Deswegen stellt Ihnen die IDE die LINQ to SQL-Klassen zur Verfügung. Fügen Sie sie Ihrem Projekt hinzu, macht sie automatisch alles, was erforderlich ist, um LINQ mit einer SQL-Datenbank zu verbinden. Sie ermöglicht Ihnen, Datenbankobjekte in den O/R-Designer zu ziehen, liest darauf automatisch die Datenbanktabellen und erzeugt die Klassen (wie die Klasse Personen), die LINQ für den Zugriff darauf benötigt.

Mit »select new« können Sie angepasste Ergebnisse für LINQ-Abfragen erzeugen, die die gewünschten Daten enthalten.

Fragen Sie Sternback und die Papierfabrik mit einem Join ab

Jetzt haben Sie alle Werkzeuge, die Sie benötigen, um die Daten von Sternback und der Papierfabrik Objekthausen zu einer endgültigen Ergebnismenge zu kombinieren.



1 **Fügen Sie Ihrem Projekt die SQL-Daten hinzu.**
Erstellen Sie eine neue Windows-Anwendung, falls Sie das noch nicht gemacht haben, und fügen Sie ihr die SQL-Datenbank KontaktDB hinzu. Fügen Sie dann die LINQ to SQL-Klassen in das Projekt ein und schreiben Sie eine einfache Testabfrage, um zu prüfen, ob alles funktioniert.

2 **Erstellen Sie die Sternback-Objekte.**
Hier ist die Liste mit den Sternback-Kundendaten. Fügen Sie diese Ihrem Projekt hinzu:

```
public class SternbackDaten {
    public string Name { get; set; }
    public Getränk Lieblingsgetränk { get; set; }
    public int GeldAusgegeben { get; set; }
    public int Besuche { get; set; }
}
public enum Getränk {
    StandardKaffee, SchokoRockoLatte, DreifachEspresso,
    ZitronengrasTee, DoppelCappuccino, Entkoffeiniert,
    SchokoMacchiato, BananaSplitImBecher,
}
```

Die Sternback-Daten stecken in einer Auflistung mit SternbackDaten-Objekten. Diese enthält eine Menge Daten – da Sie für die Werbeaktion nicht alle benötigen, müssen Sie in der LINQ-Abfrage also nur die auswählen, die Sie brauchen.

Sternback bietet eine Menge Getränke an, und jeder Kunde hat ein Lieblingsgetränk.

Außerdem brauchen Sie eine Methode, die ein paar Beispieldaten generiert:

```
public List<SternbackDaten> SternbackDatenGenerieren() {
    List<SternbackDaten> list = new List<SternbackDaten>() {
        new SternbackDaten() {
            Name = "Jana Venus", Lieblingsgetränk = Getränk.SchokoMacchiato,
            GeldAusgegeben = 255, Besuche = 50 },
        new SternbackDaten() {
            Name = "Lisa Nilson", Lieblingsgetränk = Getränk.DoppelCappuccino,
            GeldAusgegeben = 150, Besuche = 35 },
        new SternbackDaten() {
            Name = "Frank Matt", Lieblingsgetränk = Getränk.ZitronengrasTee,
            GeldAusgegeben = 75, Besuche = 15 },
        new SternbackDaten() {
            Name = "Jens Neu", Lieblingsgetränk = Getränk.BananaSplitImBecher,
            GeldAusgegeben = 60, Besuche = 10 },
        new SternbackDaten() {
            Name = "Sarah Kalter", Lieblingsgetränk = Getränk.StandardKaffee,
            GeldAusgegeben = 110, Besuche = 15 }
    };
    return list;
}
```

SternbackDaten-Generieren() nutzt Auflistungs- und Objektinitialisierer, um die Sternback-Daten einzurichten.

Diese Methode haben wir so erstellt, dass sie ein paar Namen enthält, die auch in der Adressdatenbank der Papierfabrik erscheinen. Wenn Sie andere Namen verwendet haben, sollten Sie schauen, dass Sie hier entsprechende Daten haben.

13 Verknüpfen Sie jetzt die SQL-Datenbank mit der Sternback-Auflistung

Fügen Sie dem Projekt einen Button hinzu, der eine Abfrage ausführt und die Ergebnisse auf der Konsole anzeigt. Hier ist der Code für die Abfrage:

```
List<SternbackDaten> sternbackList = SternbackDatenGenerieren();
KontaktDBDataContext kontext = new KontaktDBDataContext();

var ergebnisse =
    from sternbackKunde in sternbackList
    where sternbackKunde.GeldAusgegeben > 90
    join person in kontext.Personen
    on sternbackKunde.Name equals person.Name
    select new { person.Name, person.Firma,
                 sternbackKunde.Lieblingsgetränk };
```

Wir brauchen einen Join, um die Sternback-Daten mit den Kundendaten in der Tabelle Personen zu verknüpfen.

Hier zieht **select new** Name und Unternehmen aus der Datenbank und das Lieblingsgetränk aus den Sternback-Daten in eine Ergebnisliste.

Das Personen-Member des DataContext ist eine Auflistung, die Ihnen Zugriff auf die Personen-Tabelle der Datenbank ermöglicht.

Den Code auf die Konsole zu schreiben ist leicht:

```
foreach (var zeile in ergebnisse){
    Console.WriteLine("{0} von {1} mag {2}",
                      zeile.Name, zeile.Firma,
                      zeile.Lieblingsgetränk);
}
```

Prüfen Sie Ihre Ergebnisse – schauen Sie, ob die Abfrage wie erwartet funktioniert.

LINQPad

Es gibt ein wunderbares Werkzeug, das Sie beim Lernen und Nutzen von LINQ unterstützen kann. Es heißt LINQPad und stammt von Joe Albahari (einem der Gutachter, die dafür gesorgt haben, dass »C# von Kopf bis Fuß« halbwegs fehlerfrei ist). Sie können es hier kostenlos herunterladen:

http://www.albahari.com/linqpad.html



Name: Datum:

C#-Workshop

Invaders

Dieser Workshop gibt Ihnen eine Spezifikation für ein Programm, das Sie erstellen sollen, um die Fertigkeiten zum Einsatz zu bringen, die Sie in den letzten Kapiteln erworben haben.

Dieses Projekt ist umfangreicher als alle, die Ihnen bisher begegnet sind. Lesen Sie sich also alles erst einmal durch, bevor Sie beginnen, und gönnen Sie sich etwas Zeit. Es macht nichts, wenn Sie hängen bleiben – es passiert hier nichts Neues. Sie können also im Buch weiterlesen und später zu diesem Workshop zurückkehren.

Ein paar Entwurfsdetails haben wir bereits für Sie eingesetzt. Und wir haben sichergestellt, dass Sie alle Teile haben, die Sie benötigen ... und sonst nichts.

Ob Sie die Arbeit zu Ende führen, liegt bei Ihnen.

Die Grafiken für diesen Workshop können Sie unter *http://www.headfirstlabs.com/books/hfcsharp/#code* herunterladen. Lösungsvorschläge finden Sie dort unter der Überschrift »Check out the labs«.

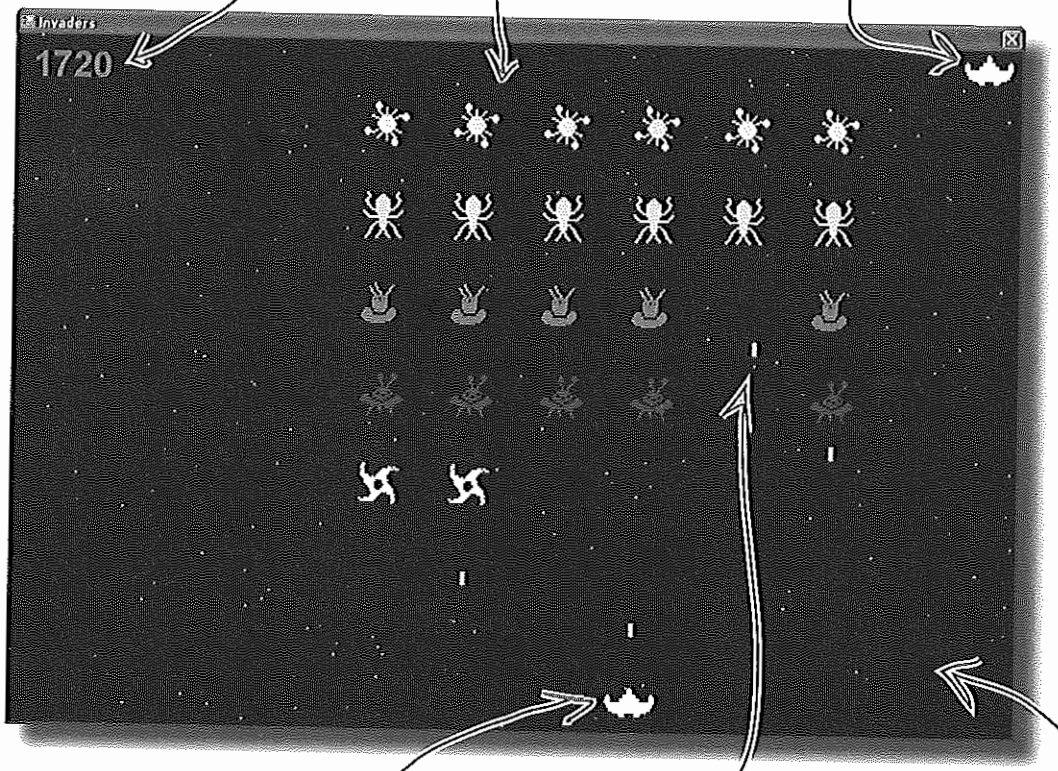
Der Urvater aller Videospiele

In diesem Workshop werden Sie einer der beliebtesten, am meisten bewunderten und am häufigsten kopierten Ikonen der Videospielgeschichte die Ehre erweisen, einem Spiel, das keiner weiteren Vorstellung bedarf. **Es ist Zeit, ein Invaders-Spiel zu schreiben.**

Während der Spieler die Invader zerstört, steigt seine Punktzahl. Sie wird in der linken oberen Ecke angezeigt.

Die Invader greifen in 30 Wellen an. Die erste Welle bewegt sich langsam und feuert jeweils nur wenige Schüsse ab. Die nächste Welle bewegt sich schneller und schießt häufiger. Sind alle 30 Aliens in einer Welle zerstört, greift die nächste Welle an.

Der Spieler beginnt mit drei Schiffen. Das erste Schiff ist im Spiel, und die anderen beiden werden in Reserve gehalten. Seine Reserveschiffe werden in der rechten oberen Ecke angezeigt.



Der Spieler bewegt das Schiff nach links und rechts und schießt auf die Invader. Trifft ein Schiff einen Invader, wird der Invader zerstört, und die Punktzahl des Spielers steigt.

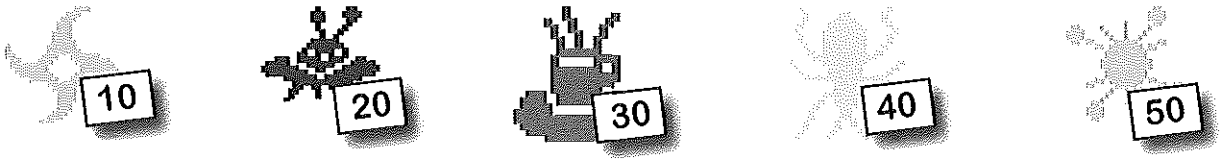
Die Invader schießen zurück. Trifft einer der Schüsse das Schiff, verliert der Spieler ein Leben. Hat er all seine Leben aufgebraucht oder erreichen die Invader den unteren Rand des Bildschirms, endet das Spiel. Es wird dann eine große >>GAME OVER<<-Meldung in der Mitte des Bildschirms angezeigt.

Die vielfarbigen Sterne im Hintergrund blinken, haben aber keine Auswirkungen auf das Spiel.

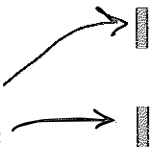
Ihre Aufgabe: Verteidigen Sie den Planeten gegen die Wellen der Invader

Die Invader greifen in Wellen an. Jede Welle ist eine kompakte Formation von 30 einzelnen Invadern. Wenn der Spieler Invader zerstört, steigt seine Punktzahl. Die untersten Invader haben die Form von Sternen und bringen 10 Punkte. Die Raumschiffe bringen 20 Punkte, die fliegenden Untertassen 30, die Wanzen 40 und die Satelliten 50. Der Spieler beginnt mit drei Leben. Verliert er alle drei Leben oder erreichen die Invader den unteren Bildschirmrand, ist das Spiel verloren.

Es gibt fünf verschiedene Typen von Invadern, die sich aber alle auf gleiche Weise verhalten. Sie beginnen oben am Bildschirm und bewegen sich auf dem Bildschirm nach links, bis sie den Rand erreichen. Dann sinken sie tiefer und bewegen sich nach rechts. Erreichen sie den rechten Rand, sinken sie weiter und bewegen sich wieder nach links. Erreichen die Invader den unteren Bildschirmrand, ist das Spiel vorüber.



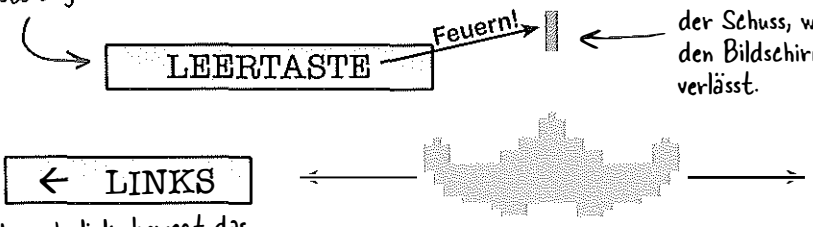
Die erste Invader-Welle kann zwei Schüsse gleichzeitig abfeuern – die Invader halten ihre Schüsse zurück, wenn bereits zwei Schüsse auf dem Bildschirm sind. Die zweite Welle feuert drei Schüsse ab, die dritte vier usw.



Die Leertaste schießt. Aber es können jeweils nur zwei Schüsse gleichzeitig auf dem Bildschirm sein. Trifft ein Schuss etwas oder verschwindet er, kann der nächste abgefeuert werden.

Trifft ein Schuss einen Invader, verschwinden beide. Ansonsten verschwindet der Schuss, wenn er den Bildschirm oben verlässt.

Das Spiel sollte festhalten, welche Tasten aktuell gedrückt werden. Werden Pfeil-nach-rechts und die Leertaste gedrückt, sollte sich das Schiff nach rechts bewegen und feuern (falls noch keine zwei Schüsse auf dem Bildschirm sind).



