ASP.NET lernen

Die Lernen-Reihe

In der Lernen-Reihe des Addison-Wesley Verlages sind die folgenden Titel bereits erschienen bzw. in Vorbereitung:

André Willms
C-Programmierung lernen
432 Seiten, ISBN 3-8273-1405-4

André Willms
C++-Programmierung lernen
408 Seiten, ISBN 3-8273-1342-2

Guido Lang, Andreas Bohne
Delphi 5 lernen
432 Seiten, ISBN 3-8273-1571-9

Walter Herglotz
HTML lernen
323 Seiten, ISBN 3-8273-1717-7

Judy Bishop
Java lernen
636 Seiten, ISBN 3-8273-1794-0

R. Allen Wyke, Donald B. Thomas
Perl 5 lernen
ca. 400 Seiten, ISBN 3-8273-1650-2

Michael Ebner
SQL lernen
336 Seiten, ISBN 3-8273-1515-8

René Martin
XML und VBA lernen
336 Seiten, ISBN 3-8273-1952-8

René Martin
VBA mit Word 2002 lernen
393 Seiten, ISBN 3-8273-1897-1

René Martin
VBA mit Office 2000 lernen
576 Seiten, ISBN 3-8273-1549-2

Dirk Abels
Visual Basic 6 lernen
425 Seiten, ISBN 3-8273-1371-6

Patrizia Sabrina Prudenzi
VBA mit Excel 2000 lernen
512 Seiten, ISBN 3-8273-1572-7

PEARSON Education

ADDISON-WESLEY

Jörg Krause

ASP.NET lernen

Anfangen, anwenden, verstehen

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

**Ein Titeldatensatz für diese Publikation ist bei
Der Deutschen Bibliothek erhältlich.**

Umwelthinweis:
Dieses Produkt wurde auf chlorfrei gebleichtem Papier gedruckt.
Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem und recyclingfähigem PE-Material.

10 9 8 7 6 5 4 3 2 1

05 04 03 02

ISBN 3-8273-2018-6

Einbandgestaltung:	Barbara Thoben, Köln
Illustration:	Lisa Herzel
Lektorat:	Sylvia Hasselbach, shasselbach@pearson.de
Korrektorat:	Haide Fiebeler-Krause, Berlin
Herstellung:	Ulrike Hempel, uhempel@pearson.de
Satz:	mediaService, Siegen
Druck und Verarbeitung:	Bosch Druck, Ergolding

Printed in Germany

Inhaltsverzeichnis

lernen

Vorwort

Steve Balmer, CEO der Firma Microsoft, ist bekannt für bühnenreife Auftritte. Der 100-Kilo-Mann tanzt dabei schon mal zur Musik von Gloria Estefan über das Podium oder schreit auf seine Mannen ein, als sei er von Sinnen. Schaut man weniger auf die Show und hört mehr auf die Worte, ist ein System erkennbar. Was auch immer er zu sagen hat, es verkörpert die Microsoft-Philosophie.

Die Welt der Computer – aus wirtschaftlicher Sicht betrachtet – wird durch die Anzahl und Qualität der verfügbaren Anwendungen bestimmt. Dies hat ganz klar zum Erfolg von Windows geführt, denn lange Zeit war es offensichtlich nicht die Produktqualität. Der letzte Schrei von Balmer ging dann auch etwa folgendermaßen: „Developer, Developer, Developer, usw.“. Im Zentrum aller Bemühung steht also fortan der Softwareentwickler, übrigens eine Gattung Mensch, die mit dem Leser dieses Buches meist übereinstimmt. Sie sind also nicht nur Mittelpunkt von Microsoft, sondern auch von Verlag und Autor.

Hinter all den Bemühungen steckt natürlich ein Hintergedanke. Gute Anwendungen schaffen die Basis für das passende Betriebssystem. Sie sind auch eine Basis, um damit gutes Geld zu verdienen. Und wenn der Softwareentwickler oder dessen Arbeit- oder Auftraggeber Geld verdienen, dann werden diese wiederum Geld investieren – in Entwicklungswerkzeuge, MSDN-Abos oder neue Computer mit neuen Betriebssystemen. Eine ganze einfache Rechnung also, die Microsoft immerhin zum größten Softwareunternehmen der Welt gemacht haben.

Wenn nun etwas neues produziert wird, das das Kerngeschäft betrifft, die Hauptzielgruppe trifft, für das Überleben und den Erfolg des Unternehmens von so elementarer Bedeutung ist, kann man davon nicht extrem viel erwarten?

Das Neue hat den Namen .Net bekommen und es ist tatsächlich ein Produkt, das extrem viel bietet. Bevor Sie sich voll in die Programmierung

stürzen und versuchen, all die neuen, komplexen und verwirrenden Sachverhalte zu ordnen, lassen Sie mich nochmals auf Philosophien zurückkommen. Das Verständnis dafür hilft tatsächlich, den steinigen Weg zum Erfolg leichter zu nehmen.

Stellen Sie sich ein Koordinatensystem vor, auf dessen X-Achse, bei Null beginnend, das Einsatzspektrum von Software aufgetragen ist. Software, die ganz einfach ist und leicht verstanden werden kann, liegt näher bei Null, komplexere Sachen werden weiter rechts platziert. Auf der Y-Achse wird der Aufwand eingetragen, denn man treiben muss, um ein bestimmtes Ziel mit der Software zu erreichen. Manche einfache Software verlangt viel Aufwand, weil bestimmte Werkzeuge fehlen, andere hat zwar diese, die Bedienung ist aber so kompliziert, dass man auch nicht schneller zum Ziel kommt.

Produkte aus dem Hause Microsoft sind auf dieser Skala jeweils an den Enden zu finden. Der Einstieg wird einem sehr leicht gemacht; bei VBScript reicht wenig Wissen und man kann losschreiben. Will man auf diesem Niveau viel erreichen, muss man sehr kompliziert programmieren und tief in die Sprache einsteigen. Will man aber extrem viel erreichen, beispielsweise Funktionen umprogrammieren, steht mit COM und VC++ eine ganz andere und sehr starke Technologie zur Verfügung. Das ist das andere Ende der Skala. Andere Firmen produzieren Software anders. Der Verfasser hat auch jahrelang Perl und PHP programmiert, Skriptsprachen aus der Unix-Welt. Dort ist der Einstieg keineswegs so einfach wie bei VBScript. Der Anfangsaufwand liegt deutlich höher. Hat man diese Stufe überwunden, kommt man sehr weit auf der Skala. Ist das Ende aber erstmal erreicht, führt kaum ein Weg daran vorbei. Mit keinem vernünftigen Aufwand geht es weiter. Solche Programme decken das Mittelfeld im Koordinatensystem ab. Persönlich neige ich dazu, der ersten Philosophie den Vorzug zu geben, weil ich mich insgesamt weniger eingeschränkt fühle, auch wenn Anfänger oft genau das Gegenteil empfinden.

ASP.NET ist in dieser Hinsicht keine Ausnahme. Es macht Spaß, weil sich erste Erfolge sehr schnell einstellen. Das Programmiersystem lädt förmlich dazu ein, einfach loszuhacken. Will man mehr, wird es sehr schnell sehr kompliziert. Die enorme Leistungsfähigkeit, die das Framework ohne jeden Zweifel bietet, erstreckt sich auf der Skala weit nach rechts, führt aber auch weit links schon zu Beschwerden. Verlieren Sie an dieser Stelle nicht den Mut. Sie werden nur wenige Tage brauchen, um über dem einen oder anderen Problem völlig verzweifelt zu sitzen. Trösten Sie sich einstweilen damit, dass ziemlich sicher ist, dass es eine ganz einfache Lösung gibt, wirklich.

Was auch immer Sie vorhaben, es gibt nichts, was nicht geht. Denn Sie stehen im Mittelpunkt, Sie sind der „Developer“ und das Produkt wurde für Sie entworfen, garantiert. Aber es ist anspruchsvoll und verlangt viel Engagement, um die bereitgestellten Leistungen wirklich freiwerden zu lassen.

In diesem Sinn: Happy Coding!

Jörg Krause
Berlin, im Mai 2002

1 Schnellstart

lernen

Dieses Kapitel gibt Ihnen einen schnellen Start in die Welt von ASP.Net. Wenn Sie noch keine Erfahrung mit ASP oder der Webserverprogrammierung haben oder noch nie mit .Net in Berührung kamen, sind Sie hier richtig. Führen Sie die Schritte dieses Abschnitts aus, um zu einer lauffähigen Entwicklungsumgebung zu gelangen. Auf dieser Basis können Sie alle folgenden Kapitel leicht bearbeiten.

1.1 Was Sie in diesem Kapitel lernen

.Net ist in seiner durch Microsofts Marketingmaschinerie präsentierten Form für Einsteiger ein unhandliches Gebilde. Je nach Ansatz und Informationsquellen werden mehr oder weniger verständliche Aussagen präsentiert. Wenn Sie „einfach nur ASP.Net" lernen wollen, kann der Anfang recht schwer sein. Betrachtet man dann noch die dicken Wälzer, die inzwischen zu ASP.Net auch erhältlich sind, scheint es einen langen Lernprozess zu geben.

Dieses Kapitel will Ihnen drei Aussagen vermitteln:

1. Der Aufbau einer einfachen Entwicklungsumgebung ist schnell möglich
2. Der Weg zum ersten lauffähigen Skript ist in kurzer Zeit zu schaffen
3. Die wichtigsten Grundlagen lassen sich in kompakter Form lernen

Das Buch ASP.Net lernen ist vor allem für Anfänger gedacht, die noch keine Erfahrung mit der Webserverprogrammierung haben oder Anwender des klassischen ASP, die noch nicht sehr tief eingestiegen sind. Allerdings sollen keine „Umstiegsszenarien" diskutiert werden. ASP.Net unterscheidet sich in sehr weiten Teilen von ASP – abgesehen davon, dass es in beiden Fällen nur um die Webserverprogrammierung geht. Wenn Sie ASP kennen und sich nicht als Profi bezeichnen, sind Sie also auch richtig. Am besten, Sie beginnen damit, das frisch Erlernte wieder

zu vergessen. Die Unterschiede sind erheblich und einige Techniken sind völlig neu. Meist sind die Änderungen für den Programmierer von Vorteil. Vieles, was vorher nur mühevoll zu programmieren war, geht nun leichter von der Hand. Gleichzeitig sind aber durch das .Net-Framework weitaus mehr Möglichkeiten vorhanden und deshalb entsteht auch ein höherer Lernaufwand – dazu später mehr.

Einige Voraussetzungen müssen gegeben sein, damit Sie erfolgreich ASP.Net lernen können. So wird die Bedienung von Windows 2000 bzw. Windows XP als bekannt angenommen. Damit sind übrigens auch gleich die nötigen Betriebssysteme genannt. Der Aufbau einer Entwicklungsumgebung setzt Windows 2000 Professional oder Windows XP Professional voraus. Vorausgesetzt wird auch, dass Sie wissen, wie Webseiten aussehen. Das betrifft Grundkenntnisse über HTML und an einigen Stellen auch JavaScript bzw. DHTML. Letzteres wird aber nur angerissen. Da es in diesem Buch auch um die Integration von Datenbanken geht, sollten Sie Erfahrung mit SQL oder MS Access haben. Eine Einführung zum Verständnis der Beispiele ist enthalten – eine professionelle Programmierung kann allein damit aber nicht erlernt werden.

1.2 Über das Buch

Damit Sie zügig mit dem Buch arbeiten können, werden in aller Kürze die Techniken gezeigt, die Sie vorfinden. Wichtige Elemente sind durch Symbole gekennzeichnet:

Tipp – Dieses Symbol kennzeichnet Passagen, die aus dem Kontext herausragen, Zusatzinformationen liefern oder einfach interessant sind.

Hinweise – Besonderheiten und abweichendes Verhalten in bestimmten Situationen wird mit dem Hinweis-Symbol gekennzeichnet.

Warnung – Hier werden typische Fallen und Fehlerquellen angesprochen oder auf Nebenauswirkungen aufmerksam gemacht.

Als zusätzliche Hilfe zum Buch können Sie die Website des Autors zu diesem Thema besuchen:

http://www.dotnet.comzept.de

Sie finden dort alle Beispiele, teilweise aktualisiert, und ergänzende Informationen als besonderen Leserservice.

1.2.1 Prinzip der Wissensvermittlung

Dieses Buch ist ein Lernbuch – es dient weniger als Nachschlagewerk, sondern soll helfen, die ersten Hürden in einer neuen Welt zu überwinden. Entsprechend werden Themen nicht vollständig und bis ins kleinste Detail behandelt, dafür gibt es die Referenz und die beiden großen ASP.NET-Bücher bei Addison Wesley. Statt dessen finden Sie für jede Station auf dem Weg zum ASP.NET-Entwickler Übungsbeispiele und viel Code. Jedes Codebeispiel ist alleine lauffähig und kann sofort ausprobiert werden. Sie finden alle Beispiele auch auf der Website zum Buch, wenn Sie sich scheuen, den Code abzutippen. Es ist aber durchaus eine gute Idee, die eine oder andere Zeile selbst zu schreiben. So werden Sie auch mit den Fehlermeldungen des Compilers und der Entwicklungsumgebung schneller vertraut.

Dieses Buch vermittelt anspruchsvolles Wissen. Es ist professionell gehalten und es enthält keine willkürlich vereinfachten Informationen. Der Leser wird direkt, schnell und ohne Voraussetzungen an den Stoff herangeführt. Aber es wird erwartet, dass intensiv mit dem Buch gearbeitet wird und die Bereitschaft, sich in kürzester Zeit mit komplexen Sachverhalte auseinanderzusetzen, vorhanden ist.

Gerade darin unterscheidet es sich von anderen „Lernen"-Büchern. Das Ziel ist nicht nur ein mehr oder weniger zufälliges Erfolgserlebnis, sondern ein Start in die Welt der professionellen Programmierung. Nichtsdestotrotz sollte klar sein, dass dies ein langer Weg ist.

Übungen

Am Ende eines Abschnitts finden Sie Übungsbeispiele. Die Website enthält dazu einige Lösungsvorschläge. Die Übungen verlangen teilweise, dass Sie sich in der Referenz über weitere Sprachelemente informieren. Dieser Weg ist typisch, denn bei 7.000 Klassen im .Net-Framework wird kein Buch eine vollständige Abbildung bieten. Die Suche nach der passenden Klasse ist Programmiereralltag.

Sie sollten für Ihre praktische Arbeit auch anerkennen, dass der enorme Umfang von .Net sehr viel Zeit zum Lernen erfordert. Die kompakte Darstellung in diesem Buch ist so aufgebaut, dass kein Weg verbaut wird und alle Themen rund um ASP.NET zumindest kurz angesprochen werden. Am Selbststudium führt aber kein Weg vorbei.

1.2.2 Wahl der Programmiersprache

Microsoft wirbt bei der Präsentation von .Net damit, dass die Wahl der Programmiersprache keine große Rolle mehr spielt. Durch die Auslagerung eines großen Teils der Funktionalität ins Framework verbleiben nur geringe Unterschiede zwischen den Sprachen. Zur Auswahl stehen

derzeit im Lieferumfang von .Net die Sprachen VB.Net, C#, MC++ und JScript.Net. Als Entwickler des alten ASP werden Sie auf den ersten Blick entweder zu VB.Net greifen oder sich mit JScript.Net beschäftigen; je nachdem, ob Sie vorher in VBScript oder JScript gearbeitet haben. Die Alternativen C# und MC++ (Managed C++) sind keine Alternative, oder? An dieser Stelle ist ein erneuter Blick auf die Philosophie angebracht, die hinter den Produkten steckt. Microsoft hat C# zur neuen „Hauptsprache" erhoben. C# ist schnell, elegant, modern und relativ leicht erlernbar. Es sind die Prinzipien, auf denen .Net aufbaut, dafür verantwortlich, dass die Wahl der Sprache nur eine geringe Rolle spielt. Was auf den ersten Blick als der wesentliche Vorteil herausgestellt wird, mutiert beim zweiten zur Nebensache. Der Effekt, dass verschiedene Sprachen denselben Code erzeugen können, ist auf die Überführung vieler Sprachelemente in die Klassenhierarchie zurückzuführen. So ist VB.Net um viele Funktionen zur Datums- und Zeichenkettenverarbeitung erleichtert worden. Statt dessen können entsprechende Klassen aus dem .Net-Framework genutzt werden. Was im Kern übrig bleibt, ist ein kleine und schlanke Sprache. Dasselbe trifft aber auch für JScript.Net und C# zu. Der Lernaufwand liegt also ganz woanders – das Framework hat deshalb eine tragende Rolle im gesamten Buch. Für Anfänger, die sich das erste Mal mit dieser Art Programmierung beschäftigen, ist die Wahl nicht von historischen Gedanken beeinflusst. Für beide Seiten wäre es also optimal, die beste und modernste Sprache zu verwenden. Diese Sprache ist in der .Net-Welt C#.

Deshalb wird in in diesem Buch ausnahmslos mit C# gearbeitet.

Die Darstellung der Sprache ist im gegebenen Umfang nicht vollständig möglich. Für die ersten ASP.NET-Seiten, für eine Datenbankabfrage und den Dateizugriff – durchaus gehobene Aufgaben – reicht der präsentierte Teil jedoch aus. Ebenso wichtig ist die Kenntnis der Klassen des Frameworks. Auch wenn der Begriff am Anfang abstrakt klingt, es ist nicht so schwer, damit zu arbeiten. Im Gegenteil, es ist ein beruhigendes Gefühl zu wissen, dass fast alle denkbaren Elementaraufgaben irgendwie schon gelöst sind. Sie müssen sich nur auf Ihre Geschäftslogik konzentrieren, auf die Erfüllung der Aufgabe, eine vernünftige Benutzerführung, das Design der Schnittstellen zum Anwender usw. – genug für eine anspruchsvolle Tätigkeit.

1.2.3 Worauf es ankommt

ASP.NET ist keine eigene Programmiersprache. Es ist eine Erweiterung des Webservers Internet Information Server (IIS) 5 bzw. IIS 5.1 um die Fähigkeit, mit Webseiten verknüpften Code auszuführen und das Ergeb-

nis der Berechnungen an den Browser zu senden. Wie das erfolgt, wird im nächsten Abschnitt noch genauer gezeigt. Der eigentliche Verarbeitungsprozess passiert also in Form von Programmen, die in der Website oder damit verknüpft vorliegen und ausgeführt werden, wenn ein Benutzer eine Seite über das Internet anfordert.

Webserver

An erster Stelle steht der Webserver. Mit der Vorgabe IIS 5 bzw. IIS 5.1 steht fest, dass nur Windows 2000 oder Windows XP verwendet werden. Beide Systeme unterscheiden sich in Bezug auf die ASP.Net-Unterstützung nur marginal. Wenn Sie Windows Me oder 98 besitzen, müssen Sie zuerst ein Upgrade auf XP Professional durchführen. Es reicht auch nicht aus, XP Home einzusetzen, denn diese Version enthält keinen IIS. Es bleibt Ihnen also nichts weiter übrig, auch für ein Entwicklungssystem einige Investitionen zu tätigen. Ebenso sind die Anforderungen an die Hardware zu beachten, die sich aber mehr nach dem Betriebssystem richten, als nach ASP.NET.

Nun wird noch das .Net-Framework selbst benötigt. Sind Sie in der glücklichen Lage, ein MSDN-Abonnent zu sein oder haben Sie Visual Studio.Net – Microsofts Enwicklungsumgebung – gekauft, sind alle notwendigen Komponenten dort enthalten. Wenn Sie aber zu Hause entwickeln und lernen, wird dies nicht unbedingt der Fall sein. Glücklicherweise zielt die Lizenz- und Preispolitik von Microsoft auf den professionellen Enwickler. Das Framework selbst ist kostenfrei.

Damit wird nur noch ein Editor benötigt, mit dem die Codes eingetippt und gespeichert werden können. Zum Abrufen wird natürlich auch ein Browser benötigt. Da standardmäßig der Internet Explorer installiert ist, wird diese Voraussetzung sicher erfüllt sein.

Zusammenfassend noch ein Mal die Voraussetzungen, die Sie schaffen müssen:

- Ein Computer mit der passenden Hardwareausstattung. Empfehlenswert sind mindestens 1 GHz und 256 MByte Speicher.
- Windows 2000 Professional mit Service Pack 2 oder Windows XP Professional.
- Microsoft Data Access Components 2.7, die für den Datenbankzugriff benötigt werden.
- Das Framework SDK selbst – wie Sie es beschaffen können, wird im folgenden Abschnitt erläutert.
- Ein Editor – dies kann, muss aber nicht Visual Studio.Net sein.

Angeboten wird von Microsoft außerdem das Installationspaket „Windows Component Upgrade“. Vom Herunterladen sollten Sie Abstand nehmen, denn es enthält neben den genannten Programmen Service

Pack 2, .Net-Framework und Data Access Components auch noch den Internet Explorer 6.0, die FrontPage-Erweiterungen und den Windows Installer. Letzterer wird nicht benötigt, solange Sie Ihre Applikationen nicht als fertig installierbares Softwarepaket fertigstellen möchten.

Visual.Studio.Net erschien erst Mitte April 2002 in deutscher Version. Außerdem ist der Preis für die einfachste Version mit ca. € 1.000 zu hoch, um sich das Paket mal eben als alternativen Editor zu installieren. Editoren zum Programmieren gibt es jedoch viele. Durchaus zu empfehlen ist der kostenfreie (Open Source) Editor „Sharp Develop“ oder der sehr preiswerte Komodo. Mit diesem Editor sind alle Beispiele des Buches entstanden. Wo Sie diese Editoren beschaffen und wie Sie sie installieren können, wird im nächsten Abschnitt beschrieben.

1.3 Eine Entwicklungsumgebung aufbauen

Dieser Abschnitt zeigt die vollständige Installation einer Entwicklungsumgebung unter Windows 2000 Professional und Windows XP Professional. Beachten Sie, dass XP Home nicht ausreicht, um ASP.NET zu verwenden.

1.3.1 ASP.NET 1.0 installieren

.Net war lange Zeit eine Marketingblase und bestand vor allem aus Ankündigungen. Wie bei einer Softwarefirma nicht anders zu erwarten, manifestiert es sich in Form eines umfassenden Systemupdates. Allerdings tangiert .Net bestehende Applikationen nicht. Sie können es installieren und später wieder entfernen.

Wenn Sie bereits eine Beta-Version von Visual Studio.Net oder dem Framework installiert haben, entfernen Sie diese bitte zuerst. Dieses Buch und alle Beispiele sind für die Final Version entwickelt worden.

Für die Beschaffung der Komponenten besuchen Sie folgende Website: *http://www.asp.net/download.aspx*. Es stehen hier zwei Versionen zur Auswahl. Zum einen das .NET FRAMEWORK REDISTRIBUTABLE, mit 21 MByte noch überschaubar groß. Dies sind die gesamten Laufzeitprogramme des Frameworks sowie C# und ASP.NET. Diese Version wird üblicherweise auf Produktionsservern (Windows 2000 Server oder Windows .Net Server) eingesetzt.

Sie finden dort auch das MICROSOFT .NET FRAMEWORK SDK. Das komplette SDK (Software Development Kit) ist 138 MByte groß. Es kann im Stück oder in Teilen zu ca. 13 MB geladen werden. Hier sind neben dem

Umfang des kleinen Paketes auch die Compiler, Dokumentationen und Beispielcodes enthalten.

Für die ersten Schritte mit ASP.NET reicht das kleine Paket aus. Die Übersetzung der Programme läuft bei ASP.NET im Hintergrund beim Aufruf der Seite ab – zusätzliche Compiler werden nicht unbedingt benötigt. Wenn Sie auf die Online-Hilfe zugreifen möchten, besuchen Sie die Website *http://www.asp.net*. Dies ist unter Umständen entspannender als 138 MByte über ISDN herunterzuladen.

Anschließend laden Sie noch die Microsoft Data Access Components 2.7 (MDAC 2.7). Diese sind auf einer eigenen Website zu finden: *http://www.microsoft.com/data/download.htm*. Die Installation ist nicht notwendig, wenn Sie Windows XP Professional verwenden; dort ist diese Version im Lieferumfang enthalten. Das Paket ist ca. 5 MByte groß und sollte in der deutschen Version geladen werden, wenn Sie ein deutsches Windows verwenden.

IIS installieren

InstallationDer IIS 5 gehört zum Lieferumfang von Windows 2000 Professional und in der Version 5.1 von Windows XP Professional. Beide Systeme installieren ihn jedoch nicht standardmäßig. Falls Sie die entsprechende Option bei der Installation nicht gewählt haben, gehen Sie wie nachfolgend beschrieben vor. Wenn Sie nicht sicher sind, ob der IIS installiert wurde, führen Sie die ersten Schritte ebenfalls aus.

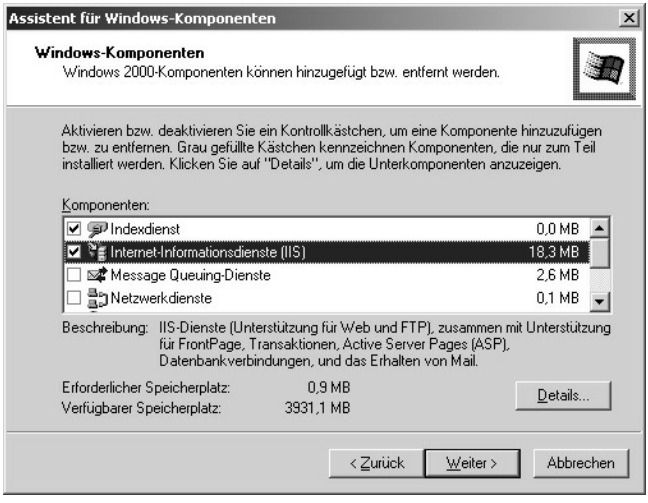


Abbildung 1.1: Installation des IIS unter Windows 2000

Öffnen Sie zuerst die Systemsteuerung und starten dort das Programm SOFTWARE. Klicken Sie dann auf WINDOWS-KOMPONENTEN HINZUFÜGEN/ENTFERNEN. Es startet der Assistent für Windows-Komponenten. In der Liste installierter Software suchen Sie den Eintrag INTERNET-INFORMATIONSDIENSTE (IIS). Wenn das Kontrollkästchen auf der linken Seite aktiviert ist, wurde der IIS bereits installiert und Sie können abbrechen.

Für den IIS können einige Optionen bestimmt werden, die Sie über die Schaltfläche DETAILS erreichen. Für den Einsatz mit ASP.NET ist es empfehlenswert, alle Optionen zu aktivieren, was auch standardmäßig der Fall sein sollte.

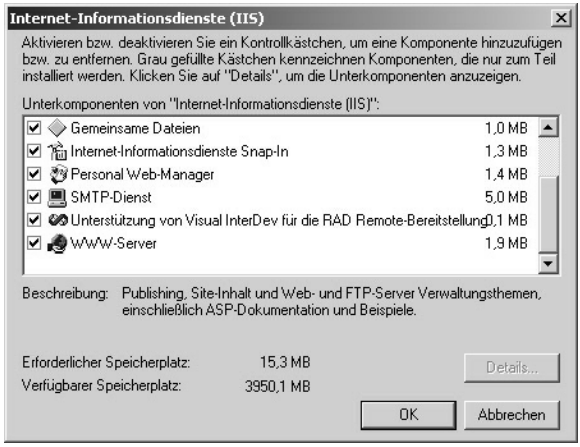


Abbildung 1.2: Auswahl der Installationsoptionen

Andernfalls aktivieren Sie das Kontrollkästchen jetzt und klicken auf WEITER. Der Installationsprozess läuft nun ab. Möglicherweise werden Sie aufgefordert, die Installations-CD einzulegen. Wenn der Assistent fertiggestellt wurde, können Sie den IIS testen.

Wie der IIS arbeitet

Der IIS startet – als Dienst WWW-PUBLISHINGDIENST – sofort nach der Installation. Ein Neustart ist dennoch erforderlich, auch wenn er nicht explizit verlangt wird. Für eine erste Orientierung sollten Sie sich das Arbeitsverzeichnis anschauen. Der IIS legt ein Verzeichnis *inetpub* an, der folgende wichtige Unterverzeichnisse enthält:

- *iissamples* – Hier sind einige Beispiele im alten ASP 3.0 zu finden.
- *scripts* – Dieses Verzeichnis kann zum Ablegen der ASP.NET-Programme genutzt werden.
- *wwwroot* – Das ist das Stammverzeichnis des Webservers; hier liegen die Dateien, die ohne weitere Pfadangabe erreicht werden können.
- *webpub* – Dieses Verzeichnis dient der Ablage von Dateien, die per WebDAV hochgeladen wurden. WebDAV ist eine HTTP-Erweiterung,

die den Zugriff auf das Dateisystem eines Webservers per HTTP erlaubt. Das Verzeichnis ist nach der Installation leer.

- *\ftproot* und *\mailroot* sind die Stammverzeichnisse des FTP- und SMTP-Servers. Wenn Sie die Installation auf einem Windows 2000 Server ausgeführt haben, finden Sie außerdem noch *\nntproot*, das Stammverzeichnis des Newsservers.

Der IIS enthält einige Dateien, die für einen ersten Test benutzt werden können. Starten Sie dazu den Browser und geben dann folgende Adresse ein: *http://localhost*

Sie sollten dann das folgende Bild sehen. Sie können nun mit der Installation des Frameworks fortsetzen.

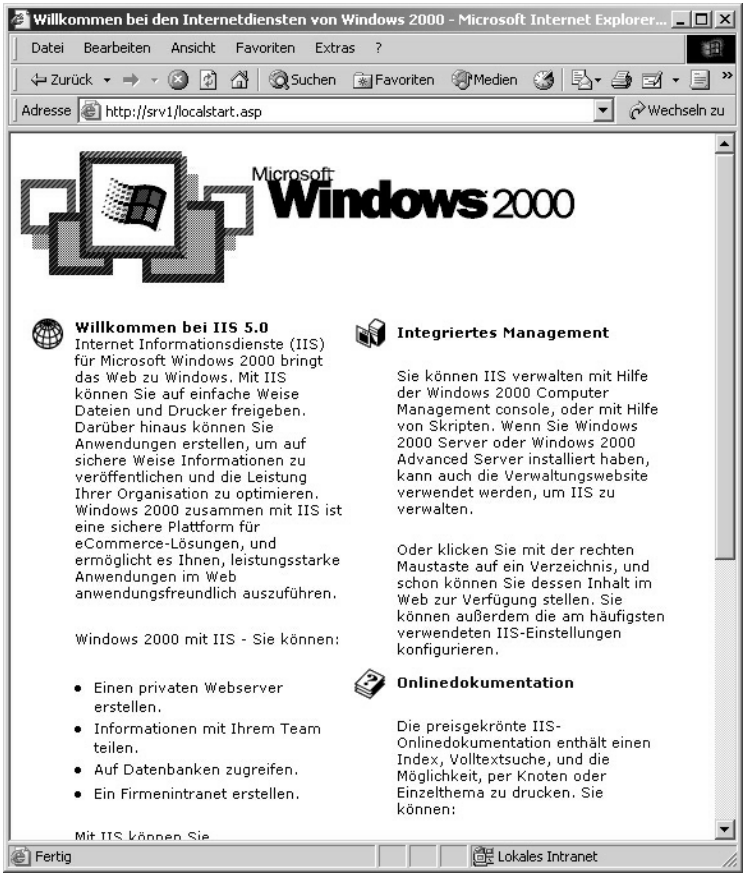


Abbildung 1.3: Der IIS wurde erfolgreich installiert

Was tun bei Fehlern?

Mit dem funktionierenden IIS können Sie nun die Installation fortsetzen. Zeigt der Browser einen Fehler an, sollten Sie erst weiter machen, wenn die Ursache gefunden ist. Gehen Sie dazu folgendermaßen vor:

- Prüfen Sie den Dienst WWW-PUBLISHINGDIENST. Er muss gestartet sein, damit der IIS arbeiten kann.
- Wenn das nicht der Fall ist, probieren Sie statt „localhost“ den Namen des Computers anzugeben.
- Möglicherweise liegt ein Konfigurationsfehler in der Netzwerkkonfiguration vor. Der Rechner sollte neben seiner IP-Adresse auch über 127.0.0.1 erreichbar sein.

Installation des Frameworks

Das Framework wird mit einem Installationsprogramm geliefert. Dieser Abschnitt beschreibt die Installation des „.Net-Framework Redistributable“ – also der kleineren Version. Starten Sie das heruntergeladene Programm. Sie müssen zuerst die Lizenzbestimmungen bestätigen und dann einen Pfad für die entpackten Dateien angeben. Dies kann beispielsweise das *temp*-Verzeichnis sein. Die Dateien die hier entpackt werden, entfernt das Installationsprogramm nach der Einrichtung wieder.

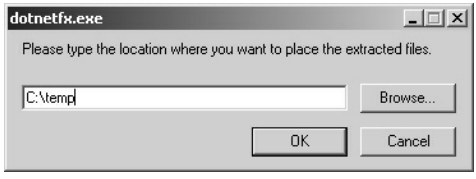


Abbildung 1.4: Entpacken in temporäres Verzeichnis

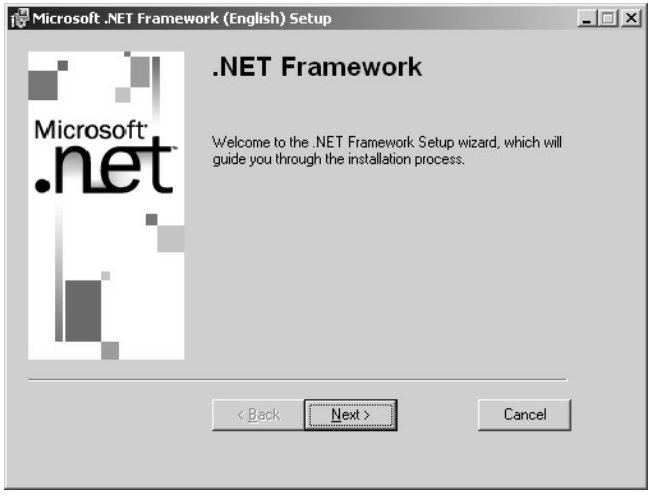


Abbildung 1.5: Start der Framework-Installation

Öffnen Sie nun dieses Verzeichnis. Dort finden Sie eine Datei *dotnetfx.exe*. Dies ist das eigentliche Installationsprogramm. Nach dem Start kommt eine Sicherheitsabfrage, ob Sie das Paket wirklich installieren möchten. Nach einem Klick auf JA läuft die Installation ab. Diese wird von einem Assistenten begleitet, der den Fortschritt anzeigt, aber keine weiteren Angaben verlangt. Klicken Sie auf WEITER.

Der zweite Schritt zeigt den Fortschritt der Installation an. Während des Vorgangs wird der WWW-PUBLISHINGDIENST angehalten. Am Ende folgt noch eine Erfolgsmeldung, die mit OK bestätigt werden muss.

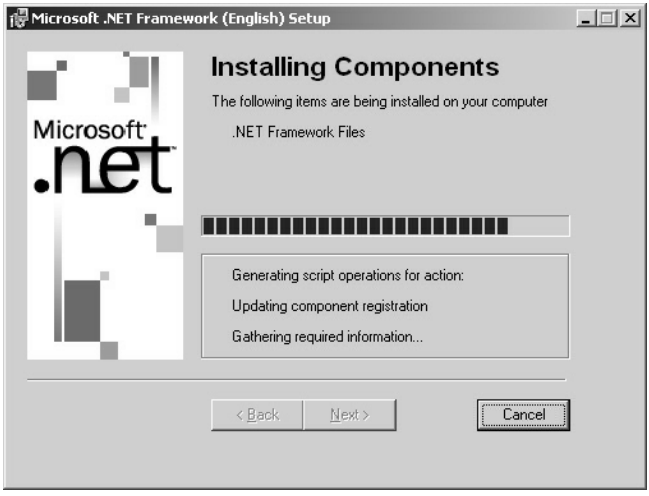


Abbildung 1.6: Fortschrittsanzeige während der Installation

An dieser Stelle steht ASP.NET bereits zur Verfügung. Die weiteren Schritte zu einem gut konfigurierten Entwicklungssystem werden wieder im IIS vorgenommen. Dafür ist die Managementkonsole des IIS das passende Verwaltungswerkzeug.

MDAC installieren

Auch wenn Sie nicht gleich mit Datenbanken arbeiten, sollten Sie jetzt MDAC installieren. Dieser Schritt ist für Windows XP nicht notwendig, da hier bereits die aktuelle MDAC-Version installiert wurde. Das Programm wird ebenfalls mit einem Installationsassistenten geliefert. Der erste Schritt umfasst wieder das Bestätigen der Lizenzbestimmungen. Dann wird der verfügbare Speicherplatz überprüft. Die Installation läuft ohne weitere Fragen ab – da es sich um Systemkomponenten handelt, ist eine Pfadangabe nicht erforderlich.

Den IIS auf ASP.NET vorbereiten

Praktisch sind ASP.NET-Anwendungen jetzt bereits lauffähig. Sie sollten den ersten Versuch dennoch nicht überstürzen. Bereiten Sie zuerst ein Verzeichnis vor, dass künftig alle Programme aufnimmt, mit denen Sie ASP.NET lernen. Dazu gehen Sie folgendermaßen vor:

1. Legen Sie unterhalb *\wwwroot* einen neuen Ordner an, beispielsweise mit dem Namen *\dotnet*.
2. Starten Sie die Managementkonsole des IIS über SYSTEMSTEUERUNG | VERWALTUNG | INTERNET-INFORMATIONSDIENSTE.
3. Wählen Sie den Eintrag STANDARDWEBSITE aus.
4. Klicken Sie mit der rechten Maustaste darauf. Aus dem Kontextmenü wählen Sie NEU und dann VIRTUELLES VERZEICHNIS.

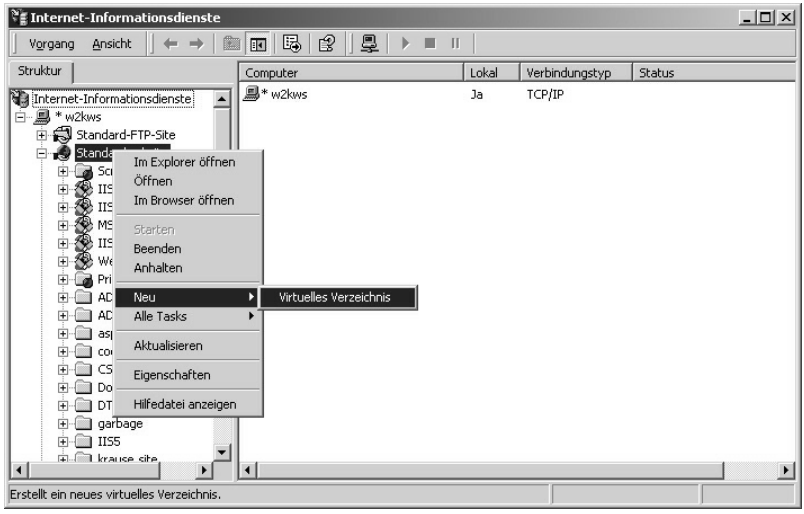


Abbildung 1.7: Anlegen eines neuen virtuellen Verzeichnisses

5. Es startet ein Assistent, der folgende Angaben verlangt:

   a. ALIAS – Ein Name, der nach außen sichtbar ist, also für den Benutzer am Browser. Die Beispiele im Buch sind unter dem Alias *aspdotnet* abgelegt.

   b. VERZEICHNIS – Hier geben Sie den Pfad zum neu angelegten Ordner an.

   c. ZUGRIFFSBERECHTIGUNGEN – In diesem Schritt geben Sie die Berechtigungen zum Zugriff an. Da ASP.NET ein Programm ist, müssen Sie nebenLESEN und SKRIPT auch AUSFÜHREN aktivieren. Für viele Beispiele im Buch werden auch Schreibberechtigungen (SCHREIBEN) erwartet. Wenn Sie DURCHSUCHEN aktivieren, können Sie die fertigen Programme leichter auswählen. Für ein Enwicklungssystem werden deshalb sinnvollerweise alle Optionen aktiviert.

      Für einen Produktionsserver gilt dies nicht unbedingt. Es folgt deshalb noch eine Warnung, genau die beschriebenen Einstellungen nicht zu verwenden. Dies können Sie aber für ein Entwicklungssystem ruhig ignorieren.

Es ist nun an der Zeit, einen ersten Test vorzunehmen. Für die Erfassung des ersten ASP.NET-Programms reicht der Windows Editor (Notepad) aus. Geben Sie folgenden Code ein:

```
<script language="c#" runat="server">
void Page_Load()
{
   datum.Text = DateTime.Now.Day.ToString() + "."
              + DateTime.Now.Month.ToString();
}
</script>
<html>
  <head><title>Erster Test</title></head>
  <body>
  <h1>Willkommen</h1>
  Nein, diesmal nicht "Hello World".<br/>
  Eine Datumsausgabe:
  Heute ist der <asp:label id="datum" runat="server"/>
  </body>
</html>
```

Listing 1.1: Das erste ASP.NET-Programm (firsttest.aspx)

Speichern: aspx

Jetzt kommt der wichtigste Teil! Speichern Sie das Dokument unter dem Namen *firsttest.aspx*. Entscheidend ist hier die Dateierweiterung. Die Verknüpfung zwischen ASP.NET und dem Webserver wird über die Dateierweiterung *.aspx* hergestellt. Speichern Sie die Datei in dem Verzeichnis, dass Sie vorher erzeugt haben. Geben Sie in der Adresszeile des Browsers den gewählten Aliasnamen an:

http://localhost/aspdotnet/firsttest.aspx

Sie sollten nun etwa folgendes Bild im Browser sehen:

Abbildung 1.8: Ausgabe, wenn die Testdatei erfolgreich ausgeführt wurde

Möglicherweise haben Sie sich vertippt. Dann reagiert der Compiler mit einer ausführlichen Fehlerbeschreibung. Wenn nicht, provozieren Sie einen Fehler im Codeteil (Umgeben von dem Tag `<script>`), um ein Gefühl für die Reaktion von ASP.NET zu bekommen.

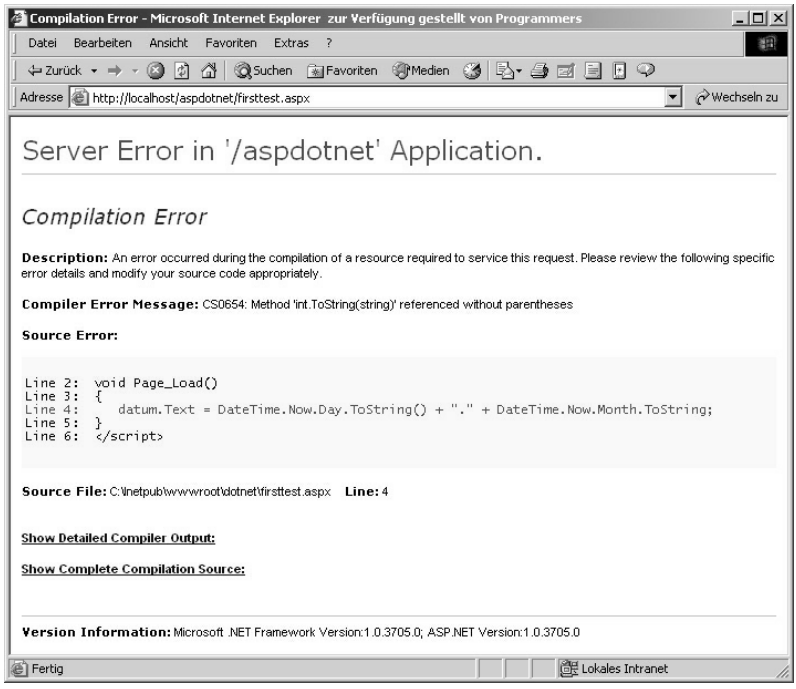


Abbildung 1.9: Fehler im Code der Seite

An dieser Stelle werden Sie sich vielleicht über das laufende Programm freuen, aber sicher ist es interessanter zu wissen, wie es funktioniert. Wenn Sie ASP kennen, wird Ihnen eventuell auch die Fehlermeldung Freude bereiten. Offenbar sind die Informationen deutlich umfangreicher und besser lesbar als im alten ASP. Was aber passierte inzwischen? Sie haben keine integrierte Entwicklungsumgebung (so genannte IDE) installiert, keinen aufwändigen Compiler, kein sichtbares Programm. Trotzdem scheint es auf Anhieb zu funktionieren. Willkommen bei .Net!

Wie es funktioniert

Der Ablauf unterscheidet sich prinzipiell nur wenig von dem des Vorgängers ASP. Sie haben im Browser die Adresse Ihres Programms eingetippt. Der Browser sendet die Anforderung an den Webserver. Im IIS besteht eine Verknüpfung zwischen der Dateierweiterung *.aspx* und der ASP.NET-Komponente. Diese besteht im Wesentlichen aus einer DLL: *aspnet_isapi.dll*. Insofern hat sich wenig geändert. Diese Komponente ist nun für die Abarbeitung der Seite verantwortlich.

Eine Analyse des Codes zeigt das Prinzip der Verarbeitung. Zuerst wird ein `<script>`-Tag eingesetzt, das dem Server zwei Dinge mitteilt: Die zu verwendende Sprache ist C# und die Verarbeitung soll auf dem Server erfolgen (`<runat="server">`).

```
<script language="c#" runat="server">
```

Nun folgt der Code der Seite. `void` ist ein Schlüsselwort in C#, das eine Funktion deklariert, die nichts zurückgibt (aber dennoch natürlich ausgeführt wird). Der Name der Funktion ist festgelegt. `Page_Load()` wird immer dann ausgeführt, wenn die Seite geladen wurde:

```
void Page_Load()
```

Dann wird einer Eigenschaft etwas zugewiesen. An dieser Stelle tangieren Sie das erste Mal die Welt der Objekte. In C#, im .Net-Framework und damit in ASP.NET gibt es praktisch nur noch Objekte. Alles ist objektorientiert. Sie finden in diesem Buch viele Informationen über das Prinzip und die Verwendung. Zuerst aber zu der Technik dieses Codes. `datum` ist ein Objekt, dass in der nachfolgenden HTML-Seite definiert wurde. Es ist aus dem Tag `<asp:label id="datum" runat="server">` entstanden. Dieses Objekt besitzt eine Eigenschaft `Text`, der eine Zeichenkette zugewiesen werden kann. Diese entsteht durch Aufrufe von Methoden des .Net-Frameworks. Aus der Klasse `DateTime.Now`, die das aktuelle Datum enthält, wird der Tag (`Day`) und der Monat (`Month`) entnommen. Weil diese Methoden keine Zeichenketten sondern Datumswerte zurückgeben, wird noch eine Konvertierung mit `ToString()` durchgeführt:

```
datum.Text = DateTime.Now.Day.ToString() + "." +
DateTime.Now.Month.ToString();
```

Damit ist das eigentliche Programm auch schon fertig.

```
}
</script>
```

Da es sich um eine HTML-Seite handelt, die letztlich an den Browser gesendet werden soll, folgt nun deren Definition:

```
<head><title>Erster Test</title></head>
<body>
<h1>Willkommen</h1>
  Nein, diesmal nicht "Hello World".<br/>
  Eine Datumsausgabe:
```

Die einzige Besonderheit ist das spezielle ASP-Tag, dass den Text ausgibt. `<asp:label>` kann nicht mehr, als die Zeichenkette, die die Eigenschaft `Text` enthält, auszugeben:

```
Heute ist der <asp:label id="datum" runat="server"/>
</body>
</html>
```

So einfach ist ASP.NET. In der Praxis kommt dann doch etwas mehr auf Sie zu, aber dies liegt weniger an der Notwendigkeit, alles zu verwenden, damit überhaupt etwas läuft, sondern mehr an den Möglichkeiten, die die gelieferte Technik bietet:

- Der gesamte Sprachumfang von C# steht zur Verfügung
- Sämtliche Klassen des .Net-Frameworks – über 7.000 insgesamt – stehen bereit
- Viele Erweiterungen für die HTML-Programmierung sind speziell in den ASP.NET-Komponenten vorhanden

Es ist also erforderlich und sinnvoll, systematisch diese neuen Dinge zu erkunden. Wenn Sie bereits ASP programmiert haben, werden Sie schnell Gefallen daran finden, denn vieles ist einfacher und eleganter geworden. Wenn Sie noch keine Erfahrung mit ASP haben, werden Sie sich sehr schnell darin zurechtfinden, da die Struktur klar und aufgabenorientiert ist.

Der nächste Schritt besteht nun aber darin, statt des sehr schlichten Editors ein praktischeres Werkzeug zu beschaffen. Neben vielen anderen freien und kommerziellen Editoren soll hier zuerst SharpDevelop vorgestellt werden.

Ein Editor: SharpDevelop

SharpDevelop wurde von Mike Krüger, einem engagierten Softwarentwickler, beginnend mit der Beta 1 des .NET-Frameworks entwickelt. Es ist Open Source, also freie Software mit offenen Quellcodes. Wenn Sie sich auch für die GUI-Programmierung in .NET interessieren, bieten die Codes eine reiche Informationsquelle. Sie finden das Programm und Informationen darüber auf folgender Seite: *http://www.icsharpcode.net*. Für dieses Buch wurde die Version 0.87c eingesetzt. Laden Sie dennoch die aktuellste Fassung. Das Paket ist ohne Quellcode 2,2 MByte groß, mit dagegen 3,6 MByte.

Installation

Heruntergeladen wird eine ausführbare Datei, die direkt das Installationsprogramm startet. Es beginnt mit der Bestätigung der Lizenzinformationen. SharpDevelop ist unter der GPL (General Public License) veröffentlicht. Geben Sie dann den Pfad an, in den installiert wird.

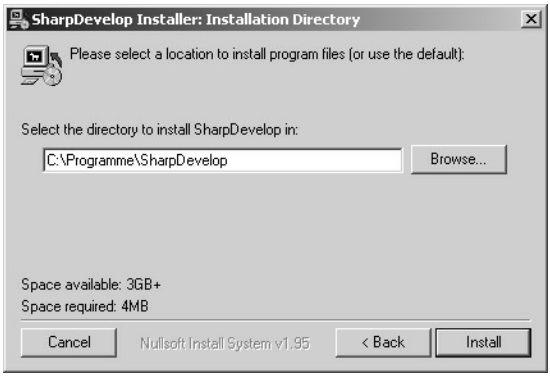


Abbildung 1.10: Installationspfad für SharpDevelop angeben

Nach der Installation, die keine weiteren Angaben mehr verlangt, können Sie das Programm sofort starten.

Konfiguration

Es ist sinnvoll, SharpDevelop nach der Installation zu konfigurieren. Dies beschränkt sich jedoch auf wenige Schritte. Stellen Sie die Sprache ein. Dazu wählen Sie im Menü EXTRAS den Eintrag OPTIONEN.

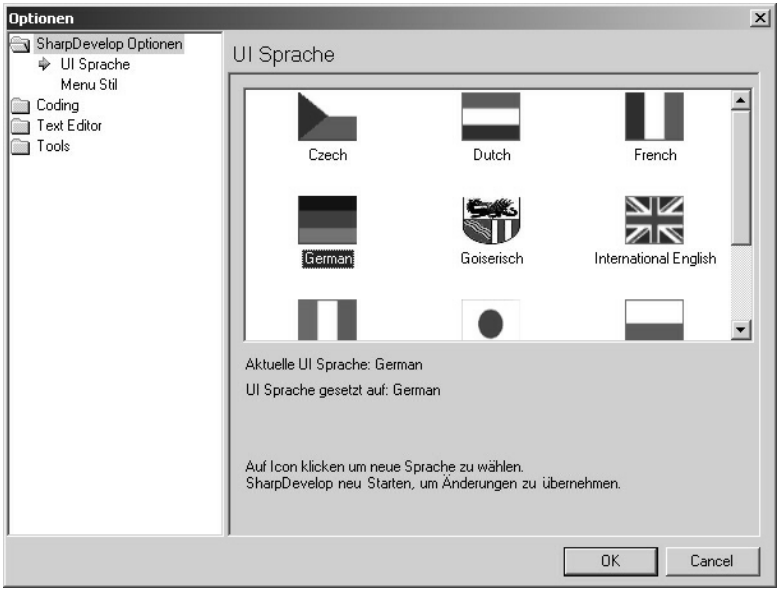


Abbildung 1.11: Auswahl der Sprache für die Menüs

Die anderen Einstellmöglichkeiten sind für die ersten Schritte nicht von Bedeutung. Der Editor ist sehr einfach, besitzt jedoch immerhin eine Projekt- und Aufgabenverwaltung. Verzichten müssen Sie derzeit jedoch auf die Vorgabe von Objektelementen, wie es Visual Studio bietet.

Noch ein Editor:

Komodo wird von ActiveState produziert, einer Firma, die sich seit Jahren um die Windows-Plattform bemüht und unter anderem Erweiterungen zu Visual Studio anbietet. Auch ihre Perl-Portierung nach ASP ist weit verbreitet. Komodo kann neben vielen anderen Programmiersprachen auch eine Syntaxhervorhebung für C# und besitzt ausgefeilte Editorfunktionen. Allerdings ist das Programm nicht kostenlos zu bekommen. Für die private Nutzung fallen $ 29,90 an.

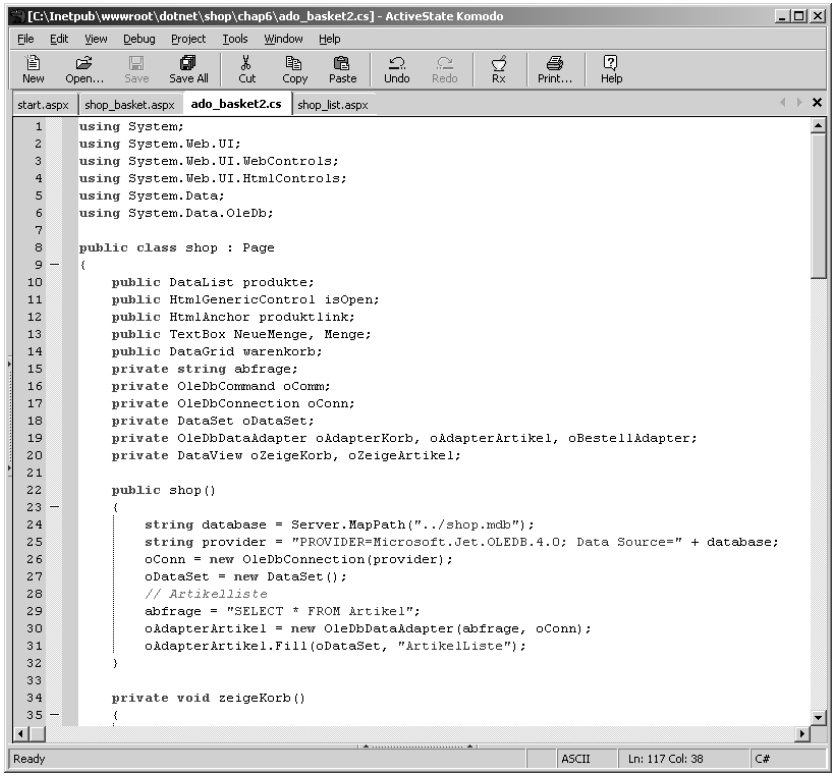


Abbildung 1.12: Komodo mit einem C#-Quelltext

Beschaffbar ist Komodo – auch als 21-Tage-Testversion – über die Website des Anbieters:

http://www.activestate.com/

1.4 Die wichtigsten Grundlagen

Nach dem die Vorbereitungen abgeschlossen sind, sollten Sie sich zurücklehnen und sich auf einen interessanten Weg in die Welt von

ASP.NET freuen. Damit es nicht zum Albtraum wird, sind jedoch ein paar theoretische Grundlagen erforderlich. Es erleichtert das Verständnis enorm, wenn Sie wissen, was im Hintergrund passiert. Das .Net-Framework bietet ein hohes Maß an Abstraktion, wenn aber Fehler auftreten, ist solides Basiswissen hilfreicher als teuere Entwicklungswerkzeuge.