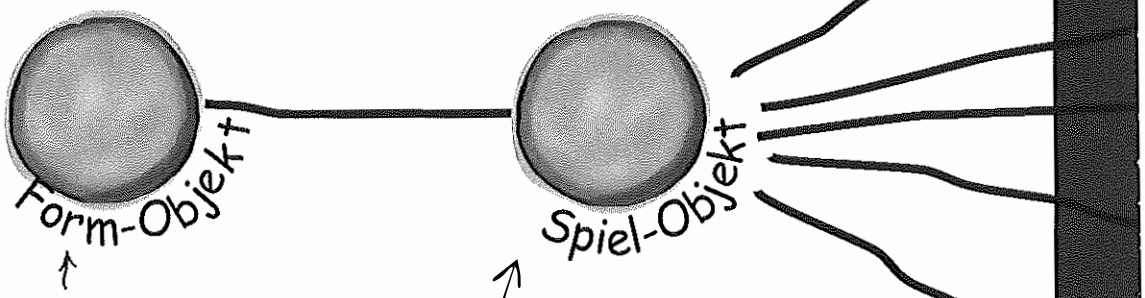
Pfeil-nach-links bewegt das Schiff zum linken Rand des Bildschirms.

Pfeil-nach-rechts bewegt das Schiff nach rechts.

Die Architektur von Invaders

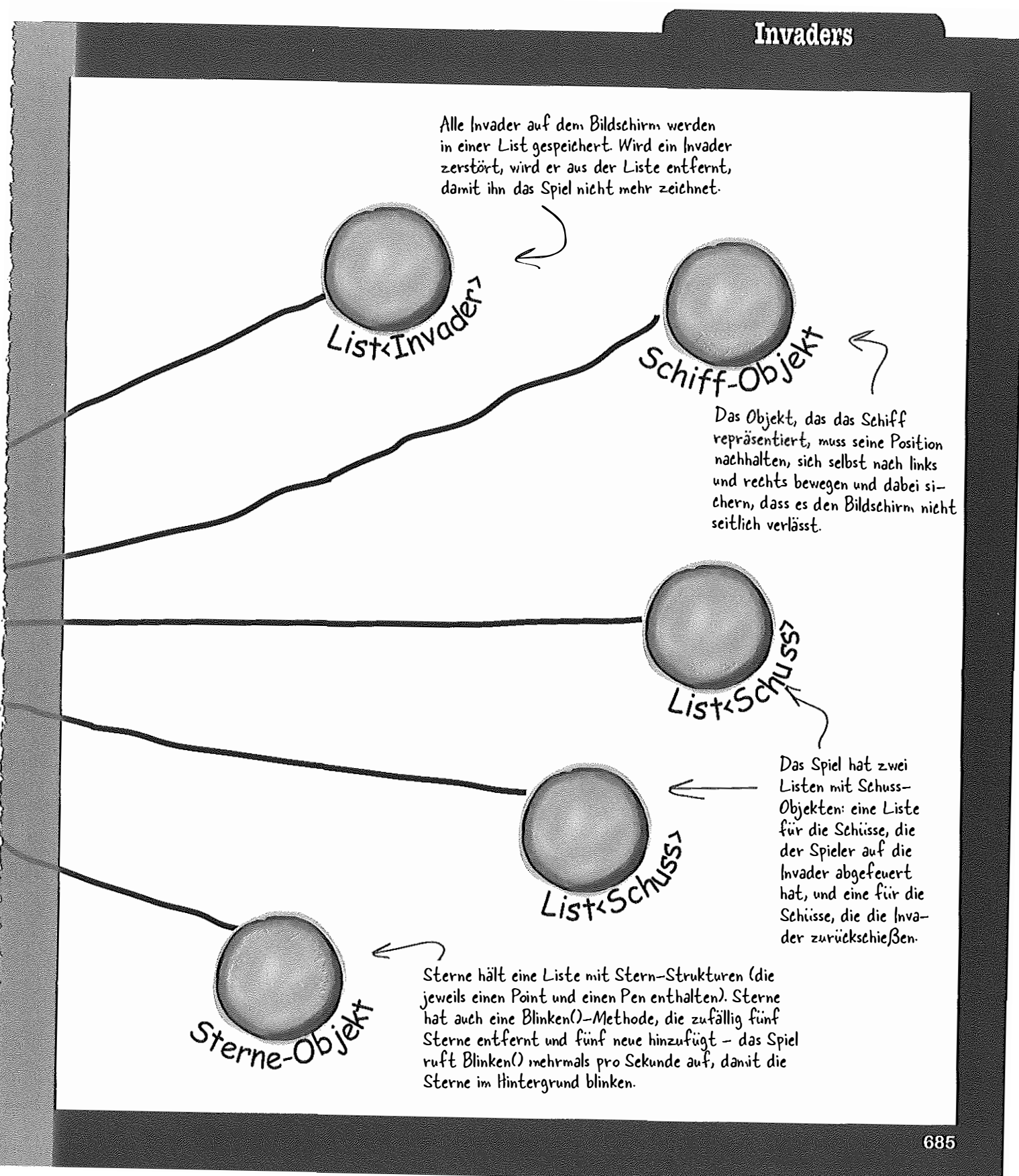
Invaders muss eine Welle mit 30 Invadern (einschließlich ihres Orts, ihres Typs und ihrer Punktzahl) festhalten, das Schiff des Spielers, die Schüsse, die Spieler und Invader aufeinander abgeben, sowie die Sterne im Hintergrund. Wie im Workshop »Die Suche« benötigen Sie eine `Spiel`-Klasse, die all das nachhält und die Formular- und Spielobjekte koordiniert.

Hier ist ein Überblick über die Dinge, die Sie erstellen müssen:



Das Formular ist ziemlich einfach. Es hat Timer, die das Spiel am Laufen halten, reicht Tastendrücke weiter und animiert die Invader und die blinkenden Sterne. Und es hat einen Paint-Event-Handler, der die Grafiken zeichnet, dazu aber einfach die Zeichnen()-Methode des Spiel-Objekts aufruft.

Das Spiel-Objekt steuert den Spielablauf. Es hält nach, wie viele Leben der Spieler noch hat und wie viele Invader-Wellen bereits angegriffen haben. Ist das Spiel vorüber, wird ein GameOver-Event abgesetzt, das dem Formular mitteilt, dass es den Spiel-Timer anhalten soll.

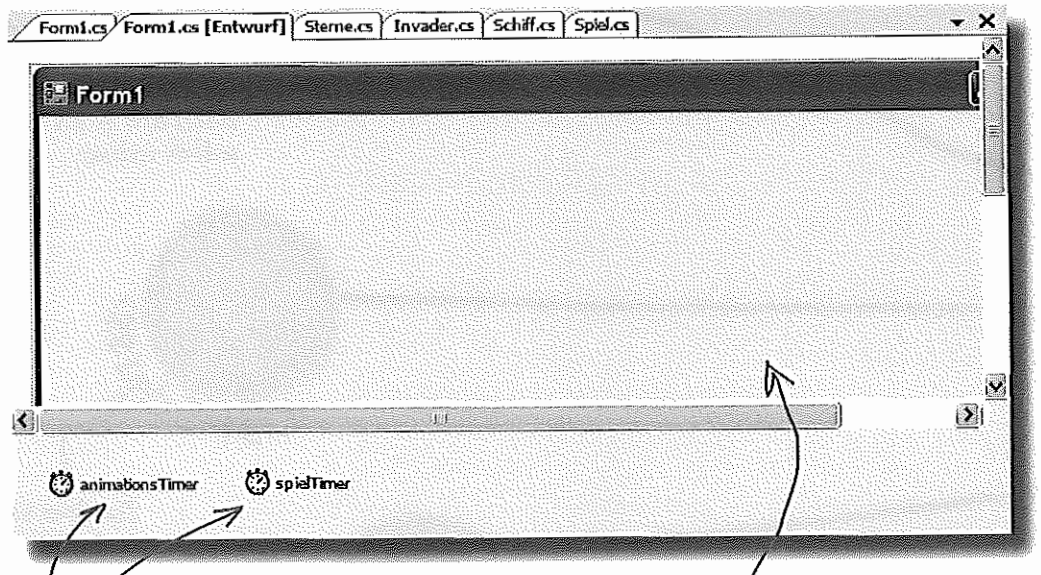

Alle Invader auf dem Bildschirm werden in einer List gespeichert. Wird ein Invader zerstört, wird er aus der Liste entfernt, damit ihn das Spiel nicht mehr zeichnet.
List<Invader>
Schiff-Objekt
Das Objekt, das das Schiff repräsentiert, muss seine Position nachhalten, sich selbst nach links und rechts bewegen und dabei sichern, dass es den Bildschirm nicht seitlich verlässt.
List<Schuss>
Das Spiel hat zwei Listen mit Schuss-Objekten: eine Liste für die Schüsse, die der Spieler auf die Invader abgefeuert hat, und eine für die Schüsse, die die Invader zurückschießen.
List<Schuss>
Sterne-Objekt
Sterne hält eine Liste mit Stern-Strukturen (die jeweils einen Point und einen Pen enthalten). Sterne hat auch eine Blinken()-Methode, die zufällig fünf Sterne entfernt und fünf neue hinzufügt – das Spiel ruft Blinken() mehrmals pro Sekunde auf, damit die Sterne im Hintergrund blinken.

Entwerfen Sie das Invaders-Formular

Das Invaders-Formular hat nur zwei Steuerelemente: einen Timer, der die Animation anstößt (die Sterne blinken lässt und die Invader animiert, indem es ihre Bilder an einen anderen Punkt bewegt), und einen Timer, der den Spielablauf steuert (die Invader von links und rechts wandern lässt, den Spieler bewegt und Spieler und Invader aufeinander schießen lässt). Daneben ist die einzige Logik auf dem Formular ein Event-Handler, der das GameOver-Event des Spiels verarbeitet, sowie KeyUp- und KeyDown-Event-Handler zur Verarbeitung der Tastatureingaben.

Das Formular setzt ein KeyDown-Event ab, wenn eine Taste gedrückt wird, und ein KeyUp-Event, wenn eine Taste losgelassen wird.

Initialisiert das Formular sein Spiel-Objekt, übergibt es ihm sein ClientRectangle, damit es die Grenzen des Formulars kennt. Die Größe des Spielfelds können Sie also einfach ändern, indem Sie die Größe des Formulars anpassen.



Sie müssen zwei Timer hinzufügen: animationsTimer und spielTimer.

Setzen Sie die FormBorderStyle-Eigenschaft des Formulars auf FixedSingle, seine DoubleBuffered-Eigenschaft auf true, schalten Sie seine MinimizeBox- und MaximizeBox-Eigenschaft ab, setzen Sie seinen Titel und geben Sie ihm die Breite, die das Spielfeld haben soll.

Der Animations-Timer kümmert sich um den Augenschmaus

Die Sterne im Spielhintergrund und die Animation der Invader haben keine Auswirkungen auf den Spielablauf und laufen weiter, wenn das Spiel ausgesetzt oder angehalten wird. Dafür benötigen wir also einen separaten Timer.



Fügen Sie Code für das Tick-Event des Animations-Timers hinzu

Ihr Code sollte einen Zähler haben, der von 0 bis 3 läuft und dann wieder zu 0 zurück. Dieser Zähler wird eingesetzt, um die vier Zellen der Invader-Animation zu aktualisieren (um eine glatte Animation zu erstellen). Er sollte außerdem die `Blinken()`-Methode des `Spiel`-Objekts aufrufen, die die Sterne zum Blinken bringt. Schließlich sollte er die `Refresh()`-Methode des Formulars aufrufen, damit der Bildschirm neu gezeichnet wird.

Diese Animation erfolgt auch, wenn das Spiel nicht läuft. Das heißt, dass das Blinken der Sterne und die Animation der Invader auch erfolgen, wenn das Spiel vorüber, angehalten oder noch nicht gestartet ist.

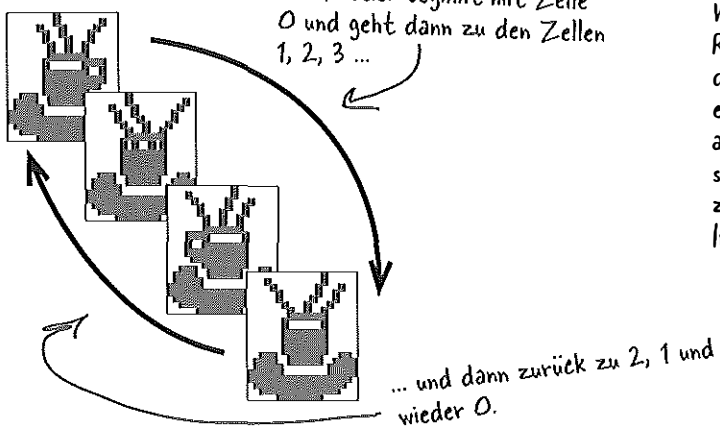
Probieren Sie es mit einem Timer-Intervall von 33ms. Das sollte Ihnen ungefähr 30 Frames pro Sekunde geben. Achten Sie aber darauf, dass Sie den Spiel-Timer auf ein kürzeres Intervall setzen. Das Schiff sollte sich schneller bewegen und das Spiel sollte schneller ablaufen, als die Sterne blinken.

Passen Sie den Timer an, damit die Animation glatter läuft

Bei einem 33ms-Intervall für die Animation sollte der Spiel-Timer auf 10ms gesetzt werden. Das sorgt dafür, dass der Spielablauf schneller läuft als die Animation (die eigentlich nur ein unwichtiges visuelles Zückerchen ist). Außerdem kann die `Laufen()`-Methode von `Spiel` (die vom Spiel-Timer angestoßen wird, über den wir gleich reden werden) mehr CPU-Zyklen in Anspruch nehmen. Ist die CPU mit dem Spielablauf beschäftigt, wartet der Animations-Timer, bis die CPU für ihn bereit ist, und wird dann abgesetzt (animiert Sterne und Aliens).

Braucht die Laufen()-Methode von Spiel mehr Zeit als die 33ms, auf die der Animations-Timer gesetzt ist, erfolgt die Animation, wenn Laufen() fertig ist.

Sie können beide Timer auch auf ein Intervall von 5ms setzen. Spiel und Animation laufen dann so schnell, wie Ihr System dies abwickeln kann (auf schnellen Rechnern kann das dazu führen, dass die Animation nervend schnell wird).



Wir haben das auf einem langsamen Rechner ausprobiert und herausgefunden, dass wir spielbare 10 Frames pro Sekunde erhalten, wenn wir den Animations-Timer auf 100ms und den Spiel-Timer auf 50ms setzen. Versuchen Sie, mit diesen Werten zu beginnen, und reduzieren Sie dann beide Intervalle, bis Sie zufrieden sind.

Auf Tastatureingaben reagieren

Bevor wir den Spiel-Timer schreiben können, benötigen wir Event-Handler für die KeyDown- und KeyUp-Events. KeyDown wird ausgelöst, wenn eine Taste gedrückt wird. Wird sie wieder losgelassen, wird KeyUp abgesetzt. Bei den meisten Tasten können wir einfach mit einer Aktion antworten, indem wir einen Schuss abgeben oder das Spiel beenden.