1.4.1 Wie dynamische Webseiten entstehen

Wenn Sie noch nie Webserver programmiert haben, müssen Sie sich zuerst um das Verständnis des prinzipiellen Ablaufs zwischen Browser und Webserver bemühen. Wenn der Benutzer eine Adresse im Browser eintippt, läuft ein recht komplexer Vorgang ab:

1. Der Browser sucht einen Nameserver, um die IP-Adresse zum URL zu ermitteln
2. Der Nameserver konsultiert gegebenenfalls weitere Server, um die IP-Adresse zu beschaffen
3. Der Browser erhält eine IP-Adresse des Servers. Wenn das Protokoll HTTP verwendet wird, ist damit auch die Portadresse festgelegt (Port 80). IP-Adresse und Port bilden eine so genannte Socket.
4. Der Browser hat eine IP-Adresse vom Provider erhalten und einen Port für die Verbindung gebildet. Damit steht auch eine Socket zur Verfügung. Zwischen beiden Endpunkten kann nun IP-Verkehr stattfinden.
5. Der Browser sendet über diese Verbindung die Anforderung der Seite. Die erfolgt mit dem Protokoll HTTP, der entsprechende Befehl lautet GET, der Vorgang wird „Request“ genannt.
6. Der Server empfängt die Anforderung und sucht die Datei. Wird sie gefunden liefert er sie aus. Dieser Vorgang wird „Response“ genannt. Wird die Datei nicht gefunden, erzeugt der Server einen Fehler. Für nicht vorhandene Dateien definiert HTTP die Fehlernummer 404.
7. Der Browser empfängt Daten oder eine Fehlermeldung und zeigt diese an.

Damit ist der Vorgang beendet. Beide Seiten „vergessen“ alles, was beim Ablauf verwendet wurde. Mit der Anforderung des nächsten Objekts wird der gesamte Ablauf wiederholt. Die Vorgänge der Namensauflösung und Adressbeschaffung laufen völlig transparent ab und sind auch bei der Programmierung kaum zu berücksichtigen. Der eigentliche Zusammenbau der Seiten ist der interessante Teil. Dies passiert in der Darstellung der Schrittfolge im Schritt 6. Diesen Punkt gilt es also genauer zu untersuchen.

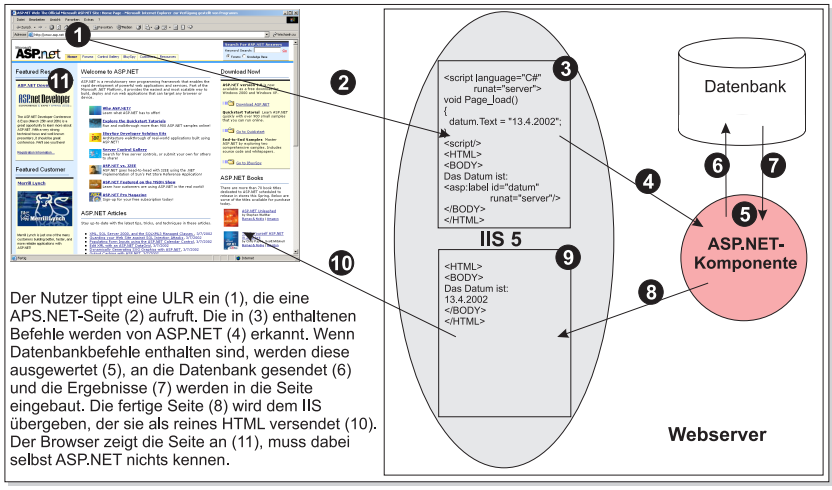


Abbildung 1.13: Ablauf bei der Auslieferung dynamischer Webseiten

Die Abbildung zeigt den Ablauf. Zusätzlich werden hier noch Schritte zur Abfrage einer Datenbank ausgeführt. Durch die Verknüpfung der Erweiterung *.aspx* wird der Webserver die ASP.NET-Komponente aufrufen, die die weitere Verarbeitung übernimmt. Am Ende entsteht eine reine HTML-Seite, die an den Browser gesendet wird. Mangels direkter Verbindung zwischen Server und Browser sind einige Besonderheiten bei der Programmierung zu beachten. So können Sie in einem Programm nicht direkt auf Ereignisse der Benutzeroberfläche reagieren, wie dies in der Windows-Programmierung der Fall ist. Wenn der Benutzer etwas anklickt, und eine Funktion in Ihrem Programm darauf reagieren soll, dann muss die Seite an den Server gesendet werden. Weiterhin müssen Sie selbst dafür Sorge tragen, dass der Status des Benutzers erhalten bleibt. Es liegt in der Natur eines Webservers, dass die Seite von vielen Teilnehmern am Netz gleichzeitig oder zeitnah angefordert werden kann. Bei komplexen Anwendungen muss die Zuordnung erhalten bleiben, was das statuslose Protokoll HTTP nicht selbst realisiert.

Für alle Probleme liefert ASP.NET interessante und hilfreiche Lösungen. Die Programmierung ist deshalb vergleichsweise einfach. Das ändert aber nichts am Prinzip oder der zugrundeliegenden Technik. Ohne das Ping-Pong-Spiel zwischen Browser und Webserver funktioniert nichts. Manchmal ist es nicht immer sichtbar, dass dieser Prozess tatsächlich abläuft, aber er wird dennoch ausnahmslos ausgeführt. Klar sollte auch sein, dass in der Webseite, wenn sie an den Browser gesendet wurde, nur noch HTML-Code steht, kein C#. Auch das spezielle Element `<asp:label/>`, das im Testprogramm eingesetzt wurde, ist durch eine HTML-konforme

Variante ersetzt worden. Schauen Sie sich den Quelltext der Seite nun im Browser an:

```
<html>
   <head><title>Erster Test</title>
   </head>
   <body>
   <h1>Willkommen</h1>
   Nein, diesmal nicht "Hello World".<br/>
   Eine Datumsausgabe:    Heute ist der <span id="datum">8.3</span>
   </body>
</html>
```

Listing 1.2: Quelltext des Testprogramms im Browser

Für das spezielle ASP.NET-Tag wurde das HTML-Tag `<span>` eingesetzt, der berechnete Text steht als fester Wert drin. Um eine Änderung zu erreichen, muss die Seite erneut beim Server angefordert werden. Welche Wege es dafür gibt, wird noch ausführlich diskutiert werden.

1.4.2 Was ist eigentlich dieses .Net?

Die Vision

.Net (sprich dott nett) besteht neben dem, was Sie als Softwarentwickler direkt tangiert, aus insgesamt drei Komponenten:

- **Die .Net-Vision** – Hier hat Microsoft seine Version der nahen Zukunft der Nutzung des Internet manifestiert. Dabei geht es darum, dass alle elektronischen Geräte über ein weltumspannendes und überall verfügbares Breitbandnetzwerk verbunden sind. Damit einher geht eine andere Art der Software- und Dienstleistungsverteilung, in deren Mittelpunkt die Webservices stehen.
- **Das .Net-Framework** – Um Entwickler wie Anwender mit den Technologien zu versorgen, wird ein großer Teil der Dienste als Framework geliefert. Sie haben es bereits erfolgreich installiert, es ist also nicht mehr und nicht weniger als ein großes Stück Software.
- **Die .Net-Server** – Um in lokalen oder globalen Netzwerken selbst Dienstleistungen anbieten zu können, mit denen die Vision in Erfüllung gehen kann, werden Enterprise Server eingesetzt. Dazu gehören der SQL Server 2000 und der BizTalk Server 2000. Beide basieren, ebenso wie die zahlreichen andern Server, noch nicht auf dem .Net-Framework, nutzen aber die Technologie. Dies gilt auch für die Windows .Net-Server, die Nachfolger der Windows 2000 Server.

Das .Net-Framework

Jeder Beschreibung über ASP.NET liegt eine Einführung in das .Net-Framework zugrunde. Da Sie bereits eine unspektakuläre Installation hinter sich haben, folgt hier nun eine etwas spektakulärere Beschreibung.

Das klingt nach viel Theorie, für dieses Buch genügen jedoch wenige Grundbegriffe. Trotzdem ist ein Blick hinter die Kulissen sinnvoll.

Das Framework ist die Kerntechnologie des Marketingbegriffs .Net. Es zerfällt bei näherer Betrachtung in folgende Teile:

- **Die .Net-Framework Klassenbibliothek**. Hier finden die bereits erwähnten Klassen, die von der Netzwerkprogrammierung über Dateisystemzugriff bis zu dynamischen Bildern alles bieten, was bei der Programmierung notwendig sein könnte.
- **Die .Net-Sprachen**. Dazu gehören im Lieferumfang C#, VB.Net, JScript.Net und MC++. Weitere können von Drittanbietern bereitgestellt werden.
- **Die Common Language Runtime (CLR)**. Dies ist die Laufzeitbibliothek, die für die Ausführung der übersetzten Programme sorgt. Dazu wird der MSIL-Code Just-In-Time, also zur Laufzeit des Programms beim ersten Start, in Maschinensprache übersetzt.
- **Die Microsoft Intermediate Language (MSIL)**. Dies ist eine Zwischensprache, in die alle in den verschiedenen .NET-Sprachen übersetzt werden und die von der CLR ausgeführt werden kann.
- **ASP.NET** – die Erweiterung des IIS zur Erstellung dynamischer Webseiten.

Compiler und Interpreter

Zwischen dem Code, den Sie in Ihren Editor eintippen und dem vom Server ausgeführten liegt ein langer Weg. Bislang gab es nur zwei Arten von derartigen Übersetzern von vom Menschen geschaffenem Code in maschinenlesbaren: Compiler und Interpreter. Compiler übersetzen den Code in einem explizit gestarteten Lauf, benötigen dazu ein besonderes Programm und sind vergleichsweise aufwändig zu bedienen. Das Testen compilierter Codes ist nicht einfach, weil jede Änderung erst nach der erneuten Übersetzung wirksam wird. Sie benötigen außerdem einen Debugger zur Fehlersuche, der Zustände während der Laufzeit des Programms abfragt und anzeigt. Auch solche Programme sind nicht trivial und erfordern einiges Verständnis für die inneren Zusammenhänge. Auf der anderen Seite gibt es Interpreter, die erst im Augenblick des Abrufes eines Programms die Übersetzung Zeile für Zeile vornehmen. Änderungen wirken sich sofort aus und Programme zur Übersetzung sind nicht notwendig. Es liegt in der Natur der Sache, das Anfänger mit Interpretern besser und schneller zurecht kommen und sich derartige Programme einer gewissen Beliebtheit erfreuen. Das alte ASP basiert auf einem Interpreter, ebenso wie die bekannten Skriptsprachen PHP und Perl. Windows-Programme in C++, C-Programme unter Unix und Java werden dagegen compiliert – die Domäne der Profientwickler.

Mit .Net ändert sich an dieser Stelle etwas. Zum einen werden ASP.NET-Programme, ebenso wie jede andere Anwendung, grundsätzlich compiliert. Das haben Sie bereits getan und es vermutlich nicht einmal bemerkt. Tatsächlich dauert der erste Aufruf einer ASP.NET-Seite etwas länger als alle folgenden. Die ASP.NET-Komponente erkennt, dass die Seite noch nicht übersetzt wurde und führt die Übersetzung sofort und ohne jede Interaktion aus. Sie können das zwar durch diverse Einträge in den Code steuern, müssen es aber nicht. Dem Compiler ist damit der Schrecken genommen. Fehlermeldungen werden in einer gut lesbaren Form an den Browser gesendet. Die Übersetzung erfolgt jedoch nicht direkt in nativen Maschinencode für Ihren Pentium-Prozessor, sondern in eine Zwischensprache – die bereits erwähnte MSIL. Dies ist ein maschinennaher Code, der sehr viel schneller abgearbeitet werden kann, als es ein Interpreter mit dem Quellcode könnte. Dieser Code wird von der Common Language Runtime (CLR) ausgeführt, letztlich eine Art spezieller Compiler. Dieser Compiler ist ein so genannter Just-In-Time-Compiler (JIT-Compiler), er übersetzt ein Stück Code beim Abruf in Maschinensprache und speichert ihn dann, sodass nur bei Änderungen eine erneute Übersetzung notwendig wird.

Darin liegt das Geheimnis der Sprachunabhängigkeit. Der MSIL-Code, den C# erzeugt, ist identisch mit dem von VB.Net oder JScript.Net. Die MSIL kann man zwar anschauen, aber sie ist nicht dafür entworfen worden, von Menschen gelesen zu werden. Wenn Sie Assembler kennen, werden Sie einige Fragmente sicher lesen können.

Die Ausführung der Übersetzung in MSIL und die Ausführung mit der CLR wird beim Abruf von ASP.NET-Programmen automatisch erfolgen. Man muss sich aber dieses Prinzip vor Augen halten, um das Laufzeitverhalten zu verstehen und auch den Zeitpunkt, an dem Fehlermeldungen ausgegeben werden. Es gibt Fehler, die treten während der ersten Übersetzungsphase auf und andere erst beim Auftreten bestimmter Daten. Sie werden beides im Laufe Ihrer Arbeit mit ASP.NET mit Sicherheit kennenlernen.

1.4.3 Begriffe aus der .Net-Welt

In der .Net-Welt ist weiterhin immer wieder von den Klassen des Frameworks die Rede, genannt Basisklassen. Diese Klassen liefern alles, was im Programmieralltag benötigt wird. Vor allem aber – und dies ist ein Unterschied zu anderen Klassensystemen – liefern sie auch ein einheitliches Typsystem. Bislang kannte jede Programmiersprache eigene Datentypen; Ganze Zahlen (Integer), Zeichenketten oder komplexe Typen wie Arrays. Wenn nun ein Teil in C# und ein anderer in VB.Net geschrie-

ben wird, beide aber reibungslos zusammenarbeiten müssen, funktioniert das nur, wenn sich auch die Datentypen angleichen. Dies würde jedoch zu Kompromissen in allen eingesetzten Sprachen führen. Deshalb sind diese Spracheigenschaften in das Common Type System (CTS) ausgelagert.

Das Common Type System

Die Wahl der Datentypen und deren Präsentation ist von großer Bedeutung bei der Programmierung. Wenn Sie bereits JScript oder VBScript programmiert haben, werden Sie den Begriff „Datentyp" nur am Rande registriert haben. Skriptsprachen arbeiten typlos oder mit sehr losen Typen, die zur Laufzeit vom System selbst vergeben werden. .Net basiert auf einem sehr strengen Typkonzept – dies gilt für alle Sprachen gleichermaßen. Sie müssen sich also stets Gedanken darüber machen, welchen Typ eine Variable besitzen soll, das heißt, welche Datenart darin gehalten wird. Typbezeichner stehen natürlich weiterhin zur Verfügung. Intern gibt es aber ein Verknüpfung zwischen dem vom Framework gelieferten und dem in der Sprache definierten Typ. Ein Datentyp des Frameworks ist beispielsweise `System.Int32`. In C# nutzen Sie für ganze Zahlen `int`, in VB.Net `Integer`. Beides wird bei der Übersetzung in `System.Int32` umgewandelt. Das Framework kennt mehr Datentypen als die Sprachen, die jeweils mit einem Basissatz ausgestattet sind. Sie können immer direkt die Datentypen des Frameworks deklarieren. Wie das erfolgt, wird in der Spracheinführung zu C# erläutert.

CTS-Datentypen

An dieser Stellen soll die Präsentation der CTS-Typen selbst genügen.

Typ	Beschreibung	Wertebereich
`System.Boolean`	Logischer Wert	True oder False
`System.Byte`	Ein Byte	0 bis 255
`System.Char`	Zeichen	Ein Unicode-Zeichen
`System.DateTime`	Zeit- und Datumswerte	64-Bit-Zahl, die den Bereich vom 1.1.0001 (Jahr 1) bis 31.12.9999 repräsentiert
`System.Decimal`	Zahlen mit 28 Dezimalstellen	
`System.Double`	64-Bit-Gleitkommazahl	$-1{,}79^{308}$ bis $+1{,}70^{308}$
`System.Int16`	16-Bit-Ganzzahl	-32.768 bis +32.768
`System.Int32`	32-Bit-Ganzzahl	-2.147.483.648 bis +2.147.483.648
`System.Int64`	64-Bit-Ganzzahl	-9.223.372.036.854.775.808 bis +9.223.372.036.854.775.808
`System.Sbyte`	8-Bit-Zahl mit Vorzeichen	-128 bis +127
`System.Single`	4-Byte-Gleitkommazahl	$-3{,}4 \times 10^{38}$ bis $+3{,}4 \times 10^{38}$

Tabelle 1.1: Typen des Common Type System

Typ	Beschreibung	Wertebereich
`System.TimeSpan`	Positive oder negative Zeitspanne	-10675199.02:48:05.4775808 bis +10675199.02:48:05.47758 (Tage.Stunden:Minuten:Sekunden.Millisekunden)
`System.String`	Unicode-Zeichenkette	Beliebig viele Zeichen
`System.Array`	Datenfeld	Feld, das beliebige andere Typen enthalten kann, auch weitere vom Typ Array
`System.Object`	Objekt	Basistyp aller Typen

Tabelle 1.1: Typen des Common Type System (Forts.)

An der Beschreibung des letzten Typs, `System.Object`, wird klar, dass allen Typen Objekte zugrundeliegen. Auf Objekte wurde noch nicht weiter eingegangen, dies soll jedoch nachgeholt werden. Denn die Welt von .Net ist eine Welt der Objekte. Sie müssen – ausnahmslos – objektorientiert programmieren. Es liegt in der Natur von Objekten, sowohl über Eigenschaften als auch Methoden zu verfügen. Die folgende Information sollten Sie, wenn Sie mit der objektorientierten Programmierung noch nicht vertraut sind, einfach als gegeben hinnehmen. `System.Object` verfügt, als Basisklasse aller Datentypen, über folgende Methoden:

Methode	Beschreibung
`Equals`	Vergleicht zwei Objekt auf Gleichheit
`GetHashCode`	Gibt einen Identifikator (Hash) des Objekts zurück. Hashes sind spezielle Arten von Aufzählungen, deren Elemente so identifiziert werden können.
`GetType`	Gibt ein Typ-Objekt zurück, mit dem der Typ selbst untersucht werden kann. Diese Art des Zugriffs auf interne Definitionen wird **Reflection** genannt.
`ToString`	Mit dieser Methode wird der Typ in eine Zeichenkette umgewandelt. Sie haben dies bereits im ersten Beispiel verwendet.

Tabelle 1.2: Methoden aus `System.Object`

Sie finden im nächsten Abschnitt eine Einführung in die Prinzipien der objektorientierten Programmierung, die Sie lesen sollten, wenn Sie damit nicht vertraut sind.

1.4.4 Überblick über das Framework

Hier ist leider nur Platz für eine sehr kompakte Einführung, aber für die ersten Schritte in ASP.NET sollte dies ausreichend sein. Zuerst ein Bild, wie man sich das gesamte Framework vorstellen kann:

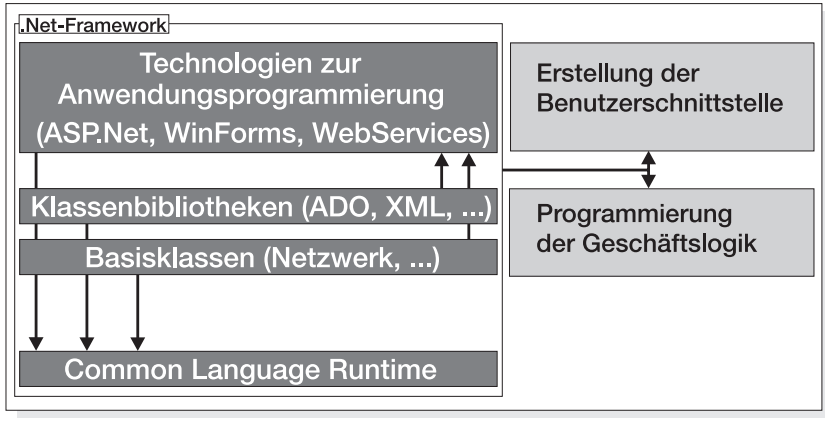


Abbildung 1.14: Grober Aufbau des Frameworks

Allen Bestandteilen liegt die Common Language Runtime zugrunde. Darauf setzen die Basisklassen auf. Basisklassen sind solche für einfache Datenmodelle (`System.Collections`), Multithreading (`System.Threading`) oder IO (`System.IO`) für den Zugriff auf das Dateisystem. Von den Basisklassen abgeleitet und ergänzt folgen die Klassenbibliotheken. Dazu gehören die Bibliotheken für den Datenbankzugriff ADO.NET, die XML-Bibliotheken oder solche für reguläre Ausdrücke. Noch komplexere Aufgaben erledigen die Klassen, die für den Anwendungsprogrammierer interessant sind: ASP.NET, WinForms und Webservices sind die wichtigsten Vertreter. In diesem Buch wird ASP.NET behandelt. Verwechseln Sie das nicht mit Webservices, die zwar auch auf Webservern ausgeführt werden, aber nicht einen Browser als Client erwarten, sondern einen anderen Computer.

Der Fokus ASP.NET bedeutet aber nicht, dass die anderen Stufen außer acht gelassen werden können. In vielen Abschnitten werden immer wieder Klassen aus allen drei Stufen der Bibliotheken verwendet werden. Dabei muss ein Zugriff auf eine Basisklasse keinesfalls komplizierter sein, als der auf eine Anwendungsklasse.

Referenz

Die Kunst beim Umgang mit dem Framework besteht im Wesentlichen darin, bei Bedarf den passenden Namensraum und darin die richtige Klasse zu finden. Die Online-Referenz ist dabei ein sehr wichtiges Arbeitsmittel, im Web unter folgender Adresse zu finden:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/cpref_start.asp

Für dieses Buch ist ein Ausschnitt aus verschiedenen Namensräumen verwendet worden. Am Anfang der Einführungen finden Sie teilweise

jeweils einen Ausschnitt aus der Klassenhierarchie. Diese Begriffe sind ein guter Startpunkt bei der Suche in der Referenz.

1.4.5 Die Welt der Objekte

Im letzten Abschnitt wurden die Datentypen des CTS vorgestellt. Zum Verständnis der Schreibweise müssen Sie sich zwangsläufig mit Objekten beschäftigen. Außerdem unterscheidet sich das Programmierprinzip des Frameworks etwas von der klassischen Programmierung. Dies wird in diesem Abschnitt erläutert.

Die Welt der Objekte

In .Net sind alle Dinge, mit denen Sie zu tun haben, Objekte. Auch wenn es nicht danach aussieht oder Sie dies nicht erwarten – es sind und bleiben Objekte. Objekte bieten für die moderne Programmierung Vorteile. Wer mit programmieren anfängt oder nur VBA oder VBScript kennt, wird sich damit schwer tun. Diesen Schritt müssen Sie am Anfang gehen, denn sonst werden Sie auch die einfachsten Beispiele nicht lesen können.

Wenn es schwer ist, warum legt Microsoft dann solchen Wert darauf, die Welt der Objekte zur alleinigen Herrschaft in der Programmierwelt zu führen? In einfachen Applikationen, die nur wenige Zeilen Code enthalten, gibt es tatsächlich kaum Vorteile. Wenn jedoch größeren Anwendungen geschrieben werden, wofür auch ASP.NET nun durchaus geeignet ist, dann wird es schwer, die Übersicht zu behalten. Der erste Schritt besteht darin, Techniken zu schaffen, die Codes wiederverwendbar machen. Dadurch werden Programme kleiner und die Fehlerquote sinkt, weil man auf geprüfte und ausgereifte Module zurückgreifen konnte. Im alten ASP war die einzige Möglichkeit der Modularisierung die Verwendung der SSI-Anweisung `INCLUDE`. Das ist völlig unzureichend, wenn man die Möglichkeiten der objektorientierten Programmierung daneben stellt.

Was ist ein Objekt?

Ein Objekt in Software – hier immer vereinfacht als Objekt bezeichnet – ist eine Sammlung von Code, der etwas aus der realen Welt konkret beschreibt. Das können Dinge wie Tiere oder Autos sein, aber auch Zustände oder Pläne wie Kalender oder Aufgaben. Man kann alles in der realen Welt als Objekt betrachten und deshalb ist die Abbildung in einer ähnlich Form in Software eigentlich genial. Objekte können in weitere, kleinere Objekte zerfallen. Stellen Sie sich ein Auto vor. Es zerfällt in Bauteile wie Räder, Türen, den Motor usw. Jedes Teil teilt sich wiederum in weitere Bauteile. Treibt man diese Überlegung sehr weit, endet man bei den Elementarteilchen. Irgendwo zwischen Elektronen und Quarks gibt es ein Basisobjekt.

Vergleichen Sie das mit einem komplexen Programm, beispielsweise Windows. Dort gibt es ganz elementare Objekte, die die Mausbewegung erfassen oder Punkte und Linien zeichnen. Aus diesen entsteht – über viele Stufen – ein bewegliches Fenster. So funktionieren Softwareobjekte.

Der Vorgang von der Abbildung realer Dinge in Software wird als Abstraktion bezeichnet. Als Programmierer muss man deshalb nicht nur logisch sondern auch abstrakt denken können. Sie versuchen dabei das, was Sie aus der Realität abbilden möchten, in ein abtraktes Modell zu packen. Je besser Ihnen das gelingt, desto einfacher und stringenter ist die Lösung. Guter objektorientierter Code hat einige Eigenschaften:

- Er repräsentiert die menschliche Sicht auf die abgebildeten Dinge
- Er hilft, Code effizienter zu schreiben
- Er hilft, Code einfacher zu schreiben
- Der Code ist leicht wiederverwendbar und leicht anpassbar
- Guter Code bildet Dinge nicht kryptisch, sondern direkt ab

Das klingt alles sehr gut, ist aber in der Praxis nur mit einiger Erfahrung zu erreichen. .Net hilft Ihnen aber dabei, denn wenn alles ein Objekt ist und C# eine objektorientierte Programmiersprache, dann kann das Schreiben entsprechenden Codes nicht schwer sein.

Objekte haben Eigenschaften

Objekte haben Eigenschaften. Das ist notwendig, um Dinge zu beschreiben. Die Wahl der passenden Eigenschaften ist wichtig. Sie sollten immer im Kontext der Verwendung betrachtet werden. Die Eigenschaften eines Autos sind beispielsweise Farbe, Länge oder Marke. Vielleicht denken Sie, dass die Leistung auch eine gute Eigenschaft wäre. Das ist zu wenig abstrahiert betrachtet. Denn das Objekt Auto enthält ein Objekt Motor, für das die Eigenschaft Leistung viel besser geeignet wäre. Ein Auto kann nämlich verschiedene Motoren haben und damit auch jeweils eine andere Leistung. Wird das Auto-Objekt aus dem Motor-Objekt abgeleitet, erbt es quasi dessen Eigenschaften mit. Sowohl Motor als auch Auto sind dann wiederverwendbar – wie in der realen Welt.

Objekte haben Methoden

Dinge haben nicht nur Eigenschaften, sie können auch Aktionen ausführen. Ein Auto kann fahren oder Türen öffnen und schließen. Steigt man in der Objekthierarchie wieder noch unten – zur Wurzel hin, werden andere Methoden interessant. So wäre für das Getriebe eine Methode „Gang höher“ oder „Gang niedriger“ interessant. Eine Methode „3. Gang“ ist dagegen nicht sinnvoll, weil es nicht immer möglich und sinnvoll sein kann, direkt in einen Gang zu schalten. Dagegen ist eine Eigenschaft „Gang“ besser geeignet, da dort der aktuelle Zustand abgefragt werden kann. Eben diese Überlegungen sollten auch bei Software

angestellt werden, was nicht so schwer ist, weil es nur um die Abbildung realer Dinge geht – theoretisch jedenfalls.

Die bisherige Beschreibung sagt noch nichts über die Verwendung aus. Es ist sinnvoll, Objekte so zu gestalten, dass man sie einsetzt, ohne das Innenleben zu kennen. Der Inhalt ist gekapselt. Sie können Auto fahren, ohne genau zu wissen, wie jedes Bauteil aufgebaut ist und funktioniert. Ebenso verhält es sich bei Objekten. Sie sollten so konstruiert sein, dass sie einfach zu benutzen sind.

Baupläne für Objekte

Objekte entstehen nicht einfach so, als Einzelstücke, sondern auf der Basis eines Bauplanes. Schließlich will man nicht nur ein Auto herstellen, sondern möglichst viele nach demselben Muster. Ein solcher Bauplan wird in objektorientierter Software eine Klasse genannt. Die Klassen des .Net-Frameworks sind also eine gigantische Sammlung von Bauplänen für Ihre Software. Wenn aus einer Klasse ein Objekt abgeleitet wird, dann spricht man von Instanziierung – das Objekt ist eine Instanz der Klasse.

Eine genaue Herleitung der Schlüsselwörter, die für Klassen und Objekte in C# genutzt werden, finden Sie im Abschnitt »Objektorientierte Programmierung« auf Seite 77 bei der Beschreibung der Sprachelemente.

Namensräume

Im Zusammenhang mir der Programmierung in .Net stösst man schnell auf den Begriff Namensraum (engl. Namespace). Namensräume haben zwei grundlegende Aufgaben:

- Sie teilen zusammengehörende Systemtypen in logische Gruppen ein.
- Sie verhindern Namenskonflikte zwischen Dateien.

Systemtypen sind in .Net alles, was die Basis für Objekte ist. Dies sind die schon erwähnten Klassen, aber auch komplexe Strukturen, Datenfelder, Aufzählungen und Schnittstellen – letztere sind quasi spezialisierte Baupläne. Jeder derartige Systemtyp liegt in irgendeinem Namensraum vor. Da damit Namenskonflikte vermieden werden, können innerhalb eines Bauplanes für Eigenschaften und Methoden Namen verwendet werden, die auch außerhalb des Bauplanes auftreten. Dies ist eine wichtige Eigenschaft objektorientierter Programmierung, denn so können Sie ihre Baupläne anderen Programmierern zur Verfügung stellen oder solche von anderen benutzen, ohne Angst zu haben, dass sich Konflikte ergeben. In VBScript war dies ein großes Problem, weil es nur einen einzigen Namensraum gab. In .Net gibt es viele – unendlich viele – durch immer neue Definitionen.

Namensräume existieren nicht physisch, sie werden also nicht dadurch gebildet, dass der Code in DLLs oder ausführbaren Dateien oder Modulen untergebracht wird. Wenn Sie sich Namensräume vorstellen wollen, denken Sie an so etwas ähnliches wie Schubladen oder Ordner. Zwei Schubladen können zwei gleiche Objekte enthalten und diese kann man trotzdem anhand der Lage in dem einen oder anderen Schubfach unterscheiden.

Die Verwendung eines Namensraumes muss in C# erklärt werden, dazu dient das Schlüsselwort `using`. Sie werden das in vielen Programmbeispielen aber nicht sehen, weil ASP.NET die wichtigsten bereits automatisch importiert. Auch dies wird natürlich noch genauer erörtert.

Übrigens wird oft auch keine richtige Klasse geschrieben und ein Objekt daraus abgeleitet. Viele ASP.NET-Beispiele sehen wie konventioneller prozeduraler Code aus. Tatsächlich nutzt aber ASP.NET Klassen des Frameworks, um aus Ihrem Code komplette objektorientierte Quellen zu erzeugen und diese dann zu übersetzen. Dies ist ein Trick, der den Einstieg in ASP.NET vereinfacht und von dem Windows-Programmierer nicht profitieren können. Mehr Vorteile für ASP.NET-Entwickler werden im nächsten Abschnitt gezeigt.

1.4.6 Vorteile für den ASP.NET-Entwickler

ASP.NET ist gegenüber ASP keine Evolution, sondern ein völlig neues Produkt. Fast alles, was an ASP kritisiert wurde, ist in ASP.NET nicht mehr zu finden.

Script Limits

ASP hatte immer weitreichende Einschränkungen, die durch die Verwendung einer Skriptsprache bedingt waren. Sie mussten deshalb oft auf Komponenten ausweichen, die in VB oder C++ geschrieben wurden – in richtigen Programmiersprachen also. Das ist nun nicht mehr notwendig. C# ist eine „richtige“ Programmiersprache und Sie müssen auf keine Funktion verzichten. Alles, was vorher in Komponenten geliefert wurde, kann nun direkt programmiert werden. Selbstverständlich lassen sich fertige Programme kompilieren und anderen zur Verfügung stellen. Damit einher geht natürlich auch ein Verlust der Einfachheit – statt typloser Programmierung muss das bereits gezeigte sehr strenge Typsystem CTS verwendet werden.

Verteilung

Die Installation fertiger ASP-Anwendungen war bislang kompliziert, wenn es sich um ASP-Code handelte, der COM-Komponenten enthielt. ASP.NET-Anwendungen lassen sich sehr leicht auf einem Entwicklungssystem erstellen und dann auf einen Produktionsserver bringen. Es genügt, die fertigen Programme zu kopieren. Weder Struktur noch

Dateien müssen geändert werden. Legen Sie auf dem Produktionsserver ein virtuelles Verzeichnis an, kopieren Sie alle Daten dort hinein – fertig. Es ist nicht notwendig, in die Metabasis des IIS einzugreifen oder Konfigurationen vorzunehmen. Alles, was konfiguriert werden muss, passiert in gut lesbaren XML-Dateien, die in den jeweils zu konfigurierenden Verzeichnissen liegen. ASP.NET liest diese Dateien und greift während der Laufzeit nur auf den Speicher zu, registriert aber sofort, wenn sich Änderungen ergeben, sodass Sie weder den Server noch den Webserver oder einen anderen Dienst neu starten müssen.

Übersetzungsvorgang

Das ASP.NET kein Interpreter ist, sondern ein Compiler, wurde bereits angesprochen. Der Vorgang des Übersetzens und der Nutzung der Codes ist weitgehend transparent, sodass Sie dafür nichts tun müssen. Dennoch läuft dies nicht im geheimen Inneren von Windows ab, sondern ist sowohl sichtbar als auch kontrollierbar. ASP.NET erzeugt in einem ersten Lauf aus der Seite, die sowohl HTML als auch Code in einer der .Net-Sprachen enthält, eine so genannte „Page“. Der Page-Compiler erstellt daraus eine `Page`-Klasse, wozu auf verschiedene Basisklassen zurückgegriffen wird (`System.CodeDOM` und `System.DLL`). Diese Klasse wird übersetzt und als Assembly abgelegt. Assemblies sind die Nachfolger der DLLs, wobei die Dateierweiterung *dll* erhalten geblieben ist. Assemblies enthalten viele Zusatz- und Konfigurationsinformationen, sodass die Verwaltung einfacher und sicherer ist, als bei DLLs. Wenn Sie das erste Beispiel ausgeführt haben, können Sie die DLL bereits sehen. Sie ist in folgendem Pfad zu finden:

%systemroot%\Microsoft.Net\Framework\v1.0.3705
Temporary ASP.NET Files

Haben Sie ein virtuelles Verzeichnis mit dem Namen *aspdotnet* verwendet, folgt dieser Name als weiterer Ordner. Darunter sind möglicherweise mehrere Verzeichnisse zu finden, die Versionsnummern enthalten. So können mehrere Versionen koexistieren. Darunter wiederum liegen die DLLs – die Assemblies. Sie müssen das nicht weiter beachten; wenn Ihnen der Pfad nicht gefällt, können Sie ihn auch ändern. ASP.NET ist sehr offen, was die Einflussnahme angeht und auch in der Darstellung dessen, was während der Laufzeit passiert.

ASP.NET in Windows

Sie arbeiten, trotz .Net-Framework und ASP.NET, natürlich immer noch im klassischen Windows 2000 oder XP. Auch die neuen .Net-Server dienen lediglich als Host für .Net, sind aber nicht selbst in .Net programmiert. Für ASP.NET ist deshalb interessant, wo die Schnittstelle zwischen dem alten und neuen System ist – es ist der Webserver IIS. ASP.NET ist eine herkömmliche ISAPI-Anwendung.

Die Konfiguration von ASP.NET wird in diesem Buch nicht behandelt. Mit den Standardeinstellungen lassen sich alle Programme ausführen. Bei einem Provider können Sie meist nicht alle Einstellungen selbst vornehmen – vor allem aus Sicherheitsgründen. Informationen darüber entnehmen Sie dem Buch „ASP.NET-Programmierung mit C#“, erschienen 2002 bei Addison Wesley.

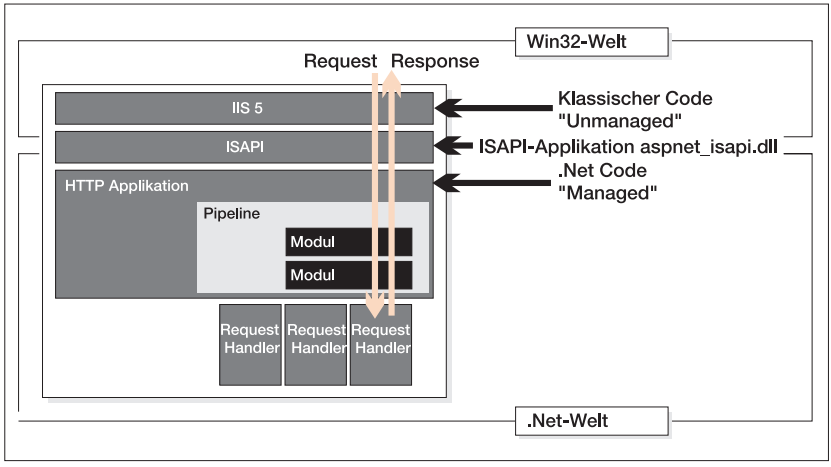


Abbildung 1.15: Struktur der Anforderungsverarbeitung in ASP.NET

Im Kern basiert ASP.NET auf einer HTTP-Applikation, der HTTP-Laufzeitumgebung. Diese Komponente empfängt von der ISAPI-Anwendung die Anforderung (so genannter Request vom Browser) und leitet sie über Module – die sie programmieren können – an den Request Handler weiter. Dieser Teil ist für die Ausführung des angeforderten Codes verantwortlich. Er kann aber auch nur einfache statische Seiten ausliefern, wenn diese zwar wegen der Erweiterung *.aspx* den Prozess in Gang setzten, aber keine Programmierung enthielten. Was im Wesentlichen hier erledigt wird, ist die korrekte Nutzung des Protokolls HTTP. Darüber müssen Sie sich keine Gedanken machen. Es ist aber auf der anderen Seite hilfreich, HTTP zu kennen. Denn dann fällt es Ihnen leichter zu verstehen, was die entsprechenden Methoden und Eigenschaften der HTTP-Handler-Klassen bedeuten und wie man sie sinnvoll in der Programmierung einsetzen kann. Der nächste Abschnitt bietet deshalb eine kompakte Einführung in das Protokoll HTTP.

1.4.7 Das Protokoll HTTP

In diesem Abschnitt erfahren Sie das Wichtigste über HTTP (HyperText Transfer Protocol), das in der Webserver-Programmierung eine herausragende Rolle spielt.

Einführung

HTTP dient der Kommunikation mit Webservern. Es gibt zwei Versionen, 1.0 und 1.1. Auf Seiten der Browser dominiert inzwischen HTTP 1.1, denn alle Browser ab Version 4 beherrschen dieses Protokoll. Der Internet Information Server 5 beherrscht die Version 1.1 vollständig.

RFC 1945

HTTP 1.0 wurde im Mai 1996 in der RFC 1945 veröffentlicht, schon im August desselben Jahres folgte HTTP 1.1.

Verbindungsloses Protokoll

Bei HTTP handelt es sich um ein verbindungs- oder statusloses Protokoll. Server und Client nehmen also nie einen besonderen Zustand ein, sondern beenden nach jedem Kommando den Prozess vollständig, entweder mit Erfolg oder mit einer Fehlermeldung. Es obliegt dem Kommunikationspartner, darauf in angemessener Weise zu reagieren.

Protokollaufbau, Header, Body

HTTP-Kommandos werden als ASCII-Text übertragen und können aus mehreren Zeilen bestehen. Die erste Zeile ist immer die Kommandozeile. Daran angehängt kann ein so genannter Message-Header (Kopf der Nachricht) folgen. Der Nachrichtenkopf enthält weitere Parameter, die das Kommando näher beschreiben. So kann ein `Content-Length`-Feld enthalten sein. Steht dort ein Wert größer als 0, folgen dem Nachrichtenkopf Daten. Die Daten werden also gleich zusammen mit dem Kommando gesendet, man spricht dann vom Body (Nachrichtenkörper) der Nachricht. HTTP versteht im Gegensatz zu anderen Protokollen den Umgang mit 8-Bit-Werten. Binärdaten, wie Bilder oder Sounds, müssen nicht konvertiert werden. Folgen dem HTTP-Kommando und den Nachrichtenkopf-Zeilen zwei Leerzeilen (Zeilenwechsel), so gilt das Kommando als beendet. Kommandos mit Nachrichtenkörper haben kein spezielles Ende-Zeichen, das `Content-Length`-Feld bestimmt, wie viele Bytes als Inhalt der Nachricht betrachtet werden.

Kommandoaufbau

Aufbau eines HTTP-Kommandos

Ein HTTP-Kommando hat immer folgenden Aufbau:

```
METHODE ID VERSION
```

Als `METHODE` wird das Kommando selbst bezeichnet. Die folgende Tabelle zeigt die HTTP-Kommandos auf einen Blick.

Kommando	Bedeutung
DELETE	Ressource löschen
GET	Ressource anfordern
HEAD	Header der Ressource anfordern
LINK	Verknüpfung zweier Ressourcen beantragen
OPTIONS	Optionen des Webservers erfragen
POST	Daten an einen Serverprozess senden
PUT	Ressource auf dem Webserver ablegen
TRACE	Kommando zurückschicken lassen
UNLINK	Verknüpfung zwischen Ressourcen löschen

Tabelle 1.3: HTTP-Kommandos

Beachten Sie, dass die Kommandos unbedingt in Großbuchstaben geschrieben werden müssen, exakt wie in der Tabelle 1.3 gezeigt. Als Ressource werden all die Objekte bezeichnet, die übertragen werden können – in der erster Linie also HTML-Dateien und Bilder.

Die ID einer Ressource kann beispielsweise eine Adresse oder ein Dateiname sein:

```
GET index.htm HTTP/1.0
```

Dieses Kommando fordert die Datei *INDEX.HTM* an.

Statuscodes

Die Antwort auf ein Kommando besteht im Senden der Daten – wenn dies gefordet wurde – und einem Statuscode. Dem Statuscode folgen optionale Felder und, bei der Übertragung von Ressourcen, die Daten. Die Statuszeile hat folgenden Aufbau:

```
VERSION STATUSCODE STATUSTEXT
```

Der Statuscode ist eine dreistellige Zahl, von der die erste Ziffer (Hunderterstelle) die Zuordnung zu einer bestimmten Gruppe anzeigt.

HTTP-Code	Bedeutung
200	Kommando erfolgreich (nach GET/POST)
201	Ressource wurde erstellt (nach PUT)
202	Authentifizierung akzeptiert (nach GET)
204	Kein Inhalt oder nicht angefordert (GET)
301	Ressource am anderen Ort
302	Ressource nicht verfügbar (temporärer Zustand)
304	Ressource wurde nicht verändert (steuert Proxy)
400	Syntaxfehler (alle Kommandos)

Tabelle 1.4: HTTP-Statuscodes

HTTP-Code	Bedeutung
401	Keine Autorisierung
403	Nicht öffentlicher Bereich
404	Nicht gefunden (`GET`)
500	Serverfehler, Fehlfunktion
502	Kommando nicht implementiert

Tabelle 1.4: HTTP-Statuscodes (Forts.)

Sie werden den Fehler 404 sicher kennen. Kennenlernen werden Sie auch den Fehler Nummer 500, der erzeugt wird, wenn ein Programm nicht funktioniert, dass Sie in ASP.NET geschrieben haben.

HTTP-Message-Header

An ein Kommando oder an die Statuszeile können weitere Felder angehängt werden. Der Aufbau lehnt an den MIME-Standard an:

```
Feldname Wert; Wert
```

MIME steht für Multipurpose Internet Mail Standard und definiert, wie bestimmte Dateiarten über Internet übertragen werden können. MIME wird nicht nur mit E-Mail, sondern unter anderem auch mit HTTP eingesetzt.

Die Nachrichtenkopffelder können in drei Hauptgruppen aufgeteilt werden:

- **F** – Frage-Felder (Request-Header-Fields), die nur in Kommandos erlaubt sind
- **A** – Antwort-Felder (Response-Header-Fields), die Statusnachrichten vorbehalten sind
- **I** – Informationsfelder (General-Header-Fields), dienen der Übertragung aller anderen Nachrichten in die eine oder andere Richtung

Eine typische Anwendung, die bei der ASP.NET-Programmierung auftreten kann, ist die Übergabe eines Nachrichtenkopfes, der einen besonderen Dateityp angibt:

```
Content-Type: application/pdf; name=aspnet.pdf
```

Freilich bietet dafür ASP.NET eine Methode innerhalb der entsprechenden Klasse an. Wenn diese Möglichkeiten aber nicht ausreichen, sind die Kenntnisse der Protokolle wichtig. Ebenso ist es hilfreich, die Begriffe zu kennen, um in großen Zahl von Klassen die passende zu finden.

Im Gegensatz zu anderen Protokollen ist die Länge eines Datenblocks im `Content-Length` festgelegt, irgendwelche Begrenzungszeichen gibt es nicht. Wichtig ist auch, dass der Server nach dem Verbindungsaufbau keine Antwort sendet. Erst das erste eintreffende Kommando löst eine

Reaktion aus. Darin ist die Ursache zu sehen, wenn der Browser nach der Anforderung eines unerreichbaren Server lange Zeit nicht reagiert. Als „Totsignal“ wird einfach eine vorgegebene Zeitspanne gewartet, in welcher der Server auf das erste Kommando reagieren sollte.

Verbindungsablauf

Eine einfache HTTP-Verbindung könnte also folgendermaßen aussehen:

```
Client: (Verbindungsaufbau des Browsers durch Nameserver, TCP/IP)
Server: (keine Antwort)
Client: GET /default.aspx HTTP/1.0
Server: HTTP/1.0 200 Document follows
        Date: Mon, 04 Mar 2002 12:23:55 GMT+100 Server: IIS 5.1,
              Microsoft Corporation
        Content-Type: text/html
        Last-Modified: Mon, 04 Mar 2002 12:23:55
        Content-Length: 1465
        JDGGF/(&:=$(?ED`D`?I`... Daten entsprechend der Längenangabe
```

Listing 1.3: Ablauf einer einfachen HTTP-Verbindung

Der Ablauf ist also recht simpel. Praktisch wird in diesem Beispiel eine Datei mit dem Namen *default.aspx* angefordert. ASP.NET startet dann, führt den Code in der Seite aus, produziert den Inhalt der Seite und gibt ihn zusammen mit den richtigen Headern an den Webserver. Dieser setzt den Code 200 – Alles OK – davor und sendet alles an den Browser. Das der Benutzer dann mit den Daten etwas anfangen kann, dafür sind Sie verantwortlich. Den Rest können Sie vorerst ASP.NET und dem IIS überlassen. Profis wissen natürlich, dass sich hier trickreich eingreifen lässt. Im Normalfall ist das aber nicht notwendig.

1.5 Aufgaben

Die folgenden Aufgaben dienen zur Überprüfung des gelernten. Die Antworten sind auf der Website zum Buch zu finden. Dort ist der gesamte Fragekurs des Buches nochmals komplett zu finden, einschließlich eines Bewertungssystems, dass Sie in der Praxis einsetzen können. Fragen finden Sie am Ende jedes Kapitels.

1. Welche Vision steckt hinter .Net?
2. Was sind die drei Bestandteile von .Net?
3. Was heißt CLR und CTS? Was bedeutet es?
4. Welche Programmiersprachen können zur ASP.NET-Programmierung verwendet werden?
5. Welches Protokoll hat in der Webserver-Programmierung eine herausragende Bedeutung?

2 Die Programmiersprache C# und die Basisklassen

lernen

Nachdem im ersten Kapitel einige grundlegende Zusammenhänge gezeigt wurden, die zum Verständnis der Webserverprogrammierung notwendig sind, wird für den praktischen Teil eine solide Basis benötigt. Dafür sind Kenntnisse einer Programmiersprache notwendig. Dieses Kapitel führt kompakt in C# ein, der wichtigsten Sprache in .Net. Es ist kein Ersatz für C#-Bücher, hilft aber, alle Programmierbeispiele des Buches, die in C# geschrieben sind, zu verstehen und nachzuvollziehen.

Vorgestellt werden auch wichtige Klassen des Namensraumes `System`, die elementare Funktionen enthalten, die in anderen Programmiersprachen Bestandteil der Sprache sind.

2.1 Was Sie in diesem Kapitel lernen

Diese Kapitel führt in die Sprache C# ein. Da diese zur Klasse der objektorientierten Programmiersprachen gehört, spielt die Vermittlung objektorientierter Techniken eine große Rolle. An einigen Stellen werden auch Basisklassen oder die für ASP.NET benötigten Klassen verwendet. Eine ausführliche Erläuterung folgt in Kapitel 3. Bevor Sie sich jedoch in die Vielfalt der Klassen und von ASP.NET stürzen, müssen Sie die Handwerkszeuge beherrschen. In diesem Buch ist dies in erster Linie C#. Sie sollten auch unbedingt das Codebeispiel aus Kapitel 1 nachvollzogen haben, denn alle folgenden Beispiele basieren auf derselben Technik.

Jede Programmiersprache enthält einige elementare Sprachteile, die sich weniger durch die Funktionalität, sondern mehr durch die Art der Notation unterscheiden. Sie finden am Anfang der jeweiligen Abschnitte eine Darstellung der theoretischen Grundlagen. Wenn Sie andere Sprachen kennen, können Sie das überlesen. Anfänger in der Programmierwelt sollten den gesamten Teil durcharbeiten. Schwerpunkte sind:

- **Einführung in die Sprache** – Variablen, Konstanten, Operatoren
- **Objektorientierte Programmierung** – Klassen, Objekte, Vererbung
- **Steueranweisungen** – Schleifen, Verzweigungen
- **Erweiterte Techniken** – Fehlerbehandlung, Ereignisbehandlung, Kollektionen

Am Ende des Kapitels folgen Übungsaufgaben, die zur Vertiefung und Festigung dienen. Auf der Website zum Buch finden Sie alle Aufgaben des Buches in Form eines Kurses, mit den Lösungen und einer Auswertung, wenn Sie die Aufgaben online ausführen.

Die Beispiele im ersten Teil sind vergleichsweise trivial. Sie sollten sie dennoch ausprobieren, verändern, damit spielen – so lernen Sie schnell die Reaktionen des Compilers auf unvermeidliche Fehler kennen.

2.2 Einführung in die Sprache

Grundlegende Programmierkenntnisse werden nur ansatzweise vorausgesetzt. Wenn Sie bereits in VBA, VBScript, JavaScript oder PHP codiert haben, können Sie diesen Abschnitt zügig durcharbeiten. Versuchen Sie auf jeden Fall, jedes Beispiel zu verstehen. Ändern Sie sie, um die Reaktion des Programms zu bobachten.

2.2.1 Schreibweise und Notation

C# gehört zur Gruppe der so genannten Klammersprachen. Das ist kein offizieller Begriff, sondern deutet nur einige Prinzipien der Notation an, die neben C# auch Sprachen wie Java, C, C++, PHP und vielen anderen eigen sind.

Zeilenabschluss

Zuerst gehört dazu der Abschluss der Befehlszeilen mit einem Semikolon:

```
C#-Anweisungen;
```

Es ist in den meisten Fällen möglich, über mehrere Zeilen zu schreiben. Zeilenumbrüche oder Leerzeichen – die so genannten Whitespaces – spielen kaum eine Rolle. Entscheidend ist der Abschluss einer Anweisung mit dem Semikolon.

Methodenaufruf

Es gibt in C# keine Prozeduren, wie in Visual Basic oder VBScript. Es gibt nur Methoden.

An dieser Stelle erwarten Sie vielleicht eher das Wort Funktionen. Tatsächlich existieren Funktionen immer – ohne jede Ausnahme – in Klassen. Sie können nicht außerhalb der objektorientierten Welt programmieren.

Funktionen in Klassen werden aber als Methoden bezeichnet. Eine Methode ist die Funktion eines Objekts.

Methodenaufrufe benötigen immer – ohne Ausnahme – runde Klammern:

```
methode();
```

Diese Klammern werden auch dann geschrieben, wenn keine Parameter übergeben werden. Ansonsten werden die erwarteten Parameter dort deklariert.

Blockbildung

Der Begriff Klammersprache stammt von der geschweiften Klammer, mit der Blöcke umfasst werden. Es gibt keine „Begin"- und „End"-Schlüsselwörter, sondern nur Klammern:

```
Aktion_fuer_mehrere_Befehle
{
   Aktion_1;
   Aktion_2;
}
```

Die Klammern dürfen lediglich entfallen, wenn nach der Aktion nur ein weiterer Befehl angesprochen wird. Das Ende eines Klammerausdrucks muss nicht mit einem Semikolon abgeschlossen werden.

Objektschreibweise

In der Einleitung wurde bereits gezeigt, dass Objekte – neben vielen anderen Merkmalen – aus Eigenschaften und Methoden bestehen. Beides wird in C# mit einer besonderen Schreibweise dargestellt. Methoden werden durch einen Punkt vom Objekt getrennt und sind durch runde Klammern gekennzeichnet:

```
Objekt_oder_Klassenname.MethodenName();
```

Der Objekt- oder Klassenname kann entfallen, wenn die Zuordnung eindeutig ist. Dadurch wirken manche Programme so, als ob prozeduraler Code geschrieben würde. Tatsächlich ist dies nicht der Fall. Der Compiler nimmt die Zuordnung nur implizit vor.