Einige Tasten, wie die Pfeil-nach-rechts- und -links-Tasten, müssen wir in einer Liste speichern, die unser Spiel-Timer einsetzen kann, um das Schiff des Spielers zu bewegen. Außerdem brauchen wir im Formular-Objekt eine Liste der Tasten, die gedrückt wurden:

Drückt der Spieler gleichzeitig die linke Pfeiltaste und die Leertaste, enthält die Liste Keys.Left und Keys.Space.

Wir benötigen eine Liste mit Tasten, damit wir nachhalten können, welche Tasten gedrückt wurden. In Kürze wird unser Spiel-Timer diese Liste für die Bewegung benötigen.

```
List<Keys> gedrückteTasten = new List<Keys>();

private void Form1_KeyDown(object sender, KeyEventArgs e) {
  if (e.KeyCode == Keys.Q)
    Application.Exit();
  if (gameOver)
    if (e.KeyCode == Keys.S) {
      // Code zum Zurücksetzen und Neustarten des Spiels.
      return;
    }
  if (e.KeyCode == Keys.Space)
    spiel.Schießen();
  if (gedrückteTasten.Contains(e.KeyCode))
    gedrückteTasten.Remove(e.KeyCode);
  gedrückteTasten.Add(e.KeyCode);
}

private void Form1_KeyUp(object sender, KeyEventArgs e) {
  if (gedrückteTasten.Contains(e.KeyCode))
    gedrückteTasten.Remove(e.KeyCode);
}
```

Die Taste »Q« beendet das Spiel.

Das Enum Keys definiert alle Tasten, deren Codes Sie eventuell prüfen möchten.

Ist das Spiel beendet, wird das Spiel zurückgesetzt und neu gestartet.

Aber das soll nur erfolgen, wenn das Spiel beendet ist. Ein Spiel, das bereits läuft, sollte durch S nicht neu gestartet werden können.

Diesen Code müssen Sie ergänzen.

Die Leertaste gibt einen Schuss ab.

Indem wir die Taste entfernen und wieder neu hinzufügen, sichern wir, dass die Taste das (aktuellste) Element in der Liste ist.

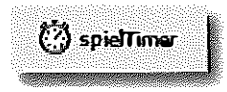
Die gedrückte Taste wird der Liste hinzugefügt, die wir gleich verwenden werden.

Wir möchten, dass die zuletzt gedrückte Taste am Anfang der Liste steht, damit das Spiel immer auf die Taste reagiert, die als letzte gedrückt wurde, wenn der Spieler mehrere Tasten auf einmal drückt. Lässt er eine los, reagiert das Spiel auf die nächste Taste in der Liste.

Wird eine Taste losgelassen, entfernen wir sie aus der Liste der gedrückten Tasten.

Der Spiel-Timer kümmert sich um Bewegung und Spielablauf

Die Hauptaufgabe des Spiel-Timers des Formulars ist der Aufruf der `Laufen()`-Methode der Klasse `Spiel`. Aber er muss auch auf gedrückte Tasten reagieren und muss deswegen die gedrückteTasten-List prüfen, um die Tastenaktionen einzusehen, die die Events `KeyDown` und `KeyUp` eingefangen haben:



Achten Sie darauf, dass die Steuerelementnamen den Namen der Event-Handler entsprechen.

```
private void spielTimer_Tick(object sender, EventArgs e) {
  if (gedrückteTasten.Count() >= 1) {
    switch (gedrückteTasten[0]) {
      case Keys.Left:
        spiel.SpielerBewegen(Richtung.Links);
        break;
      case Keys.Right:
        spiel.SpielerBewegen(Richtung.Rechts);
        break;
    }
  }
  spiel.Laufen();
}
```

gedrückteTasten ist Ihr List<Keys>-Objekt, das von den KeyDown- und KeyUp-Event-Handlern verwaltet wird. Die Taste beim Index 0 ist immer die zuletzt gedrückte Taste.

Die Events KeyUp und KeyDown nutzen das Keys-Enum, um eine Taste anzugeben. Wir nutzen Keys.Left und Keys.Right, um das Schiff zu bewegen.

Wir müssen uns nur um die Bewegung kümmern. Andere Tasten, wie die Leertaste und »Q« zumBeenden, werden im gerade geschriebenen KeyDown-Handler verarbeitet.

Abschließend rufen wir auf dem Spiel-Objekt Laufen() auf, damit das Spiel weiterläuft.

Schüsse bewegen sich nach oben und unten, der Spieler bewegt sich nach links und rechts, und die Invader gehen nach lin s, rechts und unten. Sie brauchen dieses Enum, um all diese Richtungen einzuhalten.

```
public enum Richtung {
    Links,
    Rechts,
    Hoch,
    Runter,
}
```

Ein weiteres Formulardetail: das GameOver-Event

Fügen Sie dem Formular ein privates `bool`-Feld namens `gameOver` hinzu, das `true` wird, wenn das Spiel vorüber ist. Fügen Sie dann einen Event-Handler für das `GameOver`-Event des `Spiel`-Objekts hinzu, der den Spiel-Timer anhält (aber nicht den Animations-Timer, damit die Sterne weiter blinken und die Invader weiter animiert werden), `gameOver` auf `true` setzt und die `Invalidate()`-Methode des Formulars aufruft.

Wenn Sie darangehen, den `Paint`-Event-Handler des Formulars zu schreiben, müssen Sie ihn `gameOver` prüfen lassen. Ist er `true`, sollte er in die Mitte des Bildschirms in großen, gelben Buchstaben GAME OVER schreiben. Dann sollte er in die linke untere Ecke »S drücken, um ein neues Spiel zu beginnen, oder Q, um das Spiel zu beenden« schreiben.

Hier ist ein Beispiel dafür, wie man einem Formular ohne Unterstützung der IDE ein weiteres Event hinzufügt. Das ist manuelle Programmierung.

Das GameOver-Event und sein Delegate befinden sich in der Klasse Spiel, die Sie gleich sehen werden.

Der Spiel-Timer des Formulars bringt das Spiel ans Laufen()

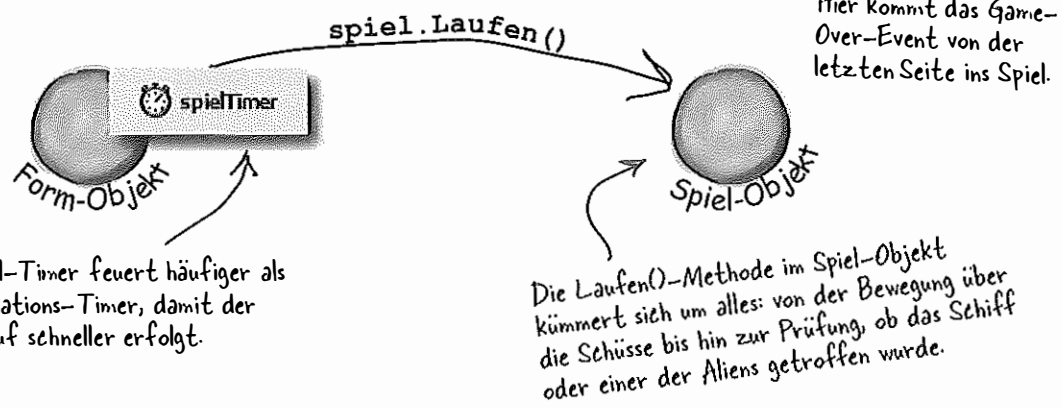
Neben der Verarbeitung der Bewegung nach links und rechts ist es die Hauptaufgabe des Spiel-Timers, die `Laufen()`-Methode des `Spiel`-Objekts aufzurufen. Das ist der Ort, an dem der Spielablauf abgewickelt wird. Das `Spiel`-Objekt kümmert sich um den Zustand des Spiels, und seine `Laufen()`-Methode rückt das Spiel einen Frame vor. Das schließt Folgendes ein:

1. **Prüfen, ob der Spieler gestorben ist**. Dazu wird die Eigenschaft `Lebend` verwendet. Stirbt der Spieler, zeigt das Spiel eine kleine Animation des kollabierenden Schiffs an (dabei nutzt es `DrawImage()`, um das Schiff zu nichts zusammenzuquetschen). Die Animation wird von der Klasse `Schiff` erledigt. `Laufen()` muss also nur prüfen, ob der Spieler tot ist. Ist das der Fall, wird das Spiel zurückgesetzt – auf diese Weise verhindert es, dass die Invader sich weiterbewegen oder schießen, während der Spieler eine kleine Pause erhält (und beobachtet, wie sein Schiff zerquetscht wird).

2. **Die einzelnen Schüsse bewegen.** Schüsse, die von den Invadern abgefeuert werden, bewegen sich nach unten, und Schüsse, die vom Spieler abgefeuert werden, bewegen sich nach oben. `Spiel` hält zwei `List<Schuss>`-Objekte, eins für die Schüsse der Invader, eins für die des Spielers. Jeder Schuss, der den Bildschirm verlässt, wird aus den Listen entfernt.

3. **Die Invader bewegen.** `Spiel` ruft die `Bewegen()`-Methode aller `Invader`-Objekte auf und sagt ihnen, in welche Richtung sie sich bewegen sollen. `Spiel` hält auch nach, wo die Invader sind, damit es weiß, ob sie eine Zeile nach unten rücken oder die Richtung ändern müssen. Dann prüft `Spiel`, ob es Zeit ist, dass die Invader das Feuer erwidern. Ist das der Fall, fügt es der `List<>` neue `Schuss`-Objekte hinzu.

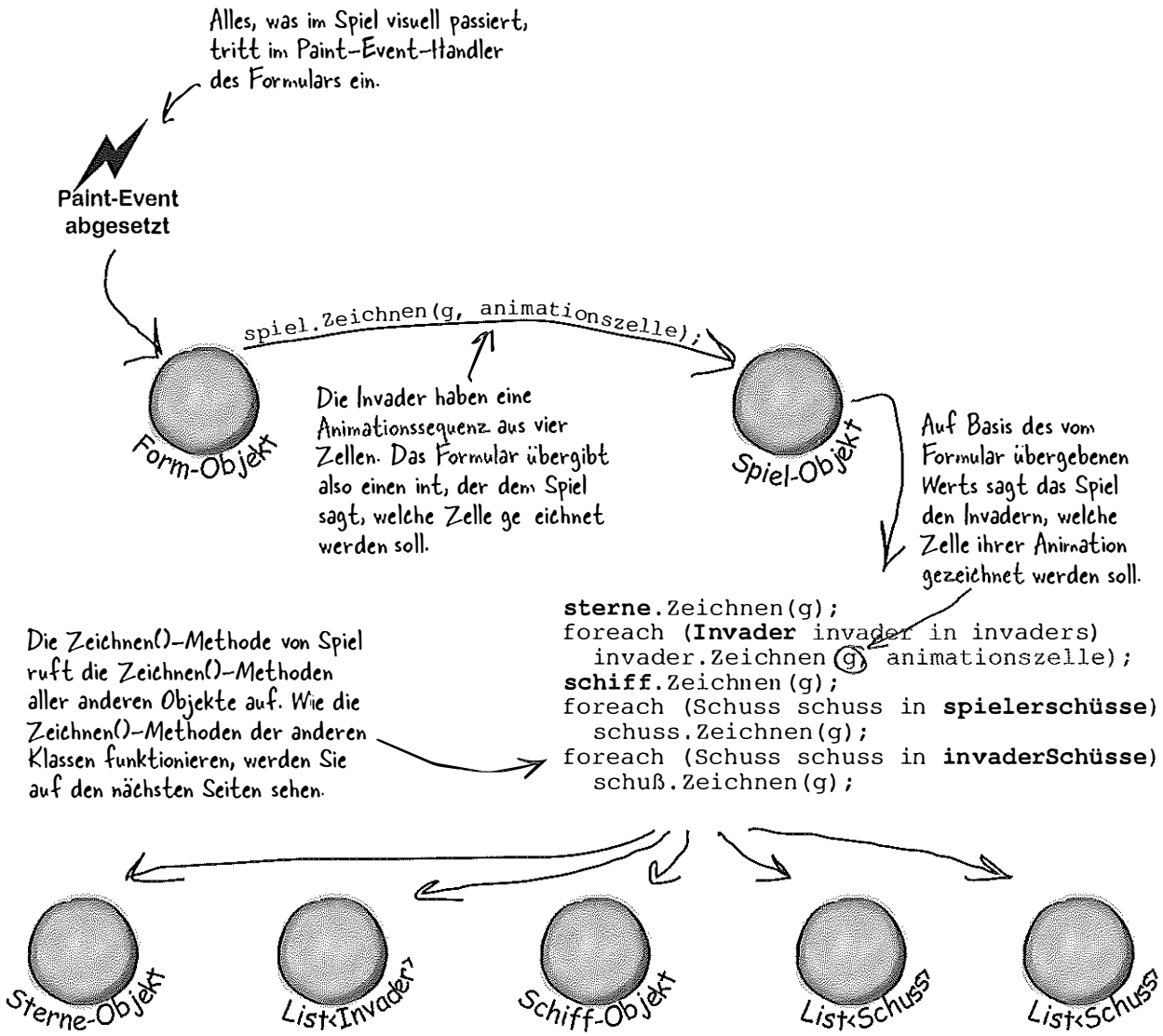
4. **Auf Treffer prüfen**. Hat ein Schuss des Spielers einen Invader getroffen, entfernt `Spiel` den Invader aus der entsprechenden `List<>`. Dann prüft das `Spiel`, ob Schüsse der Invader das Schiff des Spielers getroffen haben. Ist das der Fall, wird der Spieler getötet, indem die Eigenschaft `Lebend` auf `false` gesetzt wird. Hat der Spieler keine Leben mehr, setzt das `Spiel` ein `GameOver`-Event ab, um dem Formular zu sagen, dass das Spiel vorüber ist. Der `GameOver`-Event-Handler des Formulars hält den Spiel-Timer an, damit `Laufen()` nicht wieder aufgerufen wird.



Die Grafik in die Hand nehmen

In den vorangegangenen Workshops haben die Formulare für die Grafik Steuerelemente verwendet. Aber da Sie jetzt wissen, wie man `Graphics` und Double-Buffering nutzt, sollte sich das `Spiel`-Objekt um große Teile der Zeichnung kümmern.

Das Formular sollte also einen `Paint`-Event-Handler haben (in den Sie das Double-Buffering stecken werden). Aber den Rest der Zeichenoperationen delegieren Sie jedes Mal, wenn dieses Event abgesetzt wird, an das Spiel-Objekt.



Die Klasse Spiel aufbauen

Die Klasse Spiel ist der Controller für das Invaders-Spiel. Hier ist ein Ansatzpunkt dafür, wie diese Klasse aussehen sollte. Es bleibt Ihnen allerdings noch eine Menge Arbeit.

```
public class Spiel {
  private int punkte = 0;
  private int verbleibendeLeben = 2;
  private int welle = 0;
  private int übersprungeneFrames = 0;

  private Rectangle grenzen;
  private Random zufall;

  private Richtung invaderRichtung;
  private List<Invader> invaders;

  private Schiff schiff;
  private List<Schuss> spielerschüsse;
  private List<Schuss> invaderSchüsse;

  private Sterne sterne;

  public event EventHandler GameOver;

  // usw.
}
```

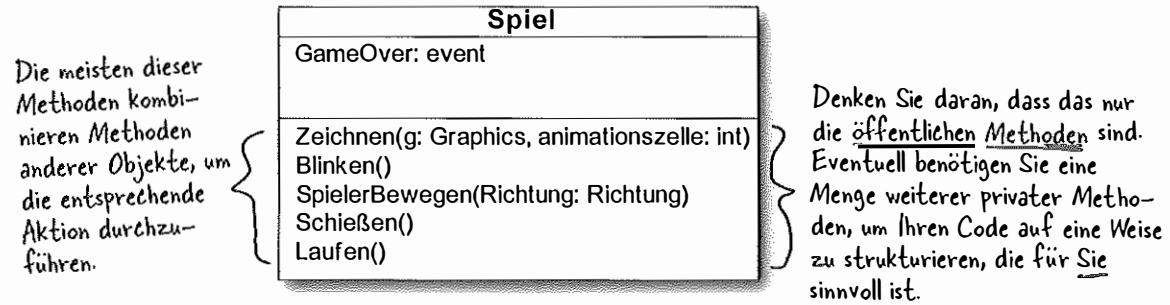
Die Felder punkte, verbleibendeLeben und welle halten einige der grundlegenden Informationen zum Spielzustand fest.

Dieses Feld nutzen Sie, um die Invader in den frühen Spielphasen zu bremsen – die erste Welle sollte 6 Frames überspringen, bevor sie sich das nächste Mal bewegt, die zweite 5 Frames, die dritte 4 usw.

Diese List<> mit Invader-Objekten hält alle Invader in der aktuellen Welle fest. Wird ein Invader zerstört, wird er aus der Liste entfernt. Das Spiel prüft regelmäßig, ob die Liste leer ist – ist sie das, sendet es die nächste Welle von Invadern.

Das Sterne-Objekt hält die vielfarbigen Sterne im Hintergrund fest.

Das Spiel-Objekt setzt sein GameOver-Event ab, wenn der Spieler stirbt und keine Leben mehr hat. Die Event-Handler-Methode werden Sie im Formular erstellen und mit dem GameOver-Event des Spiel-Objekts verknüpfen.



Die meisten dieser Methoden kombinieren Methoden anderer Objekte, um die entsprechende Aktion durchzuführen.

Denken Sie daran, dass das nur die öffentlichen Methoden sind. Eventuell benötigen Sie eine Menge weiterer privater Methoden, um Ihren Code auf eine Weise zu strukturieren, die für Sie sinnvoll ist.

Die Methoden der Klasse Spiel

Die Klasse Spiel hat fünf öffentliche Methoden, die von unterschiedlichen Events angestoßen werden, die im Fomular eintreten.

1. **Die Methode Zeichnen() zeichnet das Spiel auf ein Graphics-Objekt.**
Die `Zeichnen()`-Methode erwartet zwei Parameter: ein `Graphics`-Objekt und einen int, der die Nummer der Animationszelle enthält (eine Zahl zwischen 0 und 3). Erst sollte sie ein schwarzes Rechteck zeichnen, das das gesamte Formular ausfüllt (unter Verwendung des in `grenzen` gespeicherten Anzeigerechtecks, das vom Formular übergeben wurde). Dann sollte die Methode die Sterne, die Invader, das Schiff des Spielers und dann die Schüsse zeichnen. Abschließend sollte es in der linken oberen Ecke die Punkte und in der rechten oberen Ecke die Reserveschiffe anzeigen sowie in gelben Buchstaben eine große »GAME OVER«-Meldung schreiben, wenn `gameOver` `true` ist.

2. **Die Methode Blinken() lässt die Sterne blinken.**
Der Animations-Timer des Formulars muss die Sterne blinken lassen können, das Spiel-Objekt braucht also eine einzeilige Methode, die `sterne.Blinken()` aufruft.

Den Code für die Sterne werden wir bald schreiben.

3. **Die Methode SpielerBewegen() bewegt den Spieler.**
Die Tastatur-Event-Handler des Formulars müssen das Schiff des Spielers bewegen. `Spiel` braucht also auch eine zweizeilige Methode, die als Parameter einen `Richtung`-Wert erwartet, prüft, ob der Spieler lebt oder nicht, und dann `schiff.Bewegen()` aufruft, um diese Bewegung ausführen zu lassen.

4. **Die Methode Schießen() lässt den Spieler einen Schuss auf die Aliens abgeben.**
Die Methode `Schießen()` prüft, ob weniger als zwei Schüsse des Spielers auf dem Bildschirm sind. Ist das der Fall, sollte die Methode der List `spielerschüsse` am richtigen Ort einen weiteren Schuss hinzufügen.

5. **Die Methode Laufen() lässt das Spiel laufen.**
Der Animations-Timer des Formulars ruft die `Laufen()`-Methode des Spiels 10 bis 30 Mal pro Sekunde auf (das ist von der Geschwindigkeit der CPU Ihres Rechners abhängig). Die `Laufen()`-Methode macht alles, was das Spiel tun muss, um einen Frame weiterzurücken:

   - Das Spiel prüft anhand der Eigenschaft **`Lebend`**, ob der Spieler noch lebt. Wenn ja, ist das Spiel noch nicht vorüber – wäre es das, würde das Formular den Animations-Timer mit der Methode `Stop()` anhalten. Die Methode `Laufen()` macht also nichts, bis das Schiff wieder lebt – sie gibt einfach die Ausführung zurück.
   - Jeder Schuss muss aktualisiert werden. Das Spiel muss beide `List<Schuss>`-Objekte durchlaufen und die `Bewegen()`-Methode jedes Schusses aufrufen. Liefert die `Bewegen()`-Methode eines Schusses `false`, bedeutet das, dass der Schuss den Bildschirm verlassen hat – er wird also aus der Liste gelöscht.
   - Das Spiel bewegt dann die Invader und ermöglicht ihnen zurückzufeuern.
   - Schließlich prüft es auf Kollisionen: erst, ob sich ein Schuss mit einem Invader überschneidet (gegebenenfalls werden beide aus den entsprechenden Listen entfernt), und dann, ob der Spieler getroffen wurde. Den Klassen für die Invader und das Schiff haben wir eine `Rectangle`-Eigenschaft namens `Fläche` hinzugefügt – wir können also `Contains()` nutzen, um zu prüfen, ob ein Schuss innerhalb dieses Rechtecks liegt.

Die Klasse Spiel ausfüllen

Das Problem mit Klassendiagrammen ist, dass sie üblicherweise nicht öffentliche Eigenschaften und Methoden aussparen. Auch nachdem Sie die Methoden von Seite 693 erledigt haben, steht also noch eine Menge Arbeit aus. Hier sind einige Dinge, über die Sie nachdenken sollten:

Der Konstruktor richtet alles ein

Das `Spiel`-Objekt muss alle anderen Objekte erstellen – die `Invader`-Objekte, das `Schiff`-Objekt und die `List`-Objekte für die Schüsse sowie das `Sterne`-Objekt. Das Formular übergibt ein initialisiertes `Random`-Objekt und sein eigenes `ClientRectangle` (damit `Spiel` **die Grenzen des Spielfelds kennt**, die es einsetzt, um zu ermitteln, ob Schüsse das Spielfeld verlassen oder die Invader den Rand erreichen und tiefer sinken sowie die Richtung ändern müssen). Dann sollte Ihr Code alles andere in der Spielwelt erstellen.

Mit den meisten dieser Objekte werden wir uns auf den nächsten paar Seiten dieses Workshops befassen.

Erstellen Sie eine NächsteWelle()-Methode

Eine einfache Methode zur Erstellung der nächsten Welle von Invadern sollte hilfreich sein. Sie sollte dem Feld `invaders` eine neue `List` mit `Invader`-Objekten zuweisen, ihr die 30 Invader in 6 Spalten so hinzufügen, dass sie an ihrer Startposition sind, das Feld `welle` um eins erhöhen und das Feld `invader-Richtung` so setzen, dass sie damit beginnen, sich auf die rechte Bildschirmseite zuzubewegen. Außerdem sollten Sie das Feld `übersprungeneFrames` ändern.

Das ist ein Beispiel für eine private Methode, die uns bei der Organisation der Klasse Spiel wirklich hilft.

Ein paar weitere Ideen für private Methoden

Hier sind weitere Möglichkeiten für private Methoden, bei denen Sie erwägen können, ob sie Ihnen beim Entwurf der Klasse `Spiel` helfen:

- ✓ Eine Methode, die prüft, ob der Spieler getroffen wurde (`SpielerGetroffenPrüfen()`).
- ✓ Eine Methode, die prüft, ob ein Invader getroffen wurde (`InvaderGetroffenPrüfen()`).
- ✓ Eine Methode, die alle Invader bewegt (`InvaderBewegen()`).
- ✓ Eine Methode, die Invader zurückschießen lässt (`Zurückschießen()`).



KOPF-NUSS

Man kann geschützte und private Eigenschaften und Methoden in Klassendiagrammen zeigen, aber das werden Sie nur selten sehen. Was, glauben Sie, ist der Grund dafür?

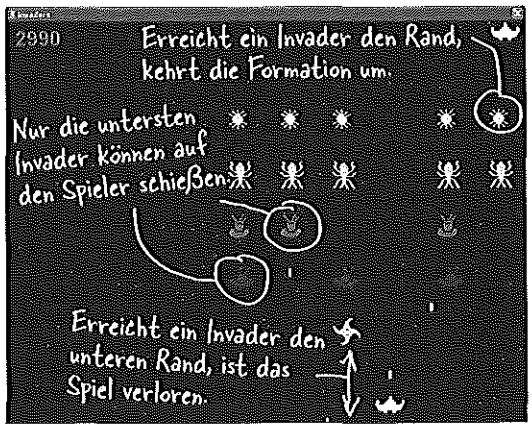
LINQ vereinfacht die Kollisionsprüfung

Beim ersten Lesen scheint das sehr komplex, aber die LINQ-Abfragen bestehen nur aus ein paar Zeilen. Ein Tipp: Machen Sie es nicht zu kompliziert!

Sie haben Auflistungen mit Invadern und Schüssen. Diese müssen Sie durchsuchen, um bestimmte Invader und Schüsse zu finden. Immer wenn Sie Auflistung und Suchen in einem Atemzug hören, sollten Sie LINQ denken und Folgendes tun:

1. **Ermitteln, ob die Invader-Formation den Bildschirmrand erreicht hat.**
Die Invader müssen ihre Richtung ändern, wenn einer von ihnen weniger als 100 Pixel vom Spielfeldrand entfernt ist. Marschieren die Invader nach rechts, muss das Spiel ihnen sagen, dass sie eine Zeile tiefer gehen und sich nach links bewegen sollen, wenn sie den rechten Formularrand erreichen. Gehen sie nach links, muss das Spiel prüfen, ob sie den linken Rand erreicht haben. Dafür sollten Sie eine private `InvaderBewegen()`-Methode schreiben, die von `Laufen()` aufgerufen wird. Zunächst sollte diese das private Feld `übersprungeneFrames` prüfen, aktualisieren und zurückkehren, falls dieser Frame übersprungen wurde. Dann sollte sie prüfen, in welche Richtung sich die Invader bewegen. Bewegen sie sich nach rechts, sollte `InvaderBewegen()` LINQ einsetzen, um in der `invaders`-Liste nach einem Invader zu suchen, der sich an einem Ort befindet, dessen X-Wert weniger als 100 Pixel vom rechten Rand entfernt ist. Findet sie einen, sollte sie die Invader anweisen, nach unten zu gehen, und dann `invaderRichtung` auf `Richtung.Links` setzen. Andernfalls sollte sie die Invader nach rechts gehen lassen. Bewegen sich die Invader nach links, sollte sie genau das Gegenteil tun: eine andere LINQ-Abfrage verwenden, um zu prüfen, ob ein Invader weniger als 100 Pixel vom linken Rand entfernt ist, und sie nach unten und dann nach rechts gehen lassen, wenn das der Fall ist.

2. **Ermitteln, welche Invader schießen können**
Fügen Sie die private Methode `Zurückschießen()` hinzu, die von `Laufen()` aufgerufen wird. Diese sollte die Ausführung zurückgeben, wenn die Schussliste der Invader bereits `welle + 1` Schüsse enthält. Gleiches sollte sie tun, wenn `zufall.Next(10) < 10 - welle` ist (das sorgt dafür, dass die Invader zufällig feuern und nicht immer). Werden beide Tests bestanden, können die Invader mit LINQ anhand von `Ort.X` gruppiert und absteigend sortiert werden (`descending`). Aus den resultierenden Gruppen kann dann zufällig eine ausgewählt werden, aus der mit `First()` der unterste Invader abgerufen wird. Dann kann der Schuss unmittelbar unter der Mitte (nutzen Sie dazu das Fläche-Feld) des Invaders der Schussliste hinzugefügt werden.



3. **Spieler- und Invader-Kollisionen prüfen**
Sie sollten eine Methode schreiben, die auf Kollisionen prüft. Es gibt drei Kollisionen, auf die geprüft werden muss, und die `Contains()`-Methode von `Rectangle` wird dabei hilfreich sein – übergeben Sie ihr einfach einen `Point`, und sie liefert `true`, wenn sich dieser Punkt im Rechteck befindet.

- ★ Nutzen Sie LINQ, um tote Invader zu finden, indem Sie die Schüsse in der Schussliste des Spielers durchlaufen und einen Invader wählen, bei dem `invader.Fläche` den Ort des Schusses enthält. Entfernen Sie den Invader und den Schuss.
- ★ Prüfen Sie mit einer Abfrage, ob ein Invader den unteren Spielfeldrand erreicht – beenden Sie dann das Spiel.
- ★ Um zu prüfen, ob der Spieler getroffen wurde, brauchen Sie kein LINQ, sondern nur eine Schleife und die `Fläche`-Eigenschaft des Spielers. (Denken Sie daran, dass **Sie eine Collection in einer foreach-Schleife nicht verändern können.**)

Die Klasse Invader schmieden

Die Klasse `Invader` hält einen Invader nach. Erstellt `Spiel` eine neue Invader-Welle, fügt es also 30 Invader-Instanzen einer `List<Invader>` hinzu. Jedes Mal, wenn die `Laufen()`-Methode von Spiel aufgerufen wird, wird die `Bewegen()`-Methode aller Invader aufgerufen, um ihnen zu sagen, dass sie sich bewegen sollen. Und jedes Mal, wenn die `Zeichnen()`-Methode von Spiel aufgerufen wird, werden die `Zeichnen()`-Methoden der einzelnen Invader aufgerufen. Sie müssen also die Methoden `Bewegen()` und `Zeichnen()` aufbauen. Außerdem sollten Sie auch eine private Methode namens `InvaderBild()` hinzufügen – diese wird sich als praktisch erweisen, wenn Sie den Invader zeichnen. Sie sollten sie in der Methode `Zeichnen()` aufrufen, um das Feld für das Bild aktuell zu halten.

```
public class Invader {
  private const int HorizontalIntervall = 10;
  private const int VertikalIntervall = 40;
  public enum Typ {
    Wanzen,
    Untertasse,
    Satellit,
    Raumschiff,
    Stern,
  }

  private Bitmap bild;

  public Point Ort { get; private set; }

  public Typ InvaderTyp { get; private set; }

  public Rectangle Fläche { get {
    return new Rectangle(ort, bild.Size); }
  }

  public int Punkte { get; private set; }

  public Invader(Typ invaderTyp, Point ort, int punkte) {
    this.InvaderTyp = invaderTyp;
    this.Ort = ort;
    this.Punkte = punkte;
    bild = InvaderBild(0);
  }

  // Hier kommen weitere Methoden hin.
}
```

Die Konstante HorizontalIntervall bestimmt, wie viele Pixel sich ein Invader bewegt, wenn er nach links oder rechts geht. VertikalIntervall gibt die Anzahl von Pixeln an, die er sinkt, wenn die Formation den Rand des Spielfelds erreicht.