Eigenschaften entsprechen Variablen in der klassischen prozeduralen Programmierung und werden ohne Klammern geschrieben, aber auch durch einen Punkt vom Klassen- oder Objektnamen getrennt:

```
Variable = ObjektName.EigenschaftsName;
```

Prinzipielle Verhaltensweisen

C# kennt noch einige Besonderheiten, die Sie wissen müssen. So wird generell zwischen Groß- und Kleinschreibung unterschieden. Eine Variable `Var` und eine mit dem Namen `var` sind nicht identisch. Weiterhin ist C# eine streng typisierte Sprache. Sie müssen – ausnahmslos – jeder Variablen mitteilen, welchen Datentyp sie enthalten darf. Manchmal

sorgen Methoden oder Befehle dafür, dass der richtige Typ erzeugt wird. Auch wenn C# dann implizit den Typ annimmt und die Notation dies nicht zum Ausdruck bringt, wird dieses Prinzip nie verletzt. Für die Änderung oder Umwandlung von Datentypen gibt es spezielle Operatoren. Dazu finden Sie genauere Informationen im nächsten Abschnitt.

2.2.2 Prinzip der Abarbeitung

Auf die Prinzipien der Arbeitsweise von Webseiten wurde bereits in Kapitel 1 hingewiesen. Alle folgenden Programme haben folgenden grundsätzlichen Aufbau:

```
<%@ Page Language="C#" %>
<html>
  <head><title>C# lernen</title></head>
  <body>
  <h1>Eingebetter Code:</h1>
  <%
  string hw = "Hello World";
  Response.Write ( hw );
  %>
  </body>
</html>
```

Listing 2.1: Standard-Skript mit eingebettetem C#-Code (basisscript.aspx)

Die erste Zeile enthält eine Seitendirektive. Darauf wird in Kapitel 3 detailliert eingegangen. An dieser Stelle schreiben Sie die Zeile so ab, wie sie ist. Sie dient der Festlegung von C# als Standardsprache. Dann folgt die HTML-Seite, die Ihr Browser anzeigen soll. Der Code selbst ist in ASP-Tags <% %> eingebettet. Dieses Prinzip hat sich gegenüber ASP nicht geändert – es sind nur weitere, alternative Wege hinzugekommen, die in Kapitel 3 genauer diskutiert werden.

Vergessen Sie nun nicht, das Programm mit der Erweiterung *.aspx* zu speichern. Führen Sie es über den Browser aus, wobei auf einem Entwicklungssystem der Name „localhost“ verwendet wird.

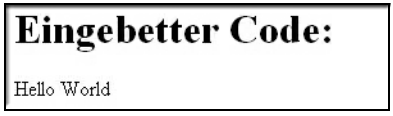


Abbildung 2.1: Ausführung des Programms basisscript.aspx

Wenn das gezeigte Bild auch bei Ihnen zu sehen ist, können Sie alle nachfolgenden Beispiele ausführen und bequem C# und ASP.NET lernen.

Es ist empfehlenswert, „sprechende“ Variablennamen zu verwenden. Dabei geht man davon aus, dass Variablen Zustände beschreiben – die Namen werden deshalb in der Regel durch Substantive gebildet. Zum Vergleich führen Methoden Aktionen aus, sie werden deshalb mit Verben bezeichnet. Häufig sind auch Namen zu sehen, die zur Trennung mehrerer beschreibender Worte Großbuchstaben verwenden. Ebenfalls üblich sind Vorsätze, die auf die Art der Daten hinweisen, die in der Variablen gespeichert werden, beispielsweise „s“ für Zeichenketten (Strings). Damit keine Missverständnisse aufkommen: Dies sind nur Vorschläge und sie haben sich in der Praxis bewährt. Solche Konventionen sind kein Zwang. Sie werden Ihnen aber helfen, in größeren Projekten den Überblick zu bewahren.

2.2.3 Variablen, Konstanten und Datentypen

Variablen dienen der Speicherung von Werten während der Verarbeitung des Programms. Der Variablenname kann beliebig lang sein und besteht aus Buchstaben und Zahlen und dem Unterstrich. Er darf jedoch nicht mit einer Zahl beginnen. Deklaration und Zuweisung kann kombiniert werden, muss aber nicht. Zuerst eine einfache Deklaration:

```
String ZeichenKette;
```

Jetzt existiert eine Variable `ZeichenKette` vom Typ `String`. Diese Variable ist noch leer. Sie kann mit dem Zuweisungsoperator gefüllt werden:

```
ZeichenKette = "Text";
```

Deklaration und Zuweisung kann auch kombiniert werden:

```
String Zeichenkette = "Text";
```

Intern verwendet C# natürlich die Datentypen, die vom Framework geliefert werden. Die folgende Tabelle zeigt, welcher Typ in C# wie dargestellt wird. Die Bedeutung der Typen wurde bereits in Tabelle 1.1 gezeigt. Die folgende Tabelle stellt nun die Typen des Frameworks und die internen Notationen von C# gegenüber. Als nicht native Datentypen werden hier solche bezeichnet, die in C# nicht direkt, sondern ausschließlich über das Framework, zur Verfügung stehen.

Typ	Beschreibung	C#-Notation
`System.Boolean`	Logischer Wert	`bool`
`System.Byte`	Ein Byte	`byte`
`System.Char`	Ein Zeichen	`char`
`System.DateTime`	Zeit- und Datumswerte	nicht nativ

Tabelle 2.1: Datentypen in C# im Vergleich zum Framework

Typ	Beschreibung	C#-Notation
System.Decimal	Zahlen mit 28 Dezimalstellen	decimal
System.Double	64-Bit-Gleitkommazahl	double
System.Int16	16-Bit-Ganzzahl	short
System.Int32	32-Bit-Ganzzahl	int
System.Int64	64-Bit-Ganzzahl	long
System.Sbyte	8-Bit-Zahl mit Vorzeichen	sbyte
System.Single	4-Byte-Gleitkommazahl	float
System.TimeSpan	Positive oder negative Zeitspanne	nicht nativ
System.String	Unicode-Zeichenkette	string

Tabelle 2.1: Datentypen in C# im Vergleich zum Framework (Forts.)

Die Zahltypen (`int`, `long` usw.) können mit dem Präfix „u“ ergänzt werden, um vorzeichenlos zu arbeiten („u“ steht für „unsigned“).

Wenn Sie Variablen deklarieren, müssen Sie nur dann das Framework direkt verwenden, wenn der Typ in C# nicht direkt verfügbar ist. Die Deklaration aus dem ersten Beispiel könnte auch folgendermaßen aussehen:

```
System.String hw = "Hello World";
```

Dies erzeugt exakt denselben Code – intern greift C# nämlich immer auf die Datentypen des Frameworks zurück. Die Nutzung der C#-Notation spart lediglich Schreibarbeit.

Das Typkonzept genauer betrachtet

Die bisherige Einführung umfasste nur die einfachen Datentypen. Das Typkonzept von C# ist weit umfassender. Unterteilen kann man es global in folgende Typenklassen:

- Werttypen
- Referenztypen

Werttypen

Die Werttypen werden verwendet, um einer Variablen einen Wert von definiertem Typ zuzuweisen. Das wurde bereits unter dem Begriff Datentypen beschrieben. Diese Werttypen umfassen außer den einfachen Datentypen auch noch Strukturen und Aufzählungen. Beides finden Sie im Abschnitt „Zugriff über Aufzählungen“ auf Seite 295.

Referenztypen

Die Referenztypen umfassen Objekte, Klassen, Interfaces, Delegates, Zeichenketten und Arrays. Der Unterschied zu den Werttypen besteht weniger in der Verwendung, als mehr in der Art der Speicherung. Vereinfacht betrachtet, werden Variablen im Speicher an definierter Stelle abgelegt und nehmen dort einen definierten Platz in Anspruch. Bei

Variablen, die sehr viele Daten enthalten, wäre sehr uneffektiv, diese in einem linearen Speicher abzulegen. Da ständig Verschiebungen ausgeführt werden, wenn sich der Inhalt vergrößert oder Lücken entstehen, wenn er sich verkleinert, würde das System langsam werden. Deshalb wurden die Referenztypen eingeführt, die im Speicher nur als Verweis konstanter Länge existieren und auf einen speziellen Speicherbereich, den Heap (Haufen) zeigen.

Boxing
Unboxing

Boxing und Unboxing sind Begriffe, die auf die rein objektorientierte Struktur von C# zurückgeführt werden können. Die Basis aller Objekte in C# ist der Typ `object`. Dieser ist seinerseits natürlich ein Referenztyp, denn die Größe ist unbestimmt. Wenn Sie nun eine Variable mit einem Werttyp in ein Objekt überführen, nennt man dies Boxing – der Wert kommt in die „Box". Damit wandert der Wert aus dem linearen Speicher auf den Heap. Den umgekehrten Weg gibt es – natürlich nur für die auf diesem Wege in die „Box" gelangten Werte – ebenso: Unboxing. Das Boxing wird ausgeführt, indem als Datentyp `object` verwendet wird.

```
int i = 66;
object boxobject;
boxobject = i;
```

Das sieht etwas eigenartig aus, denn offensichtlich fehlt eine Typkonvertierung. Was hier passiert, ist ein reguläres Verhalten – eben das erwähnte Boxing. Vergleichbar funktioniert auch das Unboxing, also die Umwandlung eines Referenztyps in einen Werttyp:

```
int k;
k = boxobject;
```

Beachten Sie, dass sich das Objekt auch den Datentyp merkt, den der Werttyp trug. Das Unboxing gelingt nur, wenn das neue Ziel denselben Datentyp trägt oder zusätzliche Konvertierungen mit den „cast"-Operatoren oder `Convert` ausgeführt werden.

Feinheiten bei der Schreibweise und Umwandlung

Schreibweisen

Für die verschiedenen Datentypen gibt es Kurzschreibweisen, die eine explizite Umwandlung sparen, wenn die Darstellung nicht eindeutig ist. So wird die Zahl 32.000 zwar oft vom Typ `int` sein, kann aber ebenso aus dem Wertebereich `float` stammen. Sie können deshalb bei der Zuweisung einen Suffix anhängen, der die Darstellung präzisiert:

```
<%
  long lngZahl;
  int intZahl;
  double dblZahl;
```

```
  lngZahl = 32000l;
  intZahl = 99;
  dblZahl = -64.55d;
  Response.Write ("Zahl Long: " + lngZahl.ToString() + "<br>");
  Response.Write ("Zahl Int: " + intZahl.ToString() + "<br>");
  Response.Write ("Zahl Double: " + dblZahl.ToString() + "<br>");
%>
```

Listing 2.2: Datentypen durch Suffixe präzisieren (dt_suffix.aspx)

Zahltypen

Würden Sie im Beispiel `int Zahl = 99L` schreiben, würde ein Compilerfehler angezeigt werden. Die implizite Umwandlung von `long` nach `int` ist nicht möglich.

C# nimmt, wenn es möglich ist, eine implizite Umwandlung vor. Die Entscheidung darüber hängt davon ab, ob sich Daten verlustfrei umwandeln lassen oder nicht. Dies ist keine Frage der konkreten Daten – die zum Zeitpunkt der Übersetzung möglicherweise nicht vorliegen – sondern der Definition der Datentypen. So wird `int` implizit nach `long` konvertiert. Der Zahlenraum `int` ist in `long` abbildbar. Umgekehrt funktioniert das nicht, denn nicht jede Zahl aus `long` kann in `int` ausgedrückt werden. Sie können aber explizit konvertieren. Durch diese bewusste Entscheidung teilen Sie mit, dass Sie gegebenenfalls Informationsverluste oder Rechenfehler in Kauf nehmen.

Alle Suffixe finden Sie in der folgenden Tabelle. Groß- und Kleinschreibung spielt hier ausnahmsweise keine Rolle.

Suffix	Datentyp
l	`long`
u	`uint`
f	`float`
d	`double`
m	`decimal`
ul	`ulong`

Tabelle 2.2: Suffixe zur Präzisierung des Datentyps

Zeichen, Zeichenketten

Zeichenketten werden immer in doppelte Anführungszeichen geschrieben. Damit ist als Datentyp `string` festgelegt. Wollen Sie ein einzelnes Zeichen mit dem Datentyp `char` darstellen, verwenden Sie einfache Anführungszeichen. Beachten Sie auch hier wieder, dass die implizite Umwandlung von `string` nach `char` nicht funktionieren kann. C# speichert Zeichen beider Zeichentypen immer als Unicode. Jedes Zeichen benötigt 16 Bit und kann Symbole aller Sprachen enthalten, auch asiatischer. Der Zeichenraum Unicode umfasst maximal 65.535 Zeichen.

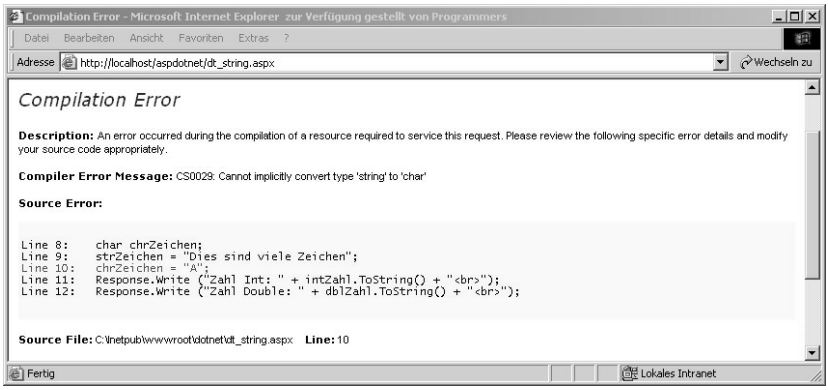


Abbildung 2.2: Die implizite Umwandlung schlägt fehl, wenn ein Datenverlust auftreten würde (Der Fehler trat hier in Zeile 10 auf)

Das folgende Beispiel zeigt den korrekten Umgang mit Zeichendatentypen:

```
<%
  string strZeichen;
  char chrZeichen;
  strZeichen = "Dies sind viele Zeichen";
  chrZeichen = 'A';
  Response.Write ("String: " + strZeichen + "<br>");
  Response.Write ("Char: " + chrZeichen + "<br>");
%>
```

Listing 2.3: Zeichenketten und Zeichen erzeugen (dt_string.aspx)

Beachten Sie, dass die Umwandlung der Variableninhalte mit `ToString()`, wie bei dem vorherigen Beispiel, nicht notwendig ist. Die Verkettung von Zeichenketten erfolgt mit dem Operator „+“. Mehr zu Operatoren und deren Anwendung folgt im nächsten Abschnitt.

Wenn Sie mit impliziter Umwandlung arbeiten möchten, hilft die folgende Tabelle. Sie zeigt, wann genau C# allein umwandeln kann. `bool` kann weder explizit noch implizit konvertiert werden. Dies ist aber meist nicht notwendig, weil C# hier einige Vereinfachungen in der Verwendung zulässt. Die implizite Konvertierung gelingt immer dann, wenn auch theoretisch – also unabhängig vom aktuellen Zustand der Daten – kein Datenverlust auftreten kann.

is

Beim Umgang mit Datentypen können auch zwei Operatoren interessant sein: `is` und `as`. Wollen Sie feststellen, ob eine Variable von einem bestimmten Typ ist, verwenden Sie `is`:

```
if (variable is string) { Response.Write ("Zeichenkette")
};
```

Ausgangstyp	Zieltyp
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long, ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

Tabelle 2.3: Implizite Umwandlung

as Die Typumwandlung kann temporär – also nur an dieser Stelle der Verwendung – mit `as` erfolgen. Der ursprüngliche Typ der Variablen bleibt erhalten. Es muss sich weiterhin um kompatible, also implizit wandelbare Typen handeln. Misslingt die Konvertierung trotzdem, wird `null` erzeugt.

```
object oi1 = "2345";
object oi2 = 2345;
Response.Write (" Ausgabe: " + (oi1 as string));
Response.Write (" Ausgabe: " + (oi2 as string));
```

Die erste Ausgabe gibt „2345“ aus, die zweite `null` – ein ganzzahliger Wert kann nicht implizit in eine Zeichenkette konvertiert werden; Sie müssten `ToString` verwenden. Das kein Compilerfehler erscheint, liegt am Datentyp `object`, der immer konvertibel ist.

Escape-Sequenzen Da als Zeichen nicht nur Buchstaben und Zahlen zulässig sind, werden auch viele Sonderzeichen aus dem ASCII-Raum erfasst. Diese stehen auch bei Unicode zur Verfügung. Da man diese nicht mit dem Editor eingeben kann, gibt es eine spezielle Notation – die Escape-Sequenz. Diese wird immer mit einem Backslash „\“ eingeleitet. Die folgende Tabelle zeigt die wichtigsten Sequenzen und deren Bedeutung. Die jeweils zwei Zeichen sind intern natürlich nur ein Zeichen mit 16 Bit:

Zeichen	Escape-Sequenz
\0	NULL
\n	Neue Zeile (Enter, New Line)
\r	Wagenrücklauf (Zeilenanfang, Carriage Return)
\f	Seitenwechsel (Form Feed)
\t	Tabulator (Tab)

Tabelle 2.4: Wichtige Escape-Sequenzen

Zeichen	Escape-Sequenz
\'	Einfaches Anführungszeichen
\"	Doppeltes Anführungszeichen
\\	Backslash
\u	Hexadezimalcode eines Unicode-Zeichens

Tabelle 2.4: Wichtige Escape-Sequenzen (Forts.)

Damit können Sie auch die Anführungszeichen in eine Zeichenkette packen, die als Begrenzung dienen:

```
string strZeichen = "Sonderzeichen wie \"Backslash\" verwenden Escape-Sequenzen";
```

Besonders häufig wird Ihnen das bei der Angabe von Pfaden unter Windows begegnen, denn dort wird der Backslash benötigt:

```
string strPath = "C:\\Dokumenten und Einstellungen\\Administrator";
```

@-Operator

Ebenso oft, wie die Erzeugung bestimmter Zeichen notwendig ist, kann der Effekt aber auch störend sein. Sie können deshalb die Auswertung der Escape-Sequenzen unterdrücken. Dazu wird der Zeichenkette – nicht der Variablen – das Zeichen `@` vorangestellt:

```
string strPath = @"C:\Dokumenten und Einstellungen\Administrator";
```

Alle anderen Zeichen, die nicht über die Tastatur erreichbar sind, werden über das Unicode-Zeichen erfasst, beispielsweise steht `\u0042` für den Buchstaben „B“. Denken Sie daran, dass Unicode mindestens 16 Bit (2 Bytes) benötigt, in hexadezimaler Schreibweise also vier Zeichen erwartet werden. Es gibt auch einen 4 Byte (32 Bit) breiten Unicode, der aber für nur selten verwendet wird. Falls Sie sich aber vertippen und weitere Ziffern folgen, wird C# dies als erweiterten Unicode interpretieren und vermutlich das falsche Zeichen verwenden.

Logische Werte

Logische Werte werden auch als Boolesch bezeichnet. Diese haben zwei mögliche Zustände: Wahr und Falsch; in C# durch die Schlüsselwörter `true` und `false` gekennzeichnet.

Woher kommt der Ausdruck Boolesch? George Boole (1815 – 1864) wurde 1815 im englischen Lincolnshire geboren. In einigen Biografien wird er als Ire bezeichnet, was genau genommen nicht korrekt ist. Allerdings verbrachte er den größten Teil seines Lebens in Irland. Er war ein englischer Mathematiker und Logiker, der einige grundlegende Regeln der Logik formulierte.

Explizite Datentyp-umwandlung

Es gibt eine weitere Möglichkeit, Datentypen explizit umzuwandeln. Dazu wird der Typ vor die umzuwandelnde Variable in runde Klammern gesetzt. Dies ist eine Art Operator – der so genannte „cast“-Operator.

Dies funktioniert halbwegs zuverlässig nur mit den Zahlentypen. Für die Umwandlung in eine Zeichenkette sollte die Methode `ToString()` verwendet werden, die immer zur Verfügung steht.

2.2.4 Kommentare

Auch in C#-Programmen gibt es die Möglichkeit, Kommentare einzubetten. Diese werden bei der Verarbeitung ignoriert. Es gibt jedoch eine spezielle Notation, Kommentare so zu gestalten, dass enthaltene Informationen als Teil der automatischen Dokumentationen betrachtet werden. Für alle anderen Fälle gibt zwei Arten von Kommentaren: Einzeilige und Mehrzeilige.

Einzeilige Kommentare

Einzeilige Kommentare beginnen am Zeilenanfang und enden, wenn die Zeile mit ↵ abgeschlossen wurde. Dies ist eine der wenigen Stellen, in denen Zeilenumbrüche in C# von Bedeutung sind:

```
string strText = "Text"; // Hier wird Text zugewiesen
```

Mehrzeilige Kommentare

Wenn sich ein Kommentar über mehrere Zeilen erstrecken soll, muss er sowohl einen definierten Anfang als auch ein definiertes Ende haben:

```
/* Dieses Programm ist in C# geschrieben
Es wurde am 8.3.2002 entwickelt */
```

Der Kommentar steht also zwischen den Zeichen `/*` und `*/`.

2.2.5 Konstanten

Konstanten sind wie Variablen zu verwenden. Sie können in Berechnungen ebenso eingesetzt werden, dürfen sich aber während der Abarbeitung eines Programms nicht ändern. Eingesetzt werden Konstanten häufig, um bestimmte Werte zu setzen, die zur globalen Einstellung dienen. Wenn Sie beispielsweise an mehreren Stellen die Farbe eines Textes angeben und sich die Änderung vorbehalten, wäre der feste Einbau der Zeichenkette „red“ nicht ratsam. Sie müssten bei Änderungen dann alle Vorkommen von „red“ suchen und ersetzen. Das funktioniert unter Umständen nicht automatisch, wenn auch Variablen mit dem Namen „reden“ vorkommen. Definieren Sie dann eine Konstante mit dem Namen „farbe“, der Sie den Wert „red“ zuweisen.

Namen der Konstanten

Prinzipiell unterliegen Konstanten den gleichen Benennungsregeln wie Variablen. Der Name kann beliebig lang sein und besteht aus Buchstaben und Zahlen und dem Unterstrich. Er darf jedoch nicht mit einer Zahl beginnen. Der Umgang mit Datentypen entspricht dem bei Variab-

len beschriebenen Prinzipien. Konstanten werden mit dem Schlüsselwort `const` deklariert:

```
<%
  const string AUTOR = "Joerg Krause";
  const string TITEL = "ASP.NET Lernen";
  Response.Write ("Autor: " + AUTOR + "<br>");
  Response.Write ("Buch: " + TITEL + "<br>");
%>
```

Listing 2.4: Verwendung von Konstanten (const_declare.aspx)

Die Zuweisung des Wertes im Augenblick der Deklaration ist obligatorisch. Die Notation des Wertes muss dem angegebenen Datentyp entsprechen.

Die Schreibweise ist wahlfrei; Großbuchstaben haben sich bewährt, um Konstanten von anderen Variablen klar unterscheiden zu können.

2.2.6 Operatoren

Selbstverständlich verfügt C# über die üblichen Operatoren. Zwei haben Sie bereits kennengelernt: Die Zuweisung mit „=“ und die Verkettung von Zeichenketten mit „+“.

Arithmetische Operatoren

C# kennt die elementaren arithmetischen Operatoren:

```
x + y; // Addition
x - y; // Subtraktion
x * y; // Multiplikation
x / y; // Division
x % y; // Modulus (Rest der Ganzzahldivision)
```

Bei der Division wird immer dann eine Gleitkommadivision durchgeführt, wenn einer der beiden Werte vom Typ `double` oder `float` ist. Für eine Ganzzahldivision müssen beide Operanden `int` (oder einer der Ganzzahltypen) sein. Um Werte um eins erhöhen oder verringern zu können, verwenden Sie die Inkrement- und Dekrementoperatoren:

```
zahl++;
zahl--;
```

Im Zusammenhang mit Zuweisungen ist interessant, ob Sie die Erhöhung (Verringerung) vor oder nach der Zuweisung vornehmen. Entsprechend schreiben Sie den Operator vor oder hinter die Variable:

```
x = y++  // x wird y zugewiesen, dann wird y erhöht
x = ++y  // y wird erhöht und dann x zugewiesen
x = y--  // x wird y zugewiesen, dann wird y verringert
x = --y  // y wird verringert und dann zugewiesen
```

In der Praxis können Sie die Operatoren für interessante Effekte einsetzen. So eignet sich der Modulus-Operator, um bei HTML-Tabellen die Zeilen abwechselnd farblich hinterlegt darzustellen. Das folgende Beispiel greift bereits einer Steueranweisung vor, der `for`-Schleife. Probieren Sie es dennoch aus, die Erklärung dazu folgt weiter hinten in diesem Kapitel.

```
<%
string rd = @"<td width=100>A</td><td width=100>B</td>
              <td width=100>C</td>";
Response.Write ("<table border=0 cellpadding=4>");
Response.Write ("<th>Spalte A</th><th>Spalte B</th><th>Spalte C</th>");
for(int i = 0; i <= 10; i++)
{
   if (i % 2 == 0)
   {
      Response.Write ("<tr bgcolor=\"Gray\">"+rd+"</tr>");
   } else {
      Response.Write ("<tr bgcolor=\"White\">"+rd+"</tr>");
   }
}
Response.Write ("</table>");
%>
```

Listing 2.5: Verwendung des Modulus-Operators (operators.aspx)

Das Erkennen jeder zweiten Zeile erfolgt durch die Berechnung `i % 2 == 0`. Wenn das Ergebnis der Berechnung 0 ist (kein Divisionsrest), wird die Zeile grau dargestellt.

@-Operator

Beachtenswert ist auch das `@`-Zeichen in der ersten Zeichenkette. Der Code erstreckt sich im Beispiel über zwei Zeilen, der Zeilenumbruch aus dem Quelltext soll aber nicht mit aufgenommen werden. Er kann mit dem `@`-Operator unterdrückt werden, der bereits zum Zeitpunkt der Übersetzung wirksam wird.

Damit sind Sie eigentlich schon mitten in der praktischen ASP.NET-Programmierung. In „Steueranweisungen“ auf Seite 63 wird gezeigt, wie `if` und `for` und viele andere Anweisungen in C# funktionieren.

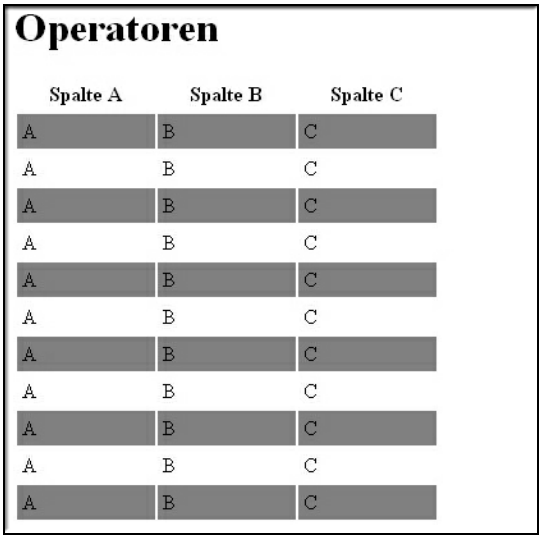


Abbildung 2.3: Ausgabe einer HTML-Tabelle mit reihenweise wechselnden Farben

Zuweisungsoperatoren

Der einfachste Operator ist der Zuweisungsoperator, der beispielsweise für die Übertragung von Werten in eine Variable Verwendung findet. Sie können die grundlegenden arithmetischen Operatoren mit diesem Operator verbinden:

```
zahl += 45;          // addiert 45 zu der Variablen zahl
zahl *= andere_zahl; // multipliziert zahl mit andere_zahl
```

Weitere Zuweisungsoperatoren sind: `+=`, `-=`, `/=`, `%=` und `*=`. Das sieht sehr einfach aus. Sie können aber mit Hilfe von Klammern komplexere Konstruktionen schaffen:

```
zahl += (faktor = 2) * 4;
```

Hier erfolgt erst eine Zuweisung (`faktor = 2`); das Ergebnis (2) wird dann mit 4 multipliziert und zu der Variablen `zahl` addiert.

Auch Ketten von Zuweisungen sind möglich:

```
int z1, z2, z3, z4;
z1 = z2 = z3 = z4 = z5 = 0;
```

Alle Variablen `zX` enthalten nun den Wert 0.

Bitoperatoren

Wenn Variablen Werte enthalten, die sich in Byte- oder Bitform darstellen lassen, können Manipulationen mit Bitoperatoren sinnvoll sein. Als Datentyp kommt beispielsweise `byte` in Betracht. Der Operator `&` führt eine binäre UND-Verknüpfung durch, `|` steht für eine ODER-Verknüpfung, während `~` den Bitwert negiert. Das exklusive Oder (`^`) ist immer dann 1, wenn einer der beiden Operatoren 1 ist (also exklusiv), nicht

aber beide. Die Operatoren entsprechen der Booleschen Algebra. Das Verhalten kann der Auflistung in der folgenden Tabelle entnommen werden.

x	y	x&y	x|y	x^y	~x	~y
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	1	0	1	0

Tabelle 2.5: Verhalten der Bitoperatoren in C#

Das folgende Beispiel zeigt das Prinzip. Denken Sie dabei an die interne Darstellung der Zahl 3 (00000011) bzw. 5 (00000101). Die negativen Zahlen kommen zustande, weil C# bei der Konvertierung das gesetzte achte Bit als Vorzeichen erkennt, -4 entspricht also 255 - 4 = 251.

```
<%
byte bNull = 3;
byte bEins = 5;
Response.Write ((bEins & bNull).ToString());
Response.Write ("<br>");
Response.Write ((bEins | bNull).ToString());
Response.Write ("<br>");
Response.Write ((~bEins).ToString());
Response.Write ("<br>");
Response.Write ((~bNull).ToString());
%>
```

Listing 2.6: Testprogramm für die Bitoperatoren (operators_bit.aspx)

Möglicherweise erscheint Ihnen die Verwendung von `ToString()` hier etwas eigenartig. Immerhin wird die Methode an die Klammer gesetzt. Aber in C# sind alle Elemente immer Objekte, also auch die Zwischenergebnisse einer Berechnung. Das Objekt `bNull`, verknüpft mit dem Objekt `bEins`, ergibt ein Objekt `(bEins & bNull)`, das natürlich in eine Zeichenkette verwandelt werden kann.

Abbildung 2.4: Wirkungsweise der Bitoperatoren

Zwei weitere Operatoren arbeiten auf Bit-Ebene: >> und <<. Beide sind mit dem Zuweisungsoperator = verknüpfbar. Mit >> werden Bits nach rechts verschoben, bei << nach links. Bedenken Sie, dass Sie Bitwerte verarbeiten. Es ist also nicht möglich, damit Zeichen in Zeichenketten zu verschieben.

Logische Operatoren

Neben den beschriebenen gibt es noch logische Operatoren, mit denen logische Ausdrücke gebildet werden können, also solche, die `true` oder `false` ergeben. Diese werden in „Verzweigungen“ auf Seite 66 erklärt.

2.3 Steueranweisungen

Jede Programmiersprache kennt Anweisungen für Verzweigungen oder Schleifen zum Steuern des Programmflusses. Die C-ähnlichen Sprachen gleichen sich auch in dieser Hinsicht. C# ist da keine Ausnahme und so ist die Zahl und Ausprägung der Steueranweisungen wenig überraschend.

2.3.1 Strukturiertes Programmieren

Zur Darstellung eignen sich bestimmte Diagramme, die in Programmablaufplänen verwendet werden. Das ist jedoch schon sehr theoretisch und soll hier nicht tiefgehend behandelt werden. Nur soviel: C# erzwingt eine strukturierte Programmierung. Anweisungen zum „wilden“ Springen in andere Strukturen, wie „goto“ in alten Basic-Versionen, fehlen völlig. Programme bestehen aus nur drei Strukturelementen:

- Folgen (Sequenzen)
- Verzweigungen (Selektionen)
- Schleifen (Iterationen)

In der Informatik finden Sie die in Klammern gesetzten Fachbegriffe.

Folgen

Folgen bilden Sie einfach durch Aufschreiben von Anweisungen. C# führt diese dann in dieser Reihenfolge aus. In manchen Fällen bezieht sich eine Anweisung auf eine Gruppe anderer. Dann werden diese in einem Block zusammengefasst. Blöcke stehen in geschweiften Klammern.

Die folgenden Abbildungen zeigen so genannte Struktogramme. Diese Darstellungen sind leicht nachvollziehbar, wenn man sie von oben nach unten liest. Alternative Zweige laufen parallel. Hat ein Element unten eine Verbindung nach oben, wird dort fortgesetzt, wenn die entsprechende Bedingung erfüllt ist.

Verzweigungen

Für Verzweigungen werden folgende Anweisungen eingesetzt:

- `if`
- `switch`

`if` kennt eine Einfachauswahl und eine Zweifachauswahl. Beides wird in den folgenden Struktogrammen gezeigt.

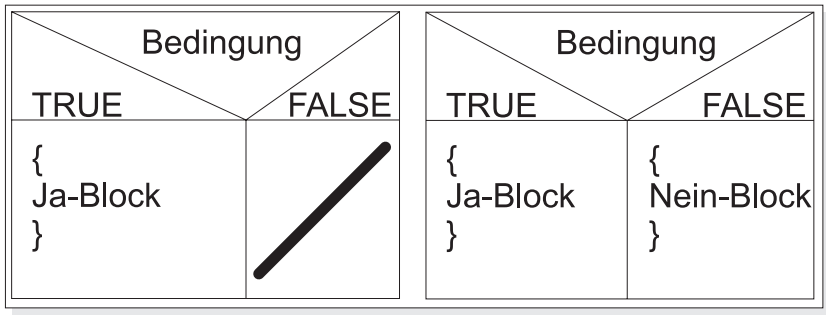


Abbildung 2.5: Struktogramme der einfachen `if`-Anweisung

`switch` erlaubt dagegen ganze Serien solcher Verzweigungen. Auch hierfür gibt es eine entsprechende Darstellung. `switch` kennt zwei Varianten: mit und ohne zusätzlichem Standardzweig.

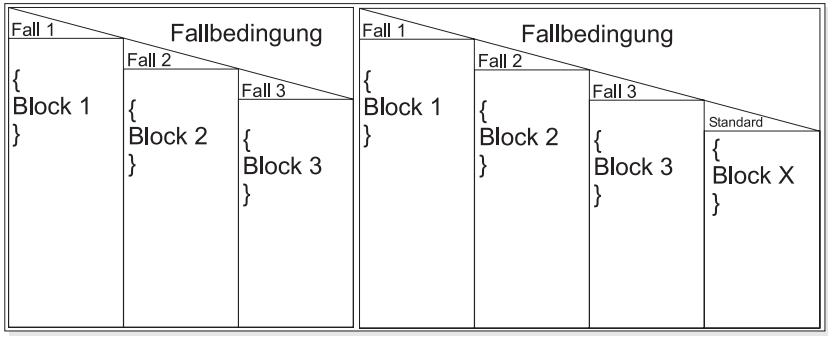


Abbildung 2.6: Struktogramme der `switch`-Anweisung

Schleifen

Schleifen können mit folgenden Anweisungen gebildet werden:

- `for`
- `foreach`
- `while`
- `do`

Normalerweise gelten die Anweisungen immer für den folgenden Befehl. Um Befehlsfolgen zu verarbeiten – was der Regelfall ist – werden Blöcke gebildet. Diese sollen deshalb gleich am Anfang des Kapitels vorgestellt werden.

Zuerst das Struktogramm der `while`-Schleife:

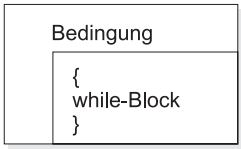


Abbildung 2.7: Die `while`-Schleife als Struktogramm

Die `do`-Schleife arbeitet ähnlich, nur der Prüfpunkt der Bedingung ist nun am Ende statt am Anfang:

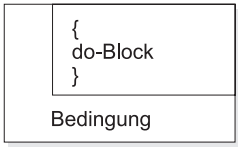


Abbildung 2.8: Das Struktogramm für `do`

Die `for`-Schleife ist komplizierter und universeller. Sie wird zum Zählen eingesetzt:

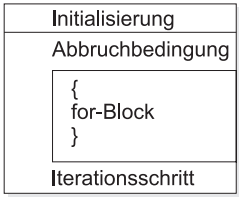


Abbildung 2.9: `for` als Struktogramm

Die Struktogramme ergeben, als Bestandteile eines Programmflussplans aufgezeichnet, eine eindeutige Anleitung der tatsächlichen Flussrichtung des Programms. Was sich damit nicht zeichnerisch darstellen lässt, funktioniert auch nicht.

2.3.2 Blöcke

C# verwendet Blöcke, um mehrere Befehle zusammenzufassen und damit als Einheit zu betrachten. Blöcke werden durch Paare geschweifter Klammern gebildet:

```
Steueranweisung
{
   // Blockanweisung
   // Blockanweisung
}
```

Blöcke und HTML

C# wird in diesem Buch immer im Zusammenhang mit ASP.NET eingesetzt. Es geht also fast ausnahmslos um die Ausgabe von HTML. Vor allem bei der Vermischung von C# und HTML müssen Sie Blöcke bilden, denn C# beendet einen nicht explizit markierten Block am Ende des HTML-Fragments. Der folgende Code funktioniert nicht:

```
<% if(test == true) %><P>HTML ist toll!</P>
```

Sie müssen hier kennzeichnen, dass der HTML-Code zur `if`-Anweisung gehört:

```
<% if(test == true) { %>
<b>HTML ist spannend, wenn es von C# erzeugt wird</b>
<% } %>
```

2.3.3 Verzweigungen

Verzweigungen sind ein elementarer Bestandteil auch kleiner Programme. Dabei wird der ein oder andere Programmteil in Abhängigkeit von einer Bedingung ausgeführt.

Bedingungen

Bedingungen sind das bestimmende Element zur Steuerung von Verzweigungen. Es gibt praktisch kaum ein Programm, das völlig ohne die Steuerung mit Bedingungen auskommt. Das Programm fällt mit Hilfe der Bedingung eine einfache Entscheidung: Ja oder Nein. Entsprechend müssen Ausdrücke, die in den Bedingungsanweisungen eingesetzt werden, logische Ausdrücke sein. Das Ergebnis muss für C# als Wahr (`true`) oder Falsch (`false`) interpretierbar sein.

Ausdrücke

Um Ausdrücke zu konstruieren, werden Operatoren benötigt. Man kann dabei logische Operatoren und Vergleichsoperatoren unterscheiden. Den eigentlichen Programmablauf steuern dann Anweisungen wie `if`, die nachfolgend vorgestellt werden. Auch die Anweisungen zur Schleifensteuerung nutzen logische Ausdrücke.

Logische Operatoren

In vielen Abfragen wird eine logische Entscheidung verlangt. Mit speziellen Operatoren können Sie Ausdrücke verknüpfen. Das Ergebnis eines korrekten logischen Ausdrucks ist immer `false` oder `true`. Die folgenden Ausdrücke zeigen die Anwendung:

```
x && y  // UND
x || $y // ODER
!x      // negiert den Wert false=true, true=false
```

Um komplexe logische Ausdrücke erstellen zu können, benötigt man auch Vergleichsoperatoren. Die folgende Tabelle gibt einen Überblick:

Vergleichsoperatoren

Operator	Bedeutung
==	Gleichheit
!=	Ungleichheit
<	Kleiner als
>=	Größer als oder gleich
<=	Kleiner als oder gleich
>	Größer als

Tabelle 2.6: Die Vergleichsoperatoren auf einen Blick

Vergessen Sie nie, dass der Gleichheitsoperator == nicht dem Zuweisungsoperator = entspricht – ein wesentlicher Unterschied zu einigen anderen Sprachen, mit dem vor allem Umsteiger zu kämpfen haben.

Der folgende Ausdruck ist syntaktisch völlig korrekt, aber trotzdem falsch und vor allem sinnlos:

```
if (var = 23)  { Response.Write var; }
```

Gemeint war sicher, dass die Variable ausgegeben wird, wenn der Inhalt gleich 23 ist. Praktisch passiert aber Folgendes: Die Variable wird durch den Zuweisungsausdruck gleich 23 gesetzt – unabhängig von ihrem vorherigen Inhalt. Dann wird die 23 ausgegeben. Die Zuweisung selbst, als Ausdruck betrachtet, wird korrekt abgearbeitet und von `if` als `true` erkannt. Wenn Sie so einen bestimmten Zustand im Programm erkennen möchten, wird Ihnen dies vielleicht nicht einmal auffallen.

Trickreiche Lösung

Es gibt eine einfache Lösung zum Verhindern solcher Fallen. Vertauschen Sie einfach die Operanden:

```
if (23 = var)
```

Dieser Ausdruck ist syntaktisch definitiv falsch. Zuweisungen können nicht „rückwärts“ arbeiten. Der Fehler wird vom Compiler sofort erkannt. Schreiben Sie dagegen den Gleichheitsoperator korrekt, ergibt sich ein gültiger Ausdruck:

```
if (23 == var)
```

Dies sieht zwar ungewöhnlich aus, ist aber sehr programmierfreundlich.

Bedingungen mit if auswerten

`if` (dt. falls oder wenn) testet eine Bedingung und führt den dahinter stehenden Block aus, wenn die Bedingung `true` ist, ansonsten wird der Block übergangen:

```
if (Bedingung) Anweisung;
```

Wenn mehrere Anweisungen von der Bedingung abhängig sind, muss ein Block gebildet werden:

```
if (Bedingung)
{
   Anweisung-1;
   Anweisung-2;
}
```

Die `if`-Anweisung besteht aus dem Schlüsselwort und dem in runden Klammern stehenden logischen Ausdruck. Wenn der Ausdruck `true` ist, wird die nachfolgende Anweisung oder der Block (in geschweiften Klammern) ausgeführt. Ist der Ausdruck `false`, wird die Programmausführung mit der nächsten Anweisung oder dem nachfolgenden Block fortgesetzt. Das folgende Beispiel erzeugt eine Ausgabe, wenn die Variable `tag` einen bestimmten Wert enthält:

```
<%
string tag = "Thursday";
if (tag == "Monday") Response.Write("Heute ist Montag");
if (tag == "Tuesday") Response.Write("Heute ist Dienstag");
if (tag == "Wednesday") Response.Write("Heute ist Mittwoch");
if (tag == "Thursday") Response.Write("Heute ist Donnerstag");
if (tag == "Friday") Response.Write("Heute ist Freitag");
if (tag == "Saturday") Response.Write("Heute ist Samstag");
if (tag == "Sunday") Response.Write("Heute ist Sonntag");
%>
```

Listing 2.7: Einfache Verwendung von `if`

Die Ausgabe ist wenig überraschend:

Abbildung 2.10: Ausgabe des Programms

Das Beispiel ist sicher primitiv. Praktisch kann immer nur eine der Alternativen erfüllt sein, denn `tag` enthält einen ganz bestimmten Wert. Allerdings besteht auch die – zumindest theoretische Chance – dass

keine der Bedingungen zutrifft. Diesen Zustand abzufangen, ist mit `if`-Anweisungen dieser Art schon schwieriger. Im Abschnitt„Mehrfachbedingungen mit switch“ auf Seite 70 finden Sie eine elegantere Alternative.

Wenn man weiter zu komplexeren Entscheidungsbäumen vordringt, wären Verschachtelungen unvermeidlich. Prinzipiell kann die `if`-Anweisung unbegrenzt verschachtelt werden, das heißt, in jedem Block kann sich wiederum eine `if`-Anweisung verstecken. Das führt in aller Regel nicht zu besonders gut lesbarem Code und sollte vermieden werden. Es gibt fast immer elegantere Lösungsmöglichkeiten. Oft ist es notwendig, nicht nur auf das Eintreten eines Ereignisses zu reagieren, sondern auch die negative Entscheidung zu behandeln. In solchen Fällen wird die `if`-Anweisung um `else` (dt. sonst) ergänzt. Die Anweisung oder der Block hinter `else` wird ausgeführt, wenn die Bedingung nicht zutrifft.

```
<%
short tag = 7;
if (tag == 0 || tag == 7)
{
   Response.Write("Wochenende!");
} else {
   Response.Write("Arbeitszeit");
}
```

Listing 2.8: `if` mit alternativem `else`-Zweig (control_ifelse.aspx)

Das letzte Beispiel könnte man auch mit `if` nach dem `else` schreiben. Praktisch wird dabei in dem `else`-Zweig noch eine weitere `if`-Anweisung eingebaut. Das folgende Beispiel zeigt eine mögliche Anwendung:

```
<%
short tag = 5;
if (tag == 0 || tag == 7)
{
   Response.Write("Wochenende!");
} else if (tag == 5) {
   Response.Write("Fast Wochenende...");
} else {
   Response.Write("Arbeitszeit");
}
%>
```

Listing 2.9: Folgen von `if` und `else` (control_ifelseif.aspx)

Der Bedingungsoperator

Der Bedingungsoperator

C# kennt eine Kurzschreibweise für einen einfachen Bedingungsbefehl, den so genannten trinären Bedingungsoperator:

```
var == "test" ? result = true : result = false;
```

Damit lassen sich Abfragen oft kürzer, aber selten lesbarer gestalten. Die drei Elemente haben folgende Bedeutung:

```
Bedingung ? Wenn true : Wenn false
```

Mehrfachbedingungen mit switch

Wenn Sie wie gezeigt mehrere aufeinanderfolgende Bedingungen gegen ein und dieselbe Variable testen möchten, ist die `if`-Anweisung sehr aufwändig. Mit `switch` steht eine weitere Anweisung zur Verfügung, die solche Listen eleganter aufbaut:

```
<%
short stunde = 9;
switch(stunde)
{
   case 8:
      Response.Write("Guten Morgen");
      break;
   case 9:
      Response.Write("Bisschen sp&auml;t heute?");
      break;
   case 10:
      Response.Write("Jetzt gibts &Auml;rger");
      break;
   case 11:
      Response.Write("Lass dich krankschreiben");
      break;
   default:
      Response.Write("Sonstwann am Tage...");
      break;
}
%>
```

Listing 2.10: Mehrfachbedingung mit `switch` aufbauen (control_switch.aspx)

Der prinzipielle Aufbau ist aus dem Beispiel ersichtlich. In der `switch`-Anweisung selbst steht die zu testende Variable, in den `case`-Abschnitten jeweils der Testwert, der auf Gleichheit getestet wird. `case 8` entspricht also `stunde == 8`.

Wichtig ist die `break`-Anweisung, die die Arbeitsweise von `switch` wesentlich beeinflusst. Wenn `switch` eine zutreffende Bedingung findet, wird nach der `case`-Anweisung mit der Ausführung des Codes begonnen. Weitere `case`-Anweisungen werden nicht ausgewertet. Der `switch`-Block muss mit `break` explizit verlassen werden.

break

Im Listing wurde bereits der Befehlsbestandteil `default` genutzt. Dieser Zweig der `switch`-Anweisung wird ausgeführt, wenn keine andere Bedingung zutrifft. Auch `default` muss mit `break` abgeschlossen werden. Die Position spielt dagegen keine Rolle. Es verbessert lediglich der Lesbarkeit, wenn es am Ende geschrieben wird.

default

2.3.4 Schleifen

Schleifen benötigen Sie, um Programmteile mehrfach durchlaufen zu lassen. Neben der Einsparung an Tipparbeit ist vor allem die variable Festlegung der Schleifendurchläufe interessant. Schleifen ohne feste Laufvariable werden durch eine Bedingung gesteuert. Der Zustand des logischen Ausdrucks bestimmt, ob die Schleife weiter durchlaufen wird oder nicht.

while

Die häufigste Schleifenart ist die `while`-Schleife (dt. während), die in fast jeder Programmiersprache zu finden ist. Die Bedingung wird mit jedem Eintritt in die Schleife getestet. Solange der Ausdruck `true` zurückgibt, wird die Schleife durchlaufen. Wenn der Ausdruck also schon beim Eintritt in die Schleife `false` ergibt, wird die Schleife überhaupt nicht durchlaufen.

```
<%
int counter = 0;
int test = 6;
while (test > counter)
{
   Response.Write("Aktueller Z&auml;hler: " + counter.ToString() + "<br>");
   counter++;
}
%>
```

Listing 2.11: Einfache `while`-Schleife (control_while.aspx)

In diesem Skript werden zuerst zwei Variablen definiert. Eine wird als Laufvariable mit wechselnden Werten eingesetzt, die andere zur Steuerung der Abbruchbedingung. Mit dem ersten Aufruf der `while`-Anwei-

sung wird die Bedingung geprüft. Es kann also vorkommen, dass der Körper der Schleife überhaupt nicht ausgeführt wird.

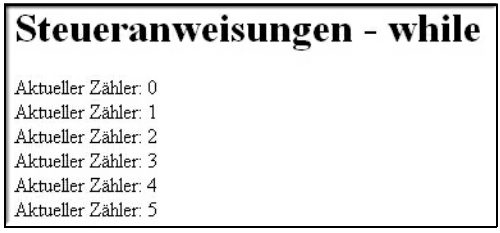


Abbildung 2.11: Ausgabe mit `while`-Schleife

do

Der Test der Bedingung am Anfang hat einen wesentlichen Nachteil, wenn der Inhalt des Blocks für die weitere Programmfortsetzung unbedingt erforderlich ist. Es ist möglich, dass die Bedingung so wirkt, dass der Inhalt nie durchlaufen wird. Um das jedoch sicherzustellen, kann die `do`-Schleife verwendet werden. Der einzige Unterschied zu `while` besteht in der Reihenfolge der Abarbeitung. Zuerst wird die Schleife einmal durchlaufen und am Ende wird die Abbruchbedingung getestet. Auch dann, wenn die Abbruchbedingung beim Schleifeneintritt `false` ist, wird der Block mindestens einmal ausgeführt.

```
<%
int counter = 0;
int test = 6;
do
{
   Response.Write("Aktueller Z&auml;hler: " + counter.ToString() + "<br>");
   counter++;
} while (counter <= test);
%>
```

Listing 2.12: Einfache Anwendung der `do`-Schleife (control_do.aspx)

break und continue

break Die Problematik der Abbruchbedingung kann oft umgangen werden, indem zusätzlich ein Notausstieg eingebaut wird. Das folgende Listing zeigt eine fehlerhaft programmierte Schleife – die Abbruchbedingung wird regulär nie erfüllt. Der Notausstieg verwendet die schon bekannten `break`-Anweisung, die die Ausführung an die nächsthöhere Programmebene zurückgibt; dies ist normalerweise die Anweisung, die auf die schließende Klammer folgt.

```
int counter = 30;
int test = 6;
while (counter > test)
{
   Response.Write("Aktueller Z&auml;hler: " + counter.ToString() + "<br>");
   counter++;
   if (counter == 50) break;
}
Response.Write("Schleifenende erreicht");
%>
```

Listing 2.13: Schleife mit Notausstieg mittels `break` (control_whilebreak.aspx)

Die Schleife entspricht der Vorhergehenden. Wenn jedoch wegen falscher Startwerte die Variable `counter` bis auf 50 erhöht wird, unterbricht `break` die Ausführung der Schleife. Anschließend setzt das Programm nach der schließenden Klammer der `while`-Anweisung fort. Der Begriff Notausstieg sollte hier nicht überbewertet werden. Die Anwendung der `break`-Anweisung ist eine reguläre Programmiertechnik. Als Anfänger sollten Sie sich aber vielleicht die eine oder andere `break`-Anweisung einbauen, um sicher durch alle Schleifen zu kommen. Profis werden besser einschätzen können, ob die gewählten Schleifenbedingungen allen Programmsituationen genügen werden. Der Einsatz von `break` ist generell nur im Zusammenhang mit `if` sinnvoll.

continue

Wenn Schleifen aus dem Block heraus beendet werden können, wäre auch ein forciertes Auslösen des Bedingungstests sinnvoll. Das können Sie mit der Anweisung `continue` erreichen. Diese setzt die Ausführung sofort mit dem Test der Schleifenbedingung fort. Ist die Schleifenbedingung in diesem Augenblick `false`, wird die Schleife mit der Anweisung `continue` verlassen.

```
<%
int counter = 0;
int test = 10;
while (counter < test)
{
   if (counter % 2 == 1)
   {
      counter++;
      continue;
   }
   Response.Write ("Aktueller Z&auml;hler: " + counter.ToString() + "<br>");
   counter++;
}
%>
```

Listing 2.14: Anwendung von continue in einer `while`-Schleife (control_continue.aspx)

Die Anwendung ist in gleicher Form auch für `do` möglich. Ebenso lassen sich die nachfolgend beschriebenen `for`- und `foreach`-Schleifen damit steuern. Allerdings ist der Einsatz in Zählschleifen wegen der komplexeren Steuerung nicht anzuraten – ein Stören des internen Zählers kann zu Programmierfehlern führen, die äußerst schwer nachvollziehbar sind.

Abzählbare Schleifen mit for

Die vorangegangenen Beispiele dienten vor allem der Erläuterung der Syntax der Befehle, die feste Vorgabe von unteren und oberen Grenzen ist keine typische Anwendung der `while`-Schleifen. In solchen Fällen setzen Sie besser `for`-Schleifen ein. Die Abbruchbedingung ist allerdings auch hier ein normaler logischer Ausdruck. Zusätzlich kann eine numerische Variable mitgeführt werden – die Zählvariable.

for

Alle Parameter dieser Anweisung sind optional. Bei vollständiger Angabe ist die `for`-Schleife jedoch komplexer als die bisher behandelten Schleifentypen:

```
for(start, bedingung, iteration);
```

Dies ist die einfachste Form der Anwendung. Das folgende Listing zeigt Schrift in verschiedenen Größen an:

```
<%
for(int i=10; i <= 24; i += 2)
{
   Response.Write ("<div style=\"font-size:" + i.ToString() + "\">");
   Response.Write ("for-Schleife</div>");
}
%>
```

Listing 2.15: Klassische `for`-Schleife (control_for.aspx)

Abbildung 2.12: Ausgabe von formatierten HTML-Code mit for

Die Schleife arbeitet mit der Laufvariablen `i`. Der Startwert ist 10. Die Schleife wird solange durchlaufen, wie `i` kleiner oder maximal gleich 24 ist. Nach jedem Durchlauf wird die Zählvariable `i` um zwei erhöht.

Die Variable in den drei Elementen der `for`-Schleife muss nicht durchgehend verwendet werden. Dies ist zwar im Hinblick auf lesbare Programme zu empfehlen, notwendig ist es jedoch nicht, wie das folgende Beispiel zeigt:

```
<%
for(int i=0, k=10; i <= 10; i++)
{
   Response.Write ("<div style=\"font-size:" + k.ToString() + "\">");
   Response.Write ("for-Schleife</div>");
   k += 2;
}
%>
```

Listing 2.16: `for`-Schleife mit mehreren Variablen im Schleifenkopf (control_for2.aspx)

In diesem Programm wird `i` wieder als Laufvariable eingesetzt. Allerdings findet sie im Block überhaupt keine Verwendung. Das ist auch nicht notwendig. Es geht hier nur darum, sicher eine bestimmte Anzahl von Durchläufen des Anweisungsblocks auszuführen. Zusätzlich wird die zur Anzeige benötigte Variable `k` in der `for`-Anweisung initialisiert. Das wirkt sich aber auf den Verlauf der Schleife nicht aus.