Der Invader nutzt das Enum Typ, um anzugeben, was für eine Art Feindschiff er ist.

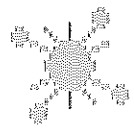
Sehen Sie sich an, was wir mit der Eigenschaft Fläche gemacht haben. Da wir den Ort und die Größe (über das Feld bild) des Invaders kennen, können wir einen Getter schreiben, der das Rectangle für die Fläche berechnet, das er bedeckt ... das bedeutet, dass Sie in LINQ-Abfragen die Contains()-Methode von Rectangle verwenden können, um Schüsse aufzuspüren, die mit einem Invader kollidiert sind.

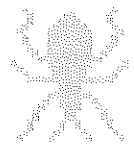
Invader
Ort: Point InvaderTyp: Type Fläche: Rectangle Punkte: int
Zeichnen(g: Graphics, animationszelle: int) Bewegen(Richtung: Richtung)

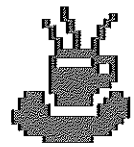
Die Methoden der Invader aufbauen

`Invader` hat drei wichtige Methoden: `Bewegen()`, `Zeichnen()` und `InvaderBild()`. Schauen wir sie uns an:

Es gibt fünf Typen von Invadern, die jeweils vier unterschiedliche Bilder für die Animationszellen haben.











Die Invader bewegen

Zunächst benötigen Sie eine Methode, um die Invader zu bewegen. Das `Spiel` sollte mit einem `Richtung`-Wert eine Richtung übergeben, und dann sollte sich der Invader bewegen. Denken Sie daran, dass sich das `Spiel`-Objekt darum kümmert herauszufinden, ob Invader sich nach unten bewegen oder die Richtung ändern müssen. Darüber muss sich Ihre `Invader`-Klasse nicht den Kopf zerbrechen.

```
public void Bewegen(Richtung Richtung) {
  // Diese Methode muss das Schiff in die
  //   angegebene Richtung bewegen.
}
```

Das Schiff und die richtige Animationszelle zeichnen

`Invader` wissen, wie sie sich selbst zeichnen. Der Invader erhält ein `Graphics`-Objekt, auf das er zeichnen kann, und die Animationszelle, die er verwenden muss, und zeichnet sich dann selbst auf das Spielfeld.

```
public void Zeichnen(Graphics g, int animationszelle)
  // Diese Methode muss das Bild des Invaders
  //   mit der richtigen Animationszelle zeichnen.
}
```

Das erforderliche Invader-Bild finden

Wir werden uns sehr häufig das richtige Bild für eine Animationszelle holen müssen, daher sollte dieser Code vielleicht in eine eigene Methode ausgelagert werden. Erstellen Sie eine `InvaderBild()`-Methode, die die erforderliche `Bitmap` für eine übergebene Animationszelle liefert.

```
private Bitmap InvaderBild(int animationszelle) {
  // Das ist eigentlich nur eine Hilfsmethode, die die
  //  richtige Bitmap für eine Animationszelle liefert.
}
```

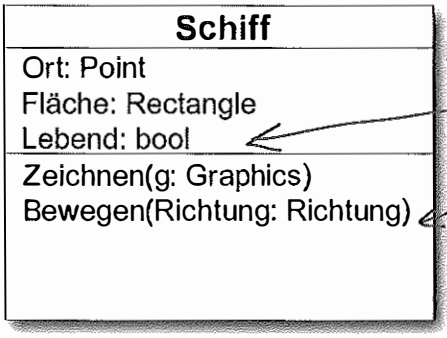
Ein Invader kennt seinen Typ. Geben Sie seiner InvaderBild()-Methode eine Zahl für die Animationszelle, kann er eine Bitmap mit der richtigen Grafik liefern.

Das Schiff des Spielers kann sich bewegen und sterben

Die Klasse `Schiff` hält das Schiff des Spielers nach. Sie ähnelt der Klasse `Invader`, ist aber noch einfacher.

Die Eigenschaften Ort und Fläche sind genau wie die in der Klasse Invader.

Wird das Schiff durch einen Schuss getroffen, setzt das Spiel die Lebend-Eigenschaft des Schiffs auf false. Das Spiel verhindert dann, dass sich die Feinde bewegen, bis die Lebend-Eigenschaft des Spiels wieder auf true gesetzt wird.



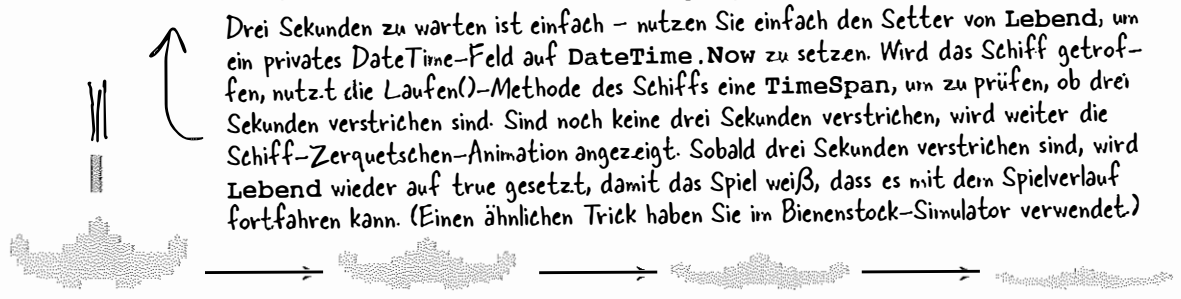
Die Methode Zeichnen() zeichnet einfach das Schiff des Spielers am richtigen Ort – es sei denn, der Spieler stirbt. Dann zeichnet sie ein Bild, das zeigt, wie das Schiff vom Schuss zerquetscht wird.

Die Methode Bewegen() nimmt einen Parameter, einen Richtung-Wert, und bewegt den Spieler in diese Richtung.

Schiff muss in seinen Konstruktor ein Rectangle mit den Grenzen des Spiels übernehmen und sicherstellen, dass sich das Schiff in Bewegen() nicht über diese hinausbewegt.

Das Schiff animieren, wenn es getroffen wird

Die Methode `Zeichnen()` sollte als Parameter ein `Graphics`-Objekt übernehmen. Dann sollte sie die Eigenschaft `Lebend` prüfen. Lebt das Schiff, zeichnet es sich selbst unter Verwendung der Eigenschaft `Ort`. Ist es tot, wird nicht die gewöhnliche Bitmap für das Schiff angezeigt, sondern das private Feld `höheTotesSchiff` verwendet, um das Schiff so zu animieren, als würde es langsam durch den Schuss zerquetscht. Nachdem das Schiff drei Sekunden tot ist, sollte die Eigenschaft `Lebend` wieder auf `true` springen.

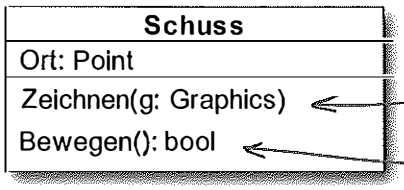


Drei Sekunden zu warten ist einfach – nutzen Sie einfach den Setter von **Lebend**, um ein privates DateTime-Feld auf **DateTime.Now** zu setzen. Wird das Schiff getroffen, nutzt die Laufen()-Methode des Schiffs eine **TimeSpan**, um zu prüfen, ob drei Sekunden verstrichen sind. Sind noch keine drei Sekunden verstrichen, wird weiter die Schiff-Zerquetschen-Animation angezeigt. Sobald drei Sekunden verstrichen sind, wird **Lebend** wieder auf true gesetzt, damit das Spiel weiß, dass es mit dem Spielverlauf fortfahren kann. (Einen ähnlichen Trick haben Sie im Bienenstock-Simulator verwendet.)

```
public void Zeichnen(Graphics g) {
  if (!Lebend) {
      Das Feld höheTotesSchiff zurücksetzen und das Schiff zeichnen.
    } else {
      höheTotesSchiff prüfen. Ist der Wert größer 0, wird das Feld um 1 vermindert
      und das Schiff mit DrawImage() etwas platter gezeichnet.
    }
}
```

Wenn geschossen wurde

`Spiel` hat zwei Listen mit `Schuss`-Objekten: eine für die Schüsse des Spielers, die sich auf dem Bildschirm nach oben bewegen, und eine für die Schüsse der Feinde, die sich auf dem Bildschirm nach unten bewegen. `Schuss` braucht nur ein paar Dinge, um zu funktionieren: einen `Point` für den Ort, eine Methode zum Zeichnen des Schusses und eine Methode zum Bewegen. Hier ist das Klassendiagramm:



Zeichnen() kümmert sich darum, dass kleine Rechteck für den Schuss zu zeichnen. Spiel ruft diese Methode jedes Mal auf, wenn der Bildschirm aktualisiert werden muss.

Bewegen() bewegt den Schuss nach oben oder unten und prüft, ob der Schuss innerhalb der Grenzen des Spiels ist.

Hier ist ein Anfang für die Klasse `Schuss`:

```
public class Schuss {
  private const int bewegungsintervall = 20;
  private const int breite = 5;
  private const int höhe = 15;

  public Point Ort { get; private set; }

  private Richtung Richtung;
  private Rectangle grenzen;

  public Schuss(Point ort, Richtung richtung,
                Rectangle grenzen) {
    this.Ort = ort;
    this.Richtung = richtung;
    this.grenzen = grenzen;
  }

  // Ihr Code kommt hierhin.
}
```

Diese Werte können Sie anpassen, um das Spiel leichter oder schwerer zu machen ... kleineren Schüssen kann man leichter ausweichen, schnelleren Schüssen schwerer.

Der Schuss aktualisiert seinen eigenen Ort in der Methode Bewegen(). Der Ort kann also eine schreibgeschützte automatische Eigenschaft sein.

Richtung ist das Enum, in dem Hoch und Runter definiert werden.

Das Spiel übergibt den Anzeigebereich des Formulars über den grenzen-Parameter des Konstruktors, damit der Schuss weiß, wann er den Bildschirm verlassen hat.

Sie müssen dafür sorgen, dass `Zeichnen()` ein `Graphics`-Objekt übernimmt und den Schuss als gelbes Rechteck zeichnet. Dann sollte `Bewegen()` den Schuss nach oben oder unten bewegen und true liefern, wenn der Schuss immer noch innerhalb der Grenzen des Spielfelds ist.

Blink, blink ... Ihre Aufgabe

Die letzte Klasse, die Sie brauchen, ist die Klasse Sterne. Es gibt 300 Sterne, und diese Klasse kümmert sich um alle. Sie bewirkt, dass bei jedem Aufruf von `Blinken()` fünf verschwinden und fünf neue angezeigt werden.

Erst benötigen Sie allerdings ein `struct` für die einzelnen Sterne:

```
private struct Stern {
  public Point point;
  public Pen pen;

  public Stern(Point point, Pen pen) {
    this.point = point;
    this.pen = pen;
  }
}
```

Jeder Stern hat einen Point (seinen Ort) und einen Pen (für seine Farbe).

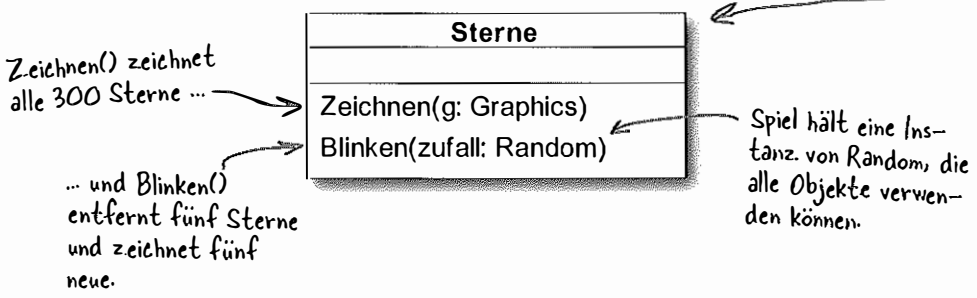
Ein Stern muss nur diese Daten festhalten ... Verhalten braucht er keines.

Ein weiterer Tipp: Beginnen Sie das Projekt mit dem Formular, der Klasse Spiel und der Klasse Sterne. Schauen Sie, ob Sie einen schwarzen Himmel mit blinkenden Sternen zeichnen können. Das verschafft Ihnen ein solides Fundament, um dann zu den anderen Klassen und Methoden überzugehen.

Die Klasse Sterne sollte eine `List<Stern>` zum Speichern von 300 `Stern struct` haben. Sie müssen einen Konstruktor für `Sterne` aufbauen, der diese Liste füllt. Der Konstruktor erhält ein `Rectangle` mit dem Anzeigebereich und eine `Random`-Instanz, die verwendet wird, um die zufälligen Punkte zu erzeugen, an die die Sterne platziert werden.

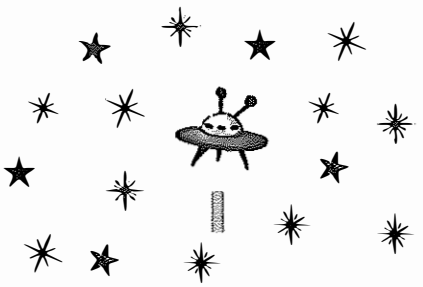
Das Stern-Struct können Sie in Sterne.cs definieren, da nur Sterne es verwenden muss.

Hier ist das Klassendiagramm für `Sterne` mit den anderen Methoden, die Sie benötigen werden:



`Zeichnen()` sollte alle Sterne in der Liste zeichnen, und `Blinken()` sollte zufällig fünf Sterne entfernen und an ihrer Stelle fünf neue Sterne zeichnen.

Sie könnten auch eine `ZufallsPen()`-Methode schreiben, über die Sie jedes Mal, wenn Sie einen Stern erzeugen, leicht eine zufällige Farbe für den Stern abrufen können. Sie sollte eine der fünf Farben liefern, die Sterne haben können, indem sie eine Zufallszahl zwischen 0 und 4 erzeugt und ein passendes `Pen`-Objekt wählt.



Und es ist immer noch mehr zu tun ...