Eigenschaften von for

Alle drei Parameter der `for`-Schleife sind optional. Ohne Bedingungstest wird die Schleife endlos durchlaufen. Sie können in diesem Fall wieder auf `break` zurückgreifen, um die Schleife mit einer zusätzlichen Bedingung zu verlassen. Ebenso kann der Test der Abbruchbedingung und die Ausführung des Iterationsschrittes mit `continue` erzwungen werden.

```
<%
int i = 0;
for(;;)
{
   i++;
   if (i % 2 == 1)
   {
      continue;
      i++;
   }
   Response.Write ("i ist jetzt: " + i.ToString() + "<br>");
   if (i > 5) break;
}
%>
```

Listing 2.17: Steuerung einer endlosen for-Schleife mit `break` und `continue` (control_for3.aspx)

Diese Schleife zählt in Zweierschritten bis maximal sechs (Test `i > 5` ist erfüllt):

Abbildung 2.13: Ausgabe der Zweierschritte (control_for3.aspx)

Der flexible Umgang mit den Schleifenvariablen kennt praktisch keine Grenzen. Es spielt offensichtlich keine Rolle, ob hier Variablen zum Einsatz kommen oder nicht. Wenn Sie nur einfach eine Liste ausgeben möchten, kann der entsprechende Befehl sehr knapp ausfallen. Mehrere Anweisungen im Parameterbereich werden durch Kommata getrennt.

```
<%
for(int i=0, j=10; i<j; i++, j--, Response.Write(i.ToString() + "<br>"));
%>
```

Listing 2.18: Extrem kompakte `for`-Schleife (control_for4.aspx)

Diese Schleife gibt die Zahlen 1 bis 5 aus, da die beiden Schleifenvariablen `i` und `j` gegenläufig zählen.

Aufzählungen und Arrays sequenziell durchlaufen

In C# und im .Net-Framework wird sehr oft mit Arrays, Kollektionen oder Aufzählungen gearbeitet. Beispiele finden Sie in den Abschnitten zu Arrays und Aufzählungen. Die grundlegende Syntax lautet:

```
foreach (typ element_variable in complextyp)
```

Danach folgt meist ein Block. Das Schlüsselwort `in` gehört zu der Anweisung. `foreach` durchläuft jedes Element eines Arrays oder einer Aufzählung und weist dieses jeweils der Variable links im Ausdruck zu. Wenn diese vorher nicht bereits deklariert wurde, kann der Datentyp auch direkt in der Anweisung angegeben werden.

2.4 Objektorientierte Programmierung

C# ist eine objektorientierte Sprache – dies wurde in den vorhergehenden Abschnitten bereits klar. Die Unterstützung in Bezug auf die Erzeugung eigener Klassen und Objekte ist deshalb außerordentlich gut. Die entsprechenden Konzepte sind für jedes Projekt von Bedeutung. Dies ist nicht zuletzt deshalb wichtig, weil der einfache prozedurale Text, der in diesem Kapitel überwiegend gezeigten Programme, intern in Klassen umgesetzt wird. Tatsächlich legt sich ASP.NET wie eine Hülle um Ihren Code und vereinfacht so die Anwendung, ohne dass intern auf irgendeine Funktion verzichtet werden würde.

2.4.1 Klassen definieren

Eine Einführung in die objektorientierte Programmierung wurde bereits in Abschnitt „Die Welt der Objekte“ auf Seite 37 gegeben. In diesem Abschnitt geht es nun um die praktische Realisierung. Der einfachste Fall ist eine Klasse, die überall im Code sichtbar ist und jederzeit verwendet werden kann. Der Sichtbereich von Klassen ist wichtig, denn C# erlaubt sehr differenzierte Einstellungen, sowohl auf Klassenebene, als auch für einzelne Elemente der Klasse.

Das Schlüsselwort zum Anlegen einer Klasse heißt `class`. Der Sichtbereich wird mit `public` eingestellt – diese Angabe wird bei Bedarf vorangesetzt. Die Angabe des Sichtbereiches selbst ist optional, der Standardwert ist `private`. Die Einleitung der Klassendefinition sieht beispielsweise folgendermaßen aus:

```
public class Name
```

Sichtbereiche der Klassenelemente

Die Definition der Klasse folgt danach in geschweiften Klammern.

Schlüsselwort	Bedeutung
`public`	Globale Sichtbarkeit
`protected`	Das Element ist nur innerhalb der aktuellen und abgeleiteter Klassen sichtbar
`internal`	Das Element ist nur in dem Programm sichtbar, in dem es definiert wurde
`private`	Private Elemente sind nur innerhalb der Definition der umgebenden Struktur sichtbar

Tabelle 2.7: Sichtbereiche von Klassen, Methoden und Eigenschaften

abstract	Die Methode enthält lediglich eine Schnittstellenbeschreibung und keine Implementierung von Funktionalität. Klassen, die davon abgeleitet werden, müssen den Code selbst implementieren.
sealed	Mit dieser Bezeichnung wird verhindert, dass die Klasse von anderen geerbt werden kann.
protected internal	Das Element ist nur im Programm und bei Ableitung aus übergeordneten Klassen sichtbar
extern	Die Definition erfolgt außerhalb der Klasse.
override	Zeigt das Überschreiben einer Methode an, die als virtual gekennzeichnet wurde.
virtual	Erlaubt explizit das Überschreiben von Methoden.

Tabelle 2.8: Zugriffsmodifikatoren der Elemente abstract

Sichtbereich

Klassen enthalten Bauanleitungen für Objekte. Die gezeigten Modifikatoren für Klassen und Elemente werden erst verständlich, wenn man die Grundprinzipien der objektorientierten Programmierung versteht.

Als „Element" werden hier allgemein die Bestandteile von Klassen bezeichnet: Variablen, Eigenschaften, Ereignisse und Methoden. Diese werden im Laufe dieses Kapitels genauer vorgestellt.

Bauanleitungen kann man am Stück schreiben, man kann sie aber auch in viele voneinander ableitende Teile zerlegen. So kann die Bauanleitung für einen Schrank auf einem Blatt Papier stehen, aber auch in mehrere kleine zerlegt werden. Die Basisanleitung beschreibt die äußeren Bauteile – also das Zusammensetzen von Seitenteilen, Dach und Boden und Türen. Dies ist die erste Klasse. Davon abgeleitet werden weitere Klassen, die jeweils Details hinzufügen, beispielsweise solche zum Einlegen der Regalbretter oder zum Einbau der Schlösser. Klassen stehen also miteinander in Beziehung. Um diese Beziehungen geht es bei den Modifikatoren. Dabei ist zu beachten, welchen Zweck eine Klasse erfüllen soll. Schreiben Sie ein einfaches Teilprogramm für die eigene Applikation, ist `public` oder `private` sicher die beste Wahl. Stellen Sie anderen Programmierern ihre Klasse zur Verfügung, ist `protected` oft angebracht. Wollen Sie eine einheitliche Struktur der Klassendefinitionen durchsetzen, die Implementierung aber anderen überlassen, käme `abstract` in Betracht. Hilfsfunktionen, die nicht woanders verwendet werden sollen, sind mit `private` oder `sealed` gut bedient.

Eigenschaften

Eigenschaften

Da Objekte Eigenschaften haben und Aktionen benötigen, die damit ausgeführt werden können, muss es auch dafür eine Schreibweise ge-

ben. Zusätzlich sind auch Variablen in allen beschriebenen Formen innerhalb der Klasse möglich. Diese bilden keine Eigenschaften, sondern stehen als lokale Speicherstellen zur Verfügung. Eigenschaften werden durch zwei spezielle Schlüsselwörter gebildet: `get` und `set`. Das folgende Beispiel zeigt dies.

```
<script language="C#" runat="server">
public class xypoint
{
   private int _xpoint;
   private int _ypoint;

   public int x
   {
      get
      {
         return _xpoint;
      }
      set
      {
         _xpoint = value;
      }
   }

   public int y
   {
      get
      {
         return _ypoint;
      }
      set
      {
         _ypoint = value;
      }
   }
}
</script>
<html>
<body>
<h1>Klassen - Klasse definieren</h1>
<%
xypoint mypoint = new xypoint();
mypoint.x = 12;
mypoint.y = 23;

Response.Write ("X=" + mypoint.x.ToString());
Response.Write ("<br/>");
```

```
Response.Write ("Y=" + mypoint.y.ToString());
%>
</body>
</html>
```

Listing 2.19: Definition einer Klasse mit zwei Eigenschaften (class_properties.aspx)

Das Listing wurde hier ausnahmsweise einmal komplett wiedergegeben. Normalerweise gilt die Vereinbarung vom Buchanfang, dass der sich immer wiederholender HTML-Code nicht mit abgedruckt wird. Dennoch ist dies hier wichtig. ASP.NET verpackt den gesamten Inhalt der Seite in eine übergeordnete Klasse. Dies erleichtert die Programmierung, weil das Einbinden der Basisklassen entfällt. Allerdings gibt es manchmal Einschränkungen bei der Implementierung eigener Klassen, die nicht nachvollziehbar sind, wenn man die Zusammenhänge nicht kennt.

Wie es funktioniert

Die Definition der Klasse umfasst zwei private Variablen vom Typ `int`. Diese speichern zwei Zahlenwerte aus einem Koordinationsystem. Nach außen – als Eigenschaft – sollen diese Werte als `x` und `y` bezeichnet werden. Eigenschaften werden folgendermaßen definiert:

```
Modifikator Datentyp Name
```

Im Beispiel definiert die Zeile `public int x` einen ganzzahligen Wert `x` als öffentliche Eigenschaft. Eigenschaften können Werte enthalten – es muss also einen Weg geben, Werte einzugeben und wieder herauszuholen. Das Eintragen der Werte erfolgt mit `set`:

```
 set { _ypoint = value; }
```

Das Schlüsselwort `value` enthält den im Augenblick der Zuweisung übergebenen Wert. Ein weiterer Parameter, wie bei Methodenaufrufen, ist nicht notwendig. Umgekehrt gibt es beim Auslesen der Werte keine Besonderheiten, die Rückgabe erfolgt in einer `get`-Definition mit `return`:

```
get { return _ypoint;  }
```

Konsequent programmieren

Freilich lässt sich das einfacher schreiben, wie das folgende Beispiel zeigt:

```
<script language="C#" runat="server">
public class xypoint
{
   public int x;
   public int y;
}
</script>
```

Listing 2.20: Alternative Deklaration von Eigenschaften

Der übrige Teil des Programms unterscheidet sich nicht vom vorherigen. Das ist zwar deutlich kürzer, aber nicht konsequent programmiert. Zum einen ist die Deklaration nicht sehr streng an den Möglichkeiten der Sprache orientiert. Es ist nicht klar, ob das Schreiben der Variablen gewollt ist oder lediglich einen Missbrauch der Definition darstellt. Zum anderen sind feinere Einstellungen nicht möglich. Bei Eigenschaften können Sie entscheiden, ob diese nur zum Lesen, nur zum Schreiben oder für beide Operationen zugelassen sind. Dabei wird für einen reinen Lesezugriff die Schreiboperation `set` entfernt. Für reine Schreib-Eigenschaften wird die Leseoperation (`get`) entfernt.

Klassen verwenden

Klassen verwenden

Nach der Deklaration der Klasse soll diese natürlich verwendet werden. Im Normalfall heißt das, aus der Bauanleitung wird ein reales Objekt erstellt. Da Klassen zur Gruppe der Typen gehören, können sie wie ein Datentyp verwendet werden. Zusätzlich ist das Schlüsselwort `new` dafür verantwortlich, dass ein neues Objekt entsteht:

```
xypoint mypoint = new xypoint();
```

Dabei ist `xypoint` der Name der Klasse und zugleich der Typ des neuen Objekts. Die verwendete Schreibweise ist der übliche, kompakte Weg. Zum Verständnis ist die Trennung der beiden Vorgänge besser geeignet:

```
xypoint mypoint;
mypoint = new xypoint();
```

Dieser Vorgang wird als Instanziierung bezeichnet – eine neue Instanz der Klasse entsteht. Auf die Eigenschaften und Methoden kann dann in der bereits oft verwendeten Punktschreibweise zugegriffen werden. Je nach dem, ob es sich um eine Zuweisung oder einen Abruf des Wertes handelt, wird `get` oder `set` verwendet. Noch ein Vorteil ergibt sich aus der Verwendung von Eigenschaften. Sie können den Rückgabewert oder den internen Speicherwert beim Schreiben und Lesen konvertieren. So wäre eine Klasse denkbar, die alle Ausgaben in Zeichenketten umwandelt, da das ständige Aufrufen von `ToString()` sehr lästig werden kann. Das folgende Beispiel zeigt, wie dies vermieden wird:

```
public class xypoint
{
   private int _xpoint;
   private int _ypoint;

   public string x
   {
      get
```

```
        {
            return (_xpoint.ToString());
        }
        set
        {
            _xpoint = Convert.ToInt32(value);
        }
    }

    public string y
    {
        get
        {
            return (_ypoint.ToString());
        }
        set
        {
            _ypoint = Convert.ToInt32(value);
        }
    }
}
```

Listing 2.21: Konvertierung des Datentypes bei der Übergabe von Eigenschaften (class_properties2.aspx)

Diese Klasse arbeitet sehr bequem für die ASP.NET-Programmierung: Die Ein- und Ausgabe erfolgt mit Zeichenketten, intern wird aber mit Zahlen gearbeitet. Zu beachten ist, dass die Zuweisung dann auch über `mypoint.x = "12"` erfolgen muss.

Konstruktoren und Destruktoren

Konstruktoren

Wenn aus Klassen Objekte abgeleitet werden, ist es oft sinnvoll, dass die Eigenschaften bereits bestimmte Standardwerte enthalten. Dazu dienen Konstruktoren. Das sind Methoden, die bei der Instanziierung des Objekts automatisch aufgerufen werden. Deklariert werden sie wie jede andere Methode. Sie entstehen, indem als Name der Name der Klasse verwendet wird.

Destruktoren

Destruktoren bearbeiten den umgekehrten Fall. Sie werden aktiviert, wenn das Objekt zerstört wird. Dies ist kritisch zu betrachten, denn bei C# handelt es sich um „managed code“, das Framework ist für die Verwaltung verantwortlich. Das Aufräumen nicht mehr benötigter Variablen im Speicher wird als „garbage collection“ bezeichnet, auf deutsch „Müll sammeln“. Dieser Vorgang ist – zeitlich betrachtet – unbestimmt. Es gibt keine Garantie, dass die Entfernung des Objekts aus dem Speicher genau dann stattfindet, wenn der Aufruf dazu im Code erfolgt.

Wenn Sie also sicher gehen müssen, dass die Entfernung tatsächlich sofort eine Aktion auslöst, können Sie auf den Destruktor nicht vertrauen. Der Destruktor entsteht, indem eine Methode deklariert wird, die den Namen der Klasse mit dem Präfix ~ (Tilde) trägt:

```
private ~xypoint()
{ /* Code des Destruktors */ }
```

Um ein Objekt explizit zu zerstören, wird es auf `null` gesetzt:

```
mypoint = null;
```

Das folgende Beispiel zeigt die Verwendung des Konstruktors. Statt späterer Zuweisung werden gleich bei der Instanziierung des Objekts Werte übernommen und in den Eigenschaften gespeichert:

```
public class xypoint
{
   private int _xpoint;
   private int _ypoint;

   public string x
   {
      get { return (_xpoint.ToString()); }
      set { _xpoint = Convert.ToInt32(value); }
   }

   public string y
   {
      get {  return (_ypoint.ToString()); }
      set {  _ypoint = Convert.ToInt32(value); }
   }

   public xypoint(int  __x, int __y)
   {
      this._xpoint = __x;
      this._ypoint = __y;
   }

}
```

Listing 2.22: Klassendefinition mit Konstruktor (class_construct.aspx)

Damit ist die Sache mit den Konstruktoren aber noch nicht abgeschlossen. Möglicherweise wollen Sie bei Anwendung der Klasse häufig Punkte erzeugen, wo `x` und `y` identisch sind. Es wäre also einfacher, den Aufruf des Konstruktors folgendermaßen zu schreiben:

```
xypoint mypoint = new xypoint(100);
```

Überladung

Das Objekt sollte dann für x den Wert 100 enthalten und für y ebenso. C# kennt dafür die Technik der Überladung. Bei der Vererbung von Klassen ist damit das Überschreiben vorhergehender Definitionen gemeint. Wenn sich aber die Parameter unterscheiden, dann koexistieren zwei oder mehr gleichnamige Methoden in derselben oder in verschiedenen Klassen. C# wählt beim Aufruf diejenige aus, die den passenden Satz an Parametern enthält.

```
<script language="C#" runat="server">
public class xypoint
{
   private int _xpoint;
   private int _ypoint;

   public string x
   {
      get { return (_xpoint.ToString()); }
      set { _xpoint = Convert.ToInt32(value); }
   }

   public string y
   {
      get {  return (_ypoint.ToString()); }
      set {  _ypoint = Convert.ToInt32(value); }
   }

   public xypoint (int __x)
   {
       this._xpoint = this._ypoint = __x;
   }

   public xypoint (int __x, int __y)
   {
       this._xpoint = __x;
       this._ypoint = __y;
   }

}
</script>
<html>
  <head><title>C# lernen</title></head>
  <body>
  <h1>Klassen - Statische Klassen</h1>
<%
xypoint mm1 = new xypoint(55);
```

```
Response.Write ("X = " + mm1.x);
Response.Write ("Y = " + mm1.y);
Response.Write ("<br/>");
xypoint mm2 = new xypoint(100, 200);
Response.Write ("X = " + mm2.x);
Response.Write ("Y = " + mm2.y);
%>
  </body>
</html>
```

Listing 2.23: Klasse mit zwei Konstrukturen (class_overload.aspx)

Beachten Sie in diesem Programm, dass der Konstruktur `xypoint()` zweimal definiert wurde. Das ist zulässig, weil sich die Parameter unterscheiden. Der erste empfängt nur einen Wert und weist ihn den beiden Eigenschaftsvariablen zu. Der zweite hat zwei Parameter, die getrennt zugewiesen werden.

Signatur

Die Erkennung der passenden Methode erfolgt anhand der so genannten Signatur. Dies ist ein Begriff, der die Eindeutigkeit einer Methode beschreibt. Die Angabe mehrerer Konstruktoren – und auch normaler Methoden – ist möglich, wenn sich deren Signatur unterscheidet.

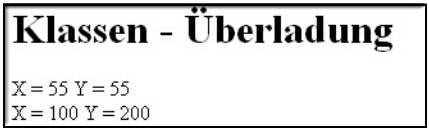


Abbildung 2.14: Arbeit mit überladenen Konstruktoren (Ausgabe von Listing 2.23)

Weitere Techniken der Methodenüberladung

C# kennt keine optionalen Parameter. Die Angabe widerspräche auch der strengen Typprüfung, denn der Compiler kann beim Übersetzen nicht mehr feststellen, ob ein Aufruf korrekt ist oder nicht. Ohnehin sind die in anderen Sprachen zulässigen optionalen Parameter eher unglückliche Ersatzlösungen. Fast immer schränken Sie mehr ein anstatt dem Programmierer mehr Freiheiten zu geben. Viel eleganter ist dies in C# gelöst. Wenn Sie ein- und dieselbe Methode verschiedenartig aufrufen möchten, schreiben Sie sie mehrfach. Intern unterscheidet C# dies durch das so genannte Interface, das ist der Kopf der Methodendeklaration.

Das folgende Beispiel enthält zwei gleichnamige Methoden (`makeHTML`):

```
<script language="C#" runat="Server">
void Page_Load()
```

```
{
    string mustertext = "Dieser Text testet Parameter";
    header.InnerHtml = makeHTML (mustertext, 'h', 3);
    para.InnerHtml = makeHTML (mustertext, 'i');
}

string makeHTML (string htmltext, char tag)
{
    htmltext = "<" + tag + ">" + htmltext + "</" + tag + ">";
    return htmltext;
}

string makeHTML (string htmltext, char tag, int modifier)
{
   htmltext = "<" + tag + modifier.ToString() + ">" + htmltext + "</" + tag +
modifier.ToString()  + ">";
    return htmltext;
}

</script>
<html lang="de">
    <head>
        <title>Methoden&uuml;berladung</title>
    </head>
    <body>
    <h1>Methoden&uuml;berladung</h1>
    <p id="header" runat="server"/>
    <p id="para" runat="server"/>
    </body>
</html>
```

Beide Methoden unterscheiden sich durch die Parameter. Welche im Augenblick des Aufrufes verwendet wird, entscheiden eben diese Parameter. Wenn Sie mit Klassen arbeiten und Klassen vererben, ist dieselbe Art der Überladung möglich. Stimmen die Parameter mit denen der Basisklasse überein, werden diese verwendet, wenn nicht, wird in einer anderen Ebene der Vererbung gesucht. Im Beispiel verhalten sich die beiden Methoden so, als ob der dritte Parameter `int modifier` optional wäre. Ob sich das Verhalten dann allerdings nur in Bezug auf den Parameter oder grundlegend unterscheidet, können Sie als Programmierer frei entscheiden. Empfehlenswert ist es, den natürlichen Zusammenhang gleichnamiger Methoden mit verschiedenen Parameterdeklarationen zu erhalten.

Abbildung 2.15: Verschiedene Ausgaben durch dieselbe, überladene Methode

Zugriff auf die Klasse in der Definition: this

Schlüsselwort this

Gleichzeitig sehen Sie hier die Verwendung eines weiteren Schlüsselwortes: `this`. Innerhalb der Klasse verweist es auf die Klasse selbst, steht also immer anstelle des Objektes, dass zu diesem Zeitpunkt noch nicht existiert. Solange der Kontext eindeutig ist, können Sie auf die Angabe verzichten. Wenn Sie jedoch Klassen vererben und dabei in der übergeordneten und der aktuellen Klasse Eigenschaften oder Methoden gleichen Namens auftreten, ist der direkte und eindeutige Verweis mit `this` notwendig. Nicht möglich ist dagegen der Einsatz in statischen Klassen.

Statische Klassen und Methoden

Manchmal werden Klassen benötigt, die lediglich ein einziges Objekt instanziieren lassen. Das ist eigentlich unnötige Schreibarbeit, denn wozu sollte ein Bauplan angelegt werden, wenn man ihn lediglich ein einzige Mal benötigt. Statische Klassen sind praktisch so etwas wie Prototypen – als fertige Objekte verpackte Klassen. Die Verwendung in C# ist sehr elegant, weil nicht die ganzen Klasse als statisch deklariert werden muss, sondern nur die Elemente, die man auch in dieser Form benötigt. Oft sind auch nur einzelne Methoden einer Klasse statisch. Dann erfolgt der Aufruf direkt, wie bei einer Funktion in der prozeduralen Programmierung.

Erweitern Sie die bereits gezeigte Klasse, um den Abstand vom Nullpunkt des Koordinatensystems zu den beiden Punkten zu berechnen:

```
public static string xyvector(int x, int y)
{
   return Math.Sqrt(x*x + y*y).ToString();
}
```

Listing 2.24: Ausschnitt aus class_static.aspx – Berechnung mit statischer Methode

Der Aufruf verzichtet gänzlich auf die Instanziierung:

```
string mm = xypoint.xyvector(100, 100);
Response.Write ("Ergebnis: " + mm);
```

Wie flexibel C# mit Typen umgehen kann, zeigt das folgende Beispiel. Es ist zwar vom Standpunkt optimierter Programmierung her etwas konstruiert – die Problemstellung lässt sich auch einfacher lösen – zeigt aber, welche Kraft in C# steckt.

```
<script language="C#" runat="server">
public class xypoint
{
   private int _xpoint;
   private int _ypoint;
   public double v;

   public string x
   {
      get { return (_xpoint.ToString()); }
      set { _xpoint = Convert.ToInt32(value); }
   }

   public string y
   {
      get {  return (_ypoint.ToString()); }
      set {  _ypoint = Convert.ToInt32(value); }
   }

   public static xypoint xyvector(int __x, int __y)
   {
      xypoint o = new xypoint();
      o.x = __x.ToString();
      o.y = __y.ToString();
      o.v = Math.Sqrt(__x*__x + __y*__y);
      return o;
   }

}
</script>
<html>
  <head><title>C# lernen</title></head>
  <body>
  <h1>Klassen - Statische Klassen</h1>
<%
xypoint mm = xypoint.xyvector(100, 100);
Response.Write ("Ergebnis Vektor: " + mm.v);
Response.Write ("<br/>");
%>
  </body>
</html>
```

Listing 2.25: Statische Methode mit Rückgabe eines Objekts (class_statico.aspx)

Das Geheimnis steckt in der statischen Methode `xyvector`. Diese ist nicht nur statisch, sie trägt auch den Rückgabetyp `xypoint`. Das ist die Klasse, in der die Methode selbst gerade erst definiert wird. Natürlich muss dann nicht die Klasse, sondern ein Objekt aus dieser zurückgegeben werden. `xypoint o = new xypoint()` legt das Objekt `o` an, das dann zurückgegeben wird. Dies ist trickreich, weil der `new`-Aufruf im Hauptprogramm entfällt. Das Objekt wird hier lediglich durch Zuweisung in den globalen Adressraum gebracht.

Abbildung 2.16: Ausgabe der Berechnung mit Hilfe einer statischen Methode

Ganz nebenbei wurde hier übrigens die Klasse `System.Math` aus dem Framework verwendet. Das funktioniert, weil ASP.NET den Namensraum `System` immer einbindet. `Math` ist dann mit allen Funktionen zu mathematischen Berechnungen verfügbar. Dies aber nur am Rande.

Vererbung und Überladung im Detail

Ein wesentlicher Aspekt objektorientierter Programmierung ist die Vererbung. Wie bei der Darstellung mit den Bauplänen bereits angedeutet, werden Klassen häufig in einer Art Hierarchie eingesetzt. Angefangen von einfachen Basisobjekten bis hin zu komplexeren Gebilden, von denen konkrete Objekte abgeleitet werden, entstehen in Projekten meist ganze Klassenbäume. Das .Net-Framework ist eine solche Hierachie – mit etwa 7.000 Klassen. Klar ist, dass diese in einer flachen Anordnung völlig unbeherrschbar wären.

Die Vererbung steht in C# mit allen Merkmalen der objektorientierten Programmierung zur Verfügung. Je nach Art der Definition werden die ererbten Eigenschaften und Methoden nicht nur übernommen, sondern lassen sich auch gezielt überschreiben. Man spricht dann vom Überladen. Dies kann natürlich auch verhindert werden, das Schlüsselwort dazu wurde bereits am Anfang der Einführung zu Klassen in C# vorgestellt: `sealed`.

In den vorangegangenen Beispielen wurde immer wieder eine Klasse verwendet, die zwei Punkte in einem Objekt zusammenfasst und verschiedene Operationen damit anstellt. Möglicherweise sollen sich spätere Formen der Punkte anders oder erweitert verhalten. Es bietet sich an, eine Basisklasse zu erstellen, die anderen als Grundlage dient:

```
public class xypoint
{
   protected int _xpoint;
   protected int _ypoint;

   public string x
   {
      get { return (_xpoint.ToString()); }
      set { _xpoint = Convert.ToInt32(value); }
   }

   public string y
   {
      get {  return (_ypoint.ToString()); }
      set {  _ypoint = Convert.ToInt32(value); }
   }

}
```

Der einzige Unterschied zu den bisher gezeigten Varianten besteht in der Definition der internen Punkte. Es soll anderen Klassen erlaubt werden, die Ganzzahlwerte direkt zu beschreiben, damit der Konvertierungsschritt entfällt. Außerhalb der Klassen – auf der Anwendungsebene also – ist es sinnvoll, die Verwendung der Eigenschaften mit der Konvertierung in Zeichenketten zu erzwingen. Das Schlüsselwort `protected` ist der geeignete Weg, dieses Verhalten zu implementieren.

Im zweiten Schritt soll nun eine Klasse entworfen werden, die die Methoden enthält. Da noch keine Methoden überschrieben werden, erfolgt nur eine einfache Vererbung:

```
public class points : xypoint
{

    public double xyvector ()
    {
        return (Math.Sqrt(this._xpoint*this._xpoint +
this._ypoint*this._ypoint));
    }

}
```

Die neue Klasse `points` erbt von der Klasse `xypoint`. Dies wird durch den Doppelpunkt angezeigt. Die Anwendung zeigt, dass nur noch die letzte Klasse verwendet wird:

```
<%
points mm1 = new points();
mm1.x = "100";
mm1.y = "45";
double vector = mm1.xyvector();
Response.Write (" X = " + mm1.x);
Response.Write (" Y = " + mm1.y);
Response.Write ("<br/>");
Response.Write (" Vektor = " + vector.ToString());
%>
```

Listing 2.26: Einsatz der Vererbung (class_inherit.aspx)

An dieser Stelle fehlt der Platz, alle Varianten der Vererbung zu diskutieren. In den folgenden Kapiteln werden diese Techniken jedoch ab und an verwendet, sodass Sie anhand der Beispiele den Einsatz lernen können. Die Vererbung ist eine Kerntechnik der objektorientierten Programmierung. Übermäßige Nutzung kann Programme jedoch unnütz kompliziert machen.

Operatorenüberladung

Neben der Überladung von Methoden können auch Operatoren überladen werden. Diese haben Sie zwar nicht selbst definiert, aber verändern können Sie das Verhalten innerhalb eigener Klassen dennoch. Wenn Sie bisher nichts mit der objektorientierten Programmierung zu tun hatten, mag Ihnen das unheimlich erscheinen, aber letztlich sind alles nur Objekte – auch die Operatoren. Das folgende Programm definiert einen neuen +-Operator, der die xy-Punkte addiert, in dem jeweils `x` zu `x` und `y` zu `y` addiert wird. Das kann man mit einer Methode `xyadd` sicher gut erledigen:

```
public xypoint xyadd(int x, int y)
```

Viel intuitiver wäre es jedoch, wenn man zwei Punkte einfach addieren könnte:

```
neuerpunkt = andererpunkt + dritterpunkt;
```

Tatsächlich ist das genau so möglich:

```
public class xypoint
{
   private int _xpoint;
   private int _ypoint;

   public string x
   {
      get { return (_xpoint.ToString()); }
      set { _xpoint = Convert.ToInt32(value); }
   }
```

```
   public string y
   {
      get {  return (_ypoint.ToString()); }
      set {  _ypoint = Convert.ToInt32(value); }
   }

   public static xypoint operator + (xypoint o1, xypoint o2)
   {
       xypoint o3 = new xypoint();
       o3.x = (Convert.ToInt32(o1.x) + Convert.ToInt32(o2.x)).ToString();
       o3.y = (Convert.ToInt32(o1.y) + Convert.ToInt32(o2.y)).ToString();
       return o3;
   }


}
```

Listing 2.27: Klasse mit einer Methode, die den +-Operator überlädt (class_overloadops.aspx)

Die Definition der Methode unterscheidet sich nur wenig von den bisher gezeigten. Klar ist, dass hier zwei Objekte addiert werden – also wird auch ein Objekt zurückgegeben. Die Methode muss außerdem öffentlich (`public`) und statisch sein (`static`). Der Rückgabewert ist hier die Klasse `xypoint` selbst. Dann folgt der entscheidende Teil: das Schlüsselwort `operator` und der Operator (+) selbst. Auch die beiden Objekte, die addiert werden sollen, werden wieder als Typ der Klasse `xypoint` deklariert.

Im Inneren der Definition finden nun die Berechnungen statt. Da die Eigenschaften der Klasse Zeichenketten erwarten, muss hier eine explizite Konvertierung stattfinden. Dies liegt an der möglicherweise unglücklichen Definition der Eigenschaften, die jedoch zugunsten einer einfachen Ausgabe deklariert wurden. Umso einfacher ist dafür die Anwendung, denn darum geht es letztlich:

```
<%
xypoint mm1 = new xypoint();
xypoint mm2 = new xypoint();
xypoint mm3;
mm1.x = "100";
mm1.y = "45";
mm2.x = "17";
mm2.y = "80";
Response.Write (" X = " + mm1.x);
Response.Write (" Y = " + mm1.y);
Response.Write ("<br/>");
```

```
mm3 = mm1 + mm2;
Response.Write (" X = " + mm3.x);
Response.Write (" Y = " + mm3.y);
%>
```

Listing 2.28: Nutzung des überladenen Operators (Fortsetzung class_overloadaps.aspx)

Das Ergebnis entspricht den Erwartungen. Sie können auch andere Operatoren überladen. Beachten Sie, dass C# weis, welche Operatoren binär sind (+, -, *, / usw.) oder unär (-, !). Entsprechend muss die Methode zum Überladen zwei oder einen Parameter haben. Zuweisungen (=) sind nicht überladbar, allerdings werden die kombinierten Zuweisungsoperatoren automatisch überladen. Das gezeigte Beispiel funktioniert deshalb auch mit += wie erwartet.

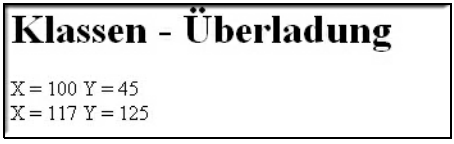


Abbildung 2.17: Überladung von Operatoren

Natürlich ist der Operator nur dann überladen, wenn die Signatur beim Aufruf stimmt. Addieren Sie zwei Zahlen, wird weiterhin die ursprüngliche Definition verwendet. Im Umkehrschluss heißt dies, dass Sie die überladenen Operatoren wieder selbst mit einer anderen Signatur überladen können. Ein C#-Programm kann also viele Arten der Behandlung des Operators + und aller anderen Operatoren enthalten.

Parameter der Methodendeklaration

Bislang wurden Methoden einfach deklariert und mit Modifikatoren versehen. Das Methoden auch Parameter akzeptieren, dürfte auch klar geworden sein. Die Parameter verhalten sich innerhalb der Methode wie lokale Variablen. Es gibt aber neben der Angabe des Datentyps noch weitere Parameterarten, die hier kurz vorgestellt werden.

Wertparameter

Die normale Form, in der Parameter übergeben werden, sind Wertparameter. Dabei wird der Wert (Inhalt) einer Variablen kopiert. Beim Aufruf der Funktion entsteht unter dem Namen des Parameters eine neue Variable, die diesen Wert zugewiesen bekommt. Zwischen der beim Aufruf verwendeten Variablen und der in der Methode lokal existierenden besteht kein Zusammenhang, außer das im Augenblick des Methodenaufrufes beide denselben Inhalt haben. In C# muss für die Nutzung dieses Verhaltens nichts weiter notiert werden.

Referenzparameter

Die Rückgabe von Daten ist auf einen Wert beschränkt, der mit `return` an die Zuweisung beim Aufruf übergeben wird. Sollen nun in einer Methode mehrere Variablen verändert werden, muss auf eine besondere Parameterform zugegriffen werden. Der Trick besteht darin, dass ein Verweis auf die ursprüngliche Variable übertragen wird. Dieser Verweis trägt in der Methode wieder den lokalen Namen, der im Methodenkopf angegeben wurde. Änderungen am Inhalt wirken sich aber direkt auf die Variable aus, die beim Aufruf verwendet wurde.

Klar ist, dass eine Unterscheidungsmöglichkeit zwischen beiden Parameterarten getroffen werden muss. Dazu dient das Schlüsselwort `ref`, das noch vor dem Datentyp als zusätzlicher Modifikator erscheint. Ein Beispiel zeigt dies:

```
<script language="C#" runat="Server">
void Page_Load()
{
    string mustertext = "Dies ist ein Test f&uuml;r Referenzparameter";
    makeBold(ref mustertext);
    fett.InnerHtml = mustertext;
}

void makeBold (ref string htmltext)
{
    htmltext = "<b>" + htmltext + "</b>";
}
</script>
<html lang="de">
    <head>
        <title>Referenzparameter</title>
    </head>
    <body>
    <h1>Referenzparameter</h1>
    Der folgende Text erscheint fett:<br/>
    <p id="fett" runat="server"/>
    </body>
</html>
```

Listing 2.29: Verwendung von Referenzparametern (method_prefparm.aspx)

Wie es funktioniert

Betrachten Sie zuerst die Methode `makeBold`. Es wurde hier ein Parameter als Referenz deklariert:

```
void makeBold (ref string htmltext)
```

Änderungen an der daraus erstellten Variablen `htmltext` wirken sich direkt auf die beim Aufruf verwendete aus. Im Beispiel wird eine Zeichenkette in `<b>`-Tags eingebettet, um sie fett auszuzeichnen:

```
htmltext = "<b>" + htmltext + "</b>";
```

Die Referenz muss auch beim Aufruf verwendet werden, denn C# wird nicht zulassen, dass Sie versehentlich eine Variable verändern, weil eine Methode sie als Referenzparameter erkennt:

```
makeBold(ref mustertext);
```

Ausgabeparameter

Es ist auch möglich, eine Variable überhaupt erst durch eine Methode erzeugen zu lassen. Dabei wird die lokale erzeugte Variable als Referenz zurückgegeben und die beim Aufruf angegebene Variable wird mit dem in der Methode berechneten Wert erzeugt. Auch dafür gibt es ein Schlüsselwort: `out`.

Das Beispiel zeigt die Verwendung:

```
void Page_Load()
{
    string mustertext = "Dies ist ein Test f&uuml;r Referenzparameter";
    string errortext;
    int error = 2;
    makeError(ref mustertext, error, out errortext);
    fett.InnerHtml = "<i>" + errortext + "</i>: " + mustertext;
}

void makeBold (ref string htmltext)
{
    htmltext = "<b>" + htmltext + "</b>";
}

void makeError(ref string htmltext, int error, out string errText)
{
    switch (error)
    {
        case 1:
            errText = "Hinweis";
            break;
        case 2:
            errText = "Warnung";
            break;
        case 3:
            errText = "Fehler";
            break;
        default:
```

```
            errText = "Keine Nachricht";
            break;
    }
    makeBold(ref htmltext);
}

</script>
```

Listing 2.30: Verwendung von Out-Parametern. Der HTML-Teil entspricht Listing 2.29 (method_poutparm.aspx)

Wie es funktioniert

Die Methode `makeError` erwartet drei Parameter. `htmltext` ist ein Referenzparameter, der an die Methode `makeBold` durchgereicht wird. `error` ist ein normaler Parameter, der eine ganze Zahl enthalten darf. Der Ausgabeparameter `errText` gibt einen Text zurück; die Variable, die den Wert aufnimmt, muss zuvor nicht initialisiert worden sein – eine Deklaration des Datentyps ist natürlich dennoch erforderlich:

```
void makeError(ref string htmltext, int error, out string errText)
```

Im Aufruf der Methode müssen die Modifikatoren für die Parameter wiederholt werden:

```
makeError(ref mustertext, error, out errortext);
```

Abbildung 2.18: Arbeit mit Referenz- und Ausgabeparametern

Wenn Sie sich nicht entscheiden können, die eine oder andere Art Parameter zu verwenden, deklarieren Sie mehrere gleichnamige Methoden mit jeweils unterschiedlichen Parameterangeben. Genutzt wird dafür die Methodenüberladung.

2.5 Komplexe Datentypen

Neben den einfachen Datentypen gibt es auch komplexe, die sich aus mehreren einfachen zusammensetzen. Auf diesen bauen wiederum sehr viele speziell für die Nutzung der Klassen des Framework entworfene individuelle Typen auf. Die in C# verfügbaren komplexen Datentypen werden in diesem Abschnitt vorgestellt.

2.5.1 Arrays

Der Umgang mit Arrays erscheint, betrachtet man sich einfache Beispiele, unproblematisch. Trotzdem bergen sie einige Tücken und ebenso auch viele Chancen für clevere Programme. Arrays sind Sammlungen von Daten unter einem gemeinsamen Namen. Ein leeres Array wird in C# erzeugt, indem eckige Klammern dem Datentyp bei der Deklaration nachgestellt werden. Damit ist nicht nur festgelegt, dass es sich um ein Array handelt, sondern auch, welche Art Daten es enthalten darf.

```
string[] arrPlz = new string[2];
```

Arrays müssen in der Größe festgelegt werden, dieser Wert ist aber zur Laufzeit änderbar. Die folgende Abbildung zeigt ein einfaches Array:

arrPlz[]

0	10999
1	12683
2	12459

Abbildung 2.19: Aufbau eines einfachen Array (arrays_strings.aspx)

Auswahl über Indizes

Im einfachsten Fall werden eindimensionale Arrays verwendet. Diese bestehen aus einer beliebigen Anzahl Elementen, die über einen numerischen Index angesprochen werden können. Der Index wird in eckigen Klammern angegeben, der Zugriff auf Elemente erfolgt ebenso. Die Indizes eines solchen Arrays sind immer numerisch und zählen ab Null. Die Angabe der Grenzen erfolgt dagegen durch Angabe der tatsächlichen Anzahl. Das folgende Beispiel zeigt, wie die Postleitzahlen gespeichert werden:

```
<%
  string[] arrPlz = new string[3];
  arrPlz[0] = "10999";
  arrPlz[1] = "12683";
  arrPlz[2] = "12459";
  Response.Write ("PLZ 1: " + arrPlz[0] + "<br>");
  Response.Write ("PLZ 2: " + arrPlz[1] + "<br>");
  Response.Write ("PLZ 3: " + arrPlz[2] + "<br>");
%>
```

Listing 2.31: Anlegen und Auslesen eines Arrays (arrays_string.aspx)

Das Speichern von Postleitzahlen in Zeichenketten ist hier nicht nur eine Demonstration. Denken Sie daran, dass diese Ziffernfolgen auch mit 0 beginnen können, was bei der Speicherung als `int` verloren geht.

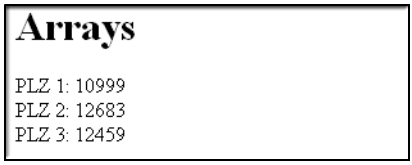


Abbildung 2.20: Ausgabe des Programms arrays_string.aspx

Die Zuweisung der Elemente kann auch bei der Deklaration erfolgen. Dann wird die Anzahl vom Compiler selbst bestimmt.

```
<%
  string[] arrPlz = {"10999", "12683", "12459"};
  Response.Write ("PLZ 1: " + arrPlz[0] + "<br>");
  Response.Write ("PLZ 2: " + arrPlz[1] + "<br>");
%>
```

Listing 2.32: Kombination von Deklaration und Zuweisung (array_string2.aspx)

Arrays ausgeben

Der Zugriff auf ein Element über den Index ist sicher oft notwendig. Ebenso häufig werden aber alle Elemente eines Arrays benötigt. Dafür kommen die Schleifenanweisungen `for` und `foreach` zum Einsatz. Die Ausgabe des einfachen Arrays mit den Postleitzahlen sieht damit folgendermaßen aus:

```
<%
  string[] arrPlz = {"10999", "12683", "12459"};
  int i = 0;
  foreach (string strPlz in arrPlz)
  {
     i++;
     Response.Write ("PLZ Nr " + i + ":"  + strPlz + "<br>");
  }
%>
```

Listing 2.33: Ausgabe des Arrays mit einer Schleife (arrays_foreach.aspx)

Hilfsweise wurde noch eine Variable eingeführt, die fortlaufend die Elemente zählt.

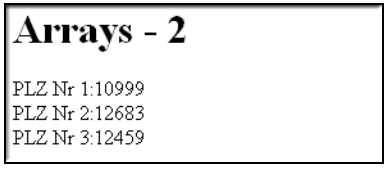


Abbildung 2.21: Ausgabe des Arrays mit foreach

Mehrdimensionale Arrays

Wenn Sie mit einer Tabellenkalkulation wie Excel arbeiten, kennen Sie schon die Struktur eines zweidimensionalen Arrays. Eine Dimension erstreckt sich über die Spalten, die andere über die Zeilen. Jede Zelle enthält aber mehrere Informationen: Formel, aktueller Wert, Name für Bezüge und Formatierung für die Darstellung. Das kann man mit einer dritten Dimension abbilden. Stellen Sie sich jetzt vor, Sie haben mehrere Arbeitsblätter, die alle dieselbe Struktur aufweisen – das wäre dann schon die vierte Dimension. Bevor Sie in Panik verfallen, ob dies noch verständlich ist, ein Tipp aus der Praxis. Wenn Sie mehr als drei Dimensionen benötigen, liegt mit höchster Wahrscheinlichkeit ein Designfehler Ihrer Datenstrukturen vor, das heißt, bei einer anderen (korrekten) Ablage der Daten wäre es viel einfacher. Versuchen Sie, Ihre Strukturen aufzumalen. Papier hat nur zwei Dimensionen und diese sind meist ausreichend.

```
<%
string[,] arrBuch = {{"ASP.NET lernen", "&euro; 24,95", "J&ouml;rg Krause"},
          {"ASP.NET mit VB.NET", "&euro; 59,90", "J&ouml;rg Krause"},
          {"ASP.NET mit C#", "&euro; 59,90", "J&ouml;rg Krause"}};
for(int k = 0; k < arrBuch.GetLength(1); k++)
{
   for(int i = 0; i < arrBuch.GetLength(0); i++)
   {
      Response.Write(arrBuch[k, i] + "<br>");
   }
   Response.Write("<hr noshade>");
}
```

Listing 2.34: Deklaration und Ausgabe eines mehrdimensionalen Arrays (arrays_formulti.aspx)

Zur Deklaration wird hier die implizite Form verwendet. Praktisch handelt es sich bei einem mehrdimensionalen Array um ein Array aus Arrays. Die Ausgabe erfolgt dennoch über Indizes, da diese von Null beginnend vergeben werden. Deshalb können hier Schleifen eingesetzt werden. Die aktuelle Ausdehnung des Arrays wird mit der Methode `GetLength(Dimension)` ermittelt. Wenn Sie die Länge eines eindimensionalen Arrays ermitteln möchten, verwenden Sie `Length` – ohne Parameter und ohne Klammern, denn dies ist eine Eigenschaft.

Der direkte Zugriff auf die einzelnen Elemente eines mehrdimensionalen Arrays erfolgt mit den schon gezeigten eckigen Klammern. Jede Dimension ist durch Kommata getrennt (`arrBuch[k, i]`).

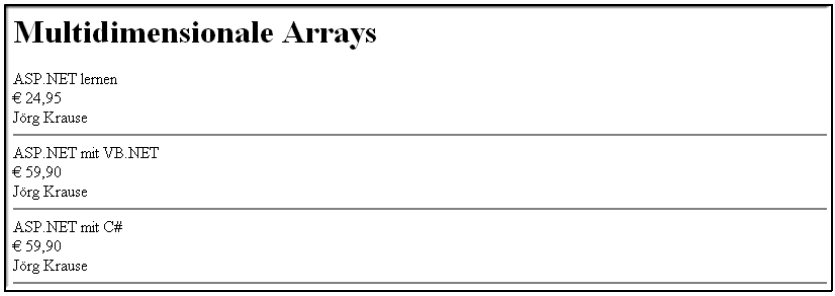


Abbildung 2.22: Ausgabe eines mehrdimensionalen Arrays mit Schleifen

Arrays verändern

Arrays lassen sich während der Laufzeit verändern – nicht nur in Bezug auf existierende Elemente, sondern auch durch Entfernen oder Hinzufügen neuer Elemente.

2.5.2 Universelle Arrays

Wenn Sie nicht wünschen, dass alle Elemente eines Arrays denselben Datentyp haben, definieren Sie es einfach mit dem universellen Typ `Object`.

```
Object[] aObject = new Object[3];
aObject[0] = 123;
aObject[1] = "Krause";
aObject[2] = 13.04;
```

Das Verfahren wird als Boxing bezeichnet – Basisdatentypen werden als Objekte verpackt. Der Zugriff erfolgt in üblicher Weise. Falls es im konkreten Fall nicht möglich ist, nehmen Sie eine explizite Typkonvertierung vor. Wird dabei der urprüngliche Datentyp (beispielsweise `String` für `aObject[1]` im Beispiel) verwendet, gelingt die Konvertierung immer.

2.6 Strukturen und Aufzählungen

C# kennt neben den einfachen Datentypen auch komplexere Gebilde. Arrays wurden bereits vorgestellt, da sie häufiger benötigt werden und einfacher anzuwenden sind. Nicht behandelt werden hier die vielfältigen Kollektionen. Diese stammen nicht aus dem Sprachumfang von C#, sondern aus dem Framework (`System.Collections`). Dieser Teil wird in „Aufzählungen und Kollektionen“ auf Seite 288 vorgestellt.

2.6.1 Strukturen

Strukturen werden ähnlich wie Klassen verwendet. Sie gehören jedoch nicht zu den Referenztypen, sondern sind Werttypen. Damit ist auch klar, dass sie eher für kleine Datenmengen geeignet sind.

Empfehlenswert ist der Einsatz von Strukturen, wenn der Inhalt nicht mehr als 16 Byte umfasst und große Mengen von Objekten aus der Struktur erzeugt werden sollen. Die Größe basiert auf der Überlegung, dass Referenztypen über Verweise im Speicher verwaltet werden und diese auch Speicherplatz benötigen. Kleine Strukturen sind deshalb nicht langsamer, solange sie nicht missbraucht werden.

Für den Einsatz als Punktspeicher käme die im folgenden Programm gezeigte Technik mit Strukturen zum Einsatz.

```
<script language="C#" runat="server">
public struct points
{
   public int x;
   public int y;
   public string name;
}
</script>
<html>
    <head><title></title></head>
<body>
<h1>Klassen - Ereignisse</h1>
<%
points p1 = new points();
p1.x = 17;
p1.y = 23;
p1.name = "Ein Punkt";
Response.Write ("X = " + p1.x.ToString() + " ");
Response.Write ("Y = " + p1.y.ToString() + "<br/>");
Response.Write ("Name = " + p1.y);
%>
```

Listing 2.35: Anlegen und Verwenden einer einfachen Struktur (struct_simple.aspx)

Dies ist die einfachste Form einer Struktur. Alle enthaltenen Variablen sind öffentlich. Einsatz und Verwendung entsprechen praktisch den bei Klassen gezeigten Techniken. Sie können auch Methoden definieren oder vorhandene überschreiben. Ebenso sind die Zugriffsmodifikatoren anwendbar. Das folgende Programm zeigt eine Struktur mit mehreren Methoden:

```
<script language="C#" runat="server">
public struct points
{
   private int __x;
   private int __y;
   public string name;

   public void setpoint(int _x, int _y)
   {
       __x = _x;
       __y = _y;
   }

   public string x()
   {
      return __x.ToString();
   }

   public string y()
   {
      return __y.ToString();
   }

}
</script>
<%
points p1 = new points();
p1.setpoint(17, 23);
p1.name = "Ein Punkt";
Response.Write ("X = " + p1.x() + " ");
Response.Write ("Y = " + p1.y() + "<br/>");
Response.Write ("Name = " + p1.name);
%>
```

Listing 2.36: Struktur mit mehreren Methoden (struct_method.aspx)

Praktisch wird die Methode `setpoint` verwendet, um die Werte definiert zu setzen. Der übrige Teil dürfte weitgehend selbsterklärend sein. Versuchen Sie, die Funktionsweise vollkommen zu verstehen. Legen Sie weitere Punkt-Objekte an und geben Sie diese zur Kontroll wieder aus.

Konstruktoren für Strukturen

Es ist naheliegend, dass in Strukturen auch Konstruktoren möglich sind. Diese haben eine besondere Aufgabe. Sie setzen, wenn sie verwendet werden, alle Werte, aus denen die Struktur besteht. Sie haben also – im Gegensatz zu den Klassen – nicht die Möglichkeit, den Inhalt des Konstruktors völlig frei zu bestimmen. Das folgende Listing zeigt einen Kon-

struktur und die Anwendung bei der Instanziierung der Klasse. Der Rest entspricht dem in Listing 2.36 gezeigten.

```
public points(int _x, int _y)
{
    __x = _x;
    __y = _y;
    name = "";
}
...
points p1 = new points(17, 23);
```

Listing 2.37: Konstruktur für eine Struktur (struct_constr.aspx)

Eine Besonderheit ist noch zu beachten. Wenn Sie, wie im Beispiel gezeigt, die Variable gleich mit `new` mit der Struktur belegen, erfolgt ohne Konstruktor eine implizite Zuweisung der Werte: Numerische Variablen werden mit 0, Boolesche mit `false` und Zeichenkettenvariablen mit einer leeren Zeichenkette gefüllt.

2.6.2 Aufzählungen

enum

In Programmen kommt es oft vor, dass Variablen nur eine fest umrissene Anzahl Werte aufnehmen. Denken Sie beispielsweise an Wochentage oder Monatsnamen. In beiden Fällen wäre der Datentyp string

zwar verwendbar, aber nicht optimal. Das Schlüsselwort, mit dem Aufzählungen erzeugt werden, heißt `enum` (vom englischen Begriff *enumeration*):

```
enum eWeekday {Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag,
Sonntag};
```

Sie verfügen damit noch nicht über eine Variable, sondern nur über eine sehr individuelle Typdefinition. Von `eWeekday` abgeleitete Variablen dürfen einen der angegebenen Werte enthalten. Definieren Sie eine Variable wie üblich, um sie mit einem der Aufzählungswerte zu belegen:

```
eWeekday tag;
```

Der Variablen `tag` können Sie nun einen der Werte zuweisen:

```
tag = eWeekday.Montag;
```

Das Programm finden Sie als Listing unter dem Namen *enum_weekday.aspx*. Bei der Ausgabe wird der erkannte Wert als Zeichenkette ausgegeben. Intern werden jedoch Zahlen zur Indizierung verwendet – daher der Name „Aufzählung“. Die Zählung beginnt mit 0 und setzt jeweils um 1 erhöht fort. Sie können aber andere Werte erzwingen:

```
<script language="C#" runat="server">
enum eWeekday {Montag = 1, Dienstag = 2, Mittwoch = 3, Donnerstag = 4,
               Freitag = 5, Samstag = 6, Sonntag = 0};
eWeekday weekday;
</script>
<%
eWeekday tag1, tag2;
tag1 = eWeekday.Dienstag;
tag2 = (eWeekday)5;
Response.Write (tag1 + "<br/>");
Response.Write (tag2 + "<br/>");
%>
```

Listing 2.38: Aufzählung mit vorgegebenen Indexwerten (enum_weekdayi.aspx)

Zugriff auf Aufzählungen

Der Zugriff ist wie vorher beschrieben über die Namen möglich. Außerdem gibt es eine spezielle Syntax für die Nutzung der Indizes:

```
tag2 = (eWeekday)5;
```

Statt einem konstanten Wert sind auch Variablen zulässig:

```
<script language="C#" runat="server">
enum eWeekday {Montag = 1, Dienstag = 2, Mittwoch = 3, Donnerstag = 4,
               Freitag = 5, Samstag = 6, Sonntag = 0};
eWeekday weekday;
</script>
<%
for(int i = 0; i < 7; i++)
{
    Response.Write ((eWeekday)i + "<br/>");
}
%>
```

Listing 2.39: Zugriff auf eine Aufzählung mit variablem Index (enum_weekdayfor.aspx)

Generell sollten Sie den Datentyp immer so wählen, dass er den für die Erfüllung der Problemstellung kleinstmöglichen Wertebereich umfasst. Aufzählungen sind ein probates Mittel dafür.

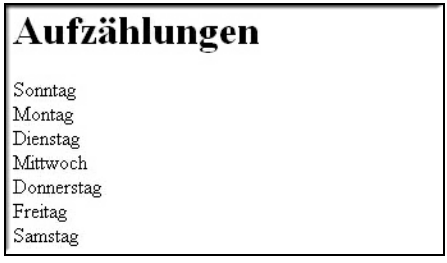


Abbildung 2.23: Ausgabe aller Elemente einer Aufzählung

Im Framework werden Aufzählungen sehr häufig verwendet. Diese sind natürlich bereits fertig definiert und müssen nur verwendet werden. Wie das geht, wird in Abschnitt „Zugriff über Aufzählungen“ auf Seite 295 beispielhaft erklärt.

2.7 Ausnahmen und Ereignisse

C# bietet einige Spracheigenschaften, die bei der Entwicklung kleiner Projekte kaum benötigt werden. Es wird aber der Zeitpunkt kommen, wo Sie größere Programme planen und umsetzen. Dann ist ein Blick auf die hier beschriebenen Methoden sinnvoll. Da hier nur wenig Platz für eine ausführliche Darstellung ist, sei außerdem auf ein gutes C#-Buch für Anfänger verwiesen. In derselben Buchreihe wie dieses Buch erscheint bei Addison Wesley das Buch „C# lernen“ von Frank Eller.

Zwei spezielle Techniken sollen dennoch kurz angerissen werden: Ausnahmen und Ereignisse.

2.7.1 Laufzeitfehler mit Ausnahmen abfangen

Programme müssen auf vielfältige Daten reagieren. Es können Bedingungen auftreten, die zu Laufzeitfehlern führen. Besonders bei Webanwendungen ist es sehr wichtig, dass der Benutzer nicht durch Fehlermeldungen irritiert wird. Ein Verstecken ist fast immer möglich, wenn man bedenkt, welche Fehlerzustände auftreten können. In der Praxis führt es freilich zu enormem Aufwand, alle erdenklichen Fehlerquellen zu erkennen, mit entsprechenden Abfragen zu belegen und Reaktionen darauf zu programmieren. Mit einer speziellen Sprachkonstruktion können Sie diese Arbeit von C# und der CLR erledigen lassen.

Die Sprachanweisung try-catch-finally

try catch finally

Die Anweisung, die hierzu verwendet wird, besteht aus mindestens zwei Schlüsselwörtern: `try` und `catch`. Der erste Teil wird von `try` umschlossen. Hier wird die Ausführung von beliebigem Code versucht. Tritt ein Laufzeitfehler auf, wird – in der C#-Sprechweise – eine Ausnahme „geworfen“. Ist keine Behandlung dafür definiert, hilft sich die CLR selbst und zeigt den Fehler an. Sie können die Ausnahme aber auch „fangen“ (engl. to catch), was mit dem Schlüsselwort `catch` erfolgt. Falls Sie Code schreiben, der immer ausgeführt werden soll, egal ob es einen Laufzeitfehler gab oder nicht, ist ein weiterer durch `finally` eingeleiteter Block notwendig.

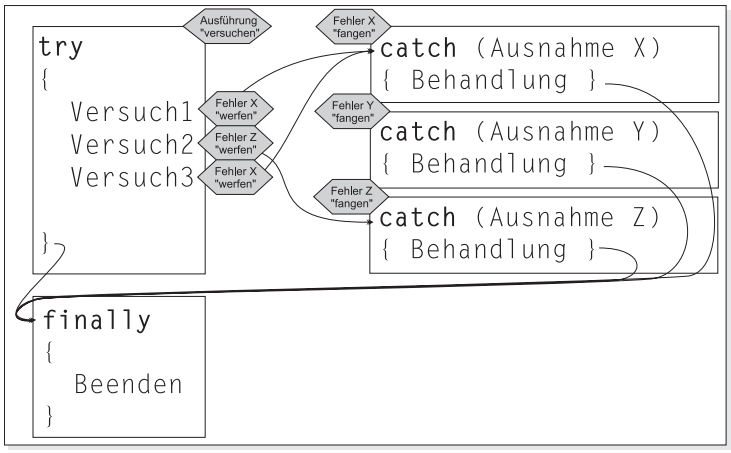


Abbildung 2.24: Ablauf eines Programms mit `try-catch-finally`-Blöcken

Das folgende Beispiel zeigt, wie die Anwendung in der Praxis erfolgt. Abgefangen wird eine Ausnahme, die bei Berechnungen häufiger auftritt: Division durch Null.

```
<script language="C#" runat="Server">
void Page_Load()
{
    int a = 16;
    int b = 2;
    string r;
    Ausnahmen rechner = new Ausnahmen();
    r = rechner.rechne(a, b, '/');
    ausgabe.InnerHtml = r;

}

class Ausnahmen
{
    public string rechne(int x, int y, char opcode)
    {
        int result;
        try
        {
            switch (opcode)
            {
                case '*':
                    result =  (x * y);
                    break;
                case '/':
                    result =  (x / y);
```

```
                    break;
                case '-':
                    result =  (x - y);
                    break;
                default:
                    result =  (x + y);
                    break;
            }
        }
        catch (Exception e)
        {
            return ("<b>Fehler:</b> " + e.ToString());;
        }
        return result.ToString();
    }
}

</script>
<html lang="de">
    <head>
        <title>Try Catch Finally</title>
    </head>
    <body>
    <h1>Try Catch Finally</h1>
    <p id="ausgabe" runat="server"/>
    <p id="fehler" runat="server"/>
    </body>
</html>
```

Listing 2.40: Abfangen einer Ausnahme mit einem `try-catch-Block`

Wie es funktioniert

In diesem Beispiel wird eine zusätzliche Klasse definiert, die neben der Ausführung der Aufgaben auch Fehler auswertet. Die Berechnungen finden in einem `try`-Block statt. Läuft dieser Block korrekt ab, werden nachfolgende `catch`-Blöcke übersprungen. Damit wird die letzte `return`-Anweisung erreicht:

```
return result.ToString();
```

Tritt dagegen eine Ausnahme auf, wird der nächste dazu passende `catch`-Block ausgeführt. Verwendet wird hier der globale Typ einer Ausnahme, `Exception`. Diese Art trifft für alle Ausnahmen zu. Abgeleitet werden diese Ausnahmen in dem in ASP.NET benötigten Umfang von der Klasse `System.SystemException`.

Die eigentlichen Klassendefinitionen sind weit im Framework verstreut, da fast überall Laufzeitfehler auftreten können. Der häufig auftretende Fehler „Division durch Null“ wird in `System.ArithmeticException` definiert.

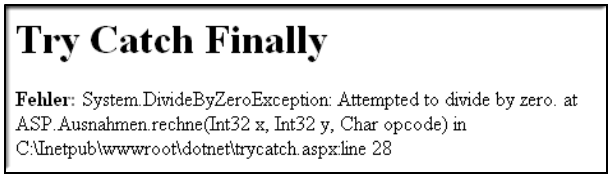


Abbildung 2.25: Ausgabe eines Laufzeitfehlers mit eigenem Code

throw

An einigen Stellen in Ihrem Programm werden Sie vielleicht Fehlerbedingungen selbst feststellen. Statt nun eine erneute Fehlerbehandlungsmethode zu schreiben, nutzen Sie eine aus einem vorhandenen `try-catch`-Block. Dazu „werfen" (engl. to throw) Sie selbst eine Ausnahme. Das Schlüsselwort heißt naheliegenderweise `throw`. Der Umgang damit ist nicht ganz einfach, weil Sie erst eine Klasse definieren müssen, die die Fehlerbehandlung übernimmt, beispielsweise eine Textausgabe. Dann ist es notwendig, eine Instanz dieser Klasse beim Auftreten der Fehlerbedingungen zu „werfen":

```
if (zahl == 0) throw new AusnahmeKlasse("Fehlermeldung");
```

Die Parameter werden im Konstruktor festgelegt.

Eine ausführliche Betrachtung der Verwendung von Ausnahmen finden Sie in dem Standardwerk „ASP.NET mit C# programmieren", von Jörg Krause und Uwe Bünning, erschienen bei Addison Wesley.

2.7.2 Ereignisse

In der Windows-Programmierung spielen Ereignisse eine sehr große Rolle. Im alten ASP war dies weniger ein Thema, weil das im Browser auftretende Ereignis nicht auf direktem Weg zum Server gelangen konnte. In ASP.NET verschmelzen beide Ansichten, denn zum einen unterstützt C# exzellent die Verwendung von Ereignissen, zum anderen bildet ASP.NET dieses Modell auf interessante Weise auch für die Webserverprogrammierung ab. Wie dies erfolgt, wird in Kapitel 3 erläutert. In diesem Abschnitt geht es um die abstrakte Abbildung in C#.

Was Ereignisse sind und wie man sie definiert

event ***delegate***

Ereignisse bestehen immer aus zwei Bestandteilen: Der auslösenden Ursache und der Reaktion darauf. Beides kann in C# direkt definiert werden. Verwendet werden dafür auch zwei getrennte Schlüsselwörter. Der Auslöser wird mit `event` gekennzeichnet, das darauf reagierende Element mit `delegate`. Die Namen sind Methode – das Ereignis wird „delegiert". Deklaration und Definition erfolgen getrennt.

Da Ereignisse komplexe Verarbeitungsschritte erfordern können, können Sie mehrere Methoden mit der Verarbeitung beauftragen. Derartiger Code ist möglicherweise am Anfang schwer lesbar, weil keine direkten Aufrufe stattfinden. Die Ereignisse treten erst zur Laufzeit auf, sodass der Zusammenhang nur durch die entsprechenden Schlüsselwörter transparent wird.

Das folgende Beispiel verwendet die bereits gezeigte Technik der Operatorenüberladung. Zusätzlich wird der Subtraktionsoperator überschrieben. Überwacht werden soll, ob die Werte negativ werden, was nicht erlaubt ist. Das Ereignis tritt also ein, wenn der Wert negativ wird. Die Behandlung, an die es delegiert wird, setzt die Werte auf Null. Der Vorgang ist nicht trivial. Zuerst das komplette Programm; eine Erklärung der für die Ereignisbehandlung wichtigen Passagen folgt danach:

```
<script language="C#" runat="server">
public class myevents
{
    public delegate void handler(xypoint o);
    public event handler handle;

    public void onhandle(xypoint o)
    {
       if (handle != null)
          handle(o);
    }
}

public class xypoint
{
   public int _xpoint;
   public int _ypoint;

   public string x
   {
      get { return (_xpoint.ToString()); }
      set { _xpoint = Convert.ToInt32(value); }
   }

   public string y
   {
      get {  return (_ypoint.ToString()); }
      set {  _ypoint = Convert.ToInt32(value); }
   }
```

```
    public static xypoint operator + (xypoint o1, xypoint o2)
    {
        xypoint o3 = new xypoint();
        o3.x = (Convert.ToInt32(o1.x) + Convert.ToInt32(o2.x)).ToString();
        o3.y = (Convert.ToInt32(o1.y) + Convert.ToInt32(o2.y)).ToString();
        return o3;
    }

    public static xypoint operator - (xypoint o1, xypoint o2)
    {
        xypoint o3 = new xypoint();
        myevents me = new myevents();
        me.handle += new myevents.handler(o3.underrun);
        o3.x = (Convert.ToInt32(o1.x) - Convert.ToInt32(o2.x)).ToString();
        o3.y = (Convert.ToInt32(o1.y) - Convert.ToInt32(o2.y)).ToString();
        if (o3._xpoint < 0) me.onhandle(o3);
        if (o3._ypoint < 0) me.onhandle(o3);
        return o3;
    }

    public void underrun(xypoint o)
    {
        o._xpoint = 0;
        o._ypoint = 0;
    }

}
</script>
<html>
    <head><title></title></head>
<body>
<h1>Klassen - Ereignisse</h1>
<%
xypoint mm1 = new xypoint();
xypoint mm2 = new xypoint();
xypoint mm3 = new xypoint();
mm1.x = "77";
mm1.y = "33";
mm2.x = "10";
mm2.y = "38";
mm3 = mm1 - mm2;
Response.Write ("X1 = " + mm1.x + "<br/>");
Response.Write ("Y1 = " + mm1.y + "<br/>");
Response.Write ("X2 = " + mm2.x + "<br/>");
Response.Write ("Y2 = " + mm2.y + "<br/>");
Response.Write ("X3 = " + mm3.x + "<br/>");
```

```
Response.Write ("Y3 = " + mm3.y + "<br/>");
%>
</body>
</html>
```

Listing 2.41: Verwendung von Ereignissen zur Programmsteuerung (events.aspx)

Wie es funktioniert

Zuerst wird in der Klasse `myevents` die Ereignisbehandlung vorbereitet. Der erste Schritt besteht in der Definition der Delegierung. Die folgende Zeile erzeugt eine mit dem Namen `handler`. Als Parameter wird ein Objekt der Klasse `xypoint` erwartet. Der Parameter ist optional, wird also nur bei Bedarf angegeben:

```
public delegate void handler(xypoint o);
```

Nun folgt die Definition des Ereignisses. Es bekommt hier lediglich einen Namen (`handle`):

```
public event handler handle;
```

Ereignisse müssen auch irgendwo ausgelöst werden. Die Quelle wird später im eigentlichen Programm festgelegt, das Ziel `onhandle` jedoch hier definiert:

```
public void onhandle(xypoint o)
```

Eine wichtige Zeile ist die Abfrage der Existenz des Ereignisses selbst. Was nämlich passiert, wird später definiert. Erfolgt diese Definition nicht – der Vorgang passiert erst zur Laufzeit – würde hier ein Laufzeitfehler aufreten. Existiert das Ereignisobjekt `handle`, wird das Ereignis ausgelöst:

```
if (handle != null) handle(o);
```

Das Objekt `o` ist der Parameter, der die ganze Strecke über nur durchgereicht wird. Zur Laufzeit ist das Objekt vom Typ `xypoint` drin.

Das Auslösen kann nur im konsumierenden Teil des Prozesses stattfinden. Das Programm, dass zur Laufzeit Berechnungen ausführt, kann bestimmte Zustände feststellen und darauf reagieren, indem das Ereignis ausgelöst wird. Im Programm wurde gegenüber der letzten Version auch der Operator überladen. Diese Methode soll mit einem Ereignis reagieren, wenn einer der beiden Werte unter 0 fällt. Dazu wird zuerst eine Instanz der Ereignisklasse benötigt.

```
myevents me = new myevents();
```

ist das neue Objekt, das das Ergebnis der Berechnung enthält. Da `o3` von der Klasse `xypoint` stammt, besitzt es alle Eigenschaften und Methoden dieser Klasse. Dazu gehört auch die Methode `underrun`, die sich um die

Fehlberechnung kümmern soll. Die folgende Zeile ist der eigentlich entscheidende Code im gesamten Programm. Dem Ereignis `handle` des Ereignisobjekts `me` wird eine neue Delegierung `handler` zugewiesen. Damit diese weiß, was sie bei Eintritt des Ereignisses tun soll, wird die Methode übergeben, die die Reaktion bearbeitet. Da es sich um eine Methode der Klasse `xypoint` handelt, wird die aktuelle Instanz `o3` verwendet:

```
me.handle += new myevents.handler(o3.underrun);
```

Jetzt folgt die Berechnung. Das Ergebnis kann ein negativer Wert sein. Eine einfache Bedingungsabfrage stellt dies fest und löst das Ereignis aus. Das fehlerhaft berechnete Objekt wird übergeben:

```
if (o3._xpoint < 0) me.onhandle(o3);
if (o3._ypoint < 0) me.onhandle(o3);
```

Nach dem das Ereignis definiert, delegiert und mit dem auslösenden Kriterium verbunden wurde, fehlt noch die Definition der Methode, die darauf letztendlich reagiert; die Methode `underrun`. Dieser Methode wird das aktuelle Objekt übergeben:

```
public void underrun(xypoint o)
```

Dann werden die beiden Werte direkt auf 0 gesetzt:

```
o._xpoint = 0;
o._ypoint = 0;
```

Die folgende Abbildung zeigt die Reaktion. Beim Y-Wert wird 33-38 berechnet; der Wert ist negativ und deshalb wird das Ergebnisobjekt auf 0 gesetzt. Ändern Sie die Zahl im Code so, dass der Wert nicht negativ wird, sehen Sie das Ergebnis der Berechnung.

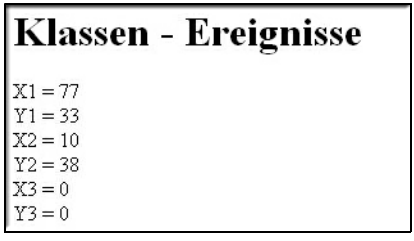


Abbildung 2.26: Verarbeitung mit Ereignissen

Wie es weitergeht

Delegates

Ereignisse sind nur eine einfache und spezielle Form der Delegierungen. Da es für die ASP.NET-Programmierung nicht zwingend notwendig ist, derartige Techniken zu verwenden, soll dieses Thema hier nicht weiter ausgebaut werden. Im Standardwerk des Verfassers, „ASP.NET-Program-

mierung mit C#“, ebenfalls bei Addison Wesley erschienen, finden Sie weitergehende Informationen dazu. Prinzipiell handelt es sich bei Delegates um einen allgemeinen Mechanismus zum Weitergeben von Funktionsaufrufen. Dies wird in C++ mit Funktionszeigern erledigt. Delegates sind in C# ein sicherer – vom Framework verwalteter – Weg, vergleichbar zu programmieren. Anfänger sollten dies erst versuchen, wenn sie im Umgang mit den übrigen Sprachelementen sicher sind.

2.8 Basisklassen des Namensraumes System

Bei der bisherigen Vorstellung haben Ihnen im Vergleich zu VBScript oder auch Visual Basic vielleicht die vielen Funktionen zur Verarbeitung von Arrays oder Zeichenketten gefehlt. Tatsächlich sind derartige Funktionen in C# nicht enthalten. Sie müssen dazu auf Klassen des .Net-Frameworks zurückgreifen. Solche elementaren Klassen sind im Namensraum `System` definiert, den ASP.NET automatisch einbindet. Wenn Sie mit hinterlegtem Code (Code Behind) arbeiten, C# also in einer eigenen Datei ablegen, ist dieser Namensraum explizit anzugeben. Informationen zu Code Behind finden Sie in Abschnitt „Hinterlegter Code: Code Behind“ auf Seite 179. Dies ist eine spezielle ASP.NET-Technik.

Wichtige Klassen

Hier werden nur die Klassen aus `System` vorgestellt, die im Rahmen dieses Buches verwendet werden und für die ersten Projekte in ASP.NET ausreichend sind:

- `Array` – Diese Klassen enthält Eigenschaften und Methoden zur Verwendung mit Arrays.
- `CharEnumerator` – Mit dieser Klasse wird auf einzelne Zeichen einer Zeichenkette verwiesen.
- `Convert` – Durch das typstrenge Konzept müssen oft Datentypen umgewandelt werden, wozu Convert eingesetzt wird.
- `Enum` – Aufzählungen lassen sich mit dieser Klasse erstellen und nutzen.
- `Math` – Mathematische Berechnungen basieren auf Eigenschaften und Methoden dieser Klasse.
- `Random` – Dient der Erzeugung von Zufallszahlen.
- `String` – Hier sind alle Eigenschaften und Methoden zur Verarbeitung von Zeichenketten zu finden.

Weitere wichtige Methoden und Eigenschaften dienen dem Umgang mit Datentypen. Wann immer Datumsoperationen oder Eigenschaften von Zeichen usw. benötigt werden, können Sie auf die Eigenschaften und Methoden von Datentypen zurückgreifen, da auch diese Objekte

sind. Mehr Informationen dazu finden Sie in „Eigenschaften und Methoden der Datentypen“ auf Seite 127.

2.8.1 Verarbeitung von Arrays

Arrays wurden bereits aus Sicht der Programmiersprache C# beschrieben. Eine ganze Palette von Eigenschaften und Methoden vereinfacht den Umgang damit. Alle Mitglieder stammen aus dem .Net-Framework, konkret aus der Klasse `System.Array`. In ASP.NET wird `System` automatisch eingebunden – dieser Namensraum enthält die Klasse. In externen C#-Dateien deklarieren Sie ihn zur Nutzung von Zeichenketten:

```
using System;
```

Typische Eigenschaften und Methoden

Größe feststellen Arrays

Die Feststellung der aktuellen Größe eines Arrays wird mit der Eigenschaft `Length` festgestellt. Hat das Arrays mehr als eine Dimension, werden alle Elemente aller Dimensionen gezählt. Mit `Rank` kann die Anzahl der Dimensionen ermittelt werden. Die Methode `GetLength` zählt dagegen die Elemente einer bestimmten Dimension.

In manchen Situationen ist es besser, mit Aufzählungstypen statt Arrays zu arbeiten. Dann können Sie die Schnittstelle `IEnumerator` des Arrays mit `GetEnumerator` verwenden. Mehr Informationen darüber finden Sie in Abschnitt „Aufzählungen und Kollektionen“ auf Seite 288. Aufzählungen verfügen über Methoden, mit denen man sich programmtechnisch durch die Elemente bewegen kann, wie beispielsweise `MoveNext`. Solche Methoden kennt die Klasse `Array` nicht.

Elemente kopieren

Ein komplettes Array können Sie an einer bestimmten Stelle in einem anderen einfügen, dazu wird `CopyTo` verwendet.

Das funktioniert aber nur mit eindimensionalen Arrays. Ein Teil wird dagegen mit `Copy` eingefügt. Der lesende Zugriff auf ein spezifisches Element erfolgt mit `GetValue`. Als Parameter werden numerische Indizes für jede Dimension erwartet, die das Array besitzt. Schreibend wird mit `SetValue` zugegriffen.

Ein neues Array mit einem bestimmten Datentyp der Elemente wird mit `CreateInstance` erzeugt:

```
Array namen = Array.CreateInstance( typeof(String), 5);
```

Dieses Array kann fünf Elemente vom Typ `String` aufnehmen. In C# kann man dafür auch Folgendes schreiben:

```
string[5] namen;
```

Um ein bestimmtes Element in einem Array zu suchen, wird `BinarySearch` verwendet. Als Parameter wird das Array und das zu suchende Objekt erwartet, zurückgegeben wird der Index. Beachten Sie, dass ein konstanter Wert mit dem Boxing-Verfahren in ein Objekt verwandelt werden kann:

Elemente suchen

```
<script runat="server" language="C#">
void Page_Load()
{
    string[] namen = {"Schmidt", "Krause", "Mueller", "Meier"};
    Object name = "Krause";
    int ni = Array.BinarySearch(namen, name);
    ausgabe.InnerHtml += "Gefundener Index: " + ni.ToString();
}
</script>
<html>
    <head>
        <title>Arrays</title>
    </head>
    <body>
        <h1>Array-Methoden</h1>
        <p id="ausgabe" runat="server"/>
    </body>
</html>
```

Listing 2.42: Durchsuchen eines Arrays (array_methods1.aspx)

In diesem Fall enthält nach Ausführung `ni` den Wert „Schmidt“. Die Suche funktioniert nur, wenn das Suchwort mit dem Arrayinhalt völlig übereinstimmt.

Soll ein Array in umgekehrter Reihenfolge gelesen werden, bietet sich die Methode `Reverse` an. Sortiert werden kann dagegen mit `Sort`. Beide Methoden sind statisch. `Sort` arbeitet nur mit eindimensionalen Arrays. Allerdings gibt es mehrere Überladungen, sodass auch komplexere Sortiervorgänge abgewickelt werden können. So kann ein Array auf der Basis eines anderen sortiert werden. Die Methode `Sort` ändert immer das Original. Das folgende Beispiel zeigt die Verwendung; die HTML-Ausgabe entspricht dem letzten Listing und wird nicht wiederholt abgedruckt:

Sortieren

```
void Page_Load()
{
    string[] namen = {"Schmidt", "Krause", "Mueller", "Meier"};
    string[] daten = {"1.12.1973", "13.4.1980", "23.8.1960", "14.9.1977"};
    Array.Sort(namen, daten);
    for (int i = 0; i < namen.Length; i++)
```

```
    {
        ausgabe.InnerHtml += "<br/>" + namen[i] + ", " + daten[i];
    }
}
```

Listing 2.43: Sortieren eines Arrays und davon abhängig eines zweiten (Ausschnitt aus array_methods2.aspx)

Die Ausgabe zeigt, dass das alphabetisch aufsteigend sortierte erste Array `namen`, das zweite mit den Geburtsdaten ebenfalls beeinflusst hat.

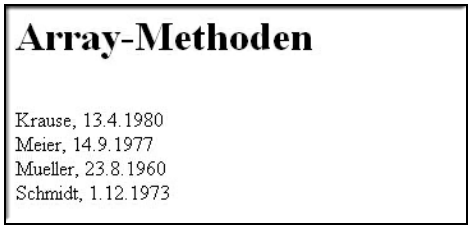


Abbildung 2.27: Sortierte Arrays

Definition eigener Suchalgorithmen

Sortieren

Eine einfache aufsteigende Sortierung wird zwar oft benötigt, mehr Sortieroptionen wären aber wünschenswert. Das .Net-Framework bietet dafür lediglich eine Implementierungsschnittstelle, Sie müssen andere Sortierkriterien selbst implementieren. Die Schnittstelle heißt `IComparer`.

Sie wird implementiert, indem auf Basis der Schnittstelle eine Klasse entwickelt wird, die eine Methode `Compare` enthält. Diese Methode erwartet zwei Objekte. Die Sortierung basiert auf dem Rückgabewert; bei 0 sind die Objekte gleich, bei einer negativen Ganzzahl ist das erste Objekt „kleiner“, bei einer positiven Zahl „größer“. Was Sie als kleiner oder größer betrachten, ist Ihnen überlassen. Das folgende Beispiel sortiert ein Array nach der Länge der Elemente, wobei angenommen wird, dass es sich um Zeichenketten handelt:

```
<script runat="server" language="C#">
public sealed class Comparer : IComparer
{
   public int Compare(Object a, Object b)
   {
      String sa = a.ToString();
      String sb = b.ToString();
      if (sa.Length < sb.Length) return -1;
      if (sa.Length > sb.Length) return 1;
      return 0;
   }
}
```

```
void Page_Load()
{
    //
    string[] namen = {"Schmidtchen", "Krause", "Mueller", "Meier"};
    string[] daten = {"1.12.1973", "13.4.1980", "23.8.1960", "14.9.1977"};
    Array.Sort(namen, daten, new Comparer());
    for (int i = 0; i < namen.Length; i++)
    {
        ausgabe.InnerHtml += "<br/> " + namen[i] + ", " + daten[i];
    }
}
</script>
```

Listing 2.44: Arrays mit eigenem Sortierkriterium (array_methods3.aspx)

Wie es funktioniert

Das Geheimnis steckt hier in der Klasse `Comparer`, die von der Schnittstelle `IComparer` erbt:

```
public sealed class Comparer : IComparer
```

Dort wird eine Methode definiert, die zwei Objekte vergleichen kann:

```
public int Compare(Object a, Object b)
```

Für das konkrete Beispiel wird angenommen, dass es sich um Zeichenketten handelt, deshalb erfolgt – nicht fehlersicher und sehr direkt – die Umwandlung mit `ToString`:

```
String sa = a.ToString();
```

Dann folgt der Vergleich der Längen, wobei zuerst festgestellt wird, ob die erste Zeichenkette kleiner als die zweite ist:

```
if (sa.Length < sb.Length) return -1;
```

Auch der umgekehrter Fall wird analysiert:

```
if (sa.Length > sb.Length) return 1;
```

Trifft beides nicht zu, müssen die Zeichenketten gleich sein:

```
return 0;
```

Damit sind alle Pfade der Methode abgedeckt, es gibt keine unerwarteten Zustände mehr. Die Anwendung erfolgt durch Übergabe einer Instanz der Klasse an die `Sort`-Methode:

```
Array.Sort(namen, daten, new Comparer());
```

Dieses Verfahren können Sie auch verwenden, wenn nur ein Array oder eine der anderen Implementierungen von `Sort` verwendet wird.

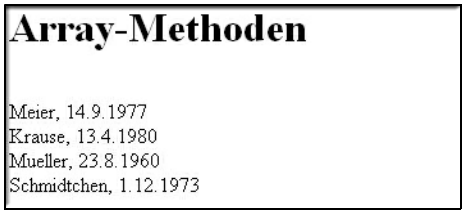


Abbildung 2.28: Sortierung eines Arrays mit eigener Sortiermethode

Wenn Sie tiefer in die Materie einsteigen möchten, müssen Sie die Technik der Schnittstellen und Implementierungen verstehen. Das gezeigte Beispiel funktioniert, ist aber auf das absolut nötige Minimum reduziert worden. Die Möglichkeiten, die das .Net-Framework hier bietet, sind enorm.

2.8.2 Mathematische Operationen

Die Sprache C# bietet außer den elementaren Rechenoperationen keine eigenen sprachlichen Elemente für Berechnungen. Alle mathematischen Methoden stammen aus dem .Net-Framework, konkret aus der Klasse `System.Math`. In ASP.NET wird `System` automatisch eingebunden – dieser Namensraum enthält die Klasse. In externen C#-Dateien deklarieren Sie ihn folgendermaßen :

```
using System;
```

Felder

Zwei statische Felder dienen als Quelle zweier Konstanten: `E` für Eulersche Zahl und `PI` für Pi.

Methoden

Abs

Alle Methoden der Klasse `Math` sind statisch. Sie können also nicht `-3.Abs` schreiben, um den absoluten Betrag zu bestimmen, sondern müssen folgende Notation wählen:

```
Math.Abs(-3).
```

Die Anwendung entspricht dem in anderen Programmierumgebungen üblichen Aufruf von eingebauten Funktionen, von der vorangestellten Klasse `Math` abgesehen.

Ceiling, Floor

Die Verwendung bereitet sicher wenig Schwierigkeiten. In der Praxis sollten Sie sich gelegentlich an einige Methoden erinnern, die trickreich zur Problemlösung eingesetzt werden können. So werden bei der Aus-

gabe langer Listen manchmal Aufteilungen in Blöcke oder Seiten benötigt. Wenn Sie 21 Elemente haben und jede Liste 5 Elemente lang sein darf, besteht die Fragestellung darin, die Anzahl der Listen zu ermitteln. Die Antwort ist 5 – die ersten 4 Listen enthalten 5 Elemente und die letzte 1. Eine einfache Division führt nicht zum Ziel: 21⁄5 = 4.2. Runden funktioniert also nicht. Sie benötigen eine Methode, die immer aufrundet: `Ceiling`. Manchmal wird auch ein generelles Abrunden benötigt: `Floor`.

Round, Pow, Sign

Mathematisch korrekt runden Sie dagegen mit `Round`, ein zweiter Parameter gibt die Anzahl der Kommastellen an. Potenzen werden mit `Pow` berechnet. Das Vorzeichen wird mit `Sign` ermittelt, wobei die Zahlen –1, 0 oder 1 zurückgegeben werden, nicht `true` oder `false`.

Min, Max

Wenn Sie andere Programmiersprachen kennen, sollten Sie wissen, das `Min` und `Max` nur zwei Parameter ermöglichen, nicht unbegrenzt viele.

Die Klasse Random

Zufallszahlen

Zufallszahlen sind immer wieder ein Thema in der Programmierung. Im Namensraum `System` gibt es die Klasse `Random`, die das Erzeugen derartiger Nummern sehr komfortabel gestaltet. Die Mitglieder dieser Klasse sind nicht statisch, Sie müssen deshalb eine Instanz der erzeugen:

```
Random rnd = new Random();
```

`rnd` ist nun das Objekt, das Zufallszahlen aller Art liefern kann. Verwenden Sie folgende Methoden:

- `Next` – Hiermit wird eine Zufallszahl erzeugt. Ohne Parameter im gesamten Wertebereich `Int32`, mit einem Parameter zwischen 0 und dem angegebenen Wert, mit zwei Parametern zwischen diesen beiden.
- `NextDouble` – Diese Methode erzeugt eine Gleitkommazahl vom Typ `double` größer 0.0 und kleiner als 1.0.
- `NextBytes` – Die Methode füllt ein beliebig großes Arrays aus Bytes mit Bytewerten (0 bis 255).

Die Verwendung aller Methoden zeigt das folgende Beispiel:

```
void Page_Load()
{
    Random rnd = new Random();
    Byte[] b = new Byte[10];
    rnd.NextBytes(b);
    for (int i = 0; i < 10; i++) {
        ausgabe.InnerHtml += i.ToString() + ":" + b[i].ToString() + "<br/>";
    }
```

```
    ausgabe.InnerHtml += "<p/>";
    ausgabe.InnerHtml += rnd.NextDouble();
    ausgabe.InnerHtml += "<p/>";
    ausgabe.InnerHtml += rnd.Next(1000, 99999);
}
```

Listing 2.45: Erzeugen von verschiedenen Zufallszahlen (randoms.aspx)

Die Ausgabe zeigt, wie die Methoden reagieren:

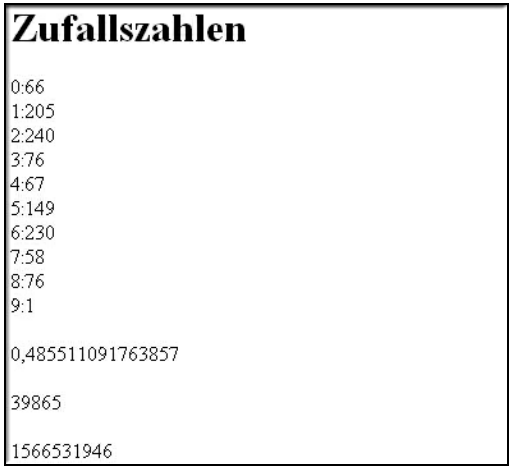


Abbildung 2.29: Ausgabe verschiedener Zufallszahlen

Startwerte

Wie üblich bei Zufallszahlen basiert der Algorithmus auf einem Startwert. Die Zuweisung erfolgt mit der Instanziierung. Der Konstruktor verwendet einen Zeitstempel, wenn kein Startwert angegeben wurde. Alternativ kann auch ein Startwert vorgegeben werden:

```
Random rnd = new Random(12345);
```

Dann beginnt die Zufallsfolge immer wieder an der gleichen Stelle.

2.8.3 Verarbeitung von Zeichenketten

Operationen mit Zeichenketten nutzen Eigenschaften und Methoden der Klasse `System.String`. In ASP.NET wird `System` automatisch eingebunden – dieser Namensraum enthält die Klasse. In externen C#-Dateien deklarieren Sie ihn folgendermaßen:

```
using System;
```

Eigenschaften

Die Klasse `String` verfügt über zwei Eigenschaften, `Chars` und `Length`. Mit `Chars` erhalten Sie Zugriff auf einzelne Zeichen der Zeichenkette. Diese Eigenschaft ist zugleich auch ein Indexer, deshalb muss die Angabe des Indizes des gewünschten Zeichens in der Arrayschreibweise erfolgen. Beachten Sie bei der Angabe des Indizes, dass dieser bei 0 beginnt. `Length` ermittelt die Länge der Zeichenkette.

```
<% @Page Language="C#" debug="true" %>
<script runat="server" language="C#">
void Page_Load()
{
    string zk = "Ich bin eine Zeichenkette";
    ausgabe.InnerHtml  = "Ausgabe:";
    ausgabe.InnerHtml += "<br/>Zeichen 7 per Indexer: " + zk[6];
    ausgabe.InnerHtml += "<br/>Länge: " + zk.Length;
}

</script>
<html>
    <head>
        <title>Zeichenketten</title>
    </head>
    <body>
        <h1>Umgang mit Zeichen und Zeichenketten</h1>
        <p id="ausgabe" runat="server"/>
    </body>
</html>
```

Listing 2.46: Zeichenketten-Eigenschaften `Chars` und `Lengths` (string_chlen.aspx)

Die Ausgabe wird hier in das HTML Server-Steuerelement `ausgabe` geschrieben. Der siebte Buchstaben wird folgendermaßen adressiert:

```
zk[6]
```

Die Länge wird durch Anwendung der Eigenschaft `Length` ermittelt: ***Length***

```
zk.Length
```

Beachten Sie, dass der Indexer `Chars` ein „Zeichen" zurückgibt. C# unterscheidet zwischen einem einzelnen Zeichen und einer Zeichenkette mit einem Zeichen.

Der Unterschied fällt auf, wenn Sie aus einer Zeichenkette, die aus Ziffern besteht, ein Zeichen entnehmen und als Ziffer verwenden möchten. Naheliegend wäre folgende Schreibweise:

```
string s = "123456";
int i = Convert.ToInt32(s[1]);
```

Erwartet wird eigentlich, dass in `i` jetzt die Zahl 2 ist. Tatsächlich steht dort 50 – der Unicode (und auch ASCII/ANSI in diesem Fall) des Zeichens „2“.

Für eine derartige Umwandlung muss also sicher gestellt werden, dass eine Zeichenkette vorliegt:

```
string s = "123456";
int i = Convert.ToInt32(s[1].ToString());
```

Auch `ToString` stammt aus der Klasse `Convert`, wird aber automatisch allen anderen Objekten vererbt, sodass die explizite Angabe nicht erforderlich ist.

Freilich stellt sich hier die Frage, ob man so umständlich auf Zeichen einer Zeichenkette zugreifen muss. Ein Blick auf die Zeichenketten-Methoden lohnt deshalb.

Methoden für Zeichenketten

Substring
IndexOf
LastIndexOf

Es gibt sehr viele Methoden, die wichtigsten werden hier vorgestellt. Häufig ist der Zugriff auf Teile einer Zeichenkette notwendig. Dann verwenden Sie `Substring`. Diese Methode verlangt zwei Parameter; den Startwert und die Anzahl der Zeichen, die herausgenommen werden sollen. Bei der Angabe der Indizes in `Substring` müssen Sie wieder darauf achten, dass die Zählung der Zeichen intern mit 0 beginnt. Das erste Auftreten eines bestimmten Zeichens wird mit `IndexOf` ermittelt, `LastIndexOf` stellt das letzte Vorkommen des Zeichens fest.

ToCharArray

Auf alle Zeichen kann auch mit `ToCharArray` zugegriffen werden – diese Methode erzeugt aus den Zeichen ein Array mit Elementen vom Typ `Char`. Die Definition erfolgt folgendermaßen:

```
char[] azk;
```

Die Methode verlangt die Angabe des ersten und letzten Zeichens, das kopiert werden soll. Im Beispiel wird die gesamte Zeichenkette verwendet, das erste Zeichen hat den Index 0, das letzte wird mit `Length` bestimmt:

```
azk = zk.ToCharArray(0, zk.Length);
```

Die Ausgabe mit `foreach` dient nur der Demonstration:

```
<% @Page Language="C#" debug="true" %>
<script runat="server" language="C#">
void Page_Load()
{
    char[] azk;
    string zk = "Ich bin eine Zeichenkette";
```

```
    ausgabe.InnerHtml  = "Ausgabe:";
    ausgabe.InnerHtml += "<br/>Teil mit 'Substring': " + zk.Substring(4, 3);
    ausgabe.InnerHtml += "<br/>Erstes 'n': " + zk.IndexOf("n");
    ausgabe.InnerHtml += "<br/>Letztes 'n': " + zk.LastIndexOf("n");
    ausgabe.InnerHtml += "<br/>";
    azk = zk.ToCharArray(0, zk.Length);
    foreach (char c in azk)
    {
        ausgabe.InnerHtml += " " + c.ToString();
    }
}

</script>
<html>
    <head>
        <title>Zeichenketten</title>
    </head>
    <body>
        <h1>Umgang mit Zeichen und Zeichenketten</h1>
        <p id="ausgabe" runat="server"/>
    </body>
</html>
```

Listing 2.47: Verwendung von Zeichenketten-Methoden (string_methods1.aspx)

Im Beispiel werden die zuvor beschriebenen Methoden verwendet. Die Ausgabe der Ergebnisse erfolgt in ein HTML Server-Steuerelement, dass mit `<p runat="server">` entsteht, durch verändern der Eigenschaft `InnerHtml`.

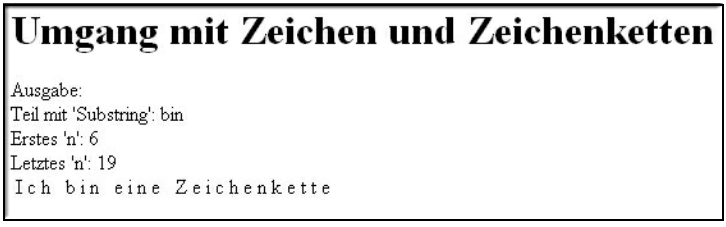


Abbildung 2.30: Wirkungsweise der Zeichenketten-Methoden

Split

Die Umwandlung in Arrays und die Rückumwandlung ist ebenfalls eine sehr oft benötigte Funktion. Nutzen Sie `Split`, um eine Zeichenkette an einem definierten Zeichen zu zerlegen. Die Fragmente werden in einem Array aus Zeichenketten abgelegt. Damit mehr als ein Trennzeichen verwendet werden kann, akzeptiert die Methode ein Array aus Zeichen. Ein zweiter Parameter begrenzt die Anzahl der Teile; dieser Parameter ist optional, ohne Angabe werden alle Teile zurückgegeben.

Join

Die Umkehrfunktion dazu ist `Join`. Als neues Trennzeichen kann nur eine Zeichenkette angegeben werden. Zwei optionale Parameter bestimmen den ersten Index und die Anzahl der Elemente des verwendeten Arrays.

```
void Page_Load()
{
    string[] astr;
    char[] ctrenn = {' ', ','};
    string zk = "Ich bin eine Zeichenkette,mit Komma";
    ausgabe.InnerHtml  = "Ausgabe:<br/>";
    astr = zk.Split(ctrenn, zk.Length);
    foreach (string str in astr)
    {
        ausgabe.InnerHtml += str + "<br/>";
    }
    string strn = String.Join("|", astr);
    ausgabe.InnerHtml += strn;
}
```

Listing 2.48: Trennen und Zusammensetzen von Zeichenketten (string_method2.aspx)

Die Trennung erfolgt hier an Leerzeichen und Kommatas. Die dafür notwendige Definition eines Zeichenarrays kann folgendermaßen aussehen:

```
char[] ctrenn = {' ', ','};
```

Die Aufteilung der Zeichenkette passiert in dieser Zeile:

```
astr = zk.Split(ctrenn, zk.Length);
```

Das Zusammensetzen mit neuem Trennzeichen erledigt `Join`:

```
string strn = String.Join("|", astr);
```

Beachten Sie hier, dass `Join` eine statische Methode ist, die nicht auf eine Zeichenketteninstanz angewendet werden kann. Schreiben Sie stattdessen den Klassennamen davor: `String`.

Abbildung 2.31: Teilen und Zusammenfügen von Zeichenketten

Bei der Annahme von Werten – beispielsweise aus Formularen – sind auch Umwandlungen nötig. Lästige Leerzeichen lassen sich mit `Trim` (am Anfang und am Ende), bzw. mit `TrimStart` und `TrimEnd` entfernen. Wenn Sie mehr als nur Leerzeichen entfernen möchten, geben Sie als Parameter ein Array aus Zeichen an, die entfernt werden sollen. Umwandlungen in Kleinbuchstaben werden mit `ToLower` vorgenommen, in Großbuchstaben mit `ToUpper`.

Leerzeichen entfernen

Wichtig sind auch Formatierungen von Zahlen, die als Zeichenkette dargestellt werden sollen. Dazu dient in erster Linie die statische Methode `Format`. Als Parameter sind mindestens eine Formatzeichenkette und das zu formatierende Objekt anzugeben. Die Formatzeichenkette enthält Formatierungsanweisungen, die diesem Aufbau folgen:

Formatierungen

```
{N [,M][:F[d]]}
```

Die Optionen `M` und `F` sind optional (durch die eckigen Klammern angedeutet, die nicht mit geschrieben werden. `N` ist eine Nummer, die einen aus der Liste der zu formatierenden Werte auswählt. `F` qualifiziert die Formatierung, die möglichen Werte sind der folgenden Tabelle zu entnehmen. `d` legt bei numerischen Werten die Anzahl der Nachkommastellen fest. Auch diese Angabe ist optional.

Formatsymbol	Bedeutung
C	Währung
D	Dezimalzahl
E	Exponentialzahl
F	Gleitkommazahl (Float)
N	Numerisch
P	Prozentwert
R	Erzeugter Wert wird so dargestellt, dass eine Rückkonvertierung möglich ist
X	Hexadezimalzahl

Tabelle 2.9: Formatieroptionen von Zahlen als Zeichenketten (F)

Der Formatwert kann groß oder klein geschrieben werden. Nachkommastellen, die nicht dargestellt werden können, werden korrekt gerundet. Alle Zeichen der Formatierungszeichenkette, die nicht in geschweiften Klammern stehen, werden unverändert ausgegeben. Sollen dennoch Zeichenketten in den Parametern mit aufgenommen werden, schreiben Sie nur die Nummer des Parameters: `{3}`. Die Formatierung von Zahlen zeigt das folgende Beispiel:

```
void Page_Load()
{
    string zk = "Jörg Krause, Berlin";
```

```
    ausgabe.InnerHtml  = "Ausgabe:<br/>";
    ausgabe.InnerHtml += zk.ToUpper();
    double zahl = 12352.4565;
    double i = 0.46;
    ausgabe.InnerHtml += "<br/>" + String.Format("{0:N2}, {1:P0}", zahl, i);
}
```

Listing 2.49: Verwendung von Umwandlungs- und Formatierungmethoden (string_methods3.aspx)

Die Ausgabe zeigt die Wirkung auf die einzelnen Werte:

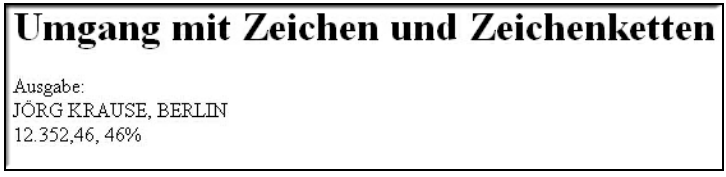


Abbildung 2.32: Formatierung von Zahlenwerten und Zeichenketten

In diesem Abschnitt wird nur auf die Formatierung von Zahlen und Zeichenketten eingegangen. Wenn als Parameter andere Datentypen eingesetzt werden, haben die Formatierzeichen eine grundlegend andere Bedeutung. In den entsprechenden Abschnitten wird darauf explizit hingewiesen.

Replace

Ersetzungen nehmen Sie mit `Replace` vor:

```
void Page_Load()
{
    string zk = "Jörg Krause, München";
    ausgabe.InnerHtml  = "Ausgabe:<br/>";
    ausgabe.InnerHtml += zk.Replace("München", "Berlin");
}
```

Die Methode ersetzt alle Vorkommen des ersten Parameters mit dem zweiten. Als Datentypen sind `String` oder `Char` erlaubt, beide Parameter müssen aber vom gleichen Typ sein.

Weitere Methoden

Die Möglichkeiten, die die Zeichenkettenfunktionen bieten, sind sehr umfangreich. Eine detaillierte Darstellung sprengt den Rahmen dieses Buches. Nutzen Sie die folgende Liste als Stichwortquelle für die Online-Referenz:

- `StartWith`, `EndWith` – Diese Methoden geben `true` zurück, wenn eine Zeichenkette mit bestimmten Zeichen beginnt bzw. endet.
- `PadLeft`, `PadRight` – Mit diesen Methoden kann eine Zeichenkette bis zu einer bestimmten Anzahl Zeichen aufgefüllt werden.

- `IndexOfAny`, `LastIndexOfAny` – Analog zu den Index-Methoden wird das Auftreten bestimmter Zeichen festgestellt, als Eingabewert kann ein Array aus Zeichen genutzt werden.
- `CopyTo` – Diese Methode kopiert eine Anzahl Zeichen an eine bestimmte Position eines Arrays aus Zeichen.
- `Concat` – Dies ist eine statische Methode, die der Verknüpfung von Zeichenketten in Situationen dient, in denen der Verknüpfungsoperator + nicht angebracht ist.

Abschließend sei noch auf den statischen Feldwert `Empty` hingewiesen, der eine leere Zeichenkette erzeugt:

```
string leer = String.Empty;
```

2.9 Eigenschaften und Methoden der Datentypen

Aus der Basisklasse `System.Type` des Namensraumes `System` werden die Datentypen abgeleitet. Der Umgang damit ist normalerweise trivial, weil C# für alle wichtigen Datentypen Wrapper verwendet und selbst Schlüsselwörter zum Zugriff bereit stellt. Da Datentypen aber auch Objekte sind, habe sie natürlich Eigenschaften und Methoden. Manchmal ist es erforderlich, darauf zuzugreifen, um beispielsweise einen Zustand zu ermitteln. In diesem Buch werden Sie oft die Methode `ToString` sehen. Sie steht praktisch jedem Datentyp zur Verfügung steht.

Die Datentypen selbst (`String`, `Byte` usw.) sind Aufzählungen aus der Klasse `Type`.

An dieser Stellen noch ein kurzer Hinweis zur Schreibweise. Wenn Sie statt `String` das Wort `string` schreiben, wird Ihr Programm unverändert funktionieren. C# unterscheidet aber streng Groß- und Kleinschreibung. Tatsächlich gibt es nur den Datentyp `String` aus der Klasse `Type`. Das Schlüsselwort `string` in C# ist nur eine Art Wrapper[1] darauf.

2.9.1 Eigenschaften der Datentypen

An erster Stelle steht oft die Analyse, ob eine Variable einen bestimmten Basisdatentyp hat. Außerdem kann festgestellt werden, wie die Variable deklariert wurde:

1. Wrapper sind hier Kapselungen von Namen in anderen Namenssystemen; C# kapselt einige Funktionen des Frameworks durch eigene Schlüsselwörter.

- Basisdatentypen – `IsArray`, `IsClass`, `IsEnum`
- Parametertypen – `IsByRef`
- Zugriffsebene – `IsPrivate`, `IsPublic`, `IsSealed` usw.
- Werttypen – `IsValueType`

2.9.2 Methoden der Datentypen

`ToString` kennen Sie bereits, damit wird jeder Typ in eine Zeichenkette konvertiert. Andere wichtige Methoden sind `Equals` zur Feststellung der Gleichheit, `GetType` zur Ermittlung des Namens des Datentyps und `GetTypeArray` zur Ermittlung eines Arrays von Datentypen aus einem universellen Array.

```
<% @Page Language="C#" debug="true" %>
<script runat="server" language="C#">
void Page_load()
{
    Object[] aObject = new Object[3];
    aObject[0] = 123;
    aObject[1] = "Krause";
    aObject[2] = 13.04;
    Type[] aTypes = Type.GetTypeArray(aObject);
    ausgabe.InnerHtml = "Datentypen des Arrays:<br/>";
    for(int i = 0; i < aTypes.Length; i++)
    {
       ausgabe.InnerHtml += aTypes[i].FullName + "<br/>";
    }
}
</script>
<html>
    <head>
        <title>Datentypen</title>
    </head>
    <body>
        <h1>Datentypen</h1>
        <p id="ausgabe" runat="server"/>
    </body>
</html>
```

Listing 2.50: Ermittlung von Datentypen (type_get.aspx)

Dieses Skript ist weniger spektakulär als es aussieht. Sie werden nur selten in die Verlegenheit kommen, wirklich Datentypen festzustellen. Wenn es erforderlich ist, können Sie aber die Methoden leicht auf jede Variable anwenden. Denken Sie daran, dass alle Variablen von Datentypen des Frameworks abstammen und damit Objekte sind und deshalb

über diese Eigenschaften und Methoden verfügen – zusätzlich zu den kontextspezifischen.

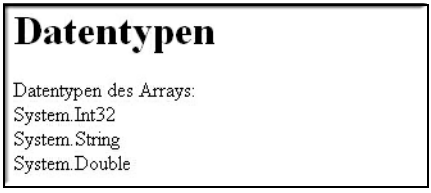


Abbildung 2.33: Ausgabe von Datentypen

2.10 Fragen und Übungsaufgaben

1. Erklären Sie den Unterschied zwischen Referenztypen und Werttypen.
2. Schreiben Sie ein kleines Programm, das den unteren Teil des ASCII-Zeichensatzes ausgibt (Zeichen 1 bis 127). Hinweise:
   - Verwenden Sie zuerst eine `for`-Schleife. Versuchen Sie dann, dasselbe Problem mit `while` zu lösen.
   - Für die Ausgabe verwenden Sie Konvertierungen des Typs `Char`.
3. Der folgende Code gibt die aktuelle Stunde als Zahl in der Variablen `stunde` zurück:

   ```
   DateTime heute = new DateTime();
   int stunde = heute.Hour;
   ```

   Schreiben Sie damit ein Programm, das passend zur Tageszeit eine nette Begrüßung erzeugt.
4. Erzeugen Sie ein Array, dass die Namen und Geburtsdaten (als Zeichenkette) einiger Freunde enthält. Geben Sie das Array dann komplett mit einer Schleifenanweisung aus.
5. Schreiben Sie eine statische Klasse, die eine Methode zur Berechnung des Logarithmus zu einer beliebige Basis enthält. Die Rechenvorschrift lautet: $\log_2$ (wert) / $\log_2$ (basis).
6. Schreiben Sie ein Programm, das die Würfelfunktion des Spieles Kniffel simuliert. Geworfen werden fünf Würfel. Geben Sie die Würfelergebnisse als Sternchen aus: *** für 3.

3 ASP.NET praktisch programmieren

lernen

Dieses Kapitel führt direkt in die Welt der ASP.Net-Programmierung ein. ASP.Net basiert zum einen auf der Nutzung spezielle Klassen des .Net-Frameworks und zum anderen auf der Behandlung von .aspx-Seiten mit einem speziellen Verarbeitungszyklus. Beides wird an Hand praktischer Beispiele erläutert.

3.1 Was Sie in diesem Kapitel lernen

Bislang ging es lediglich um Grundlagen wie Webserver und C#. Sie haben zwar schon einige ASP.NET-Programme ausprobiert, aber echte ASP-Programmierung war dies noch nicht. Auf die vielfältigen Prozesse während der Datenübertragung zwischen Browser und Server kann programmtechnisch zugegriffen und natürlich auch beeinflusst werden. Darum geht es in diesem Kapitel:

- *Grundlagen der Seitenverarbeitung* – vom Lesen des Codes bis zum Senden der Daten
- *Webserver-Objekte* – Zugriff auf den Datenstrom
- *Übertragung von Daten von Seite zu Seite mit der GET-Methode* – lernen Sie, wie Sie HTTP richtig nutzen.
- *Grundlagen der Formularverarbeitung*
- *Sitzungen und Applikationen*, Wiedererkennung von Benutzern
- *Cookies* programmieren
- *Strukturierung Ihrer Applikation durch hinterlegten Code*
- *Steuerung des Compilers durch Direktiven*

Die Einführung in Formulare ist nur ein erster Schritt in diese Richtung. Im nächsten Kapitel wird dieses umfassende Thema systematisch und ausführlich vorgestellt.