Sie meinen, das Spiel sähe ziemlich gut aus? Mit ein paar weiteren Ergänzungen können Sie das Qualitätsniveau weiter heben:

Schaffen Sie animierte Explosionen

Lassen Sie die Invader explodieren, nachdem sie getroffen wurden, und zeigen Sie dann kurz eine Zahl an, die dem Spieler sagt, was der Invader wert war.

Fügen Sie ein Mutterschiff hinzu

Ab und an kann ein Mutterschiff, das 250 Punkte wert ist, das Spielfeld oben durchqueren. Trifft der Spieler dieses, erhält er einen Bonus.

Fügen Sie Schilde hinzu

Fügen Sie gleitende Schilde hinzu, hinter denen der Spieler sich verbergen kann. Sie können einfache Schilde erzeugen, durch die die Feinde und der Spieler nicht hindurchschießen können. Aber wenn Ihr Spiel so richtig abheben soll, können Sie zusätzlich Schilde erstellen, die Spieler und Invader nach einer gewissen Anzahl von Treffern durchbrechen.

Versuchen Sie, die Schilde bei höheren Levels weniger lang halten zu lassen.

Fügen Sie Sturzbomber hinzu

Erstellen Sie eine besondere Art von Feind, der sich im Sturzflug auf den Spieler stürzt. Ein Sturzflieger sollte aus der Formation ausbrechen, auf den Spieler zufliegen, nach dem Angriff am unteren Rand des Bildschirms entlangfliegen und wieder seine Position einnehmen.

Fügen Sie weiter Waffen hinzu

Rüsten Sie auf! Intelligente Bomben, Laser, Lenkwaffen ... es gibt unendlich viele Arten von Waffen, die Spieler und Invader einsetzen können, um einander anzugreifen. Versuchen Sie, dem Spiel drei weitere Waffen hinzuzufügen.

Erweitern Sie die grafische Darstellung

Auf http://examples.oreilly.de/german_examples/hfcsharpger finden Sie weitere Grafikdateien für einfache Schilde, ein Mutterschiff und mehr. Wir bieten Ihnen würfelige Pixelgrafiken, um einen stilisierten 80er-Look zu erzeugen. Können Sie eigene Grafiken gestalten, um dem Spiel einen neuen Look zu geben?

Bei einem guten Klassendesign sollten Sie die Grafiken mit minimalen Code-änderungen auswechseln können.

Hier können Sie sich beweisen! Haben Sie eine coole Version des Spiels entwickelt? Gehen Sie ins Head First C#-Forum und zeigen Sie sich: www.headfirstlabs.com/books/hfcsharp/.

Anhang A: Was übrig bleibt

Die Top-5 der Themen, die es nicht ins Buch geschafft haben



Der Spaß fängt gerade erst an!

Wir haben Ihnen viele wunderbare Werkzeuge gezeigt, mit denen Sie mit C# richtig **mächtige Software** aufbauen können. Aber es war unmöglich, **alle Werkzeuge, Technologien und Techniken** in dieses Buch einzuschließen – so viele Seiten hat es einfach nicht. Wir mussten einige *sehr harte Entscheidungen* in Bezug darauf treffen, was wir mit aufnehmen und was wir weglassen. Hier sind einige Themen, die es nicht geschafft haben. Aber auch wenn wir zu ihnen nicht gekommen sind, denken wir trotzdem, dass sie **wichtig und nützlich** sind, und möchten Ihnen einen kleinen Ausblick auf sie bieten.

1. LINQ to XML

XML – oder E**X**tensible **M**arkup **L**anguage – ist ein Format für Dateien und Datenstreams, das komplexe Daten als Text darstellt. Das .NET Framework bietet Ihnen einige mächtige Werkzeuge zum Erstellen, Laden und Speichern von XML-Dateien. Halten Sie XML-Daten einmal in den Händen, können Sie sie mit LINQ abfragen. Geben Sie am Anfang der Datei »`using System.Xml.Linq;`« an und nutzen Sie diese Methode – sie generiert ein XML-Dokument mit Stammkundendaten von Sternback.

```
private static XDocument GetSternbackDaten() {
    XDocument dok = new XDocument(
       new XDeclaration("1.0", "utf-8", "yes"),
       new XComment("Sternback Stammkundendaten"),
       new XElement("SternbackDaten",
           new XAttribute("FilialienName", "Im Park"),
           new XAttribute("Ort", "Berlin-Mitte"),
           new XElement("Person",
               new XElement("PersonenDaten",
                   new XElement("Name", "Jana Venus"),
                   new XElement("PLZ", 11215)),
               new XElement("Lieblingsgetränk", "Schoko-Macchiato"),
               new XElement("GeldAusgegeben", 255),
               new XElement("Besuche", 50)),
           new XElement("Person",
               new XElement("PersonenDaten",
                   new XElement("Name", "Lisa Nilson"),
                   new XElement("PLZ", 11238)),
               new XElement("Lieblingsgetränk", "Doppel-Cappuccino"),
               new XElement("GeldAusgegeben", 150),
               new XElement("Besuche", 35)),
           new XElement("Person",
               new XElement("PersonenDaten",
                   new XElement("Name", "Frank Matt"),
                   new XElement("PLZ", 11217)),
               new XElement("Lieblingsgetränk", "Zitronengras-Tee"),
               new XElement("GeldAusgegeben", 75),
               new XElement("Besuche", 15)),
           new XElement("Person",
               new XElement("PersonenDaten",
                   new XElement("Name", "Hans Neu"),
                   new XElement("PLZ", 11217)),
               new XElement("Lieblingsgetränk", "Banana-Split im Becher"),
               new XElement("GeldAusgegeben", 60),
               new XElement("Besuche", 10)),
           new XElement("Person",
               new XElement("PersonenDaten",
                   new XElement("Name", "Sarah Kalter"),
                   new XElement("PLZ", 11215)),
               new XElement("Lieblingsgetränk", "Boring Coffee"),
               new XElement("GeldAusgegeben", 110),
               new XElement("Besuche", 15))));
    return dok;
}
```

XML-Dateien laden und speichern

Ein XDocument-Objekt können Sie auf die Konsole schreiben oder in einer Datei speichern, und Sie können eine XML-Datei in ein XDocument-Objekt laden:

```
XDocument dok = GetSternbackDaten();
Console.WriteLine(dok.ToString());
dok.Save("SternbackDaten.xml");
XDocument anderesDok = XDocument.Load("SternbackDaten.xml");
```

Die Load()- und Save()-Methoden von XDocument lesen oder schreiben XML-Dateien. Seine ToString()-Methode liefert den gesamten Inhalt als einen großen XML-String.

Ihre Daten abfragen

Hier ist eine einfache LINQ-Abfrage, die die Sternback-Daten über das XDocument abfragt:

```
var daten = from element in dok.Descendants("Person")
    select new { getränk = element.Element("Lieblingsgetränk").Value,
        betrag = element.Element("GeldAusgegeben").Value,
        plz = element.Element("PersonenDaten").Element("PLZ").Value };
foreach (var p in daten)
    Console.WriteLine(p.ToString());
```

Die Methode Descendants() liefert eine Referenz auf ein Objekt, das Sie gleich in LINQ einstöpseln können.

Sie wissen bereits, dass Sie mit LINQ Methoden aufrufen und als Teil der Abfrage verwenden können. Mit der Methode Element() funktioniert das ziemlich gut.

Und Sie können auch komplexere Abfragen ausführen:

```
var plzGruppen = from element in dok.Descendants("Person")
    group element.Element("Lieblingsgetränk").Value
    by element.Element("PersonenDaten").Element("PLZ").Value
        into plzGruppe
        select plzGruppe;
foreach (var gruppe in plzGruppen)
    Console.WriteLine("{0} Lieblingsgetränk in {1}",
                      gruppe.Distinct().Count(), gruppe.Key);
```

Element() liefert ein XElement-Objekt, dessen Eigenschaften Sie nutzen können, um bestimmte Werte in Ihrem XML-Dokument zu prüfen.

Daten aus einem RSS-Feed lesen

Mit LINQ to XML können Sie einige sehr mächtige Dinge tun. Hier ist eine einfache Abfrage **zum Lesen von Artikeln in unserem Blog**:

```
XDocument blog = XDocument.Load("http://www.stellman-greene.com/feed");
Console.WriteLine(blog.Element("rss").Element("channel").Element("title").Value);
var einträge = from eintrag in blog.Descendants("item")
    select new { Title = eintrag.Element("title").Value,
        Date = eintrag.Element("pubDate").Value};
foreach (var eintrag in einträge)
    Console.WriteLine(eintrag.ToString());
```

Die XDocument.Load()-Methode bietet mehrere Überladungen. Diese zieht die XML-Daten aus einer URL.

Packen Sie einen Button auf ein Formular, achten Sie darauf, dass die Datei >>using System.Xml.Linq;<< angibt, geben Sie diese Abfrage in den Event-Handler ein und schauen Sie sich an, was der Code auf der Konsole ausgibt.

Wir haben die URL unseres Blogs, Building Better Software, verwendet: http://www.stellman-greene.com/

2. Umgestalten (Refactoring)

Umgestalten bedeutet, die Struktur von Code zu ändern, ohne sein Verhalten zu ändern. Wenn Sie komplexe Methoden schreiben, sollten Sie gelegentlich zurücktreten und schauen, wie Sie sie ändern könnten, damit sie leichter zu verstehen sind. Glücklicherweise sind in die IDE einige sehr nützliche Umgestaltungswerkzeuge eingebaut. Es gibt alle möglichen Arten von Codeänderungen, die Sie durchführen können – hier sind einige, die wir häufig verwenden.

Eine Methode extrahieren

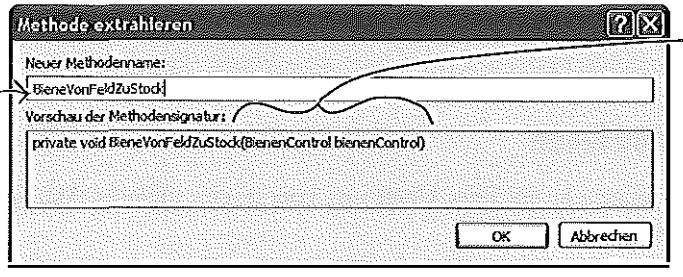
Als wir den Code für den steuerelementbasierten Renderer für Kapitel 13 schrieben, haben wir ursprünglich diese foreach-Schleife eingeschlossen:

```
foreach (Biene biene in welt.Bienen) {
    bienenControl = BienenControlSuchen(biene);
    if (biene.InStock) {
        if (feldForm.Controls.Contains(bienenControl)) {
            feldForm.Controls.Remove(bienenControl);
            bienenControl.Size = new Size(40, 40);
            stockForm.Controls.Add(bienenControl);
            bienenControl.BringToFront();
    } else if (stockForm.Controls.Contains(bienenControl)) {
        stockForm.Controls.Remove(bienenControl);
        bienenControl.Size = new Size(20, 20);
        feldForm.Controls.Add(bienenControl);
        bienenControl.BringToFront();
    }
    bienenControl.Location = biene.Ort;
}
```

Diese vier Zeilen bewegen ein BienenControl vom Feld in den Stock.

Und diese vier Zeilen bewegen ein BienenControl vom Stock aufs Feld.

Einer unserer Gutachter, Joe Albahari, wies uns darauf hin, dass das etwas schlecht lesbar sei. Er schlug vor, die **zwei Vier-Zeilen-Blöcke in Methoden herauszuziehen**. Wir wählten also den ersten Block aus, klickten mit rechts darauf und wählten »Umgestalten→Methode extrahieren...« – es sprang dieses Fenster auf:



Den Namen für die neue Methode haben wir eingegeben. Wir haben uns für den passenden Namen BieneVonFeldZuStock() entschieden.

Die IDE hat den von uns ausgewählten Code untersucht, dabei festgestellt, dass er eine BienenControl-Variable namens bienenControl verwendet, und diese als Parameter hinzugefügt.

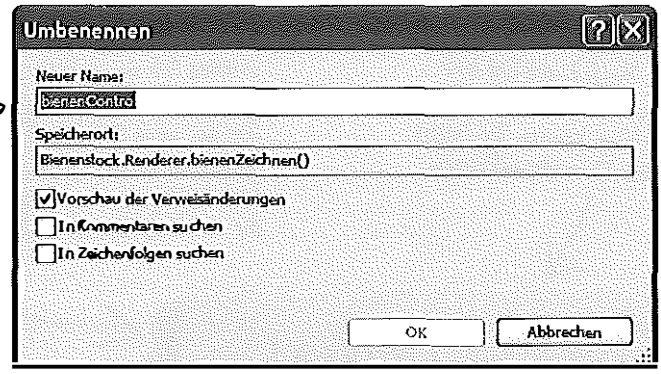
Dann extrahierten wir die anderen vier Zeilen in eine Methode namens BieneVonStockZuFeld(). So sah die foreach-Schleife anschließend aus – sie ist viel besser lesbar:

```
foreach (Biene biene in welt.Bienen) {
    bienenControl = BienenControlSuchen(biene);
    if (biene.InStock) {
        if (feldForm.Controls.Contains(bienenControl))
            BieneVonFeldZuStock(bienenControl);
    } else if (stockForm.Controls.Contains(bienenControl))
        BieneVonStockZuFeld(bienenControl, biene);
    bienenControl.Ort = biene.Ort;
}
```

Eine Variable umbenennen

In Kapitel 3 haben wir erklärt, wie intuitive Namen für Klassen, Methoden, Felder und Variablen die Lesbarkeit Ihres Codes verbessern. Die IDE kann Ihnen wirklich helfen, wenn es darum geht, den Dingen in Ihrem Code Namen zu geben. Klicken Sie einfach auf eine Klasse, Variable, Eigenschaft, Methode, Konstante usw. – also auf fast alles, das Sie mit einem Namen versehen können – und wählen Sie »Umgestalten→Umbenennen«. Das haben wir mit dem »bienenControl« im Code für den Simulator gemacht. Es wurde dann dieses Fenster geöffnet:

In diesem Fenster können Sie einen neuen Namen für das Element wählen. Würden wir stattdessen z.B. >>Bobbo<< verwenden, würde die IDE den Code durchgehen und jedes Vorkommen in >>Bobbo<< ändern.



Die IDE macht beim Umbenennen ganze Arbeit. Nennen Sie eine Klasse um, ändert sie jede Anweisung, die sie instantiiert oder nutzt. Sie können an beliebiger Stelle des Codes auf ein beliebiges Vorkommen des Namens klicken. Die IDE führt die Änderung an allen Punkten Ihres Programms durch.