3.2 Grundlagen der Seitenverarbeitung

Dieser Abschnitt gibt einen grundlegenden Überblick über die Seitenverarbeitung in ASP.NET, den Aufbau von Applikationen und die Art der Verarbeitung von Code und Daten. Mit diesen Grundlagen ausgestattet, werden die wichtigsten Techniken in den nachfolgenden Abschnitten leicht verständlich. Vieles, was in diesem Teil beschrieben wird, ist gegenüber klassischem ASP völlig neu.

Voraussetzungen

Voraussetzung für alle folgenden Programme ist die Existenz eines virtuellen Verzeichnisses und die Bildung einer Applikation in diesem. Diese vorbereitenden Maßnahmen wurden bereits in Kapitel 1 gezeigt. Haben Sie noch keine Applikation angelegt, führen Sie die folgenden Schritte aus:

1. Öffnen Sie die Managementkonsole INTERNET-INFORMATIONSDIENSTE.
2. Suchen Sie im Verzeichnisbaum das virtuelle Verzeichnis (in Kapitel 1 wurde der Name *dotnet* vorgeschlagen).
3. Wechseln Sie zur Registerkarte VERZEICHNIS
4. Klicken Sie nun im unteren Teil auf ERSTELLEN. Lassen Sie die übrigen Optionen unverändert.
5. Klicken Sie auf ÜBERNEHMEN und dann auf OK.

Die folgende Abbildung zeigt den Dialog nach dem Anlegen der Applikation und mit dem entsprechenden Symbol im Verzeichnisbaum der Managementkonsole.

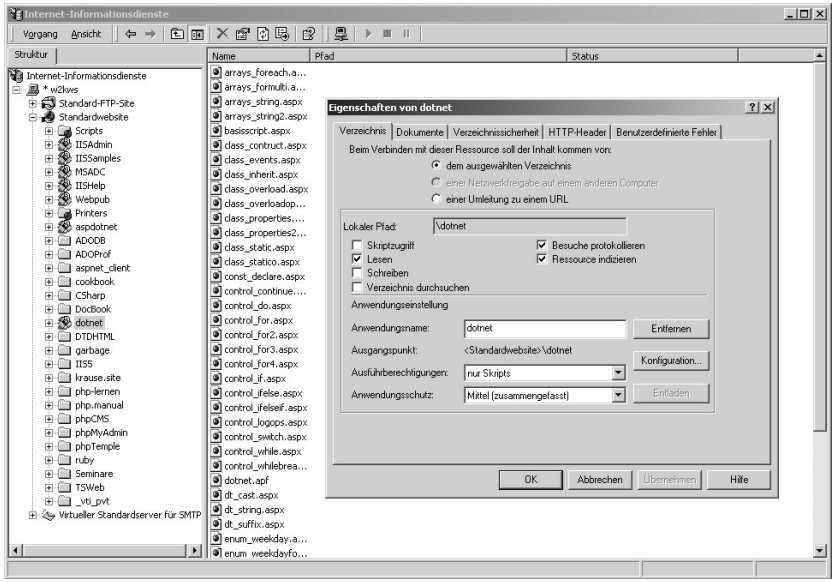


Abbildung 3.1: Anlegen einer Applikation für das Verzeichnis dotnet

3.2.1 Wie die Seite im Server verarbeitet wird

Für den korrekten Umgang mit ASP.NET-Applikationen müssen Sie einige Prinzipien der Seitenverarbeitung kennen. ASP.NET arbeitet ereignisorientiert. Das Prinzip der objektorientierten Programmierung gilt auch hier – ohne jede Ausnahme. Den Ablauf beim Laden und Ausführen einer Webseite auf dem Server kann man als Folge von Ereignissen betrachten. Sie können davon profitieren, da alle Ereignisse in entsprechend benannten Methoden ausgeführt werden.

Nichtsdestotrotz können Sie weiterhin Code einfach in die HTML-Seite einbetten. Dieser wird nach den Initalisierungsfunktionen fortlaufend entsprechend dem Auftreten in der Seite ausgeführt. Da die ASP.NET-Komponente aus der HTML-Seite erst eine reguläre Klasse erstellt – wenn Sie C# verwenden, natürlich darin –, wird der Code in der Reihenfolge der Erfassung abgelegt. Davon weichen spezielle Methoden ab, die durch Ereignisse zu bestimmten, genau definierten, Zeitpunkten aktiviert werden.

Page_Init
Page_Load
Page_Unload

Der Ablauf beginnt mit einer Methode `Page_Init`, die immer zuerst aufgerufen wird. Danach wird die Seite komplett analysiert und in das interne Klassenmodell übernommen. Sind alle Elemente analysiert, wird `Page_Load` ausgeführt. Diese Methode stellt also sicher, dass die HTML-Elemente alle erfasst wurden. Dann werden die Ereignisse ausgeführt, die Steuerelemente (engl. Controls) auf der Seite ausgelöst haben. Das tritt beispielsweise auf, wenn ein Benutzer ein Formular abgeschickt hat. Sind keine Steuerelemente vorhanden, wird dieser Schritt übergangen. Dann werden alle Codes der Seite ausgeführt, die nicht in irgendwelchen Methoden stehen, ausgenommen der Inhalt von `Page_Unload`. Wurde die Seite abgearbeitet und an den Browser gesendet, wird noch diese letzte Methode ausgeführt.

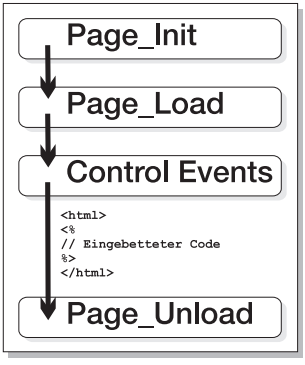


Abbildung 3.2: Ablauf der Codeausführung einer ASP.NET-Seite

PreRender
Error

Alle Ereignismethoden sind als `public void` zu kennzeichnen. Sie müssen aber nicht alle deklarieren, sondern nur die, die sie wirklich benötigen. Am häufigsten dürfte `Page_Load` zum Einsatz kommen. Damit stellen Sie sicher, dass alle Elemente der Seite bekannt sind. Neben diesen, in der normalen Programmierung häufig benötigten Ereignissen gibt es weitere, die nur unter bestimmten Bedingungen von Interesse sind, beispielsweise `PreRender` vor der Analyse der Seite oder `Error` beim Auftreten von Fehlern.

Die Tatsache, dass die Verarbeitung der HTML-Elemente auf der Seite von Bedeutung ist, deutet das grundlegende Prinzip an. ASP.NET sucht die Seite nach besonders gekennzeichneten Elementen ab und stellt diese im Code als Objekte zur Verfügung. Sie müssen also nicht mehr wie im alten ASP zwingend in die HTML-Seite schreiben, sondern können Code und Layout von vornherein weitgehend auseinander halten.

3.2.2 Einbettung von Code

Um es vorweg zu nehmen – die Einbettung von Code ist keine gute Idee. Es ist zwar bequem und für Programmierer, die bereits ASP-Erfahrung haben, auch gewohnt, aber es gibt in ASP.NET bessere, stringentere Techniken. Im vorhergehenden Kapitel über C# wurde diese Technik dennoch manchmal verwendet. Für derart kleine Codeschnipsel ist es durchaus legitim. Bei größeren Programmen sollten Sie versuchen, das zu vermeiden. Alle folgenden Beispiele verzichten zur Gewöhnung darauf ganz.

Der Vollständigkeit halber folgen die Methoden der Codeverarbeitung auf einen Blick. Code vor der HTML-Seite betten Sie in `<script>`-Tags ein:

```
<script language="C#" runat="server">
```

Innerhalb der HTML-Seite werden die alten ASP-Tags verwendet:

```
<% Methode(); %>
```

Hier kann auch die Kurzschreibweise zur Ausgabe von Variablen verwendet werden:

```
<% = variable; %>
```

Intern wird auch lediglich `Response.Write()` aufgerufen. `Response` ist ein Server-Objekt, dass in diesem Kapitel noch vorgestellt wird.

3.2.3 Spezielle Funktionen in ASP.NET

Code-Behind

Das ganze lässt sich perfektionieren, wenn der Code insgesamt ausgelagert wird. Diese Technik wird als Code-Behind bezeichnet. Dabei wird das jeweils pro Seite abzulegende C#-Programm als *cs*-Seite gespeichert. In der Webseite selbst wird lediglich ein Verweis untergebracht, der in der ersten Zeile steht. Ausführlich wird dies in „Hinterlegter Code: Code Behind“ auf Seite 179 beschrieben.

Seitendirektiven

Einige grundlegende Einstellungen zur Verarbeitung der Seite können an den Anfang gestellt werden. Die so genannten Seitendirektiven setzen beispielsweise die Vorauswahl der Programmiersprache oder die Cacheeinstellungen. Sie finden eine genaue Betrachtung in „Direktiven“ auf Seite 182.

3.3 Einführung in die Formularverarbeitung

Schwerpunkt der Entwicklung in ASP.NET bildet die Programmierung von Formularen. Gegenüber dem Vorgänger ASP 3.0 gibt es hier die größten Unterschiede und Fortschritte. Es ist wichtig, gute Formulare zu entwerfen, denn die Qualität der Benutzerschnittstelle entscheidet letztlich über den Erfolg einer Website.

3.3.1 Ablauf der Formularverarbeitung

In allen bisherigen Systemen für die Webserverprogrammierung waren die Vorgänge im Server und auf dem Client völlig voneinander getrennt. Ablauf und Darstellung entsprachen den Gegebenheiten der Protokolle. Der Client fordert mit einem HTTP-Request eine Ressource auf dem Server an, der Server erkennt dies, erstellt gegebenenfalls eine Seite auf Grundlage von Berechnungen und sendet diese an den Client. Dort wird die Seite dargestellt und auf Benutzereingaben gewartet. Der Programmierer ist nun für die Abbildung der Objekte der Website in seinem Datenmodell selbst verantwortlich. Betrachtet er HTML als losgelöste Sammlung von Zeichen, entsteht der von ASP und anderen Skriptsprachen bekannte zersplitterte Code dynamischer Webseiten. Sendet der Benutzer die Seite – beispielsweise ein ausgefülltes Anmeldeformular – wieder an den Server, muss der gesamte Status wieder durch aufwändige Programmierung ermittelt, die Daten analysiert und die Reaktionen durch Erzeugen von HTML-Code präsentiert werden.

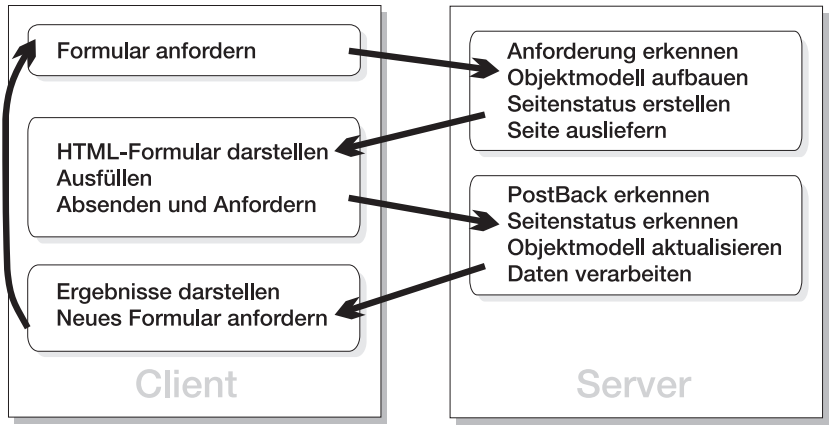


Abbildung 3.3: Ablauf der Formularverarbeitung zwischen Browser und Server

Das Spiel zwischen Client und Server hat sich nicht geändert. An den Umständen des Protokolls HTTP und den Möglichkeiten der Darstellung mit HTML kann auch ASP.NET nichts verbessern. Allerdings ist es den Entwicklern gelungen, ein raffiniertes Modell der Abbildung der Vorgänge und Elemente im Objektmodell des Programms zu entwerfen.

Server Steuerelemente

Basis dieses Modells sind die so genannten Server Steuerelemente (Server-Steuerelemente). Diese werden als spezielle Tags – im XML-Format – in die Webseite eingebettet. Jedes Steuerelement besteht aus zwei Teilen. Zum einen die Darstellung in HTML bzw. XML innerhalb des Seitencodes. Zum anderen als Objekt im Objektmodell des Programms. Genauer betrachtet ist es ein Objekt im Objektmodell der Klasse `Page`, die die aktuelle Website jeweils abbildet. Sie können also mit den Elementen im HTML-Code über Objekte in Beziehung treten. Dazu gehören ganz simple Aufgaben, beispielsweise die Gestaltung verändern, den dargestellten Text erzeugen oder den Status abfragen. Da bei einigen Elementen, wie eben die in Formularen verwendeten, wo der Benutzer in Interaktion mit dem Programm tritt, muss auch ein Zugriff auf diese Daten erfolgen. Im Hintergrund heißt dies, der in Abbildung 3.3 gezeigte Ablauf findet auch hier statt. ASP.NET sorgt nur intern für eine stringente Einbettung in das Objektmodell der Seite.

3.3.2 Status eines Formulars

Es wurde bereits angedeutet, das ASP.NET beim Umgang mit Formularen einiges vereinfachen kann. Wie sehr sich das auswirkt, zeigt das erste Beispiel:

```
<script language="c#" runat="server">
void Page_Load()
{

}
</script>
<html>
  <head><title>Formulare</title></head>
  <body>
  <h1>Formulare - Ein Test</h1>
  Tragen Sie einen Text ein:
  <form runat="server">
    <input type="Text" name="Content" id="Content" runat="server"/>
    <input type="Submit" value="Absenden"/>
  </form>
  </body>
</html>
```

Listing 3.1: Einfaches Formular ohne Programmcode (form_simplecontrol.aspx)

Auf den ersten Blick kann dieses Formular nicht viel tun. Im alten ASP würde es nicht einmal gesendet werden, weil im `<form>`-Tag das Attribut `action` fehlt. Tatsächlich kann es aber eine ganze Menge: Es wird an die aufrufende Seite zurück gesendet und behält dabei den Inhalt des Feldes. Das mehr an den Browser gesendet wurde, als im Listing zu sehen war, verrät ein Blick in den HTML-Quelltext:

```
<html>
  <head><title>Formulare</title></head>
  <body>
  <h1>Formulare - Ein Test</h1>
  Tragen Sie einen Text ein:
  <form name="_ctl0" method="post" action="form_simplecontrol.aspx"
id="_ctl0">
<input type="hidden" name="__VIEWSTATE" value="dDwxNzU2ODcxNTcyOzs+/
CJSAEmAbo0oaWsXVv53LOTiHi4=" />

    <input name="Content" id="Content" type="Text" value="xxfg" />
    <input type="Submit" value="Absenden"/>
  </form>
  </body>
</html>
```

Listing 3.2: Dasselbe Formular im Browser

Viewstate

Tatsächlich hat ASP.NET hier einiges verändert. Das `<form>`-Tag ist mit den nötigen Attributen versorgt worden. Außerdem wurde ein verstecktes Feld hinzugefügt, dass den so genannten Viewstate enthält. Damit

findet der Server den Zusammenhang zwischen dem vorher gesendeten Formular und dem vom Benutzer zurückgesendeten wieder. Dies ist notwendig, weil die Verbindung zwischen Client und Server nach der ersten Transaktion beendet wird.

runat="server"

Verantwortlich für das Auslösen dieses Verhaltens ist das Attribut `runat="server"`. Selbstverständlich sind diese in der HTML-Seite verschwunden, denn ASP.NET sendet nur reines HTML an den Browser. Alle Erweiterungen, die in irgendeiner Form programmiert werden, sind allein für den Aufbau des Objektmodells bestimmt.

An dieser Stelle ist natürlich interessant, wie das Formular im Objektmodell dargestellt wird. Bislang wurde progammtechnisch nicht eingegriffen.

3.3.3 Formulare im Objektmodell

Alle Elemente, die im HTML-Code als Server-Steuerelemente deklariert wurden (mit `runat="server"`) stehen im Programm als Objekt zur Verfügung. Der Zugriff erfolgt über den Namen. Der Name basiert auf der Angabe im Attribut `id`. Entsprechend kann auf die Eigenschaften zugegriffen werden.

```
<script language="c#" runat="server">
void Page_Load()
{
   Content.Value = "Programmatisch";
}
</script>
<html>
  <head><title>Formulare</title></head>
  <body>
  <h1>Formulare - Ein Test</h1>
  Tragen Sie einen Text ein:
  <form runat="server">
    <input type="Text" name="Content" id="Content" runat="server"/>
    <input type="Submit" value="Absenden"/>
  </form>
  </body>
</html>
```

Listing 3.3: Zugriff auf HTML-Objekte zur Laufzeit des Programms

Das Beispiel zeigt, wie die Eigenschaft `value` des Objekts `Content` per Programm gefüllt wird.

Abbildung 3.4: Das Formular mit vorausgefülltem Wert

IsPostBack

Leider funktioniert jetzt das Formular nicht mehr. Denn die Methode `Page_Load` wird immer ausgeführt und damit kann sich der Wert nicht mehr ändern. Der Zustand „Formular gesendet“ oder „Formular neu“ muss also unterschieden werden. Dafür gibt es eine spezielle Eigenschaft des Objekts `Page` – `IsPostBack`.

Diese ist `true`, wenn das Formular vom Client empfangen oder `false`, wenn es erst neu erzeugt und an den Client gesendet wurde. Dasselbe Programm noch einmal, diesmal füllt es das Feld aber korrekt aus und lässt sich vom Benutzer überschreiben:

```
<script language="c#" runat="server">
void Page_Load()
{
   if (!IsPostBack)
   {
      Content.Value = "Programmatisch";
   }
}
</script>
<html>
  <head><title>Formulare</title></head>
  <body>
  <h1>Formulare - Ein Test</h1>
  Tragen Sie einen Text ein:
  <form runat="server">
    <input type="Text" name="Content" id="Content" runat="server"/>
    <input type="Submit" value="Absenden"/>
  </form>
  </body>
</html>
```

Der einzige Unterschied ist die Verwendung der Abfrage des Status in der Methode `Page_Load`. Selbstverständlich verfügt das Objekt `Page` über eine ganze Reihe derartiger Eigenschaften, Ereignisse und Methoden. Die Eigenschaften können Sie der folgenden Tabelle entnehmen:

Eigenschaft	Bedeutung
Application	Erlaubt Zugriff auf das Applikationsobjekt. Dies wird in „Daten senden und empfangen“ auf Seite 144 behandelt.
Cache	Kontrolliert einen Zwischenspeicher mit Seiteninformationen.
ClientTarget	Hiermit werden browserabhängige Informationen gesteuert.
EnableViewState	Die Eigenschaft schaltet den Viewstate ein und aus.
ErrorPage	Definition einer Fehlerseite, die bei Laufzeitfehlern an den Browser gesendet wird.
IsPostBack	Diese Eigenschaft erkennt, ob das Formular vom Client gesendet wurde.
IsValid	Die Eigenschaft is `true`, wenn alle überwachten Elemente des Formulars korrekt ausgefüllt wurden.
Request	Zugriff auf das `Request`-Objekt.
Response	Zugriff auf das `Response`-Objekt
Server	Zugriff auf das `Server`-Objekt
Session	Zugriff auf das `Session`-Objekt
SmartNavigation	Hiermit wird eine besondere Form des Seitenaufbaus kontrolliert, die auf den Internet Explorer zugeschnitten ist.
Trace	Zugriff auf das `Trace`-Objekt
TraceEnabled	Schaltet Trace ein und aus.
User	Benutzerinformationen
Validators	Eine Kollektion der Elemente der Seite, bei denen die Eingabe überwacht wird.

Tabelle 3.1: Eigenschaften, Ereignisse und Klassen der Klasse `Page`

Die Eigenschaften decken insgesamt mehr ab, als für die reine Formularbehandlung notwendig ist. Informationen zu den Objekten `Session` und `Application` finden Sie in „Sitzungen und Applikationen“ auf Seite 160.

3.4 Standardobjekte und Programmierprinzipien

Gerade am Anfang bereitet es Schwierigkeiten zu verstehen, wo die Unterschiede zwischen den verschiedenen Arten serverseitigen Zugriffs auf HTML liegen. Dabei machen es die Bezeichnungen der Klassen, die den Elementen zugrundeliegen, nicht unbedingt einfacher.

3.4.1 Standardobjekte

Im Namensraum `System.Web` sind eine Vielzahl von Klassen zu finden, die der Programmierung von ASP.NET-Applikationen dienen. Wegen der vielfältigen Beziehungen untereinander und einiger Vereinfachungen, wie beispielsweise der automatischen Instanziierung, ist das System dahinter nicht sofort erkennbar. Prinzipiell müssen Sie sich die an der Abarbeitung der Applikation beteiligten Komponenten und Vorgänge vor Augen halten. Die folgende Abbildung nennt die wichtigsten Objekte und wie diese an der Erstellung der Seiten beteiligt sind.

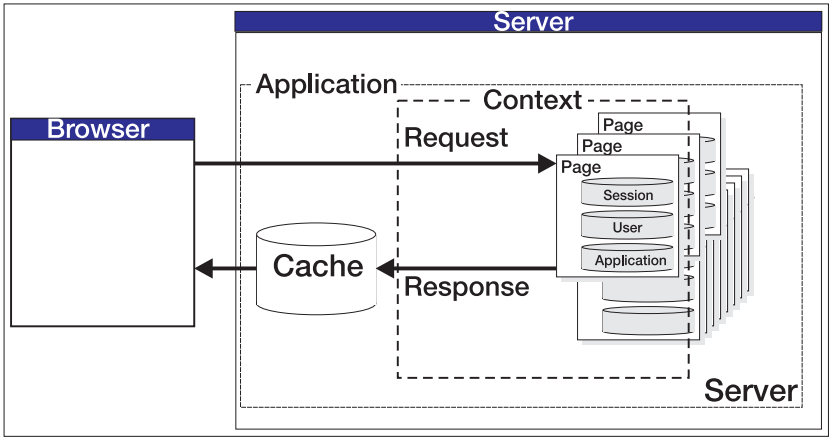


Abbildung 3.5: Zuständigkeit der Objekte in ASP.NET

In der Hierarchie ganz oben steht das Objekt `Application`. Es wird automatisch aus der Klasse `HttpApplication` instanziiert. Dieses Objekt enthält verschiedene Eigenschaften, die den Zugriff auf die anderen Objekte erlauben. Allerdings werden auch diese implizit instanziiert. Die wichtigsten, die auch in diesem Buch Verwendung finden, sind folgende:

- `Application` – Die Eigenschaft `Application` gibt selbst ein Objekt `Application` zurück, dass allerdings von der Klasse `HttpApplicationState` abgeleitet wird. Die Namensgleichheit mag irritierend sein, ASP.NET erkennt das korrekte Objekt aber am Kontext der Verwendung.
- `Session` – Mit diesem Objekt wird das Sitzungsmanagement gesteuert.
- `Request` – Hiermit erhalten Sie Zugriff auf alle Informationen, die im Zusammenhang mit der Anforderung vom Browser stehen, beispielsweise den Querystring (URL-Parameter) oder Formularinhalte.
- `Response` – Dieses Objekt dient der Kontrolle des Ausgabedatenstromes. Unbewusst haben Sie schon `Response.Write` verwendet, eine Methode zum Ausgeben von Daten an den Browser.

- `Cache` – Mit diesem Objekt wird die Zwischenspeicherung von Daten kontrolliert.
- `Server` – Dieses Objekt stellt den Zugriff auf Serverfunktionen sicher.
- `Page` – Das Objekt der aktuell angeforderten (aktiven) Seite

Das Objekt `Application` wird einmal beim Start der Applikation instanziiert. Es steht dann die gesamte Laufzeit über zur Verfügung. Bei der Gestaltung von Web-Applikationen sollten Sie daran denken, dass diese zwar prinzipiell aus einzelnen Seiten bestehen. Jeder Benutzer löst im ersten Schritt eine Anforderung aus (Request). Es entsteht das `Request`-Objekt. Normalerweise wird eine *aspx*-Seite aufgerufen. Damit entsteht ein `Page`-Objekt. Wenn auf einer Seite weitere Elemente wie Bilder untergebracht sind, löst der Browser nach dem Empfang der Antwort (Response) weitere Anforderungen aus. Diese führen aber nicht zwingend zu weiteren oder erneuten Seitenaufrufen. Andererseits kann ein HTML-Frameset zum Abruf mehrerer Seiten führen, ebenso wie der Nutzer beim Surfen weitere Anforderungen auslöst. Der Benutzer selbst wird – unabhängig von der Anzahl der Anforderungen – innerhalb einer Sitzung als Einheit betrachtet. Es existiert für ihn also nur ein `Session`-Objekt. Alle Seiten mit externem Code zusammen bilden wiederum eine Applikation, die ihrerseits spezifische Eigenschaften hat, die die Art der Verarbeitung kontrollieren. Dazu gehört auch die Arbeitsweise des Zwischenspeichers des Ausgabedatenstromes, repräsentiert im `Cache`-Objekt. Noch globaler ist das `Server`-Objekt zu betrachten, das sehr allgemeine Zustände erreichbar werden lässt. Dazu gehört beispielsweise die Fehlerverwaltung und die Bereitstellung von Hilfsfunktionen zur Kodierung oder Dekodierung.

Weitere Klassen

Aus den `Response`- und `Request`-Objekten können weitere abgeleitet werden, die wiederum durch eigene Klassen ergänzt werden. Dazu gehört die Behandlung von Cookies, die eine Klasse zur Erstellung verwenden, die die Darstellung jedes Cookies als Objekt erlaubt und die entsprechende Kollektion im Objekte `Response`, die den Zugriff auf zu sendende Cookies erlaubt bzw. `Request`, wo die empfangenen Cookies als Kollektion bereitgestellt werden.

3.4.2 Steuerelemente der Formulare: Web Forms

„Web Forms“ ist ein Oberbegriff für die Gestaltung von Formularen. In der ASP.NET-Programmierung spielen Formulare eine zentrale Rolle. Web Forms entstehen, in dem auf einer ASP.NET-Seite ein Formular verwendet wird. Alternativ zur *aspx*-Seite kann auch ein nutzerdefiniertes Steuerelement entworfen werden, gekennzeichnet durch die Erweite-

rung *ascx*. Diese Elemente lassen sich als komplexe, vorbereitete Sammlungen von HTML-Elementen verstehen.

HTML Server-Steuerelemente

Der programmatische Zugriff auf HTML ist am einfachsten und in vielen Fällen auch ausreichend beim Aufbau von Formularen. Die Abbildung im Framework erfolgt mit den *HTML Server-Steuerelementen* (HTML Server Controls). Praktisch gibt es für jedes Steuerelement dieser Klasse eine direkte und eindeutige Entsprechung in HTML. HTML-Elemente werden programmtechnisch verfügbar gemacht, in dem sie mit dem Attribut `runat="server"` versehen werden. HTML Server-Steuerelemente werden ausführlich im Abschnitt „Was Sie in diesem Kapitel lernen“ auf Seite 191 behandelt.

Web Server-Steuerelemente

Sowohl den Zugriff auf einzelne HTML-Tags als auch auf Sammlungen mehrerer Elemente erlauben die *Web Server-Steuerelemente* (Web Server Controls). Interessant sind diese Steuerelemente, weil sie eine sehr einfache programmtechnische Verwaltung in einem Objekt auch dann erlauben, wenn zur Darstellung viele HTML-Tags notwendig sind. Einige sind aber auch nur für ein Element zuständig, sodass sich Überschneidungen mit den einfacheren HTML Server-Steuerelemente ergeben. Sie können eigene Web Server-Steuerelemente entwerfen – dies sind die Benutzer-Steuerelemente (User Controls). Web Server-Steuerelemente werden im Abschnitt „Web Server-Steuerelemente (Web Server Controls)“ auf Seite 217 behandelt.

Kontroll-Steuerelemente

Sollen die Eingaben der Benutzer vom Programm analysiert werden, bieten sich die *Kontroll-Steuerelemente* (Validation Controls) an. Diese erlauben sowohl eine client- als auch serverseitige Kontrolle. Für den Einsatz im Browser liefert ASP.NET browserunabhängige JavaScript-Bibliotheken mit. Kontroll-Steuerelemente werden ausführlich im Abschnitt „Kontroll-Steuerelemente (Validation Controls)“ auf Seite 240 behandelt.

Benutzer-Steuerelemente

Benutzer-Steuerelemente (User Controls) erleichtern den Entwurf modularisierter Webformulare. Sie können häufig benutzte HTML-Elemente zusammenfassen und mehrfach in Seiten einbinden. Die interne Darstellung als Objekt erleichtert den Zugriff vom Programm aus. Eine Anwendung ist auch der Entwurf von Bibliotheken mit Steuerelementen für spezielle Zwecke, beispielsweise mobile Clients. Benutzer-Steuerelemente werden im Abschnitt „HTML Server-Steuerelemente (HTML Controls)“ auf Seite 192 kurz vorgestellt.

3.5 Daten senden und empfangen

Von zentraler Bedeutung bei der ASP.NET-Programmierung ist die Kontrolle des Sendens und des Empfangens von Informationen.

3.5.1 Den Datenfluss steuern

Der Datenaustausch zwischen Browser und Webserver besteht aus den Prozessen Anforderung (Request) und Antwort (Response). Wann immer Daten übertragen werden, erlauben die gleichnamigen Objekte den Zugriff über Methoden und Eigenschaften. Beide Objekte sind bereits instanziiert und bilden immer die aktuelle Anforderung bzw. Antwort ab, sodass nur eine einzige Instanz nötig und möglich ist. Ergänzend ist auch das Objekt `Server` von Interesse, dass Hilfsfunktionen enthält.

Antwortkontrolle und Datenausgabe

Die Klasse `HttpApplication` stellt eine Eigenschaft `Response` bereit, die ein Objekt der Klasse `HttpResponse` instanziiert. Dieses steht unter dem Namen `Response` zur Verfügung. Der Vorgang weicht vom üblichen Vorgehen der Ableitung von Objekten ab, damit zum einen die Programmierung vereinfacht wird, zum anderen aber auch eine wenigstens teilweise Kompatibilität zum alten ASP 3.0 erreicht wird.

Response.Write
Response.WriteFile

Die erste Methode des Objekts, `Write`, haben Sie bereits verwendet. Damit wird ein Text an den Browser ausgegeben. Statt Text kann auch der Inhalt einer Datei ausgegeben werden. Dazu wird `WriteFile` eingesetzt. Diese Methode besitzt zwei optionale Parameter, die Startpunkt und Länge des Textes angeben, gezählt in Byte aus der Datei.

```
<script language="C#" runat="Server">
string footer = "footer.html";
</script>
<html>
    <head>
        <title>Response.WriteFile</title>
    </head>
    <body>
        <hr/>
        <%
        Response.WriteFile(footer);
        %>
    </body>
</html>
```

Listing 3.4: Ausgabe einer Datei am Ende einer Seite (response.writefile.aspx)

Die Ausgabe an den Browser kann prinzipiell mit zwei Methoden erfolgen. Zum einen wird jeder Text, der erzeugt wird, sofort gesendet. Der Browser stellt diesen meist fortlaufend dar.

Response.BufferOutput

Der Effekt kann verdeckt ablaufen, wenn HTML-Tabellen verwendet werden oder der Aufbau der Seite nicht klar ist, bis ein bestimmtes Element auftritt. Hier spielt also auch die Gestaltung eine Rolle. Für einfachen Text gilt aber, dass gesendete Daten auch sofort erscheinen.

Das Verhalten kann modifiziert werden. Sie können auch einen Pufferspeicher verwenden, der alle Ausgaben sammelt. Erst wenn die Seite vollständig abgearbeitet wurde, werden alle Daten gesendet.

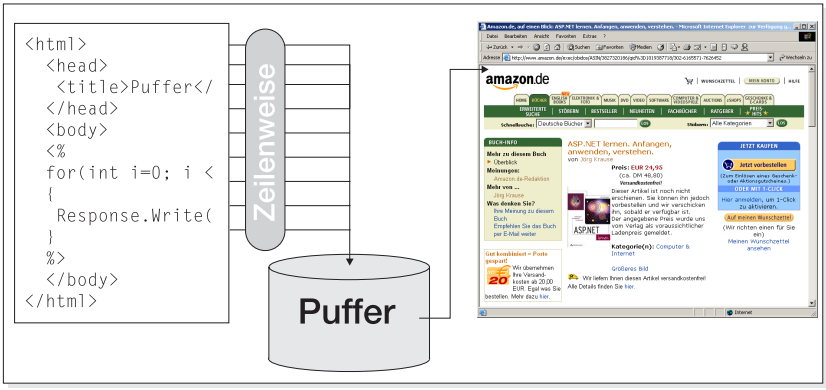


Abbildung 3.6: Arbeitsweise des Puffers: Zeilenweise Ausgaben werden gesammelt und am Ende als Block gesendet

Dies ist die Standardeinstellung – ASP.NET verwendet eine Ausgabepufferung. Das folgende Beispiel zeigt, wie die Pufferung abgeschaltet werden kann:

```
<script language="C#" runat="Server">
void Page_Init()
{
    Response.BufferOutput = false;
}
</script>
<html>
    <head>
        <title>Response.Write</title>
    </head>
    <body>
        Ausgabe ohne Ausgabepuffer:
        <br/>
        <%
        for (int i = 0; i < 10; i++)
```

```
        {
            Response.Write (i.ToString());
            for (int j = 0; j < 10; j++)
            {
                Response.Write ('.');
            }
            Response.Write ("<br/>");
        }
        %>
    </body>
</html>
```

Listing 3.5: Ausgabe ohne Puffer (response.write.aspx)

Ausgabepuffer

Die Einstellung erfolgt über die Eigenschaft `BufferOutput`. Übergeben Sie `true`, um die Pufferung einzuschalten oder `false`, um sie nicht zu verwenden. Die beiden Schleifen geben viele Punkte aus, sodass der Effekt gut beobachtet werden kann. Möglicherweise erscheinen die ersten Zeilen nicht punkteweise, sondern als Block. Das liegt nicht an Fehlern im Puffer, sondern daran, dass der Browser am Anfang mit der Darstellung nicht nachkommt.

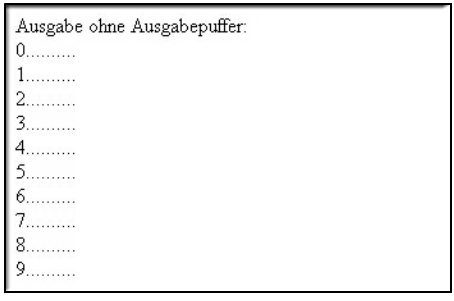


Abbildung 3.7: Ohne Ausgabepuffer erscheinen die Punkte nacheinander

Tatsächlich ist die Ausgabe spürbar langsam, wenn man bedenkt, dass die innere Schleife wenig Code enthält, der verzögernd wirken könnte. Der Verzicht auf den Ausgabepuffer ist außerordentlich uneffektiv. Aber es gibt sinnvolle Anwendungen. Wenn beispielsweise eine Datenbankabfrage längere Zeit dauert, kann eine gelegentliche Ausgabe von Zeichen an den Browser für den Benutzer hilfreich sein und vom Anklicken der Stopp-Schaltfläche abhalten.

Zeitüberschreitung

Derart lange Aktionen können öfter auftreten, wenn komplexe Abfragen ausgeführt werden. Die ASP.NET-Komponente übernimmt vom IIS den Timeout-Wert von 90 Sekunden für die Abarbeitung einer Seite. Wird innerhalb dieser Zeit das Programm nicht fertiggestellt, wird der Prozess recycled und der Browser zeigt eine Fehlermeldung an. Der Wert

kann natürlich programmatisch verändert werden. Da diese Einstellung für den gesamten Serverprozess gilt, wird das `Objekt Server` eingesetzt. `Server` steht ebenfalls automatisch und als einzige Instanz der Klasse `HttpServerUtility` bereit. Das folgende Programm setzt eine kurze Timeout-Zeit und provoziert die Zeitüberschreitung durch eine Endlosschleife.

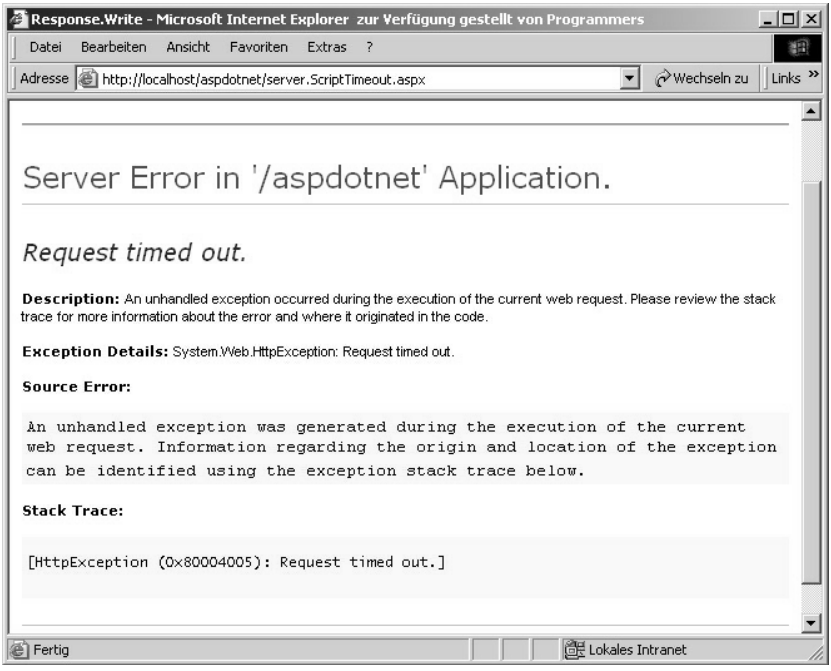


Abbildung 3.8: Fehler durch eine Zeitüberschreitung beim Programmablauf

```
<script language="C#" runat="Server">
void Page_init()
{
    Server.ScriptTimeout = 4;
}
</script>
<html>
    <head>
        <title>Response.Write</title>
    </head>
    <body>
        Ausgabe ohne Ausgabepuffer:
        <br/>
        <%
        do
        {
```

```
        } while (true);
        %>
    </body>
</html>
```

Listing 3.6: Abbruch einer Endlosschleife durch Timeout (server.ScriptTimeout.aspx)

Wenn Sie das Programm ausprobieren, wird nach Ablauf der Zeit ein Fehlerereignis erscheinen.

Es ist sicher sinnvoll, auf derartige Fehler reagieren zu können. Verwenden Sie `try`- `catch`-Anweisungen dafür.

3.5.2 Daten von Seite zu Seite übertragen

Besteht Ihre Applikation aus mehreren Seiten, müssen oft Daten zwischen diesen ausgetauscht werden. Dafür gibt es mehrere Wege:

- Cookies
- Sessions (Sitzungen)
- HTTP-Methode POST
- HTTP-Methode GET

Die Standardmethoden

Cookies werden in „Cookies“ auf Seite 173 behandelt. Die zu speichernden Daten werden dabei an den Browser übertragen und von diesem wieder zurückgesendet. `Session`-Objekte speichern Variablen intern und nutzen den Sitzungsstatus, um dem Benutzer seine Werte zuzuordnen. Ausführlich wird dies in „Sitzungen und Applikationen“ auf Seite 160 diskutiert. Die Standardmethoden in der Webserverprogrammierung sind jedoch POST und GET. POST ist das Kommando, das der Browser nutzt, um Formulardaten zu senden. Dies wird ausführlich in Kapitel 4 vorgestellt. Wenn nur mit Hyperlinks gearbeitet wird, verwendet der Browser zur Anforderung von Ressourcen das Kommando GET.

Daten per URL übertragen

Die ganze Technik beruht darauf, die zu übertragenden Daten in einer ganz bestimmten Art und Weise an den URL anzuhängen:

```
http://www.seite.de/target.aspx?vari1=Wert1&vari2=Wert2
```

Dieser Link überträgt zwei Variablen: `vari1` und `vari2`, die die Zeichenketten „Wert1“ bzw. „Wert2“ enthalten. Auf der Seite *target.aspx* werden diese im `Request`-Objekt bereit gestellt.

Drei spezielle Zeichen finden dabei Verwendung: Das Trennungszeichen zwischen der URL und den angehängten Parametern ist das Fragezeichen. Jedes einzelne Wertepaar wird mit einem &-Zeichen getrennt. Zwischen Variable und Wert steht ein Gleichheitszeichen. Sie können theoretisch beliebig viele Werte übertragen. Beachten Sie aber, dass die Browser ganz unterschiedliche Längen für die komplette URL akzeptieren. Der Internet Explorer akzeptiert ca. 2.000 Zeichen. Die gesamte Parameterschlange nach dem Namen der Seite wird als „Querystring" bezeichnet.

Request.QueryString

Die Auswertung wird entsprechend mit der Methode `Request.QueryString` vorgenommen. Diese Methode extrahiert die einzelnen Werte aus dem URL. Schauen Sie sich das folgende Beispiel an, das zur Bestellung von Büchern dient:

```
<script language="C#" runat="server">
void Page_Load()
{
    string self = Request.ServerVariables["SCRIPT_NAME"];
    link1.InnerHtml = "Der Unbesiegbare";
    link1.HRef = self + "?book=1";
    link2.InnerHtml = "Der Schnupfen";
    link2.HRef = self + "?book=2";
    link3.InnerHtml = "Sterntageb&uuml;cher";
    link3.HRef = self + "?book=3";
    link4.InnerHtml = "Eden";
    link4.HRef = self + "?book=4";
    int booknumber = Convert.ToInt32(Request.QueryString["book"]);
    result.Text = "Ihre Bestellung: ";
    switch (booknumber)
    {
        case 1:
            result.Text += link1.InnerText;
            break;
        case 2:
            result.Text += link2.InnerText;
            break;
        case 3:
            result.Text += link3.InnerText;
            break;
        case 4:
            result.Text += link4.InnerText;
            break;
        default:
            result.Text = "Sie haben noch keine Wahl getroffen";
            break;
    }
}
```

```
</script>
<html lang="en">
    <head>
        <title>Request.QueryString</title>
    </head>
    <body>
        <h2>Willkommen in unserem Buchladen</h2>
        <b>Ihre Bestellung bitte:</b><br/>
        <a id="link1" runat="server"/><br/>
        <a id="link2" runat="server"/><br/>
        <a id="link3" runat="server"/><br/>
        <a id="link4" runat="server"/><br/>
        <br/>
        <asp:label id="result" runat="server"/>
    </body>
</html>
```

Listing 3.7: Übertragung von Daten per GET (request.querystring.aspx)

Wie es funktioniert

Das Programm nutzt serverseitige Steuerelemente, die im Detail in Kapitel 4 diskutiert werden. Basis der Seite bilden die Hyperlinks, die mit `<a>`-Tags erzeugt werden:

```
<a id="link1" runat="server"/>
```

Dies unterscheidet sich insofern von HTML, als dass hier der notwendige Parameter `href` und der Inhalt des Tags dynamisch in der Methode `Page_Load` erzeugt werden. An den Browser wird folgende Zeile gesendet:

```
<a href="/aspdotnet/request.querystring.aspx?book=1" id="link1">
  Der Unbesiegbare
</a>
```

Erzeugt wird diese am Anfang der Methode `Page_Load`. Dort wird zunächst der Name der Seite ermittelt:

```
string self = Request.ServerVariables["SCRIPT_NAME"];
```

Dann wird der Text zugewiesen, der als Link erscheinen soll:

```
link1.InnerHtml = "Der Unbesiegbare";
```

Der entscheidende Punkt ist das Anhängen der GET-Parameter, die später ausgewertet werden sollen:

```
link1.HRef = self + "?book=1";
```

Nach dem sich die Seite selbst aufruft, müssen die übertragenen Parameter noch ausgewertet werden. Dazu wird die Buchnummer aus der Kollektion `QueryString` ermittelt. Die Umwandlung in eine Zahl erleich-

tert die Weiterverarbeitung. Welche Konvertierung Sie vornehmen, hängt von Ihren Daten ab. Standardmäßig werden die Informationen als Zeichenkette bereitgestellt:

```
int booknumber = Convert.ToInt32(Request.QueryString["book"]);
```

Der `switch`-Zweig sorgt dann dafür, dass das Ergebnis an das Element `<asp:label/>` übertragen wird. Die folgende Abbildung zeigt das Ergebnis nach einem Klick auf einen Link. Beachten Sie die Anzeige der Parameter in der Statuszeile des Browsers:

Abbildung 3.9: Arbeitsweise des Beispiels

Solange Ziffern übertragen werden, bereitet der Umgang mit dem Querystring wenig Probleme. Wenn Sie beliebige Werte übertragen möchten, muss jeder einzelne Wert in die URL-Form gebracht werden. ASP.NET übernimmt das für Sie automatisch. Werden die Buchnamen direkt übermittelt, ist dies gut zu beaobachten, wie der folgende Ausschnitt des URL zeigt:

Abbildung 3.10: Kodierung von Sonderzeichen im URL

Neben den Leerzeichen werden vor allem die Zeichen `&` und `=` kodiert, da diese als Parametertrennzeichen benötigt werden. Die Darstellung erfolgt in der Form `%HH`, wobei `HH` der hexadezimale Code des Zeichens ist.

3.5.3 Header erzeugen und analysieren

Response.AppendHeader

Normalerweise sorgt ASP.NET für die Erzeugung der richtigen Header. Es gibt jedoch Fälle, wo Sie eigene Header erzeugen müssen. Voraussetzung ist immer, dass Header noch nicht gesendet wurden. Deshalb ist die Nutzung nur in der Methode `Page_Init` sinnvoll. Hier ist sichergestellt, dass die Seite selbst noch nicht analysiert und gesendet wurde. Die Header werden gesendet, wenn das erste Zeichen des Body der Antwort ausgegeben wird. Header werden mit der Methode `Response.AppendHeader` der Liste der Header hinzugefügt. Die Methode `Response.ClearHeaders` löscht

alle bereits erfassten Header. Es wird ein Fehler generiert, wenn die Header bereits gesendet wurden.

Zwei wichtige Header werden häufiger eingesetzt: *Location* und *Contenttype*. Beide müssen nicht von Hand erzeugt werden; dafür stehen spezielle Eigenschaften des Objekts `Response` zur Verfügung. Mit Location steuern Sie Weiterleitungen, dies wird im Abschnitt „Weiterleitungen“ auf Seite 155 erläutert.

3.5.4 Servervariablen

Request.ServerVariables

Der Browser übermittelt mit jeder Anforderung einige Informationen an den Server, die in Programmen sinnvoll eingesetzt werden können. Zusammen mit Statusinformationen des Servers ergibt sich eine Sammlung aktueller Daten über die aktuelle Verbindung. Das Objekt `Request` stellt diese Information über die Eigenschaft `ServerVariables` zur Verfügung.

Wichtige Servervariablen

Nicht alle Werte werden für die praktische Programmierarbeit wirklich benötigt. Die wichtigsten sind:

- `HTTP_REFERER` – Wenn Ihre Seite durch Anklicken eines Hyperlinks auf einer anderen Seite erreicht wurde, enthält diese Variable die URL der Seite, von welcher der Nutzer kam
- `HTTP_USER_AGENT` – Der Typ des Browsers wird angezeigt. Sie können auswerten, welche Browser Ihre Nutzer bevorzugen, und die Gestaltung der Seiten daran ausrichten.
- `REMOTE_ADDR` – Dieses Feld enthält die IP-Adresse, mit der der Browser die Verbindung hergestellt hat.
- `QUERY_STRING` – Diese Variable enthält die Zeichenkette nach dem Fragezeichen, dem Trennzeichen für die Übertragung von Parametern zum Server.
- `SCRIPT_NAME` – Der virtuelle (relative) Pfad der aktuellen ASP.NET-Seite. Damit können Sie Seiten automatisch mit sich selbst referenzieren, ohne den Speicherort zu kennen.
- `SERVER_NAME` – Der Name des Webservers oder die IP-Adresse.
- `PATH_TRANSLATED` – Der physische Pfad der ASP.NET-Seite auf der Festplatte des Webservers.

Wie der Zugriff auf die gesamte Kollektion erfolgt, zeigt das folgende Beispiel:

```
<%
NameValueCollection sv;
sv  = Request.ServerVariables;
foreach (string element in sv)
```

```
{
   Response.Write ("<b>" + element + "</b> : ");
   Response.Write (sv[element] + "<br/>");
}
%>
```

Listing 3.8: Auslesen aller Servervariablen (request.servervariables.aspx)

Die Antwort bietet wertvolle Informationen über die aktuelle Verbindung:

LOCAL_ADDR : 127.0.0.1
PATH_INFO : /aspdotnet/request.servervariables.aspx
PATH_TRANSLATED : C:\Inetpub\wwwroot\dotnet\request.servervariables.aspx
QUERY_STRING :
REMOTE_ADDR : 127.0.0.1
REMOTE_HOST : 127.0.0.1
REQUEST_METHOD : GET
SCRIPT_NAME : /aspdotnet/request.servervariables.aspx
SERVER_NAME : localhost
SERVER_PORT : 80
SERVER_PORT_SECURE : 0
SERVER_PROTOCOL : HTTP/1.1
SERVER_SOFTWARE : Microsoft-IIS/5.0
URL : /aspdotnet/request.servervariables.aspx
HTTP_CONNECTION : Keep-Alive
HTTP_ACCEPT : */*
HTTP_ACCEPT_ENCODING : gzip, deflate
HTTP_ACCEPT_LANGUAGE : de,en-us;q=0.8,en-au;q=0.5,en-gb;q=0.3
HTTP_AUTHORIZATION : Basic QWRtaW5pc3RyYXRvcjpjbGVtZW5z
HTTP_COOKIE : ASP.NET_SessionId=33dwkwumswsoid2wxabq1j45
HTTP_HOST : localhost
HTTP_REFERER : http://localhost/aspdotnet/
HTTP_USER_AGENT : Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705)

Abbildung 3.11: Ausschnitt aus der Liste der Servervariablen

In der Praxis wird die vollständige Liste nur selten benötigt. Das folgende Programm zeigt, wie eine ganz bestimmte Information beschafft werden kann. Es zeigt eine Version des Sperrens einer Seite vor unbefugten Nutzern:

```
<script language="C#" runat="Server">
void Page_Load()
{
    string referer = Request.ServerVariables["HTTP_REFERER"];
    if (referer == "http://localhost/aspdotnet/request.servervariable.aspx")
    {
        message.Text = "OK, du bist hier richtig";
    } else {
        message.Text = "Leider falsch. Du kamst von: " + referer;
    }
}
</script>
```

```
<html>
    <head>
        <title>Request.Servervariables</title>
    </head>
    <body>
        <asp:label id="message" runat="Server"/>
        <hr/>
        <a href="request.servervariable.aspx">Restart</a>
    </body>
</html>
```

Listing 3.9: Schutz einer Seite durch Feststellung der Herkunft des hinführenden Links

Wie es funktioniert

Hier wird zuerst die Herkunft abgefragt, indem die Servervariable `HTTP_REFERER` ermittelt wird:

```
string referer = Request.ServerVariables["HTTP_REFERER"];
```

Dann wird der erwartete Wert mit dem tatsächlichen verglichen und eine Nachricht auf der Seite erzeugt. Wenn Sie das Programm aus der Liste aller Programme heraus starten, ist die Herkunft natürlich nicht die Seite selbst. Es wird die Fehlermeldung angezeigt. Klicken Sie auf den Link, ruft sich die Seite selbst auf und die Bedingung stimmt. Entsprechend ändert sich die Meldung.

3.5.5 Inhaltstyp bestimmen

Neben der Datenübertragung stellt sich auch die Frage, was eigentlich übertragen wird. Browser stellen nicht nur HTML, sondern auch die im Quelltext angegebenen Bilder, Plug-Ins und Applets dar oder dienen dem Herunterladen von Word- oder Exceldateien bis hin zu ZIP-Archiven.

Standard für den Inhalt der Seite: MIME

Response.ContentType

Der Content-Header enthält Informationen darüber, welche Art Datei zu erwarten ist. Die Kodierung erfolgt nach dem MIME-Standard. Mögliche Angaben sind „text/HTML“, „image/GIF“, „application/msword“ usw. Mit der Eigenschaft `ContentType` können Sie die Werte setzen. Der häufigste Wert ist vermutlich „text/HTML“, der für jede Übertragung einer ASP.NET- oder HTML-Datei eingesetzt wird. Dieser Header muss nicht erzeugt werden, dies erledigt ASP.NET für Sie.

Sie können aber auch einfachen Text übertragen. Wenn Sie beispielsweise ein Lernprogramm über HTML schreiben, dann kann es unter Umständen sinnvoll sein, einen Quelltext im Browser des Studenten

anzuzeigen und ihn anschließend zur Ausführung zu bringen. Aber wie wird dem Browser HTML abgewöhnt? Setzen Sie dazu einfach den Header *Content-type* auf Text, wie es im folgenden Beispiel gezeigt wird.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<script language="C#" runat="Server">
void Page_Init()
{
    Response.ContentType = "text/plain";
}
</script>
<b>
<%
Response.Write("Dieser Text ist ein Test.");
%>
</b>
```

Listing 3.10: Ausgabe von Text im Browser forcieren (response.contenttype.aspx)

Die Ausgabe zeigt, dass der Internet Explorer sich von der Angabe überzeugen ließ und die Seite nicht als HTML interpretiert. Allerdings gibt es einen zweiten Erkennungsmechanismus, der auf die umschließenden `<html>`-Tags reagiert (die hier bewusst weggelassen wurden). Insofern ist der Internet Explorer nicht völlig von diesem Header abhängig. Testen Sie Ihre Applikation mit mehreren Browsern, um die Reaktion kennenzulernen.

Die folgende Abbildung zeigt den Effekt:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<b>
Dieser Text ist ein Test.
</b>
```

Abbildung 3.12: Ausgabe einer HTML-Seite als einfachen Text

3.5.6 Weiterleitungen

Es gibt Situationen, in denen eine Seite beendet wird und die nächste oder eine bestimmte Seite automatisch erreicht werden soll. Die Methode `Redirect` des Objekts `Response` löst dieses Problem. Intern wird ein Header mit dem Namen `Location` erzeugt.