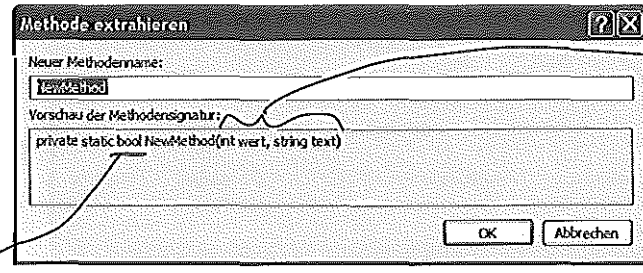
Eine Testbedingung konsolidieren

Hier ist ein sauberer Weg, das Feature »Methode extrahieren« zu verwenden. Öffnen Sie ein Programm, fügen Sie ihm einen Button hinzu und stecken Sie in seinen Event-Handler diesen Code:

```
private void button1_Click(object sender, EventArgs e) {
    int wert = 5;
    string text = "Hallo du";
    if (wert == 36 || text.Contains("du"))
        MessageBox.Show("Buh!");
}
```

Markieren Sie alles in der if-Anweisung: `wert == 36 || text.Contains("du")`. Klicken Sie dann mit rechts darauf und wählen Sie »Umgestalten→Methode extrahieren...«. Dann wird dieses Fenster eingeblendet:

Jede Testbedingung wird zu einem bool ausgewertet. Die IDE erzeugt also eine Methode, die einen bool zurückliefert, und ersetzt die Testbedingung durch einen Aufruf dieser Methode.



Der Ausdruck nutzt die zwei Variablennamen wert und text. Die IDE hat diese also als Methodenparameter mit diesen Namen hinzugefügt.

Das verbessert nicht nur die Lesbarkeit Ihres Codes, sondern liefert Ihnen auch eine Methode, die Sie an anderer Stelle wiederverwenden können!

3. Einige unserer Lieblingskomponenten in der Toolbox

In diesem Buch ging es darum, C# zu lernen und nicht alle Vor- und Nachteile der Komponenten, die mit .NET ausgeliefert werden. Wir haben allerdings unsere Vorlieben und dachten, wir sollten sie mit Ihnen teilen.

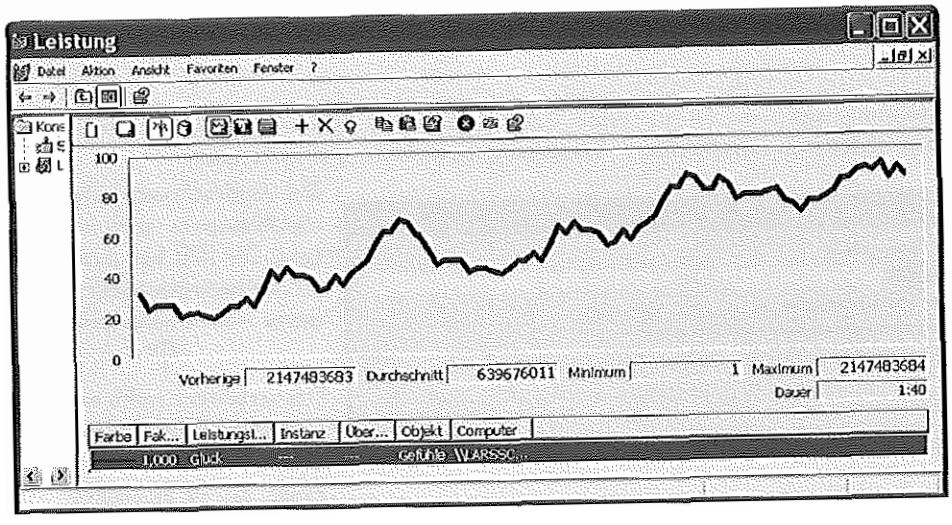
BackgroundWorker ist eine dieser nicht visuellen Komponenten (wie Timer), die Sie auf Ihr Formular ziehen können. Sie macht etwas wirklich Praktisches – mit ihr können Sie leicht Anwendungen mit mehreren Threads aufbauen.

Ziehen Sie sie einfach auf Ihr Formular und klicken Sie doppelt darauf (oder instantiieren Sie sie, wenn Sie möchten – aber vergessen Sie nicht, sie zu entsorgen!). Fügen Sie dann einen Event-Handler hinzu, der die Arbeit ausführt, die im Hintergrund laufen soll.



Rufen Sie darauf die Methode RunWorkerAsync() auf, wird die Methode DoWork() angestoßen – aber der entscheidende Punkt ist, dass sie in einem anderen Thread läuft. Das bedeutet, dass sie ihre Arbeit zeitgleich mit den anderen Dingen verrichtet kann, die Ihr Programm ausführt.

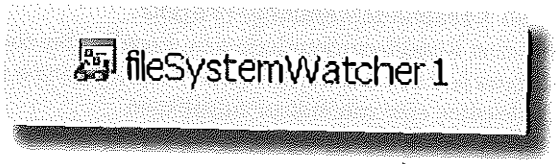
Seinen Fortschritt kann Ihr Prozess Ihnen berichten, indem er die ReportProgress()-Methode von BackgroundWorker aufruft – dieses bewirkt, dass das ProgressChanged-Event abgesetzt wird.



Diesen »Glück«-Zähler in der Kategorie »Emotionen« haben wir mit einem PerformanceCounter eingerichtet und mit Daten gefüttert.

FileSystemWatcher überwacht das Dateisystem und achtet darauf, ob sich darin etwas ändert.

Die Eigenschaft Filter setzen Sie auf die Dateitypen, die überwacht werden sollen (z.B. *.txt), die Eigenschaft Path auf das Verzeichnis, das überwacht werden soll. Über IncludeSubdirectories können Sie angeben, ob auch Unterverzeichnisse überwacht werden sollen.



Haben Sie ihn eingerichtet, überwacht er für Sie das Verzeichnis auf neue oder geänderte Dateien. Sobald eine Datei hinzugefügt, geändert oder gelöscht wird, setzt er sein Changed-Event ab. Außerdem gibt es auch noch die Events Created, Deleted und Renamed, über die spezifische Ereignisse überwacht werden können.

Setzt der FileSystemWatcher ein Event ab, übergibt er Informationen zur geänderten (oder hinzugefügten oder gelöschten) Datei über ein FileSystemEventArgs-Objekt.

Wenn Sie ein Programm schreiben, das kontinuierlich läuft, ist es hilfreich, es zu überwachen. Windows wird mit einem raffinierten Werzeug namens Performance Monitor (perfmon.exe) ausgeliefert, über das Sie Prozesse überwachen können.

Mit der Komponente PerformanceCounter können Sie Informationen zu Ihrem Programm für die Windows Leistungsüberwachung bereitstellen. Nutzen Sie Increment() und Decrement() oder setzen Sie die Eigenschaft RawValue. Sobald Sie das tun, können Sie die Daten in der Leistungsüberwachung beobachten.



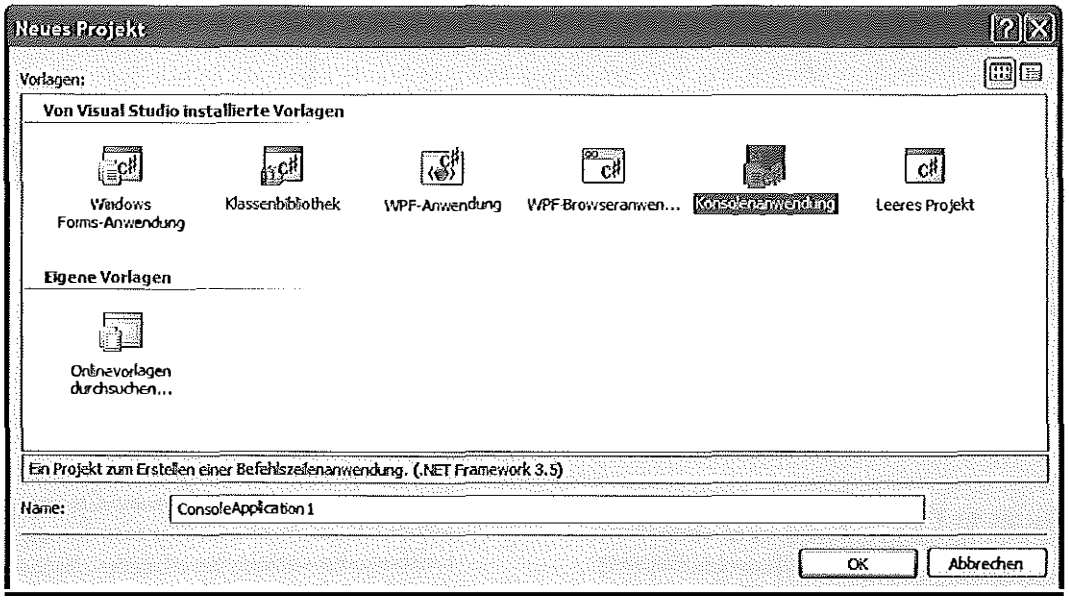
Windows hält seine Leistungszähler in Kategorien. Sie müssen also eine Kategorie erstellen – das können Sie mit Methoden in System.Diagnostics tun. Dann müssen Sie Ihren PerformanceCounter nur noch mit der von Ihnen erstellten Kategorie verbinden und damit beginnen, die Diagnoseinformationen an Ihren Liebling zu senden.

4. Konsolenanwendungen

Die meisten C#-Bücher beginnen mit Konsolenanwendungen. Das fanden wir langweilig. Es ist deutlich befriedigender, Programme aufzubauen, die gut aussehen ... oder, besser gesagt, die überhaupt nach etwas aussehen. Und das tut ein Konsolenprogramm nicht. Aber manchmal muss man Konsolenanwendungen schreiben. Glücklicherweise ist das ziemlich einfach. Es geht so:

1 Erstellen Sie ein Konsolenanwendung-Projekt.

Jedes Projekt kann eine Konsolenanwendung sein. Erstellen Sie ein Projekt, wählen Sie im Menü Projekt »Eigenschaften« und ändern Sie den »Ausgabetyp« in »Konsolenanwendung«. Aber es ist leichter, eine von null an aufzubauen.



2 Die IDE fügt nur eine Datei hinzu - Program.cs.

Diese hat einen leeren Einstiegspunkt ... und das war's.

```
class Program
{
    static void Main(string[] args)
    {

    }
}
```

Hier ist ein einfaches Projekt für Sie: Nehmen Sie den Hex-Dumper aus Kapitel 9 und wandeln Sie ihn in eine Konsolenanwendung um. Lassen Sie ihn einen Dateinamen lesen, den Sie von der Kommandozeile übergeben, und geben Sie ihren Inhalt als Hex-Dump aus. Lassen Sie ihn die Daten von der Standardeingabe nehmen (mit Console.ReadLine()) und geben Sie diese Daten als Hex-Dump aus. Schlagen Sie dann den Unix-Befehl »od« nach und probieren Sie, diesen in C# nachzubauen.

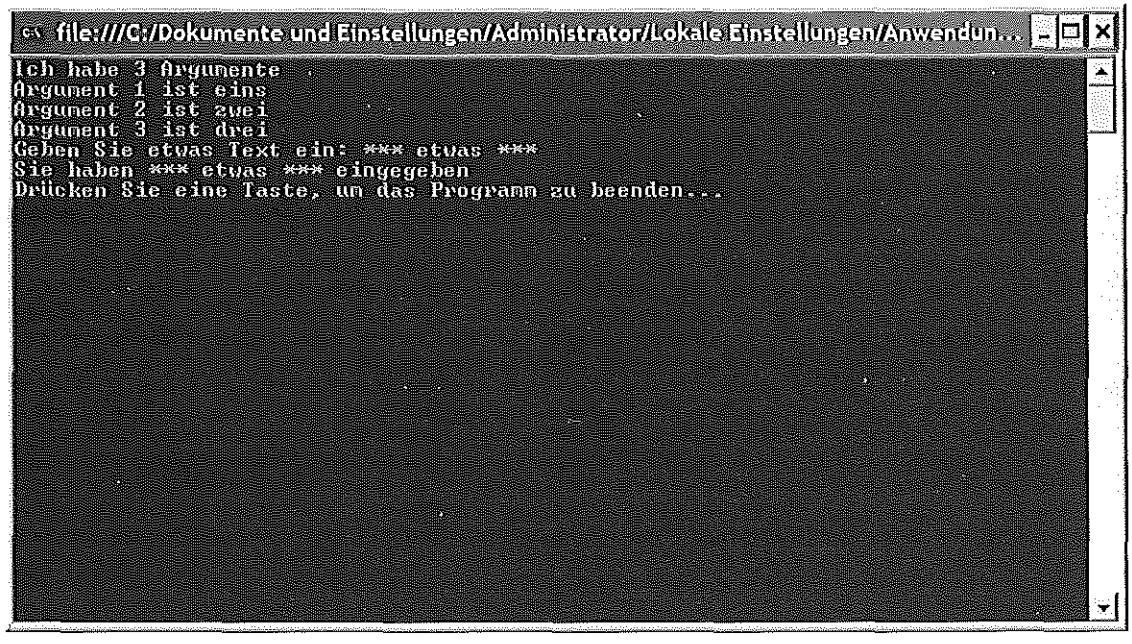
3 Nutzen Sie den Parameter args für Kommandozeilenargumente.

Ihr Einstiegspunkt nimmt einen Parameter an, ein String-Array namens `args`, das die Kommandozeilenargumente enthält. Sie wissen bereits, wie Sie die Methode Console.WriteLine() verwenden – es gibt noch weitere nützliche Methoden für die Konsole einschließlich ReadLine() und ReadKey().

```
class Program {
    static void Main(string[] args) {
        Console.WriteLine("Ich habe {0} Argumente", args.Length);
        for (int i = 1; i <= args.Length; i++)
            Console.WriteLine("Argument {0} ist {1}", i, args[i - 1]);
        Console.Write("Geben Sie etwas Text ein: ");
        string input = Console.ReadLine();
        Console.WriteLine("Sie haben {0} eingegeben", input);
        Console.WriteLine("Drücken Sie eine Taste, um das Programm zu beenden...");
        Console.ReadKey();
    }
}
```

4 Debuggen Sie Ihr Programm in einem Konsolenfenster.

Wenn Sie Ihr Programm debuggen, öffnet die IDE ein Konsolenfenster. Die Methoden ReadLine() und ReadKey() erhalten ihre Eingabe aus diesem Fenster – geben Sie einfach etwas darin ein. Und statt in das Ausgabefenster zu schreiben, schreibt eine Konsolenanwendung in das Konsolenfenster. Die Kommandozeilenargumente können Sie auf der »Debuggen«-Seite des Eigenschaften-Fensters des Projekts setzen.