Weiterleitungen intern

Die Methode `Redirect` benutzt einen speziellen Statuscode des Webservers, den Code „302 Object Moved“. Jede Anforderung eines Browsers wird mit einem bestimmten Statuscode beantwortet. Der Browser entscheidet dann, wie damit zu verfahren ist. Wenn die `Response.Redirect`-Methode aufgerufen wird, sendet der Webserver zuerst die Antwort

„302 Object Moved“ an den Browser. Der neue URL wird mitgeliefert; normalerweise sollten alle Browser diesen dann ansteuern. Um zu verstehen, wie `Redirect` funktioniert, lesen Sie die folgenden zwei Zeilen Code, die exakt die Funktionsweise wiedergeben:

```
Response.Status = "302 Object Moved"
Response.AppendHeader = "Location", "http://www.neuerserver.de/ziel.aspx"
```

Response.Redirect

Besser ist natürlich die folgende Schreibweise:

```
Response.Redirect
```

Das folgende Beispiel besteht aus zwei Listings. Das erste zeigt die Seite, die nur mit einem speziellen Namen erreicht werden kann. Dieser muss zuvor in einem Formular eingegeben werden. Ist das nicht erfolgt, wird eine Weiterleitung eingesetzt, um zu diesem Formular zu gelangen:

```
<script language="C#" runat="Server">
void Page_Init()
{
    if (Request.Form["name"] != "Demo")
    {
        Response.Redirect("register.aspx");
    }
}
</script>
<html>
  <head>
    <title>Response.Redirect</title>
  </head>
  <body>
    Die Anmeldung wurde korrekt ausgef&uuml;hrt.
  </body>
</html>
```

Listing 3.11: Einsatz einer Weiterleitung (response.redirect.aspx)

Das Ziel besteht nur aus HTML – ein einfaches Formular:

```
<html>
  <head>
    <title>Response.Redirect</title>
  </head>
  <body>
    Ihre Anmeldung bitte;<br/>
    <form method="post" action="response.redirect.aspx">
```

```
        Ihr Name: <input type="text" name="name" />
        <input type="Submit" value="Anmelden" />
    </form>
  </body>
</html>
```

Listing 3.12: Formular zur Erfassung des Anmeldenamens

Obwohl als Ziel hier mit `action="response.redirect.aspx"` eindeutig auf die zuvor gezeigte Seite verwiesen wird, bleibt das Formular sichtbar, bis der Name korrekt eingegeben wurde (im Beispiel „Demo“). Tatsächlich springt der Browser mittels `Redirect` zwischen den beiden Seiten hin und her.

3.5.7 Übergabe der Programmsteuerung

Die im letzten Abschnitt gezeigte Technik mit *Response.Redirect* ist gut geeignet, um den Benutzer dauerhaft auf eine andere Site zu leiten. Umfangreiche Anwendungen benötigen jedoch andere Verfahren. Im Objekt `Server` sind zwei Methoden zu finden, die einmal die Ausführung direkt an eine andere Seite übertragen (`Transfer`) oder ein anderes Programm ausführen und dann im ursprünglichen fortsetzen (`Execute`).

Steuerung weitergeben: Server.Transfer

Server.Transfer

`Response.Redirect` leitet die Übertragung der Ausführung auf ein anderes Skript über eine Anforderung des Browsers ein.

Ein solcher Umweg hat den Nachteil, das Zeit und Bandbreite verloren geht. Außerdem hat der Client Einfluss auf den Ablauf. Dafür kann die Übertragung auch auf eine andere Site im Internet außerhalb der eigenen Domäne erfolgen. Letzteres ist mit `Server.Transfer` nicht möglich. Nach der Ausführung der Methode wird die Codeausführung mit der aufgerufenen Seite fortgesetzt, eine direkte Rückkehr ist nicht möglich.

Browsertyp

Das folgende Beispiel zeigt, wie die Methode eingesetzt werden kann. Die Ausführung wird hier anhand des Browsertyps gesteuert:

```
<script language="C#" runat="Server">
void Page_Init()
{
    HttpBrowserCapabilities bc = Request.Browser;
    if (bc.Type == "IE6" || bc.Type == "IE5")
    {
        Server.Transfer("start_ie.aspx?" + bc.Type);
    } else {
        Server.Transfer("start_other.aspx?" + bc.Type);
```

```
    }
}
</script>
```

Listing 3.13: Übertragung der Programmausführung in Abhängigkeit vom Browsertyp (server.transfer.aspx)

Beispielhaft wird hier eine der Antwortseiten gezeigt, die den Querystring auswertet, um von der Information über den erkannten Browsertyp zu partizipieren.

```
<script language="C#" runat="server">
void Page_Load()
{
    btype.Text = Request.QueryString.ToString();
}
</script>
<html lang="en">
    <head>
        <title>IE erkannt</title>
    </head>
    <body>
        Startseite f&uuml;r IE!
        <br/>
        Erkannter Typ: <asp:label id="btype" runat="server"/>
    </body>
</html>
```

Listing 3.14: Auswertung des Browsertyps auf der anderen Seite

Die Ausgabe ist wenig spektakulär. Bei `Response.Redirect` konnte der Benutzer den Umschaltprozess beobachten. Der Browser zeigte den Namen der aufgerufenen Seite in der Adresszeile an, bis die Weiterleitung ausgeführt wurde. Dann erschien der neue Name. Bei der Übertragung der Ausführung mit `Server.Transfer` ist der Prozess nicht zu bemerken – sogar der alte Seitenname bleibt bestehen.

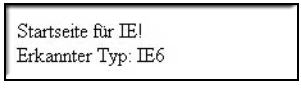


Abbildung 3.13: Alter Seitenname und neuer Inhalt. So arbeitet Server.Transfer

Im Gegensatz dazu arbeitet `Server.Execute` wie der Aufruf eines Unterprogramms.

Steuerung zeitweilig delegieren: Server.Execute

Diese Methode ruft innerhalb einer ASP.NET-Seite eine andere auf, führt diese aus und setzt nach dem Aufruf an der ursprünglichen Stelle fort. Ausgaben, die während der neuen Aufforderung entstehen, werden an der Stelle des Aufrufes ausgeben. Standardmäßig wird der Dateiname einer auszuführenden Seite angegeben:

```
Server.Execute("name.aspx");
```

Die Methode ist allerdings mit einem zweiten Parametersatz überladen. Dabei ist die Angabe eines `StringWriter`-Objekts erlaubt, das den erzeugten Datenstrom aufnehmen kann. Dadurch wird der Inhalt manipulierbar. Das folgende Beispiel nutzt die Methode, um HTML-Code aus einer anderen Datei in Abhängigkeit von bestimmten Bedingungen zu holen.

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.IO" %>
<html>
  <head>
    <title>Server.Execute</title>
    <script language="C#" runat="server">

    void Page_load()
    {
        StringWriter result = new StringWriter();
        string self = Request.ServerVariables["SCRIPT_NAME"];
        link.HRef = self + "?start=next";
        if (Request.QueryString["start"] == "next")
        {
            Server.Execute("otherse.aspx", result);
        } else {
            Server.Execute("firstse.aspx", result);
        }
        welcome.Text = result.ToString();
     }
     </script>
  </head>
  <body>
    <h1>Willkommen auf unserer Website</h1>
    <p>
    <asp:label id="welcome" runat="server"/>
    </p>
    Wir möchten Sie auf unseren Seiten über Aktuelles aus unserem
    Unternehmen informieren.
    Wählen Sie eine Information:
    <ul>
```

```
      <li><a id="link" runat="server" >Kontaktseite</a>
      <!-- weitere Links -->
    </ul>
  </body>
</html>
```

Listing 3.15: Einbinden externer Daten mit Server.Execute (server_execute.aspx)

Wie es funktioniert

In den beiden Dateien *otherse.aspx* und *firsts.aspx* steht nur HTML-Code, kein C#-Programm. Beim ersten Aufruf der Seite ist kein Parameter im Querystring. Die folgende Prüfung misslingt deshalb:

```
if (Request.QueryString["start"] == "next")
```

Nun wird der `else`-Zweig ausgeführt und die Daten aus *firstse.aspx* werden an das `StringWriter`-Objekt übergeben.

Der Zugriff auf das Dateisystem soll hier nicht weiter beschrieben werden. Dateioperationen dieser Art werden in „Zugriff auf das Dateisystem“ auf Seite 300 genauer betrachtet.

Ein `StringWriter`-Objekt kann Text enthalten, in diesem Fall die Ausgabe der Ausführung von *firstse.aspx*. Das ist in diesem Fall der dort untergebrachte Quelltext. Dieser Text wird dann an das Label `welcome` übergeben:

```
welcome.Text = result.ToString();
```

Der Text erscheint an der Stelle, wo das Label definiert wurde:

```
<asp:label id="welcome" runat="server"/>
```

Damit sich die Seite selbst aufruft, wurde zuvor der Seitenname ermittelt:

```
string self = Request.ServerVariables["SCRIPT_NAME"];
```

Dieser wird dem `<a>`-Tag im HTML-Teil zugewiesen:

```
link.HRef = self + "?start=next";
```

Entscheidend ist hier der Parameter, der angehängt wird und beim nächsten Aufruf zur Ausführung der anderen Datei führt.

3.6 Sitzungen und Applikationen

Wie am Anfang bereits angedeutet, ist die Wiedererkennung von Benutzern nicht Bestandteil des Protokolls HTTP. Die Zeit vom Abruf der ersten Seite durch einen bestimmten Benutzer bis zum Verlassen der Seite wird als Sitzung (engl. Session) bezeichnet. Das Sitzungsmanagament ist

deshalb für die ASP.NET-Programmierung von großer Bedeutung. Allerdings wird dies in ASP.NET im Kontext der Applikation betrachtet. Der Status der Applikation enthält unter anderem Informationen über den Status einer Sitzung.

3.6.1 Sitzungsmanagement

Allgemein geht es bei allen Vorgängen rund um das Management von Sitzungen um die Erhaltung des Status einer Operation. So lässt das Anforderungs-Antwort-Spiel in HTTP nicht zu, dass Informationen zum Benutzer parallel zum Vorgang erhalten bleiben. ASP.NET ersetzt diesen Mangel durch eigene Methoden, die wiederum auf protokollkonformen Wegen basieren. Insgesamt steht drei Wege zur Verfügung, den Status zu kontrollieren.

Prinzip

Sitzungen

Die Kontrolle des Sitzungsmanagements erfolgt mit dem Objekt `Session`. Hiermit kann der Status einer Sitzung erhalten und kontrolliert werden. Der Standardwert für die Laufzeit einer Sitzung beträgt 20 Minuten, einstellbar im IIS und über Eigenschaften des `Session`-Objekts. Standardmäßig basiert das Sitzungsmanagement auf Cookies. Im Gegensatz zum alten ASP funktioniert es jedoch auch dann, wenn der Client Cookies nicht akzeptiert. Dann sorgt die ASP.NET-Komponente selbst dafür, dass der Status per HTTP-GET übertragen wird.

Prinzip der Sitzungssteuerung

Um den Status eines Benutzers zu erhalten, wird intern eine spezielle Identifikationsnummer erzeugt und mitgeführt – die „Sitzungs-ID" (Session-ID).

Diese ID wird an den Browser übertragen. Der Browser sendet sie bei der nächsten Anforderung einer Seite wieder zurück und der Server kann daran erkennen, um welche Sitzung es sich handelt. Die Übertragung der ID basiert auf einem von drei Wegen:

- *Cookies* – Hier wird die ID als Sitzungs-Cookie übertragen und der Browser speichert sie ab. Bei jeder Anforderung werden die Cookie-Daten ausgelesen und übertragen. Sitzungs-Cookies werden automatisch gelöscht, wenn die Sitzung beendet ist.
- *HTTP-GET* – Diese Methode erweitert den URL um einen entsprechenden Parameter, der als Attribut die ID enthält.
- *HTTP-POST* – Wenn Formulare verwendet werden, wird ein verstecktes Feld (`type="hidden"`) erzeugt und zusammen mit den Formulardatei zurück gesendet.

Prinzipiell versucht ASP.NET zuerst Cookies einzusetzen. Hat der Benutzer seinen Browser so eingestellt, dass Cookies überhaupt nicht akzeptiert werden, wird der URL um die Sitzungsparameter ergänzt, sodass die nächste Anforderung per GET die Informationen wieder enthält. Die von ASP.NET kontrollierten Formulare (Web Forms) enthalten die Sitzungs-IDs ohnehin, sodass hier keine besonderen Maßnahmen notwendig sind. Natürlich können Sie die Verwendung von Cookies generell unterbinden, damit Benutzer mit rigiden Sicherheitseinstellungen nicht durch unnütze Alarme beunruhigt werden.

Ablage von Daten

Das alleine reicht zwar, um den Benutzer zu erkennen; die mit ihm verbundenen Daten müssen aber auch gespeichert werden. Denn ohne diese Informationen ist das Sitzungsmanagement nicht sinnvoll. Die einfachen Beispiele im Kapitel 2 kamen ganz ohne aus, hier wurden verschiedene Benutzer nicht mit verschiedenen Informationen versorgt. Für die Ablage der Daten gibt es mehrere Wege:

- Gemeinsame Speicherbereiche
- Interne Speicherbereiche
- Dateien
- Datenbanken

ASP.NET verwendet drei der vier möglichen Methoden: Gemeinsame Speicherbereiche mit ASP.NET (In-Process). Hier werden die Variablen im Speicher abgelegt und bei erneuten Aufrufen zur Verfügung gestellt. Diese Methode ist sehr schnell, aber nicht skalierbar. Wenn Sie mit einem Lastverteilungssystem arbeiten möchten oder diese Technik von Ihrem Provider verwendet wird, funktioniert es nicht. Denn die in einem lokalen Speicher abgelegten Variablen stehen auf einem anderen System nicht zur Verfügung. Die zweite Möglichkeit ist ein spezieller Dienst, der als Out-Of-Process außerhalb von ASP.NET läuft und die Verteilung über mehrere Server erlaubt. Die dritte Möglichkeit ist eine Datenbank, vorzugsweise wird dies der SQL Server 2000 sein. In diesem umfangreichsten, teuersten und leistungsfähigsten Szenario werden mehrere Webserver um einen SQL Server gruppiert. Die Lastverteilung verteilt die Anforderungen auf die Webserver, die ihrerseits Daten über den Datenbankserver austauschen. Der Datenbankserver selbst nutzt natürlich auch den Speicher zum schnellen Zugriff auf die Daten.

Sitzungs-Cookies

Die Sitzungs-Cookies sind in ASP.NET leicht zu erkennen, wenn Sie einen Cookie-Manager einsetzen. Im Netscape 6.2 ist dies besonders einfach.

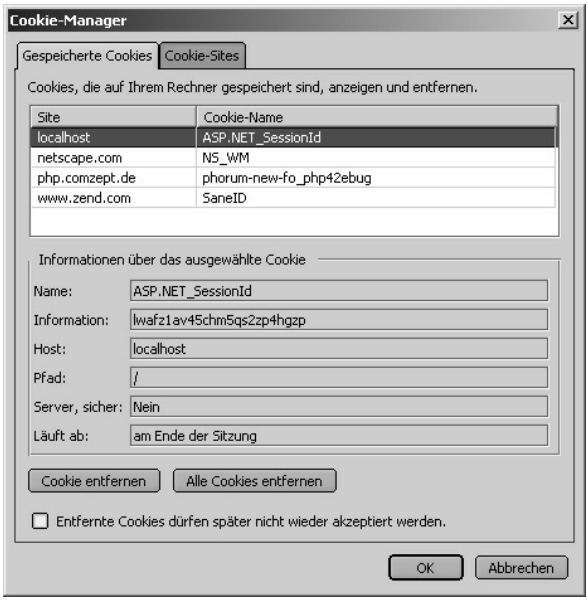


Abbildung 3.14: ASP.NET-Sitzungscookie im Cookiemanager

In der ID selbst werden keine Daten gespeichert, sie dient nur der Wiedererkennung. Wenn Sie das Sitzungsmanagement bildlich betrachten, kann man es sich so vorstellen, als würde jeder Abruf des Programms den Code kopieren und dann in einer eigenen Instanz ausführen (tatsächlich wird es genau so auch gemacht). Die folgende Abbildung zeigt dies nochmals:

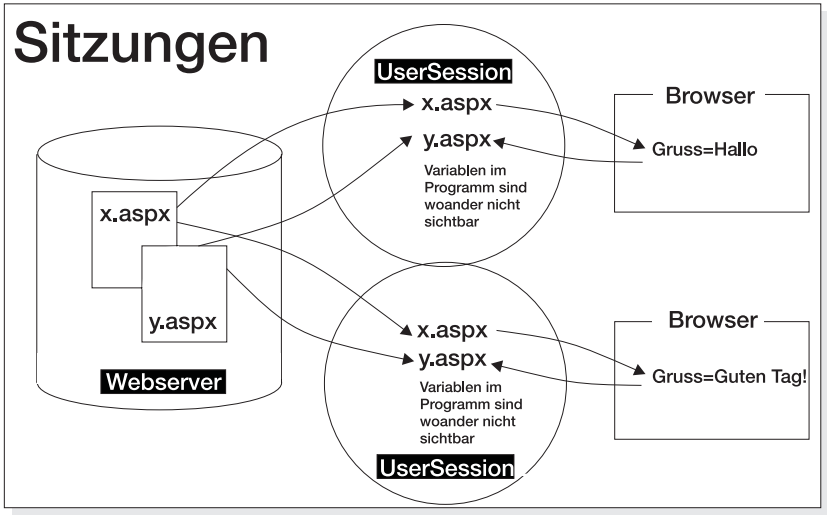


Abbildung 3.15: Dasselbe Programm speichert für verschiedene Benutzer verschiedene Inhalte

Praktische Anwendung

Anwendung

Das folgende Programm zeigt, wie eine Sitzungsvariable erzeugt und gespeichert wird:

```
<script language="C#" runat="server">
void Page_Init()
{
    Session["Farbe"] = "red";
}
</script>
```

Listing 3.16: Erzeugen und speichern einer Sitzungsvariablen (Ausschnitt aus session1.aspx)

Beachten Sie hier, dass als Methode zum Aufruf des Codes `Page_Init` eingesetzt wird. Cookies sind Header-Informationen, die vor dem Senden der Seite zusammengestellt werden müssen. Da standardmäßig davon ausgegangen werden muss, dass Cookies verwendet und akzeptiert werden, wird der Aufruf in `Page_Load` misslingen, denn bei Ausführung dieser Methode ist die Seite bereits fertiggestellt und Header können nicht mehr hinzugefügt werden.

Das folgende Programm zeigt, wie auf den gespeicherten Wert wieder zugegriffen werden kann:

```
<%@ Page Language="C#" debug="true"%>
<script language="C#" runat="server">
void Page_Load()
{
   string color;
   color = Session["Farbe"].ToString();
   currentcolor.Text = color;
}
</script>
<html>
  <head><title>C# lernen</title></head>
  <body>
  <h1>Sessions</h1>
  Dieses Skript liest eine Sessionvariable:
  <b><asp:label id="currentcolor" runat="server"/></b>
  <br/>
  Klicken Sie auf den Link, um wieder auf die erste Seite zu gelangen.
  <br/>
  <a href="session1.aspx">Zur ersten Seite</a>
  </body>
</html>
```

Listing 3.17: Vollständiges Beispiel zum Abruf der Session-Variablen (session2.aspx)

Im Gegensatz zum Senden spielt der Zeitpunkt des Abrufs der Informationen beim Empfang keine Rolle. Der Browser sendet immer alle Cookies zu der Domain zurück, von der er sie empfangen hat. Im `Session`-Objekt stehen diese Informationen deshalb immer zur Verfügung. Der eigentliche Speicherort für die Werte steht natürlich auch immer bereit, egal ob es sich um eine Datenbank oder den lokalen Speicher handelt. Beachten Sie, dass Daten immer als Objekt und in serialisierter Form gespeichert werden.

Am Ende geht es hier immer nur darum, ein Objekt aus der aktuellen Seite so zu speichern, dass es dem Benutzer auf der nächsten Seite wieder zugeordnet werden kann. Über die Methode müssen Sie sich dabei am Anfang keine Gedanken machen. Der Standard ist In-Process, also innerhalb des ASP.NET-Prozesses im IIS. Das `Session`-Objekt selbst bietet neben Eigenschaften und Methoden vor allem eine Kollektion von gespeicherten Objekten. Dies sind meist Zeichenketten oder Zahlen, die Verwendung ist aber diesbezüglich nicht eingeschränkt. Diese universelle Vorgehensweise stellt sicher, dass alle Arten von Datentypen gespeichert werden können. Serialisierung ist ein Vorgang, bei dem komplexe Datentypen als Zeichenkette abgelegt werden, sodass später der ursprüngliche Typ mit Daten wieder entstehen kann. Zeichenketten sind der einzige zuverlässige Weg der Speicherung außerhalb der ASP.NET-Komponente. Wenn Sie Klassen speichern wollen, müssen diese mit dem Attribut `<serializable>` gekennzeichnet werden.

Das Session-Objekt

Eigenschaften

Das `Session`-Objekt besitzt einige Eigenschaften, mit denen es überprüft werden kann:

- `IsCookieless` – Wenn die Eigenschaft `true` zurückgibt, werden Cookies nicht verwendet.
- `IsReadOnly` – Wenn die Sitzung als Nur-Lese-Sitzung gekennzeichnet wurde, wird `true` zurückgegeben. Ist Schreiben nicht erlaubt, ist ASP.NET in der Behandlung etwas schneller.
- `Mode` – Diese Eigenschaft enthält einen Wert einer Aufzählung, der angibt, welche Speichermethode verwendet wird. Mögliche Werte sind `Off`, `InProc`, `StateServer` oder `SQLServer`. `InProc` ist der Standardwert.
- `IsNewSession` – Wenn die Sitzung mit der aktuellen Anforderung erzeugt wurde, ist diese Eigenschaft `true`.
- `SessionID` – Diese Eigenschaft enthält die Session-ID.
- `TimeOut` – Tragen Sie hier die Anzahl Minuten ein, die eine Sitzung ohne Aktivität laufen darf. Der Standardwert beträgt 20. Löst der Benutzer innerhalb dieser Zeit keine Anforderung aus, wird die Session-ID verworfen und bei einer dann folgenden Anforderung eine neue erzeugt.

Alle Eigenschaften außer `TimeOut` sind nur lesbar. `IsReadOnly` wird über einen Parameter der `@Page`-Direktive gesteuert:

```
<% @Page EnableSessionState="ReadOnly" %>
```

Andere Parameter sind `true` (ein und schreibbar) bzw. `false` (aus).

Die vollständige Liste ist in der Referenz zu ASP.NET zu finden. Neben den für dieses Objekt spezifischen sind die für Kollektionen typischen Eigenschaften `Count`, `Item` und `Keys` zu finden. Informationen zum Umgang mit Kollektionen finden Sie im „Aufzählungen und Kollektionen“ auf Seite 288.

Methoden

Das Objekt kennt auch einige Methoden:

- `Abandon` – Abbruch der Sitzung
- `Add` – Fügt ein weiteres Objekt der Kollektion hinzu. `Session["Wert"] = 13` entspricht `Session.Add("Wert", 13)`.
- `Clear`, `RemoveAll` – Mit `Clear` werden alle Werte gelöscht, mit `RemoveAll` alle Objekte.

Aus der Basisklasse werden die Methoden `ToString`, `Equals` und `GetType` geerbt.

Session-Ereignisse

ASP.NET kennt für Sitzungen auch Ereignisse. Diese führen zur Ausführung von Methoden in der Datei *global.asax*, wenn sie definiert sind. Es gibt keinen Zwang, die Ereignisse zu verwenden oder anderweitig zu bedienen. Ein Anwendungsbeispiel und Informationen zum Umgang mit *global.asax* finden Sie im Kapitel „Die Datei global.asax“ auf Seite 172. Verfügbar sind folgende Ereignisse:

- `OnStart` – Diese Ereignis wird ausgelöst, wenn eine neue Sitzung startet.
- `OnEnd` – Endet eine Sitzung, egal aus welchen Gründen, wird dieses Ereignis gestartet.

3.6.2 Applikationsmanagement

Sitzungen sind immer benutzerorientiert. Dies ist nicht hilfreich, wenn Daten zwischen Benutzern ausgetauscht werden sollen. Es gibt deshalb auch eine Sicht auf die in ASP.NET ablaufenden Prozesse aus dem Blickwinkel der gesamten Applikation.

Prinzip

Ein typischer Weg, große Datenmengen allen Benutzern bereit zu stellen, sind Datenbanken. Kleinere Datenmengen, die sich während der

Laufzeit häufig ändern und deren Inhalt von den aktuellen Aktionen der Benutzer abhängt, sind darin nicht immer optimal aufgehoben. ASP.NET kennt dafür applikationsweite Variablen. Die Verwaltung erfolgt im Rahmen des Applikationsmanagements. Ebenso wie für die Sitzungen gibt es ein spezielles Objekt dafür: `Application`. Instanzen werden ähnlich dem `Session`-Objekt automatisch durch die ASP.NET-Komponenten gebildet. Die Verwendung entspricht einer statischen Klasse (auch wenn es keine solche ist).

Applikation

Eine Applikation umfasst alle Dateien, Seiten, Module und Programme im Sichtbereich eines virtuellen Verzeichnisses auf einem einzelnen Webserver. Der Datenaustausch kann also nur innerhalb dieses Kontextes stattfinden. Innerhalb einer solchermaßen gebildeten Applikation gibt es eine ganze Reihe spezieller Ereignisse, auf die Sie reagieren können aber nicht müssen. Zehn Ereignisse werden immer ausgelöst, wenn eine *aspx*-Seite angefordert wird.

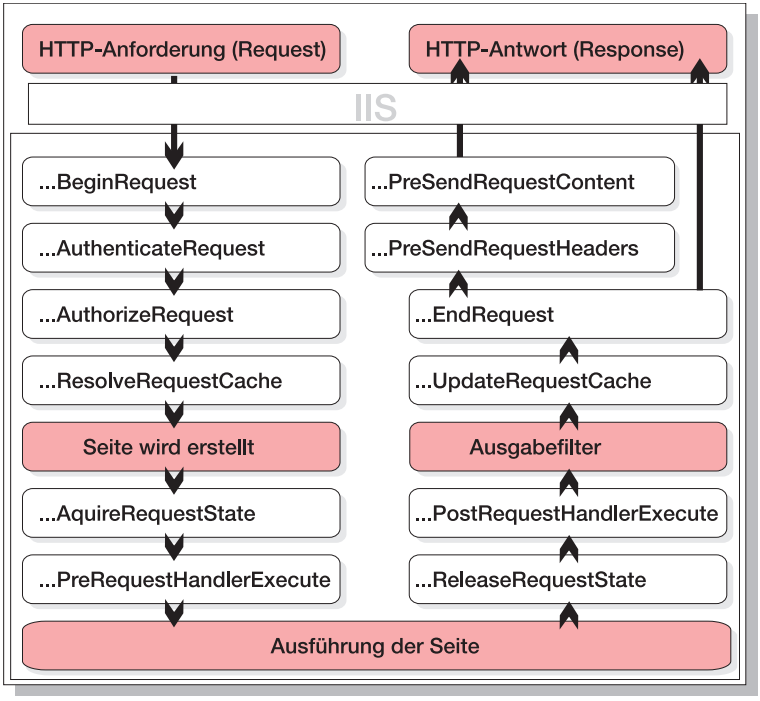


Abbildung 3.16: Ablauf des Aufrufes der zehn Standardereignisse einer Applikation

Ereignisse

Die Behandlung dieser und der nachfolgend beschriebenen bedingten Ereignisse finden in der Datei *global.asax* statt. Der Start der Ereignisse bei der Abarbeitung der Seite (`Page_Init`, `Page_Load` usw.) geschieht innerhalb des Prozessschrittes „Ausführung der Seite“ in Abbildung 3.16. Die einzelnen Ereignisse haben folgende Bedeutung:

- `BeginRequest` – Dieses Ereignis wird bei jeder Anforderung zuerst ausgelöst. Hier können Aktionen ausgeführt werden, die unabhängig von irgendeinem anderen Teil des Programms benötigt werden.
- `AuthenticateRequest`, `AuthorizeRequest` – Diese Ereignisse dienen der Behandlung der Authentifizierung (Erkennung) bzw. Autorisierung (Rechtevergabe).
- `ResolveRequestCache` – Wenn der interne Zwischenspeicher verwendet wird, liefert ASP.NET Seiten aus dem Cache aus, ohne erneut den Code auszuführen. Dieses Ereignis wird unmittelbar vor der Prüfung der Gültigkeit des Zwischenspeichers aufgerufen. Es erlaubt damit die Ausführung von Code unabhängig vom Zustand des Zwischenspeichers oder seiner Verwendung.
- `AcquireRequestState` – An dieser Stelle werden die Sitzungsdaten rekonstruiert. Wenn eine eigene Sitzungsverwaltung implementiert werden soll, sollte der Code hier ausgeführt werden.
- `PreRequestHandlerExecute` – Dies ist das letzte Ereignis vor der Ausführung der Anforderung. Diese führt unter normalen Umständen und bei einfachen Seitenabrufen zu einem `Page`-Request.
- `PostRequestHandlerExecute` – Dies ist das erste Ereignis nach der Ausführung der Anforderung.
- `ReleaseRequestState` – In diesem Schritt werden die Sitzungsdaten aktualisiert. Änderungen daran sind nun nicht mehr möglich.
- `UpdateRequestCache` – Falls der Zwischenspeicher aktualisiert werden soll und dazu spezifische Aktionen notwendig sind, kann dieses Ereignis verwendet werden.
- `EndRequest` – Dies ist das letzte Ereignis, bevor die Daten an den IIS zur Auslieferung übergeben werden. Danach werden Kopf und Körper der Antwort gesendet.

Ungepufferte Ausgaben behandeln

Bei der Beantwortung einer Anfrage gibt es zwei Strategien, die angeforderten Daten zu übertragen. Entweder wird der gesamte Inhalt gesammelt und anschließend komplett übertragen. Diese Strategie verwendet einen Ausgabepuffer, um die Daten zu sammeln. Dann gilt der Ablauf, den die bereits beschriebenen Ereignisse behandeln. Wird dagegen die Seite ungepuffert aufgebaut, werden immer wieder kleinere Fragmente an den IIS zum Senden übergeben. Während die linke Seite in Abbildung 3.16 unverändert und einmalig zur Ausführung gelangt, funktioniert das mit der Ausgabe nicht mehr. Deshalb gibt es zwei spezielle Ereignisse:

- `PreSendRequestHeaders` – Dieses Ereignis wird ausgelöst, bevor die Kopfzeilen gesendet werden.
- `PreSendRequestContent` – Auch wenn mehrere Fragmente einer Antwort übertragen werden, gibt es nur einen Zeitpunkt, an dem die Kopfzeilen vollständig sind. Dieses Ereignis löst aus, wenn die Header vollständig vorliegen und die Übertragung der Inhalte beginnt.

  Am Ende der Übertragung wurden die anderen Antwort-Ereignisse dennoch ausgelöst.

Bedingte Ereignisses

Neben den unbedingten Ereignissen gibt es auch bedingte. Diese treten nur auf, wenn die folgenden Bedingungen innerhalb einer Applikation auftreten:

- `OnStart` – Dieses Ereignis tritt auf, wenn die Applikation das erste Mal aufgerufen wird. Dies passiert logischerweise nur ein einziges Mal.
- `OnEnd` – Auch dieses Ereignis ist einmalig. Es wird beim Stoppen der Applikation aktiv.
- `Error` – Wenn Fehler auftreten, können Sie diese mit diesem Ereignis behandeln. Es ist mächtiger als `Page_Error`, da alle Laufzeitfehler einer Applikation abgefangen werden können.
- `OnDisposed` – Dieses Ereignis tritt auf, wenn eine Applikation vollständig entfernt wird, beispielsweise über die Managementkonsole des IIS.

Alle Applikationsereignisse werden in der Datei *global.asax* verarbeitet.

Applikations-Variablen

Ebenso wie bei den Sitzungen kennen auch Applikationen Variablen. Diese stehen dann allen Benutzern zur Verfügung. Von den Sessionvariablen unterscheiden sich Applikationen durch drei Eigenschaften:

1. Applikationsvariablen basieren nicht auf Cookies.
2. Der Webserver muss keine Session mitführen, um mit der Applikationsvariablen zu arbeiten.
3. Die Verwendung ist risikolos und unabhängig vom Browser oder seinen Einstellungen.

Das Einsatzspektrum der Variablen ist weit gefächert. Es gibt viele Anwendungen, die davon profitieren. Wenn Sie immer wieder auf dieselben veränderlichen Informationen zugreifen möchten, wie beispielsweise einen Nachrichtendienst oder Tipp des Tages, sind Applikations-Variablen einsetzbar. Speichern Sie den Wert in der Applikations-Variablen und er steht allen Nutzern überall zur Verfügung. Andere typische Einsatzfälle sind Bannersteuerungen oder Besuchszähler, so genannte Counter.

Das folgende Programm zeigt, wie eine Applikations-Variable erzeugt und mit einem Wert belegt wird:

```
<script language="C#" runat="Server">
void Page_Load()
{
    Application["Version"] = "0.9.3";
}
</script>
<html>
    <head>
        <title>Applikations-Variablen</title>
    </head>
    <body>
        Auf dieser Seite wird eine Applikationsvariable erzeugt und gesetzt.
        <br/>
        Auf der <a href="application_getvar.aspx">folgenden Seite </a> wird
        sie wieder ausgelesen.
    </body>
</html>
```

Listing 3.18: Setzen einer Applikations-Variablen (application_setvar.aspx)

Auf einer anderen Seite kann dann auf diese Variable zugegriffen werden:

```
<script language="C#" runat="Server">
void Page_Load()
{
    string version = Application["Version"].ToString();
    ver.Text = version;
}
</script>
<html>
    <head>
        <title>Applikations-Variablen</title>
    </head>
    <body>
        Auf dieser Seite wird eine Applikationsvariable ausgelesen.
        <br/>
        Version = <asp:label id="ver" runat="Server"/>
    </body>
</html>
```

Listing 3.19: Nutzung einer Applications-Variablen (application_getvar.aspx)

Intern werden Applikations-Variablen als Objekte gespeichert, deshalb ist die Anwendung von `ToString` angebracht. Wie schon bei den Sitzun-

gen gezeigt, ist das Objekt direkt instanziiert und muss nicht abgeleitet werden. Es ist dennoch nicht statisch. Die Variablen bilden eine Aufzählung, auf die direkt über einen Indexer zugegriffen werden kann. Daher ist die Kurzschreibweise `Application["Name"]` zulässig.

Eine gute Anwendung ist ein Hitzähler für Ihre Webseite. Damit der Zähler auch exakt arbeitet, ist es sinnvoll, über die interne Arbeitsweise nachzudenken. Der Webserver liefert Seiten an Nutzer, wann immer diese die Seiten anfordern. So entstehen parallel laufende Prozesse. Da alle Prozesse Zeit brauchen, um ausgeführt zu werden, ergibt sich möglicherweise ein Problem. Wenn zwei Nutzer gleichzeitig eine Seite aufrufen, werden die Werte parallel verarbeitet. Wenn der Ursprungswert der Variablen 4 ist, schreibt Nutzer 1 mit seiner Sitzung den Wert 5 zurück. Bis dahin hat aber auch Nutzer 2 die Seite gestartet, ebenfalls den Wert 4 ermittelt und 5 zurückgeschrieben. Danach steht der Zähler auf 5 und nicht, wie es richtig wäre, auf 6. Das `Application`-Objekt kennt deshalb zwei besondere Methoden, die dazu gedacht sind, andere Prozesse vorübergehend zu stoppen – `Lock` und `UnLock`. Diese Methoden werden aus der Klasse `HttpApplicationState` implizit abgeleitet und stehen unmittelbar zur Verfügung. Die folgende Abbildung zeigt, wie die Methoden eingesetzt werden können, um den Zähler korrekt ablaufen zu lassen.

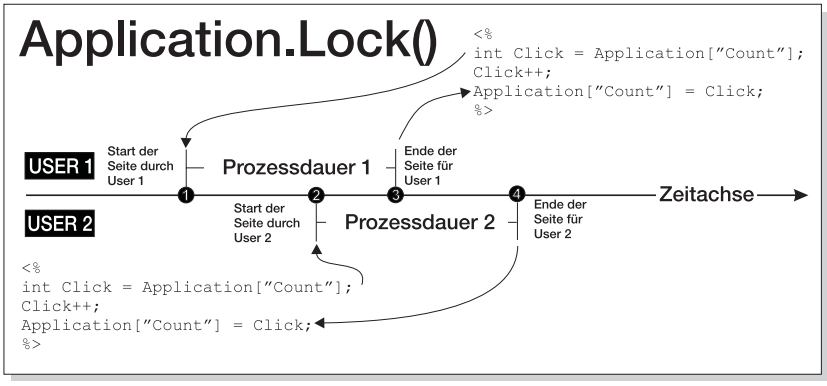


Abbildung 3.17: Ablauf von Seitenabrufen und Einsatz von Application.Lock()

Daneben sind auch Methoden und Eigenschaften zum Zugriff auf die Variablen-Kollektion vorhanden, die sich nicht von anderen Kollektionen unterscheiden.

Eine praktische Anwendung finden Sie im folgenden Abschnitt.

3.6.3 Die Datei global.asax

Der Umgang mit *global.asax* ist recht einfach. Alle in den beiden letzten Abschnitten angesprochenen Ereignisse werden an Methoden geleitet, die sie hier definieren können. Alle Methoden sind vom Typ `void`. Applikations-Ereignisse tragen den Präfix *Application_On*, Sitzungsereignisse dagegen *Session_On*. Wenn Sie also eine Behandlung der Ereignisse wünschen, dann muss lediglich die passende Methode dazu definiert werden.

Die Datei *global.asax* liegt im Stammverzeichnis der Applikation. Dies ist bei virtuellen Verzeichnissen das Startverzeichnis des Verweises, so wie es im IIS eingerichtet wurde. Allein das Ablegen der Datei an der richtigen Stelle reicht aus, um auf Applikations- und Sitzungsereignisse reagieren zu können. Das folgende Beispiel zeigt eine Datei *global.asax*, die gleich mehrere Aufgaben erfüllt:

```
<script language="C#" runat="Server">
void Application_OnEndRequest()
{
    string hits = Application["HitCounter"].ToString();
    Response.Write ("<hr/>Sie sind Besucher " + hits + ".</body></html>");
}
void Application_OnStart()
{
   Application["HitCounter"] = 0;
   Application["Version"] = "0.9.3";
}
void Session_OnStart()
{
   Application.Lock();
   Application["HitCounter"] = Convert.ToInt32(Application["HitCounter"])
                               + 1;
   Application.UnLock();
}
</script>
```

Listing 3.20: Ereignisverarbeitung in global.asax

Wie es funktioniert

Verwendet wird hier das Ereignis `Application_OnStart`. Beim Start der Applikation oder beim ersten Aufruf der Datei wird die entsprechende Methode ausgeführt. Die Applikations-Variablen `HitCounter` und `Version` werden erzeugt und gesetzt. Mit dem Start jeder neuen Sitzung (Ereignis `Session_OnStart`) wird die Applikation verriegelt, der Zähler `HitCounter` erhöht und die Applikation wieder freigegeben. Außerdem wird am Ende jeder Anforderung – also jedes einzelnen Seitenabrufes – eine Zeile erzeugt, die jede Seite abschließt. Beachten Sie, dass dies der letzte Text ist,

der einer HTML-Seite hinzugefügt werden kann. Um korrektes HTML zu erzeugen, sollten die abschließenden Tags `</body>` und `</html>` mit übergeben und aus den Seiten der Applikation entfernt werden.

3.7 Cookies

Im Abschnitt über Sitzungen und Applikationen wurden Cookies implizit verwendet – zur Speicherung des Status. Cookies sind aber auch unabhängig davon vielfältig einsetzbar.

3.7.1 Cookies als Informationsspeicher

Cookies dienen der Speicherung von Informationen für einen kurzen Zeitraum. Wie dies funktioniert und warum es Probleme mit Cookies gibt, wird in diesem Abschnitt erläutert. Cookies (dt. Kekse) haben einen völlig irreführenden, verharmlosenden Namen. Aber sie sind bekannt und oft verteufelt als der Angriffspunkt des bösen Hackers aus dem Web, der sich an den privaten Dateien der Surfer zu schaffen machen will.

Die Ursache für den ganzen im Prinzip unbegründeten Ärger ist ein Artikel von Jon Udell in der Märzausgabe 1997 der Zeitschrift BYTE, einer der größten amerikanischen Computerfachzeitschriften. Dort wurde berichtet, dass Cookies Informationen auf Anforderung des Servers auf der lokalen Festplatte des Nutzers speichern und natürlich auch lesen können. Daraus wurde geschlussfolgert, dass der Server private Daten vom Computer des Surfers lesen kann. Des Weiteren wurde behauptet, dass andere Server wiederum die Daten lesen können, die schon im Cookie gespeichert sind, wodurch das Auslesen privater Kennwörter möglich sein soll. All das hat zur Verdächtigung der Cookies beigetragen und dazu geführt, das viele Nutzer die Funktion abschalten oder sogar die Cookiedateien regelmäßig löschen. In der Maiausgabe 1997 der BYTE wurde der gesamte Artikel revidiert und richtig gestellt – zu spät, wie sich herausstellte. Die sensationslüsterne Presse hatte wieder etwas Negatives am Web entdeckt und wollte sich das Spielobjekt nicht wegnehmen lassen. Der Pro-Cookie-Artikel wurde totgeschwiegen.

Aus heutiger Sicht muss man sagen, das tatsächlich eine Reihe von Missbrauchsmöglichkeiten bestehen, die vor allem die Privatsphäre des Nutzers betreffen. Eine Sicherheitslücke, die ein generelles Abschalten rechtfertigen würde, sind Cookies aber definitiv nicht.

Cookies wurden von Netscape erfunden und sind seit der ersten Version des Navigators dabei. Später wurde daraus ein Standard, der auch vom World Wide Web Consortium W3.ORG unterstützt wird. Cookies sind eine oder mehrere Dateien, die der Browser anlegt und in denen der sendende Server auf Wunsch Informationen unterbringen und wieder auslesen kann. Der Sinn von Cookies ist die Wiedererkennung des Nutzers bei einer späteren Session. Cookies lösen also ein gravierendes Problem des HTTP-Protokolls. Cookies können temporär sein, also am Ende einer Sitzung wieder gelöscht werden, andere sind permanent und werden nie oder sehr viel später gelöscht.

Aufbau und Prinzip

Cookies werden zwischen Server und Browser durch HTTP-Header übertragen. Durch Senden eines Set-Cookie-Headers wird ein Cookie in der Cookiedatei erzeugt. Soll beispielsweise der Name eines Nutzers gespeichert werden, sieht der zugehörige Header folgendermaßen aus (gesendet wird der Text allerdings in einer Zeile):

```
Set-Cookie: UserName=Roger+Waters;
            path=/;
            domain=comzept.de;
            expires=Tuesday, 01-Jan-99 00:00:01 GMT
```

Der neue Eintrag in der Cookiedatei wird jetzt erstellt. Die erste Zeile nach dem Header-Namen *Set-Cookie* enthält den Namen des Cookies und dessen Inhalt. Die Variable *path* schränkt die Rückgabe des Eintrags auf Seiten ein, die von dem benannten Pfad aus anfragen. Mit *domain* wird die Rückgabe auf die angegebene Domain eingeschränkt. Server, die aus anderen Domains das Cookie abfragen, erhalten keine Antwort. Das ist auch der Grund, warum fremde Server – nicht für sie bestimmte Einträge – tatsächlich nicht lesen können. *expires* gibt das Datum an, an dem der Eintrag ungültig ist und vom Browser gelöscht wird. Allerdings kann der Browser den Eintrag auch schon früher löschen, wenn die Datei zu groß wird. Der Cookie wird, wenn nun Domain und Pfadangabe stimmen, in jede Anfrage eingebaut, die der Browser an den Server stellt. Für jedes Verzeichnis im Webserver können Sie also eigene Cookies erzeugen.

3.7.2 Cookies praktisch verwenden

Cookies werden in ASP.NET umfassend unterstützt. Da die Aussendung im Header der Serverantwort erfolgt, werden zu sendende Cookies als Kollektion im Objekt `Response` gesammelt. Umgekehrt sind die vom Browser übermittelten Cookies als Kollektion im Objekt `Request` zu finden. Zur Bildung der Cookies selbst wird die Klasse `HttpCookie` verwen-

det, mehrere Cookies werden in Objekten der Klasse `HttpCookieCollection` gespeichert.

Damit Sie in den nächsten Beispielen den Effekt sofort sehen, ist es empfehlenswert, in den Sicherheitseinstellungen des Internet Explorers, Registerkarte DATENSCHUTZ, für alle Cookies die Option EINGABEAUFFORDERUNG zu wählen. Sie werden dann jedesmal aufgefordert, empfangene Cookies zu bestätigen. In dem dann angezeigten Dialog können Sie auf DETAILS klicken, um zu sehen, was Ihr Programm tatsächlich gesendet hat.

Cookie setzen und überprüfen

Die folgende Seite setzt ein Cookie mit einer Laufzeit von zwei Stunden:

```
<script language="C#" runat="Server">
void Page_Init()
{
    HttpCookie phonecookie = new HttpCookie("");
    phonecookie.Name = "Musiker";
    phonecookie.Value = "Roger Waters";
    phonecookie.Expires = DateTime.Now.AddHours(2);
    Response.Cookies.Add(phonecookie);
}
</script>
<html>
  <head>
    <title>Response.Cookies</title>
  </head>
  <body>
    Das Cookie wurde gesetzt. Klicken Sie auf den
    <a href="request.cookies.aspx">Link</a>, um es wieder auszulesen.
  </body>
</html>
```

Listing 3.21: Erzeugen und Senden eines Cookies (response.cookies.aspx)

Das Programm nutzt die Methode `Page_Init` zur Ausgabe der Cookies. Damit wird sichergestellt, dass die Header noch verändert werden können. Cookies werden als HTTP-Header übertragen. Dies würde mit `Page_Load` auch funktionieren, da aber keine Elemente der fertigen Seite benötigt werden, ist die Wahl der Methode etwas direkter. Zuerst wird ein Cookie-Objekt erzeugt:

```
HttpCookie phonecookie = new HttpCookie("");
```

Cookie-Daten setzen

Dann werden die Eigenschaften gesetzt, die unbedingt notwendig sind. Der Name wird mit `Name` festgelegt. Alternativ kann die Angabe auch

gleich im Konstruktor erfolgen. Die folgende Zeile würde die ersten beiden ersetzen:

```
HttpCookie phonecookie = new HttpCookie("Musiker");
```

Dann wird der Wert des Cookies mit `Value` gesetzt. Zuletzt ist noch das Verfallsdatum wichtig, mit der Eigenschaft `Expires` festgelegt. Die Darstellung in der korrekten Datumsform erledigt ASP.NET wieder automatisch. Deshalb können hier einfach Methoden der Klasse `DateTime` verwendet werden. `Now` ermittelt die aktuelle Zeit und `AddHours(2)` rechnet zwei Stunden hinzu. Detaillierte Informationen zu den Datums- und Zeitmethoden finden Sie in „Datum und Zeit“ auf Seite 278. Wenn Sie im Browser die Anzeige des Cookies forciert haben, sehen Sie folgende Ausgabe in der Detailansicht:

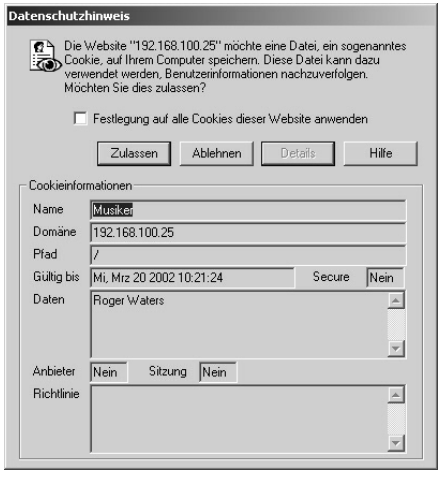


Abbildung 3.18: Anzeige des gesendeten Cookies im Internet Explorer

Für die praktische Nutzung ist natürlich das Auslesen auf einer anderen Seite wichtiger. Das folgende Beispiel zeigt dies:

```
<script language="C#" runat="Server">
void Page_Load()
{
    string cv = Request.Cookies["Musiker"].Value;
    cookievalue.Text = cv;
}
</script>
<html>
  <head>
    <title>Request.Cookies</title>
  </head>
  <body>
```

```
    Es wurde folgendes Cookie mit dem Name Musiker erkannt:<br/>
    Inhalt: "<asp:label id="cookievalue" runat="Server"/>"
  </body>
</html>
```

Listing 3.22: Auslesen empfangener Cookies (request.cookies.aspx)

Cookie empfangen

Der Zugriff auf empfangene Cookies erfolgt über `Request.Cookies`. Auch dies ist wieder eine Kollektion. Es besteht die Möglichkeit, über den Index direkt zuzugreifen, wie dies im Beispiel gemacht wird. Der Wert des Cookies steht in der Eigenschaft `Value` zur Verfügung. Denken Sie daran, dass jedes Cookie wiederum ein Objekt der Klasse `HttpCookie` ist. Die ersten beiden Zeilen der Methode `Page_Load` könnten auch folgendermaßen geschrieben werden:

```
HttpCookie oCookie = Request.Cookies["Musiker"];
cookievalue.Text = oCookie.Value;
```

Die Instanziierung eines Objekt lohnt freilich nur, wenn Sie planen, mehr damit zu tun, als nur den Inhalt auszulesen.

Cookie-Kollektionen verwenden

Bislang wurde zwar von der `Cookie`-Kollektion gesprochen, aber diese nicht tatsächlich verwendet, da nur ein einziges Cookie vorhanden war. Das folgende Beispiel erzeugt eine Kollektion mit Hilfe der Klasse `HttpCookieCollection` und sendet drei Cookies. Das darauffolgende Listing stellt alle Cookies dar, die der Browser gesendet hat.

```
<script language="C#" runat="Server">
void Page_Init()
{
    string[,] aValues = {{"Musiker", "Roger Water"},
                         {"Band", "Pink Floyd"},
                         {"Album", "The Wall"}};
    for(int i = 0; i < aValues.GetLength(0); i++)
    {
        string sCookName = aValues[i, 0];
        string sCookValue = aValues[i, 1];
        HttpCookie cTemp = new HttpCookie(sCookName);
        cTemp.Value = sCookValue;
        Response.Cookies.Add(cTemp);
    }

}
</script>
```

Listing 3.23: Senden mehrerer Cookies aus einem Array (response.cookiescoll.aspx)

Wie es funktioniert

In diesem Programm werden Daten aus einem zweidimensionalen Array entnommen und jeweils einem `Cookie`-Objekt zugewiesen:

```
cTemp.Value = sCookValue;
```

Dieses wird dann an die `Cookie`-Kollektion des `Response`-Objekts angehängt:

```
Response.Cookies.Add(cTemp);
```

Haben Sie die Eingabeaufforderung im Browser wieder eingeschaltet, werden die drei Cookies nacheinander angezeigt.

```
<script language="C#" runat="Server">
void Page_Load()
{
    HttpCookieCollection MyCookieColl = Request.Cookies;
    HttpCookie MyCookie;
    String[] CookieNames = MyCookieColl.AllKeys;
    for (int i = 0; i < CookieNames.Length; i++)
    {
        MyCookie = MyCookieColl[CookieNames[i]];
        Response.Write ("<b>Cookie:</b> " + MyCookie.Name + "<br>");
        Response.Write ("<b>Wert:</b> " + MyCookie.Value + "<br>");
    }
}
</script>
```

Listing 3.24: Auslesen aller empfangenen Cookies (request.cookiescoll.aspx)

Wie es funktioniert

Zum Auslesen wird eine Instanz der Klasse `HttpCookieCollection` direkt benutzt.

```
HttpCookieCollection MyCookieColl = Request.Cookies;
```

Diese wird dann mit einer Schleife durchlaufen, zuvor werden jedoch die Namen extrahiert. In der Kollektion bilden die Namen die Schlüssel, deshalb kommt die Eigenschaft `AllKeys` zum Einsatz:

```
String[] CookieNames = MyCookieColl.AllKeys;
```

In der Schleife kann dann der Zugriff über die so erhaltene Liste der Schlüssel (= Cookienamen) erfolgen:

```
MyCookie = MyCookieColl[CookieNames[i]];
```

Die Ausgabe nutzt dann die Eigenschaften `Name` und `Value` des Cookie-Objekts.

3.8 Hinterlegter Code: Code Behind

Bei der Arbeit mit größeren Applikationen ist die Verbindung von Code und HTML in einer Datei nicht immer praktikabel. Eine strengere Trennung – Code in einer Datei und HTML in einer anderen – ist besser geeignet. Wie am Anfang schon gezeigt wurde, stellt ASP.NET einige Automatismen bereit, die das Erstellen einfacher Seiten stark erleichern. So fehlte in allen vorangegangenen Beispielen die in C# (ebenso wie in C/C++) notwendige Methode `main`. Ein Blick in die Quelle des Compilers verrät, dass die kurzen Programme mit Hilfe der Reflection-Technik in ein reguläres C#-Programm übersetzt wurden. Direkter und offensichtlicher programmieren Sie in hinterlegtem Code – Code Behind genannt.

Außer bei kleinen Übungsdateien, wie sie in diesem Buch häufig vorkommen, sollten Sie immer Code Behind verwenden. Wird als Entwicklungsumgebung Visual Studio .Net verwendet, wird diese Technik immer genutzt.

3.8.1 Hinterlegten Code erzeugen

Wenn der Code getrennt wird, muss eine reguläre Klasse erstellt werden. Die Ablage erfolgt in einer *cs*-Datei, die der Compiler übersetzen muss. Da Sie in dieser Klasse Zugriff auf die HTML-Elemente haben wollen, müssen diese Objekte verfügbar gemacht werden. Dazu stellt ASP.NET die *aspx*-Seite als Klasse zur Verfügung. Diese trägt den Namen `Page`. Die Definition ihrer eigenen Klasse würde dann folgendermaßen aussehen:

```
public class MyClass : Page
```

Weiterhin ist zu beachten, dass ASP.NET den Import der wichtigsten Namensräume des .Net-Frameworks für Sie erledigt. Das ist nicht mehr der Fall, wenn Sie Code Behind verwenden. Sie müssen dann selbst die nötigen Anweisungen erteilen. In C# erfolgt dies mit der Anweisung `using`:

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using System.Collections;
```

Die gezeigten Namensräume sollten für die meisten Aufgaben mit ASP.NET ausreichen. Alle anderen bindet auch ASP.NET nicht automatisch ein, sodass sich hier bei Code Behind keine Änderungen ergeben.

3.8.2 Hinterlegten Code verwenden

Im nächsten Schritt muss eine Methode gefunden werden, den externen Code in der *aspx*-Seite zu verwenden. Dazu wird die Seitendirektive `@Page` eingesetzt:

```
<% @Page Inherits="classname" src="path/file.cs" %>
```

Sie können also die externe Code-Datei und die verwendete Klasse getrennt erreichen. Dadurch ist es möglich und oft sinnvoll, mehrere Klassen in einer Datei unterzubringen.

Interessant ist die Möglichkeit, die *cs*-Datei vorab zu komplieren und dann als Binärdatei bereit zu stellen. Liegt diese im Verzeichnis *bin*, reicht die Angabe der benötigten Klasse:

```
<% @Page Inherits="classname" %>
```

Beachten Sie, dass die Verwendung von `Response.Write` in externen Klassen wenig sinnvoll ist. Sie sollten daran denken, dass die Ausgabe im Teil `Page_Load` vor der Erstellung der Seite erfolgt. Ausgaben sollten besser mit HTML Server-Steuerelemente oder Web Server-Steuerelemente erfolgen.