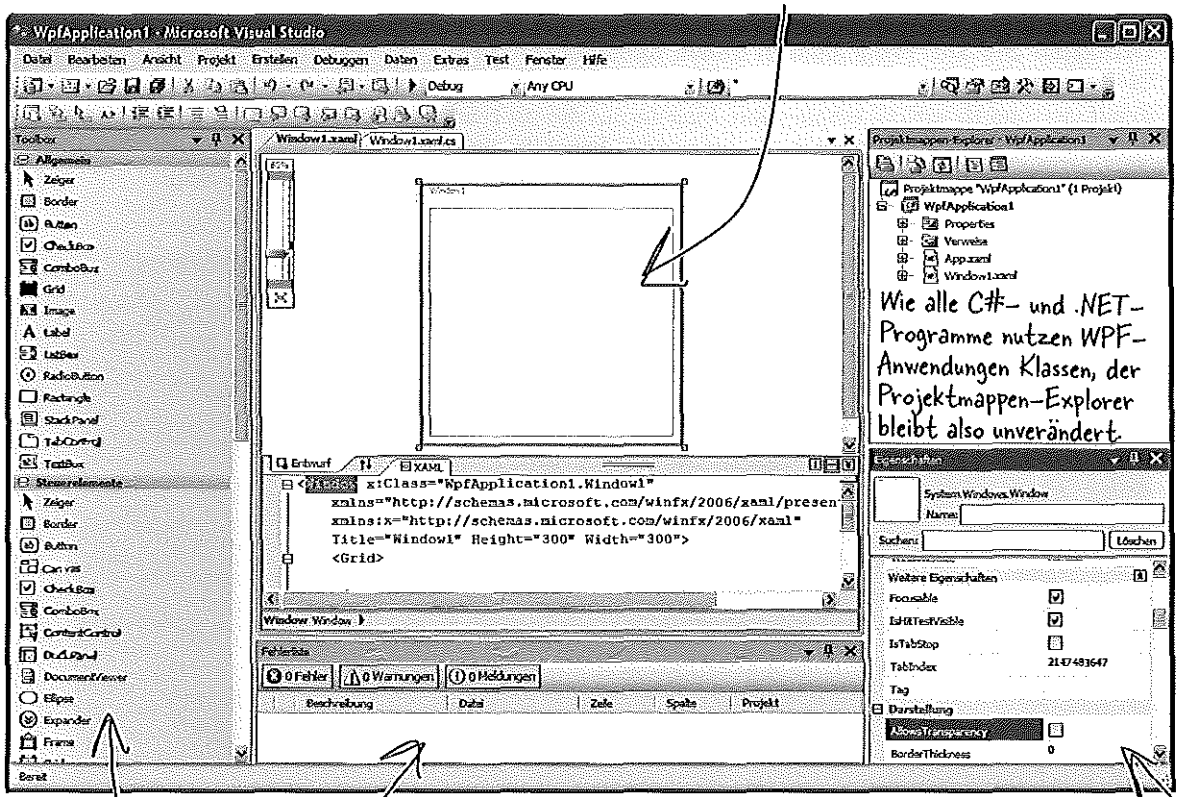
5. Windows Presentation Foundation

Das Windows Presentation Foundation oder WPF ist Microsofts kommende Plattform für den Aufbau grafischer Anwendungen. Es ist ziemlich beeindruckend – es bietet ein XML-basiertes Layout, skalierbare Steuerelemente, ein vollständig neues Steuerelementsystem, 2-D- und 3-D-Grafiken und -Animationen, Textfluss- und Dokumentformatierungen, und es gibt sogar ein plattformübergreifendes Browser-Plug-in, das es einsetzt.

Aber obgleich WPF eine richtig coole und sehr mächtige Technologie ist, stellt es kein besonders geeignetes Werkzeug dar, um C# zu erklären. Und das war schließlich unser Ziel – C#-Konzepte so schnell und einfach wie möglich in Ihren Kopf zu kriegen.

Nehmen Sie sich einen Moment Zeit und erstellen Sie eine neue WPF-Anwendung. Erzeugen Sie einfach mit der IDE ein neues Projekt, aber keine Windows-Anwendung. **Wählen Sie stattdessen WPF-Anwendung.** Ihnen sollte gleich der Unterschied in der IDE auffallen.

Der größte Unterschied, den Sie sehen sollten, ist, dass der Form-Designer ganz anders aussieht als früher. Wir werden ihn uns gleich ansehen.



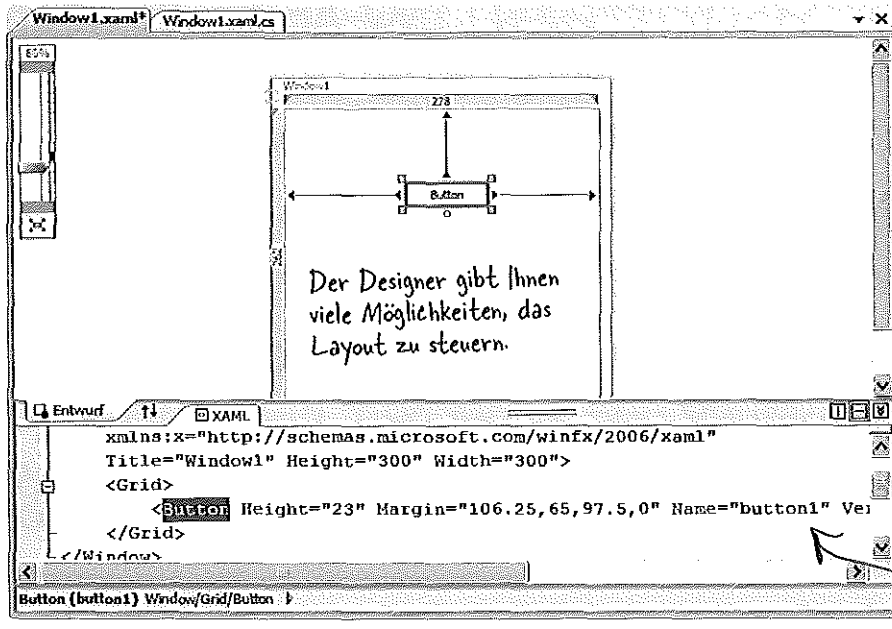
Sehen Sie sich die Toolbox genau an – sie enthält einen ganz anderen Satz Steuerelemente.

Hier sind die vertrauten Fenster mit der Fehlerliste und der Ausgabe, die Sie bereits verwendeten.

Die Eigenschaften sehen ganz anders aus. Das liegt daran, dass Sie hier die Attribute in einer XAML-Datei ändern und nicht mehr die Eigenschaften von Objekten.

Ziehen Sie einen Button aus der Toolbox auf das Formular. Wäre das eine Windows-Anwendung, würde die IDE Form1.Designer.cs Code für das neue Steuerelement hinzufügen. Aber WPF funktioniert anders – es nutzt die XML-basierte Sprache XAML, um das Layout der Benutzerschnittstelle zu definieren.

Ziehen Sie diesen Schieber hoch und runter, um rein- und rauszuzoomen. Auch wenn Sie weit hineinzoomen, sieht Ihre Benutzerschnittstelle immer noch gut aus – sie wird nicht pixelig.



XAML steht für »Extensible Application Markup Language«. Es ist eine XML-basierte Sprache, die WPF-Anwendungen nutzen, um festzulegen, wohin die Steuerelemente und andere UI-Elemente kommen.

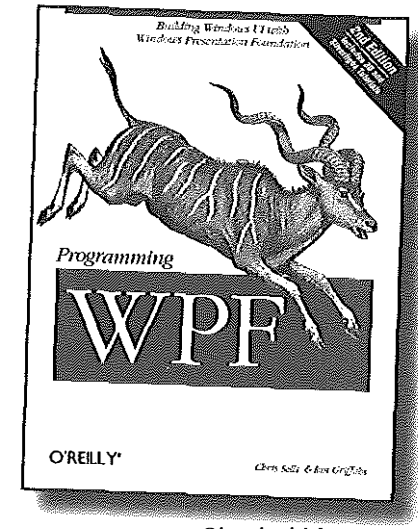
Die IDE bietet einen sehr mächtigen XML-Editor, der für die Arbeit mit XAML optimiert ist.

Gehen Sie in den XML-Editor und fügen Sie einen zweiten Button hinzu, indem Sie die fett gedruckte Zeile unten in den XAML-Editor eingeben. Sehen Sie, wie Ihnen IntelliSense hilft, all die XML-Tags einzugeben?

```
<Grid>
    <Button Height="23" Margin="98,43,105,0" Name="button1"
                  VerticalAlignment="Top" Click="button1_Click">Button</Button>
    <Button Height="23" Margin="5,5,100,20" Name="button2"
                  VerticalAlignment="Top" Click="button2_Click">Noch ein Button</Button>
</Grid>
```

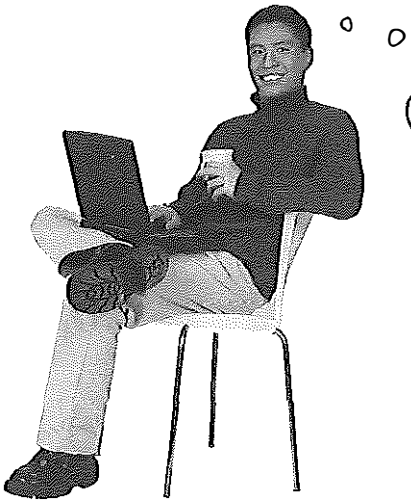
Wenn Sie zum »**Click="*button2_Click*"**«-Teil der Zeile kommen, sollten Sie den Namen des Event-Handlers nicht eingeben. Nutzen Sie stattdessen das IntelliSense-Fenster, um der IDE zu sagen, dass sie einen neuen Event-Handler hinzufügen soll. Sobald Sie die Zeile abschließen, erscheint der zweite Button im Entwurf. Wechseln Sie nun zur Registerkarte Window1.xaml.cs, finden Sie die neue button2_Click-Methode.

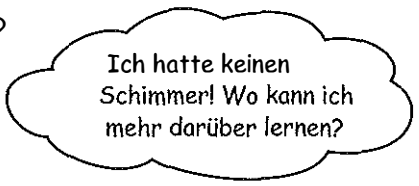
Das ist alles, was wir hier über WPF und XAML erzählen möchten. Aber nachdem Sie jetzt die Mittel haben, in die WPF-Programmierung einzusteigen, empfehlen wir Ihnen ausdrücklich, einen Blick in ***Programming WPF*** von Chris Sells und Ian Griffiths zu werfen. Es ist auf der O'Reilly-Website verfügbar: **http://www.oreilly.de/**.



Wussten Sie, dass Sie mit C# und dem .NET Framework ...

- Mit fortgeschrittenen LINQ-Abfragen noch viel mehr Macht über Ihre Daten erhalten? Objekte in eine XML-Datei serialisieren können?
- Mit eingebauten Klassen auf Websites und andere Netzwerkressourcen zugreifen können?
- Ihren Programmen eine fortgeschrittene Verschlüsselung und Sicherheit hinzufügen können?
- Komplexe Anwendungen mit Multi-Threading erstellen können?
- Ihre Klassen so verteilen können, dass andere Sie nutzen können?
- Mit regulären Ausdrücken ausgefeilte Suchoperationen auf Texten vornehmen können?

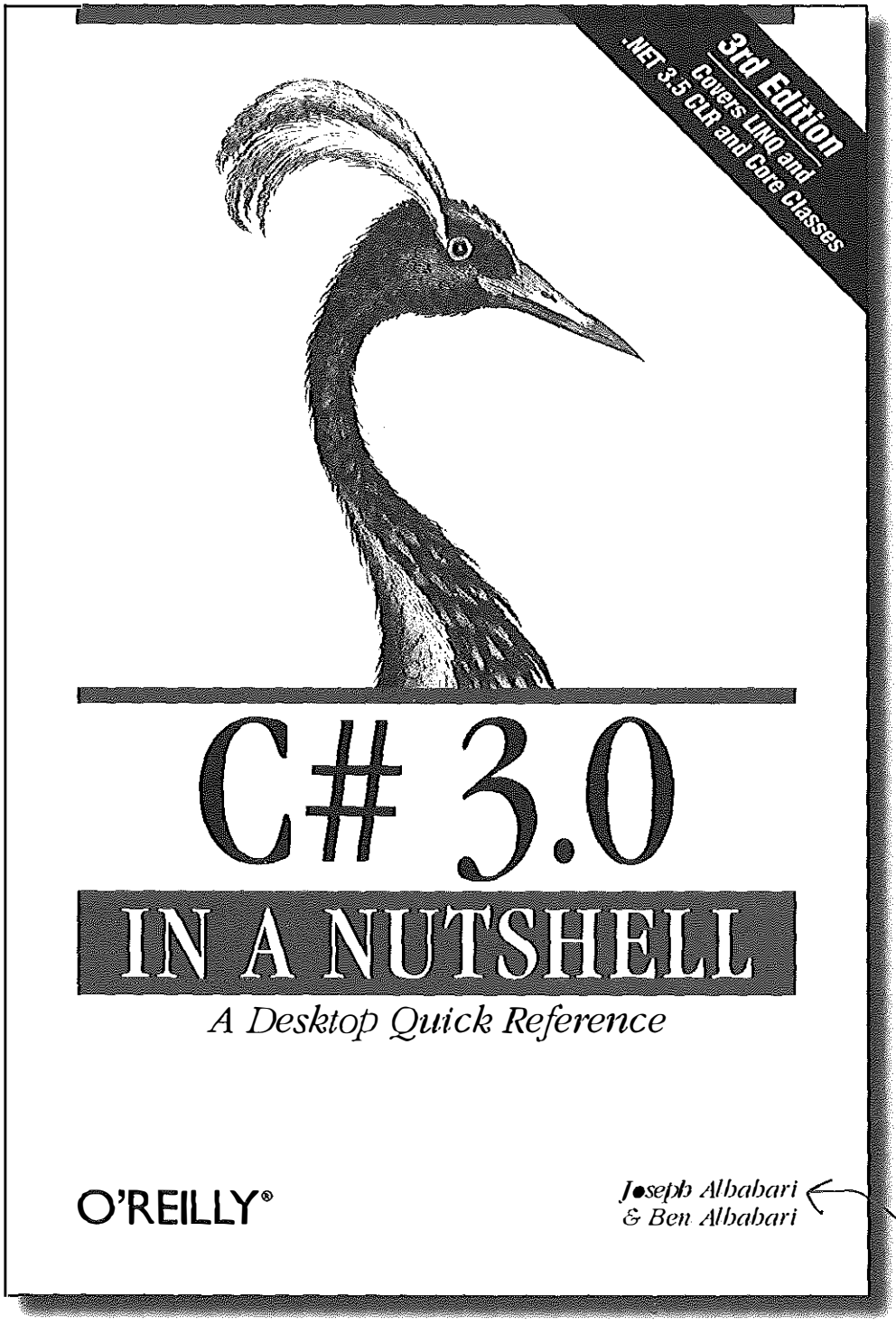




Es gibt ein wunderbares Buch, das all das erklärt!

Es heißt ***C# 3.0 in a Nutshell*** und wurde von Joseph Albahari und Ben Albahari geschrieben. Es ist ein ausführliches Handbuch zu allem, was C# zu bieten hat. Sie werden fortgeschrittenere Features der Sprache C# kennenlernen sowie die grundlegenden Klassen und Werkzeuge des .NET Framework, und Sie werden **noch mehr darüber erfahren**, was bei C# hinter den Kulissen tatsächlich abläuft.

Gehen Sie zu **http://www.oreilly.de/**.



Joseph Albahari hat uns sehr geholfen, indem er dieses Buch einer gründlichen technischen Überprüfung unterzogen hat. Vielen Dank für all deine Hilfe, Joe!