Ein Beispiel zeigt, wie hinterlegter Code praktisch verwendet wird. Zuerst eine *aspx*-Datei, die nur noch HTML und Direktiven enthält:

```
<%@ Page Inherits="ButtonStyle" src="codebehind.cs" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="de">
    <head>
        <title>Schaltfl&auml;che gestalten</title>
    </head>
    <body>
        <h1>Schaltfl&auml;che gestalten:</h1>
        <form runat="server" >
            <input type="button" id="btn" runat="server"
               onServerClick="clickBtn"/>
        </form>
        <div id="Bestaetigung" runat="server"/>
    </body>
</html>
```

Listing 3.25: Verwendung externen Codes (codebehind.aspx)

Interessant ist nur die erste Zeile, die die zu verwendende Klasse und die Quelldatei festlegt. Die eigentliche Arbeit steckt in der *cs*-Datei:

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using System.Collections;

public class ButtonStyle : Page
{
    public HtmlInputButton btn;
    public HtmlGenericControl Bestaetigung;

    void Page_Load()
    {
        btn.Value = "Ich bin ein schicker Button";
        btn.Style["font-family"] = "Arial";
        btn.Style["font-weight"] = "bold";
        btn.Style["color"] = "green";
        btn.Style["font-size"] = "24pt";
    }

    public void clickBtn (object sender, EventArgs e)
    {
        HtmlInputButton btn = (HtmlInputButton) sender;
        Bestaetigung.InnerHtml = "Meine Gestaltung basiert auf:<br/>";
        IEnumerator keys = btn.Style.Keys.GetEnumerator();
        while (keys.MoveNext())
        {
            String key = (String) keys.Current;
            Bestaetigung.InnerHtml += key + "='" + btn.Style[key] + "'<br/>";
        }
    }
}
```

Listing 3.26: Code Behind-Datei (codebehind.cs)

In hinterlegtem Code müssen Sie alle Deklarationen – einschließlich der Namensräume – selbst erledigen. Die ersten Zeilen bestehen praktisch immer aus den entsprechenden `using`-Anweisungen. Danach folgt die Klasse, im Beispiel mit dem Namen `ButtonStyle`. Diese Klasse muss von `Page` abgeleitet werden. `Page` ist die aktuelle Seite und durch die Vererbung wird der Zugriff auf Steuerelemente und Eigenschaften überhaupt erst möglich. Innerhalb der Klassen stehen die Methoden `Page_Load`, `Page_Init` usw. natürlich unverändert zur Verfügung. Wollen Sie hier Code ausführen, überladen Sie die entsprechende Methode, wie im Beispiel für `Page_Load` gezeigt.

Damit aus der Seite heraus auf die instanziierten Seitenobjekte zugegriffen werden kann, wie das beispielsweise für die Schaltfläche `btn` gezeigt wird, müssen diese als `public` deklariert werden:

```
public HtmlInputButton btn;
```

Außerdem ist natürlich der richtige Datentyp anzugeben. Der Zugriff über globale Klassen, wie `HtmlControl` statt `HtmlInputControl` funktioniert zwar, schränkt aber die verfügbaren Eigenschaften und Methoden entsprechend ein.

3.9 Direktiven

Viele grundlegende Operationen lassen sich über Seitendirektiven steuern. Diese stehen am Anfang der Seite und gelten für Übersetzung oder Ausführung. Die Direktiven sollten immer an den Anfang der Seite gesetzt werden, auch wenn dies nicht in allen Fällen zwingend notwendig ist.

3.9.1 Übersicht

Folgende Direktiven sind verfügbar:

- @Page – Hiermit werden Vorgänge gesteuert, die unmittelbar mit der Ausführung der Seite zu tun haben.
- @Import – Mit dieser Direktive werden Namensräume aus dem .Net-Framework importiert. Dies ist notwendig, wenn Code Behind nicht verwendet wird, da in ASP.NET die Anweisung `using` nicht eingesetzt werden kann, mit der üblicherweise Namensräume importiert werden.
- @Implements – Diese Direktive importiert eine Schnittstellenbeschreibung (Interface) aus dem .Net-Framework.
- @Register – Hiermit werden Benutzer-Steuerelemente (User Controls) angemeldet, sodass der Compiler diese finden und einbinden kann.
- @Assembly – Die Nachfolger der DLLs in Windows sind in .Net Assemblies. Sie enthalten praktisch bereits übersetzten Code in der Intermediate Language. Mit dieser Direktive können Sie Assemblies direkt nutzen.
- @OutputCache – Jede Seite, die auf dem Server erzeugt wird, kann zwischengespeichert werden. Dies erhöht die Systemleistung und verringert die Prozessorlast. Für den korrekten Aufbau der Seite im Browser ist die die Kontrolle dieses Vorgangs wichtig. Die Direktive steuert dies.
- @Reference – Mit dieser Direktive werden Benutzer-Steuerelemente zur Laufzeit übersetzt und verbunden.

Die wichtigsten Direktiven werden nachfolgend vorgestellt – in dem Rahmen, indem sie in diesem Buch Verwendung finden.

3.9.2 Die Seitendirektive @Page

Die Seitendirektive `@Page` definiert Bedingungen zur Abarbeitung einer Seite. Sie steht immer am Seitenanfang:

```
<% @Page Attribut="Wert" ... %>
```

Die wichtigsten Attribute finden Sie in der folgenden Tabelle:

Attribut	Parameter	Bedeutung
`Buffer`	`true`, `false`	Steuert den Ausgabepuffer ein. `true` ist der Standardwert.
`ClassName`	Name	Name einer Klasse, die für diese Seite übersetzt wird, wenn mehrere Klassen zu Auswahl stehen.
`ContentType`	z.B. image/gif	Der Name eines MIME-Typs, mit dem der Inhalt gesendet werden soll.
`Debug`	`true`, `false`	Schaltet serverseitiges Debuggen ein. Der Standardwert ist `false` (aus).
`Trace`	`true`, `false`	Schaltet serverseitiges Ablaufverfolgung ein. Der Standardwert ist `false` (aus).
`Description`	Text	Infotext zur Seite, der von ASP.NET ignoriert wird.
`Inherits`	Name	Name einer Klasse einer hinterlegten Code-Datei (Code Behind).
`Src`	Name, URL	Quelle einer Code-Datei (Code Behind)
`Language`	C#, VB, JScript.Net	Name der Sprache, die im Inline-Code (<%%> -Tags) erwartet wird.
`Culture`	Name	Name des Sprachpaketes, z.B. „de-DE“, zur Einstellung der Währung, Zahlenformate usw.

Tabelle 3.2: Die wichtigsten Attribute der Direktive `@Page`

Einsatz der Seitendirektive zur Fehlersuche

Debug

Die Attribute `Debug` und `Trace` helfen bei der Fehlersuche. Mit Debug wird der Ablauf des Programms auf der Serverseite überwacht. Tritt ein Fehler auf, kann die ASP.NET detaillierte Informationen über die Ursache bereitstellen. In Produktionsumgebungen wird das Attribut entfernt, weil die Überwachung Leistung kostet. Wenn Sie in manchen Situationen unzureichende Informationen über den Fehler finden oder die Angabe der Zeile mit der Ursache fehlt, fügen Sie der Seitendirektive `debug="true"` hinzu.

Wenn Sie noch mehr Informationen über den Gesamtzustand des Systems benötigen, um die Ursache eines Laufzeitfehlers zu finden, ergänzen Sie das Attribut `trace="true"`.

Server Error in '/aspdotnet' Application.

Could not find a part of the path "C:\WINNT\system32\xx".

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.IO.DirectoryNotFoundException: Could not find a part of the path "C:\WINNT\system32\xx".

Source Error:

```
Line 23:                ausgabe.Text += "<br/>Erzeugt am : " + dinfo.CreationTime.ToShortDateString();
Line 24:                ausgabe.Text += "<br/><u>Unterverzeichnisse:</u>";
Line 25:                foreach (DirectoryInfo subdir in dinfo.GetDirectories())
Line 26:                {
Line 27:                    ausgabe.Text += "<br/>" + subdir.FullName;
```

Source File: C:\Inetpub\wwwroot\dotnet\readdir.cs **Line:** 25

Stack Trace:

```
[DirectoryNotFoundException: Could not find a part of the path "C:\WINNT\system32\xx".]
   System.IO.__Error.WinIOError(Int32 errorCode, String str) +287
   System.IO.Directory.InternalGetFileDirectoryNames(String fullPath, Boolean file) +251
   System.IO.DirectoryInfo.GetDirectories(String searchPattern) +382
   System.IO.DirectoryInfo.GetDirectories() +13
   dir.Page_Load() in C:\Inetpub\wwwroot\dotnet\readdir.cs:25
   System.Web.Util.ArglessEventHandlerDelegateProxy.Callback(Object sender, EventArgs e) +10
   System.Web.UI.Control.OnLoad(EventArgs e) +67
   System.Web.UI.Control.LoadRecursive() +29
   System.Web.UI.Page.ProcessRequestMain() +724
```

Version Information: Microsoft .NET Framework Version:1.0.3705.0; ASP.NET Version:1.0.3705.0

Session Id:	ztulde45fze5ixmkbfaywt45	**Request Type:**	POST
Time of Request:	21.4.2002 19:28:07	**Status Code:**	500
Request Encoding:	Unicode (UTF-8)	**Response Encoding:**	Unicode (UTF-8)

Category	Message	From First (s)	From Last (s)
aspx.page	Begin Init		
aspx.page	End Init	0,000090	0,000090
aspx.page	Begin LoadViewState	0,000528	0,000437
aspx.page	End LoadViewState	0,001009	0,000481
aspx.page	Begin ProcessPostData	0,001080	0,000071
aspx.page	End ProcessPostData	0,001150	0,000071
Unhandled Execution Error	Could not find a part of the path "C:\WINNT\system32\xx". at System.IO.__Error.WinIOError(Int32 errorCode, String str) at System.IO.Directory.InternalGetFileDirectoryNames(String fullPath, Boolean file) at System.IO.DirectoryInfo.GetDirectories(String searchPattern) at System.IO.DirectoryInfo.GetDirectories() at dir.Page_Load() in C:\Inetpub\wwwroot\dotnet\readdir.cs:line 25 at System.Web.Util.ArglessEventHandlerDelegateProxy.Callback(Object sender, EventArgs e) at System.Web.UI.Control.OnLoad(EventArgs e) at System.Web.UI.Control.LoadRecursive() at System.Web.UI.Page.ProcessRequestMain()	0,001811	0,000660

Control Id	Type	Render Size Bytes (including children)	Viewstate Size Bytes (excluding children)

Application Key	Type	Value
HitCounter	System.Int32	1
Version	System.String	0.9.3

Name	Value	Size
ASP.NET_SessionId	ztulde45fze5ixmkbfaywt45	42

Abbildung 3.19: Ausgabe eines Laufzeitfehlers mit `trace="true"`

Name	Value
Cache-Control	no-cache
Connection	Keep-Alive
Content-Length	185
Content-Type	application/x-www-form-urlencoded
Accept	*/*
Accept-Encoding	gzip, deflate
Accept-Language	de,en-us;q=0.8,en-au;q=0.5,en-gb;q=0.3
Authorization	Basic QWRtaW5pc3RyYXRvcjpjbGVtZW5z
Cookie	ASP.NET_SessionId=ztulde45fze5ixmkbfaywt45
Host	localhost
Referer	http://localhost/aspdotnet/file_readdir.aspx
User-Agent	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705)

Name	Value
__VIEWSTATE	dDwxMDgwMDE5NzY4O3Q8O2w8aTwzPjs+O2w8dDxwPHA8bDxUZXh0Oz47bDxHZWJlbiBTaWUgZWlu IFZlcnplaWNo
dirname	xx
senden	Anzeigen

Name	Value
ALL_HTTP	HTTP_CACHE_CONTROL:no-cache HTTP_CONNECTION:Keep-Alive HTTP_CONTENT_LENGTH:185 HTTP_CONTENT_TYPE:application/x-www-form-urlencoded HTTP_ACCEPT:*/* HTTP_ACCEPT_ENCODING:gzip, deflate HTTP_ACCEPT_LANGUAGE:de,en-us;q=0.8,en-au;q=0.5,en-gb;q=0.3 HTTP_AUTHORIZATION:Basic QWRtaW5pc3RyYXRvcjpjbGVtZW5z HTTP_COOKIE:ASP.NET_SessionId=ztulde45fze5ixmkbfaywt45 HTTP_HOST:localhost HTTP_REFERER:http://localhost/aspdotnet/file_readdir.aspx HTTP_USER_AGENT:Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705)
ALL_RAW	Cache-Control: no-cache Connection: Keep-Alive Content-Length: 185 Content-Type: application/x-www-form-urlencoded Accept: */* Accept-Encoding: gzip, deflate Accept-Language: de,en-us;q=0.8,en-au;q=0.5,en-gb;q=0.3 Authorization: Basic QWRtaW5pc3RyYXRvcjpjbGVtZW5z Cookie: ASP.NET_SessionId=ztulde45fze5ixmkbfaywt45 Host: localhost Referer: http://localhost/aspdotnet/file_readdir.aspx User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705)
APPL_MD_PATH	/LM/W3SVC/1/Root/aspdotnet
APPL_PHYSICAL_PATH	C:\Inetpub\wwwroot\dotnet\
AUTH_TYPE	Basic
AUTH_USER	W2KWS\Administrator
AUTH_PASSWORD	clemens
LOGON_USER	W2KWS\Administrator
REMOTE_USER	W2KWS\Administrator
CERT_COOKIE	
CERT_FLAGS	
CERT_ISSUER	
CERT_KEYSIZE	
CERT_SECRETKEYSIZE	
CERT_SERIALNUMBER	
CERT_SERVER_ISSUER	
CERT_SERVER_SUBJECT	
CERT_SUBJECT	
CONTENT_LENGTH	185
CONTENT_TYPE	application/x-www-form-urlencoded
GATEWAY_INTERFACE	CGI/1.1
HTTPS	off
HTTPS_KEYSIZE	
HTTPS_SECRETKEYSIZE	
HTTPS_SERVER_ISSUER	
HTTPS_SERVER_SUBJECT	
INSTANCE_ID	1
INSTANCE_META_PATH	/LM/W3SVC/1
LOCAL_ADDR	127.0.0.1
PATH_INFO	/aspdotnet/file_readdir.aspx
PATH_TRANSLATED	C:\Inetpub\wwwroot\dotnet\file_readdir.aspx
QUERY_STRING	
REMOTE_ADDR	127.0.0.1
REMOTE_HOST	127.0.0.1
REQUEST_METHOD	POST
SCRIPT_NAME	/aspdotnet/file_readdir.aspx
SERVER_NAME	localhost
SERVER_PORT	80
SERVER_PORT_SECURE	0
SERVER_PROTOCOL	HTTP/1.1
SERVER_SOFTWARE	Microsoft-IIS/5.0
URL	/aspdotnet/file_readdir.aspx
HTTP_CACHE_CONTROL	no-cache
HTTP_CONNECTION	Keep-Alive
HTTP_CONTENT_LENGTH	185
HTTP_CONTENT_TYPE	application/x-www-form-urlencoded
HTTP_ACCEPT	*/*
HTTP_ACCEPT_ENCODING	gzip, deflate
HTTP_ACCEPT_LANGUAGE	de,en-us;q=0.8,en-au;q=0.5,en-gb;q=0.3
HTTP_AUTHORIZATION	Basic QWRtaW5pc3RyYXRvcjpjbGVtZW5z
HTTP_COOKIE	ASP.NET_SessionId=ztulde45fze5ixmkbfaywt45
HTTP_HOST	localhost
HTTP_REFERER	http://localhost/aspdotnet/file_readdir.aspx
HTTP_USER_AGENT	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705)

Abbildung 3.19: Ausgabe eines Laufzeitfehlers mit `trace="true"` (Forts.)

3.9.3 Die Direktive @Import

Mit dieser Direktive werden weitere Namensräume eingebunden. Sie kann so oft angewendet werden, wie Namensräume notwendig sind:

```
<% @Import namespace="System.Data" %>
```

ASP.NET importiert einige Namensräume automatisch, sodass die Anwendung nicht immer notwendig ist:

- System
- System.Collections
- System.Collections.Specialized
- System.Configuration
- System.IO
- System.Text
- System.Text.RegularExpressions
- System.Web
- System.Web.Caching
- System.Web.Security
- System.Web.SessionState
- System.Web.UI
- System.Web.UI.HtmlControls
- System.Web.UI.WebControls

Wenn `@Import` erforderlich ist, wird es in diesem Buch explizit erwähnt.

3.9.4 Die Direktive @Register

Mit dieser Direktive werden Benutzer-Steuerelemente (User Controls) der Seite bekannt gemacht:

```
<% @Register tagprefix="tagprefix" Tagname="tagname" Src="pathname" %>
```

Festgelegt wird der Präfix, den das Element auf der Seite verwendet, der Name und der Pfad, wo die *ascx*-Datei gespeichert wurde. Mehr Informationen dazu finden Sie im Abschnitt „Benutzer-Steuerelemente (User Controls)“ auf Seite 257.

3.10 Das Projekt

Im Laufe dieses Buches soll ein Projekt entwickelt werden, das wesentliche Teile der Programmierprinzipien enthält und eine gute Basis für eigene Arbeiten ist. Es wird nur soweit an dieser Stelle ausgebaut, wie die bereits gezeigten Techniken es erlauben.

Das Projekt realisiert einen einfachen Shop. In der ersten Stufe werden folgende Leistungen von der Applikation erwartet:

- Startseite mit verschiedenen Links zu statischen Seiten
- Link zum Shop
- Anzeige der Produkte des Shops mit Preisen; Auswahlmöglichkeit eines Produkts durch den Benutzer

Die fertigen Dateien finden Sie auf der CD im Verzeichnis *shop*.

3.10.1 Die nächsten Schritte

Im nächsten Kapitel wird dann ein Formular zur Vervollständigung der Bestellung angezeigt. Außerdem soll die Wahl der Menge möglich sein und eine Anzeige der Bestätigung erfolgen. Weiterhin sollen die vorerst im Code definierten Artikel in leichter erreichbare Dateien ausgelagert werden. Die eigentliche Ausführung der Bestellung wird erst später eine Rolle spielen. Zuerst wird dies mit E-Mail und Textdateien erfolgen, danach mit einer Datenbank und elementaren Administrationsfunktionen.

Der Shop enthält einige Softwareprodukte. Der Käufer soll nach der Bestellung zum einen eine Bestellbestätigung erhalten. Zum andere soll er über einen speziellen Link mit einer Codenummer zu der Stelle geführt werden, wo die Software geladen werden kann. Der Vorgang des Herunterladens ist in der Bestellinformation zu vermerken.

3.10.2 Strategische Planung

Ein solches Projekt verlangt in Anbetracht der Endausbaustufe eine gute Planung. Der erste Schritt soll nur einige einfache Funktionen enthalten. Diese teilen sich – mindestens – in folgende Seiten auf:

- Startseite
- Produktseite
- Die Startseite dient hier nur der Einführung und ist vorläufig nur HTML:

```
<html lang="de">
    <head>
        <title>Software-Shop</title>
    </head>
    <body>
        <h1> Software-Shop </h1>
        <ul>
            <li><a href="agb.aspx">AGB</a></li>
            <li><a href="kontakt.aspx">Kontakt</a></li>
            <li><a href="shop.aspx">Shop</a></li>
            <li><a href="support.aspx">Support</a>

            <ul>
                <li><a href="download.aspx">Download</a></li>
                <li><a href="faq.aspx">FAQ</a></li>
            </ul>
            </li>
        </ul>

    </body>
</html>
```

Listing 3.27: Startseite des Softwareshops (start.aspx)

Die Produktseite enthält die Definition der Produkte. Die Speicherung der Produkte erfolgt in einem Array. Die Ausgabe erfolgt direkt in ein HTML Server-Steuerelement:

```
<script language="C#" runat="Server">

void Page_Load()
{
    string[,] ProduktArr = new string[3,3];
    const int ProdName = 0;
    const int ProdPreis = 1;
    const int ProdID = 2;

    ProduktArr[0,ProdName] = "Codeschreiber 1.0";
    ProduktArr[0,ProdPreis] = "6299,9";
    ProduktArr[0,ProdID] = "A0001";

    ProduktArr[1,ProdName] = "Datenbanker 1.1";
    ProduktArr[1,ProdPreis] = "99,9";
    ProduktArr[1,ProdID] = "A0002";

    ProduktArr[2,ProdName] = "Projektor 2.0";
    ProduktArr[2,ProdPreis] = "39";
    ProduktArr[2,ProdID] = "A0003";
```

```
    produktliste.Text += "<ul>";
    for (int i=0; i < ProduktArr.GetLength(0); i++)
    {
        produktliste.Text += "<li>" + ProduktArr[i, ProdName] + " (&euro; "
                          + ProduktArr[i, ProdPreis] + ")";
    }
    produktliste.Text += "</ul>";
}

</script>

<html lang="de">
    <head>
        <title>Shop</title>
    </head>
    <body>
        <h1> Shop </h1>
        Unsere Produkte:
        <asp:label id="produktliste" runat="server"/>
    </body>
</html>
```

Listing 3.28: Erster Entwurf, Produktliste aus einem Array erzeugen (shop.aspx)

Das Array wird hier zuerst mit einer festen Grenze definiert. Damit der Zugriff auf die Elemente besser lesbar wird, werden für die Indizes außerdem Konstanten definiert:

```
const int ProdName = 0;
```

Dann werden die Produkte definiert. Im HTML-Code befindet sich ein Web Server-Steuerelement, dass schon mehrfach verwendet wurde:

```
<asp:label id="produktliste" runat="server"/>
```

Diesem Element werden die Daten in Form einer unsortierten Liste zugewiesen, jeweils durch Zugriff auf die Eigenschaft `Text`:

```
produktliste.Text = "<ul>";
```

Dann folgt eine Schleife, mit der das Array durchlaufen wird. Jeder Artikel ergibt einen Schleifendurchlauf:

```
for (int i=0; i < ProduktArr.GetLength(0); i++)
```

Die Methode `GetLength` gibt die Anzahl der Elemente einer bestimmten Dimension des Arrays zurück. In der Schleife werden dann `<li>`-Tags erzeugt, die die Produktnamen und Preise enthalten:

```
produktliste.Text += "<li>" + ProduktArr[i, ProdName] + " (&euro; "
                  + ProduktArr[i, ProdPreis] + ")";
```

Für einen Shop ist die Ausgabe sicher noch wenig beeindruckend. Der Code ist aber eine gute Basis für die nächsten Schritte.

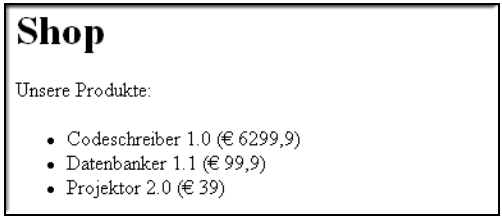


Abbildung 3.20: Ansicht der Produktliste

Mit den im nächsten Kapitel vermittelten Informationen über Formulare und Server-Steuerelemente wird der Shop endlich interaktiv. Im Kapitel 6 werden dann die Datenbankfunktionen das Projekt abrunden.

3.11 Fragen und Übungsaufgaben

1. Nennen Sie die Objekte, die für die Übertragung der Daten von und zum Server verantwortlich sind.
2. Schreiben Sie ein Programm, dass die IP-Adresse des Benutzers ermittelt und ihm anzeigt.
3. Schreiben Sie ein Programm, dass Benutzerzugriffe eindeutig zählt. Ein einmal erkannte IP-Adresse soll nicht erneut gezählt werden.

   Tipp: Verwenden Sie das Objekt `Application`.
4. Schreiben Sie ein Programm mit zwei HTML-Seiten und einer hinterlegten Code-Datei (Code Behind). Die erste Seite soll eine Liste von Farben zur Auswahl stellen. Die Auswahl soll in einem Cookie gespeichert werden. Ruft der Benutzer jetzt oder später die zweite Seite auf, erscheint diese in der ausgewählten Farbe. Das Cookie soll einen Tag gültig bleiben.

   Tipp: Überprüfen Sie im Browser die korrekter Werte des Cookies.

4 Webformulare: Web Forms

lernen

Dieses Kapitel behandelt den Aufbau und die Programmierung von Webformularen. ASP.NET unterstützt Formulare umfassend und durch mehrere Arten von Steuerelementen. Diese werden systematisch anhand praktischer Beispiele vorgestellt.

4.1 Was Sie in diesem Kapitel lernen

Im letzten Kapitel wurden die Grundlagen der Programmierung dynamischer Webseiten mit ASP.NET erklärt und es wurde bereits angedeutet, dass der Behandlung von Formularen eine besondere Bedeutung zukommt. In diesem Kapitel geht es um HTML im Allgemeinen und Formulare im Besonderen.

ASP.NET bietet hier eine völlig neue Art der Webserverprogrammierung, die sich in ihrer Art der klassischen GUI-Programmierung (Erstellung von Windows-Anwendungen) immer mehr annähert. Dennoch werden die zugrunde liegenden Protokolle nicht umgangen – proprietäre Erweiterungen sind nicht zu finden.

In diesem Kapitel werden folgende Themen detailliert behandelt:

- *HTML Server-Steuerelemente* – Wie Sie HTML-Tags erzeugen, verändern und entfernen, direkt aus dem Programm heraus.
- *Web Server-Steuerelemente* – Komplexe Darstellungen mit nur einem Tag, das ist der Traum der Webprogrammierer. Leistungsfähige Tags, ohne das Prinzip der HTML-Kompatibilität zu verletzen, das kann mit diesen Steuerelementen erreicht werden.
- *Überwachung von Feldinhalten* – Mit Kontroll-Steuerelementen können Sie sehr einfach und effektiv den Inhalt von Formularen überwachen.

Für fortgeschrittende Leser wurde auch ein Abschnitt über Benutzer-Steuerelemente aufgenommen, ein ideales Werkzeug zur Modularisierung von größeren Anwendungen und ein Ausblick auf die Möglichkeiten, die ASP.NET wirklich bietet.

Wir sprechen deutsch!

Zwei Dinge zum Verständnis sollten vorab erwähnt werden. Zum einen wurden fast durchgehend deutsche Begriffe für die entsprechenden Techniken verwendet, auch wenn dies in der derzeitigen Fachliteraturlandschaft einmalig ist. Es gibt aber nach Meinung des Verfassers keinen Grund, gedankenlos englische Wörter zu übernehmen und damit die Lesbarkeit sinnlos zu erschweren. Die englischen Herkunftswörter sind dennoch mit erwähnt, damit die Suche in der englischen Referenz und im Googlenet nicht erschwert wird.

Zum anderen ist dieses Kapitel anspruchsvoll und verlangt aktive Mitarbeit. Sie sollten jedes Beispiel ausprobieren, verändern, die Reaktion beobachten und solange damit spielen, bis Sie Ihr Ziel erreicht haben und das Programm fehlerfrei läuft und Sie auch wissen, warum des funktioniert.

4.2 HTML Server-Steuerelemente (HTML Controls)

Namensraum

In der Klassenhierarchie des .NET-Frameworks werden die HTML Server-Steuerelemente im Namensraum `System.Web.UI.HtmlControls` definiert. UI steht für User Interface, es handelt sich also um eine Klasse, die der Gestaltung der Benutzerschnittstelle dient. Auch dieser Namensraum wird von ASP.NET automatisch bereitgestellt. Lediglich beim Einsatz der Code Behind-Technik müssen Sie den Zugriff explizit deklarieren. Dann schreiben Sie an den Anfang der Datei folgende Zeile:

```
using System.Web.UI.HtmlControls;
```

Ereignisse

Wie viele andere Objekte, kennen die HTML Server-Steuerelemente Ereignisse. Mit entsprechend benannten Methoden können Sie im Code auf diese Ereignisse reagieren. Der Name ist frei wählbar, er wird im HTML-Element selbst als Attribut `id` festgelegt. Beachten Sie, dass dieses Verarbeitungsprinzip nichts am üblichen Ablauf zwischen Browser und Webserver ändert. Wenn Sie auf ein Klick-Ereignis reagieren möchten, wird der Browser das Formular an den Server absenden. Wenn der Benutzer einen Link anklickt, wird er den GET-Befehl ausführen, wenn der Link außerhalb eines Formulars liegt. Innerhalb wird JavaScript verwendet, um die im Link verknüpften Informationen per POST zu übertragen. In jedem Fall handelt sich also um eine Anforderung (Request) des Clients. Damit der Server nun zuordnen kann, welches Formular

zurückgesendet wird, wird ein verstecktes Feld mit Statusinformationen erzeugt, dem so genannten Viewstate. Der Viewstate kann für jedes Element ein- und ausgeschaltet werden. Bei Links müssen Sie selbst für die exakte Zuordnung sorgen, wenn es erforderlich ist. Allerdings können Sie dafür Sitzungen verwenden, die entweder mit Cookies oder dem Querystring arbeiten. Im Detail wird das anhand der folgenden Beispiele gezeigt.

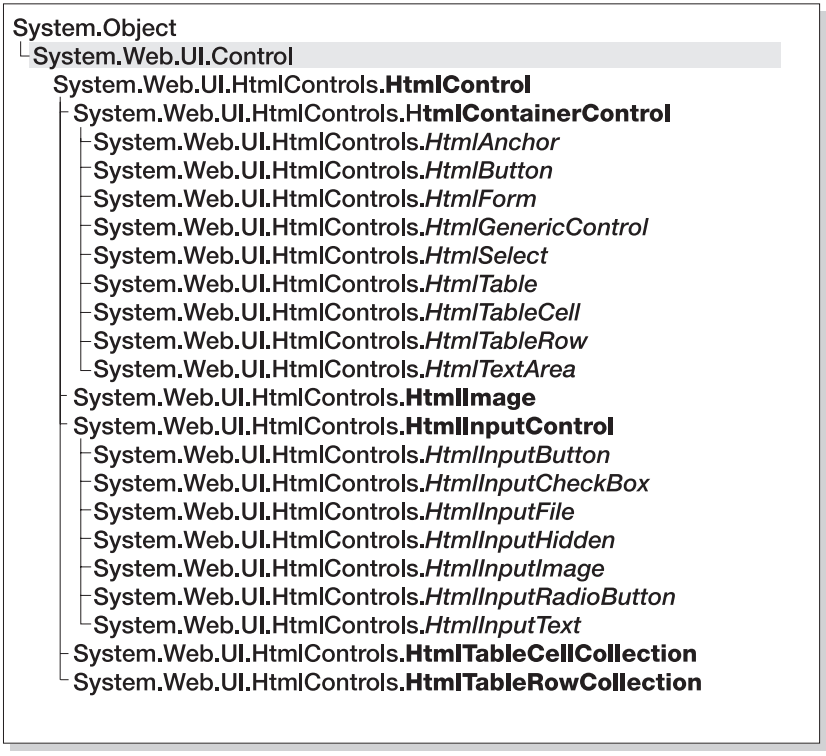


Abbildung 4.1: HtmlControl im Framework und untergeordnete Klassen

Der vollkommen auf HTML und HTTP aufbauende Verarbeitungszyklus der HTML Server-Steuerelemente führt auch zu einer sehr wichtigen Aussage: Alle Steuerelemente sind vollständig HTML-kompatibel und lassen sich mit allen Browsern darstellen. Es handelt sich nicht um Nachfahren der ActiveX-Controls oder ähnlicher proprietärer Erweiterungen.

4.2.1 Prinzipieller Umgang mit HTML Server-Steuerelementen

Da jedes HTML Server-Steuerelement eine Entsprechung in HTML besitzt, ist eine direkte Gegenüberstellung der Namen mit den Tags sinnvoll. Die folgende Tabelle zeigt die explizit vorhandenen Klassen und die passenden HTML-Tags:

HTML-Tag	Klassenname
`<a>`	`HtmlAnchor`
`<button>`	`HtmlButton`
Alle sonstigen Tags, die als Container auftreten, beispielsweise `<b>`, `<i>` usw.	`HtmlContainerControl`
Alle sonstigen Tags, die alleinstehend auftreten, beispielsweise ` `	`HtmlControl`
`<form>`	`HtmlForm`
Dient der Generierung neuer Elemente	`HtmlGenericControl`
`<img>`	`HtmlImage`

`<input type="button">`
`<input type="reset">`
`<input type="submit">`
	`HtmlInputButton`

`<input type="checkbox">`	`HtmlInputCheckBox`

`<input type="text">`
`<input type="file">`
`<input type="submit">`
	`HtmlInputControl`

`<input type="file">`	`HtmlInputFile`
`<input type="hidden">`	`HtmlInputHidden`
`<input type="image">`	`HtmlInputImage`
`<input type="radio">`	`HtmlInputRadioButton`
`<input type="text">` `<input type="password">`	`HtmlInputText`
`<select>`	`HtmlSelect`
`<textarea>`	`HtmlTextArea`
`<input type="checkbox">`	`HtmlCheckBox`
`<table>`	`HtmlTable`
`<td>`	`HtmlTableCell`
`<tr>`	`HtmlTableRow`
Mehrere Zellen einer Reihe	`HtmlTableCellCollection`
Mehrere Reihen einer Tabelle	`HtmlTableRowCollection`
`<tr>`	`HtmlTableRow`

Tabelle 4.1: HTML-Tags und Zuordnung der Klassen des Namensraumes `HtmlControls`

Der Schwerpunkt liegt eindeutig bei Formularen, die aber erfahrungsgemäß auch den meisten Aufwand verursachen und die von der Unterstützung in ASP.NET am ehesten profitieren. Eine Sonderstellung nimmt `HtmlGenericControl` ein, eine Klasse, die der Erzeugung neuer Objekte dient, die dann der Elemente-Kollektion der Seite hinzugefügt werden können. Angesprochen wird das verfügbare korrespondierende Objekt natürlich nicht über seinen Klassennamen, sondern über die Instanz. Der Name der automatisch gebildeten Instanz wird aus dem Attribut `id` abgeleitet.

Besteht die gesamte Seite nach der Analyse aus HTML Server-Steuerelementen, ist die Manipulation leicht möglich. Betrachtet man die Elemente als Hierarchie, ist auch die Anordnung an einer bestimmten Stelle möglich. So kann HTML sehr gut programmatisch manipuliert werden.

Universeller Zugriff auf HTML-Tags

HTMLControl
HTMLContainer-Control

Wie bereits in der Tabelle angedeutet, gibt es nur für die wichtigsten HTML-Tags eine direkte Entsprechung in Form einer eigenen Klasse. Der Zugriff ist aber generell auf alle Tags möglich, wenn diese mit `runat="server"` zum Bestandteil der Hierarchie gemacht werden. Diese Elemente werden automatisch von der abstrakten Klasse `HtmlControl` bzw. `HtmlContainerControl` abgeleitet. Der Unterschied besteht lediglich darin, wie die Elemente auftreten dürfen. So sind einige zwingend allein stehend, wie beispielsweise `<hr/>` oder `<br/>`. Containerelemente treten paarweise auf, dazu gehören `<b></b>` usw.

Eine wichtige Eigenschaft ist `Attributes`. Dies ist eine Kollektion, da Elemente mehr als ein Attribut enthalten können. Das folgende Beispiel zeigt die Anwendung:

```
<script language="C#" runat="server">
void Page_load()
{
   head1.Attributes["style"] = "color:" + Request.QueryString["header"];
}
</script>
<html>
  <head>
        <title>HTMLControl</title>
  </head>
  <body>
    <h1 style="" id="head1" runat="Server">Willkommen auf unserer Website
    </h1>
    Viel Spaß mit unseren Angeboten:
```

```
    <ul>
        <li><a href="htmlcontrol.aspx?header=blue">Blaue Überschrift sehen
            </a>
        </li>
        <li><a href="htmlcontrol.aspx?header=red">Rote Überschrift sehen
            </a>
        </li>
        <li><a href="htmlcontrol.aspx?header=green">Grüne Überschrift sehen
            </a>
        </li>
    </ul>
  </body>
</html>
```

Listing 4.1: Anwendung von HTML Server-Steuerelemente (htmlcontrol.aspx)

Wie es funktioniert

Zum HTML Server-Steuerelement wird hier das Tag `<h1>` bestimmt. Manipuliert wird das Attribut `style`, mit dem sich die Gestaltung über Cascading Style Sheets (CSS) bestimmen lässt. Die Definition erfordert neben dem Attribut `runat="server"` auch die Vergabe eines Namens über `id`:

```
<h1 style="" id="head1" runat="Server">
```

Der Name des Objekts ist nun `head1`. Der Zugriff auf ein bestimmtes Attribut erfolgt über die bereits verwendete Schreibweise für Kollektionen:

```
head1.Attributes["style"] =
```

Die Attribute sind sowohl schreib- als auch lesbar.

Umgang mit der Objekthierarchie

HTML-Tags können innerhalb der Seite nicht nur sequenziell, sondern auch hierarchisch auftreten:

```
<b><i></i></b>
```

In dieser Folge ist `<i>` dem Tag `<b>` untergeordnet. Da alle Objekte innerhalb der Seite instanziiert werden, spielt das nicht zwingend eine Rolle bei der Programmierung. Wenn Sie aber auf die Hierarchie Rücksicht neben müssen, gibt es einen Weg, die Anordnung zu ermittelt. `HtmlControl` kennt dazu die Eigenschaft `ControlCollection`. Dies ist eine Kollektion der Objekte, die dem betreffenden Objekt untergeordnet sind. Der Effekt ist natürlich nur sichtbar, wenn es sich um Containerelemente handelt. Einfache Tags können in der Kollektion zwar selbst enthalten sein, haben aber selbst keine Kindelemente.

Neben dem Zugriff auf Tags kann natürlich auch auf jedes Attribut zugegriffen werden. Da häufig CSS verwendet wird und das `style`-Attribut in HTML wiederum viele Anweisungen enthalten kann, kann zusätzlich auf jeden einzelnen Stylewert zugegriffen werden. Beides, Attribute und Styles, bilden Kollektionen, wenn mehrere Werte auch vorhanden sind.

Attribute

Damit bei der Abfrage das Programm nicht hängenbleibt, wenn ein Element ohne die entsprechenden Attribute oder Kindelemente angetroffen wird, ist noch die Methode `HasControls` von Bedeutung. Sie gibt `true` zurück, wenn weitere Kindelemente existieren. Typisch für den Umgang mit Hierarchien sind rekursive Methoden. Diese rufen sich selbst auf und können beliebig tiefe Verschachtelungen analysieren. Das folgende Beispiel zeigt eine solche Anwendung:

```
<script language="C#" runat="server">
void Page_load()
{
      getControls(Page);
}

void getControls(Control c)
{
    if (c.HasControls())
    {
        if (c is HtmlControl)
        {
            HtmlControl hc = (HtmlControl) c;
            ausgabe.Text += ("&lt;" + hc.TagName);
            ICollection hk = hc.Attributes.Keys;
            foreach(string ss in hk)
            {
                ausgabe.Text += (" " + ss + "=\"" + hc.Attributes[ss] + "\"");
            }

            ausgabe.Text += ("&gt; (ID = \"" + hc.ClientID + "\")<br/>");
        }
        if (c.HasControls())
        {
            foreach(Control cs in c.Controls)
            {
                getControls(cs);
            }
        }
    }
}
</script>
<html>
```

```
  <head>
        <title>HTMLControl</title>
  </head>
  <body>
    <h1>Willkommen auf unserer Website</h1>
    Viel Spaß mit unseren Angeboten:
    <ul runat="server" type="disk">
      <li runat="server" style="color:red">Blaue Überschrift sehen</li>
      <li runat="server" type="square" style="color:blue; font-weight:bold">
           Rote <b runat="server">Überschrift</b> sehen</li>
      <li runat="server">Grüne Überschrift sehen</li>
    </ul>
    Servercontrols:<br/>
    <pre><asp:label id="ausgabe" runat="server"/></pre>
  </body>
</html>
```

Listing 4.2: Auslesen aller HTML Server-Steuerelemente einer Seite mit Attributen (htmlcontrolcontainer.aspx)

Wie es funktioniert

Die Tags, die untersucht werden können, sind im HTML-Teil des Beispiels mit `runat="Server"` gekennzeichnet. Die Ausgabe der Analyse erfolgt mit einem Web Server-Steuerelement:

```
<asp:label id="ausgabe" runat="server"/>
```

Web Server-Steuerelemente werden im nächsten Abschnitt beschrieben. Der Einsatz erfolgt hier, damit das Ausgabe-Tag nicht selbst zum Bestandteil der Hierarchie wird und die Behandlung damit unnötig erschwert.

Das eigentliche Programm startet in `Page_Load`. Hier wird die Methode `getControls` aufgerufen, als Argument wird die Seite selbst übergeben. Der Zugriff auf `Page` wäre zwar auch direkt möglich, die Methode soll sich aber rekursiv aufrufen und dabei wechselt der Parameter später.

Rekursive Aufrufe werden oft verwendet, wenn eine beliebig verschachtelte Struktur durchlaufen werden soll. Dabei ruft sich eine Methode selbst auf, bis das Ende der Elemente erreicht ist.

Die Definition erwartet, dass ein Wert vom Typ `Control` übergeben wird:

```
void getControls(Control c)
```

Damit wird beim ersten Aufruf eine implizite Typumwandlung erforderlich, die C# selbstständig erledigt. In der Variablen `c` stehen jetzt nur noch alle Objekte vom Typ `Control`. Nun wird überprüft, ob das Argument weitere Steuerelement enthält:

```
if (c.HasControls())
```

Ist das der Fall, wird untersucht, ob es sich um HTML Server-Steuerelemente vom Typ `HtmlControl` handelt. Nicht alle Kindelemente des Objekts `Page` müssen dies sein:

```
if (c is HtmlControl)
```

Wenn das so ist, wird eine explizite Typumwandlung durchgeführt; das Objekt wird mehrfach benötigt. Beachten Sie, dass dies unbedingt erforderlich ist, da die Tatsache, dass es sich um ein HTML Server-Steuerelement handelt, noch nicht zur Umwandlung des Typs `Control` führt. Sie können nicht erwarten, dass der Compiler dies erkennt, da es aus Sicht der Programmlogik nicht zwingend eindeutig ist:

```
HtmlControl hc = (HtmlControl) c;
```

`hc` ist nun tatsächlich ein Objekt vom Typ `HtmlControl`, das auch sicher ein HTML Server-Steuerelement ist. Diese Objekte besitzen eine Reihe von Eigenschaften; die wichtigsten werden am Ende des Abschnitts zusammengefasst. Zuerst wird der Name des Tags ausgelesen und dem Anzeigeelement zugewiesen:

```
ausgabe.Text += ("&lt;" + hc.TagName);
```

Zum Auslesen gibt es viele Verfahren. Hier wird die Kollektion über die Schlüssel erreicht. Die Variable `hk` bildet eine einfache Kollektion vom Typ `ICollection`. Die Eigenschaft `Keys` gibt die Schlüssel aus:

```
ICollection hk = hc.Attributes.Keys;
```

Nun werden alle Attribute anhand der Schlüssel mit einer `foreach`-Schleife durchlaufen. Hier wird direkt auf die Attribute zugegriffen. Dies ist möglich, da die Kollektion intern ein C#-Indexer ist. Die Betrachtung der vielfältigen Möglichkeiten von Kollektionen finden Sie in „Aufzählungen und Kollektionen“ auf Seite 288.

```
ausgabe.Text += (" " + ss + "=\"" + hc.Attributes[ss] + "\"");
```

Eine weitere Eigenschaft ermittelt die interne ID des Elements. Die Vergabe einer ID ist nicht zwingend notwendig, wenn dies auch dringend zu empfehlen ist. ASP.NET vergibt selbstständig eindeutige IDs. In diesem Beispiel sollen diese angezeigt werden. Dazu wird die Eigenschaft `ClientID` abgefragt. Wird keine ID angegeben, sind `ID` und `ClientID` identisch:

```
ausgabe.Text += ("&gt; (ID = \"" + hc.ClientID + "\"");
```

Der letzte Teil der Methode `getControls` dient dem rekursiven Aufruf. Dazu wird wieder überprüft, ob das Objekt weitere untergeordnete Steuerelemente enthält:

```
if (c.HasControls())
```

Ist das der Fall, werden diese mit `foreach` durchlaufen und die Methode für jedes Element aufgerufen. Durch die Rekursion wird dieser Prozess beliebig tief fortgesetzt. Irgendwann ist `HasControls` nicht mehr `true`. Dann läuft die Methode zum Ende durch und löst die rekursiven Aufrufe wieder auf.

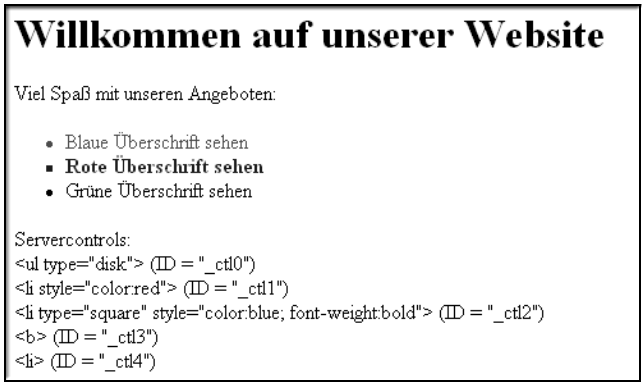


Abbildung 4.2: Ausgabe serverseitig verwalteter Elemente mit Attributen

Methoden und Eigenschaften

Wenn Sie derartige Zugriffe auf die HTML Server-Steuerelemente entwerfen, sollten Sie die wichtigsten Methoden und Eigenschaften kennen. Die Tabellen am Ende des Abschnitts fassen diese zusammen.

Container verhalten sich im Prinzip ebenso, zusätzlich kann aber der Inhalt kontrolliert werden.

Besonderheiten der Container-Steuerelemente

Container-Steuerelemente können Text enthalten. Wenn Sie beispielsweise eine Fehlermeldung nur bei Bedarf anzeigen lassen möchten, können Sie den Text vom Programm aus leicht beeinflussen. Das folgende Beispiel zeigt die Anwendung:

```
<script language="C#" runat="server">
void Page_load()
{
    switch (Convert.ToInt32(Request.QueryString["test"]))
    {
        case 0:
            errormessage.InnerText = "";
            break;
        case 1:
            errormessage.InnerText = "Super, das war richtig";
            break;
        case 2:
```

```
            errormessage.InnerText = "Schade, falscher Link";
            break;
    }
}
</script>
<html>
  <head>
        <title>HTMLControl</title>
  </head>
  <body>
    <h1>Willkommen auf unserer Website</h1>
    <div style="background-color:#eeeee; font-family:Arial"
          runat="Server" id="errormessage"/>
    Wählen Sie den den richtigen Link:<br/>
    <ul>
        <li><a href="htmlcontainer.aspx?test=2">Versuch 1</a></li>
        <li><a href="htmlcontainer.aspx?test=1">Versuch 2</a></li>
        <li><a href="htmlcontainer.aspx?test=2">Versuch 3</a></li>
    </ul>
  </body>
</html>
```

Listing 4.3: Anwendung von Container Controls

Wie es funktioniert

Die Seite enthält einen serverseitigen Container:

```
<div runat="Server" id="errormessage"/>
```

Auf das Objekt mit dem Namen `errormessage` kann nun außer mit den bereits bei `HtmlControl` beschriebenen Eigenschaften und Methoden auch mit den Eigenschaften `InnerText` und `InnerHtml` zugegriffen werden. Damit ist die Manipulation des eingeschlossenen Textes möglich.

InnerText
InnerHtml

Die Eigenschaft `InnerText` schreibt den Text nicht nur in das Tag, sondern maskiert auch die für HTML wichtigen Zeichen, „<“, „>“ und „&“; durch die Entitäten „<“, „>“ und „&“ dargestellt. Wenn HTML ausgegeben werden soll, ist deshalb `InnerHtml` zu verwenden.

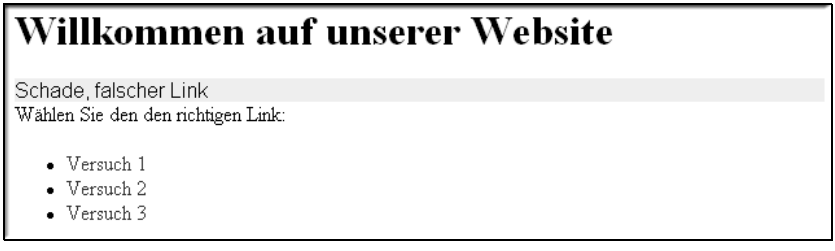


Abbildung 4.3: Erzeugen von Text zur Laufzeit

Gemeinsame Eigenschaften und Methoden der HTML Server-Steuerelemente

Dieser Abschnitt zeigt die wichtigsten Eigenschaften und Methoden der HTML Server-Steuerelemente in tabellarischer Form.

Eigenschaft	Bedeutung
`Attributes`	Kollektion der Attribute
`Style`	Kollektion der Style-Anweisungen des Attributes `style`
`Controls`	Kollektion der untergeordneten Control-Objekte
`TagName`	Name des Tags
`Parent`	Zugriff auf das übergeordnete Control-Objekt
`ID`	ID wie im Attribut `id` angegeben
`ClientID`	Automatisch vergebene, interne ID
`Visible`	Steuert die Sichtbarkeit, `true` = Sichtbar, `false` = Unsichtbar
`Disabled`	Steuert, ob das Element aktiv ist. Dies ist nicht bei allen Elementen anwendbar.

Tabelle 4.2: Wichtige Eigenschaften von `HtmlControl`

Methode	Bedeutung
`ToString`	Umwandlung in eine Zeichenkette
`RenderControl`	Schreibt das veränderte Element in ein Objekt vom Typ
`HtmlTextWriter`	zur späteren Ausgabe
`DataBind`	Bindet Datenbankinformationen an das Element (nur für Listenelemente und Tabellen interessant)

Tabelle 4.3: Wichtige Methoden von `HtmlControl`

Der Umgang mit Datenbankinformationen wird in Kapitel 6 behandelt.

Basisoperationen mit Steuerelementen

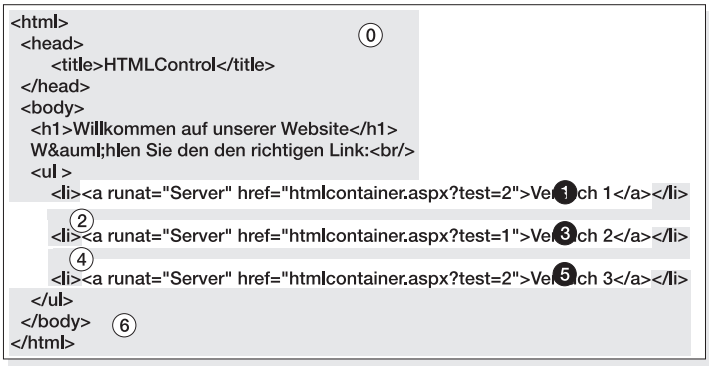
Wenn die gesamte Seite eine Hierarchie von Server-Steuerelementen darstellt, und Sie diese manipulieren können, ist auch das Entfernen und Erzeugen möglich.

Steuerelemente entfernen

LiteralControl

Das Entfernen ist mit den Mitteln einer Kollektion möglich. Alle Steuerelemente einer Ebene sind in solchen Kollektionen vom Typ `ControlCollection` enthalten. Zum Entfernen wird die Methode `Remove` verwendet, die ein Steuerelement als Argument erwartet, bzw. `RemoveAt`, wo ein numerischer Index erfoderlich ist. Wenn alle Elemente entfernt werden sollen, kann `Clear` eingesetzt werden. Der Index ist leicht zu ermitteln. In der Objekthierarchie tauchen nur die Steuerelemente explizit auf, die Sie mit `runat="server"` gekennzeichnet haben. Alles übrige wird als Ob-

jekt vom Typ `LiteralControl` verfügbar gemacht. Wie das bei einem typischen HTML-Quelltext aussieht, ist der folgenden Abbildung zu entnehmen.




```
<html>
 <head>
     <title>HTMLControl</title>
 </head>
 <body>
  <h1>Willkommen auf unserer Website</h1>
  W&auml;hlen Sie den den richtigen Link:<br/>
  <ul >
     <li><a runat="Server" href="htmlcontainer.aspx?test=2">Ver ch 1</a></li>
     <li><a runat="Server" href="htmlcontainer.aspx?test=1">Ver ch 2</a></li>
     <li><a runat="Server" href="htmlcontainer.aspx?test=2">Ver ch 3</a></li>
  </ul>
 </body>
</html>
```

Abbildung 4.4: Indizierung der Controls einer HTML-Seite

Als `HtmlControl` oder `HtmlContainerControl` werden nur die Elemente erfasst, die entsprechend gekennzeichnet sind. Alles übrige wird zusammengefasst und steht als `LiteralControl` als Text zur Verfügung. Die interne Indizierung beginnt mit 0. In der Abbildung sind dies die Elemente 0, 2, 4 und 6. Um beispielsweise aus dem gezeigten Code den zweiten Link zu entfernen, eignet sich folgende Vorgehensweise:

```
<script language="C#" runat="server">
void Page_load()
{
    if (Page.HasControls())
    {
        Page.Controls.RemoveAt(3);
    }
}
</script>
```

Listing 4.4: Entfernen eines HTML Server-Steuerelemente (htmlcontrol_CollectionRemove.aspx)

Hinzufügen

Das Hinzufügen von Elementen ist ebenso möglich, setzen Sie dazu die Methode `Add` ein:

```
<script language="C#" runat="server">
void Page_load()
{
    if (Page.HasControls())
    {
        HtmlGenericControl LiElement= new HtmlGenericControl("li");
        LiElement.InnerText = "Neues Element";
```

```
        Liste.Controls.Add(LiElement);
    }
}
</script>
<html>
  <head>
        <title>HTMLControl</title>
  </head>
  <body>
    <h1>Willkommen auf unserer Website</h1>
    Wählen Sie den den richtigen Link:<br/>
    <ul runat="Server" id="Liste">
        <li runat="Server"><a href="htmlcontainer.aspx?test=2">Versuch 1</a>
        </li>
        <li runat="Server"><a href="htmlcontainer.aspx?test=1">Versuch 2</a>
        </li>
        <li runat="Server"><a href="htmlcontainer.aspx?test=2">Versuch 3</a>
        </li>
    </ul>
  </body>
</html>
```

Listing 4.5: Hinzufügen von HTML-Code (htmlcontrol_CollectionAdd.aspx)

Wie es funktioniert

Hier wird zuerst ein neues Steuerelement erzeugt:

```
HtmlGenericControl LiElement= new HtmlGenericControl("li");
```

Dem Konstruktor wird dabei der Name des Elements als Zeichenkette übergeben, in diesem Fall entsteht also ein `<li>`-Tag. Dieses Tag ist ein Container und kann deshalb mit Text gefüllt werden:

```
LiElement.InnerText = "Neues Element";
```

Dann wird das Element der Kollektion der Steuerelemente des Tags `<ul>` hinzugefügt. Die Liste wird über den Namen adressiert:

```
Liste.Controls.Add(LiElement);
```

Nach Ausführung der Seite werden vier Listenelemente angezeigt:

Abbildung 4.5: Seite mit programmatisch erzeugtem HTML (htmlcontrol_CollectionAdd.aspx)

4.2.2 Klickereignisse verarbeiten

Für die Übertragung von Daten dienen normalerweise Formulare, denn die Erhaltung der Werte erfolgt über den Viewstate, der nur als verstecktes Feld zur Verfügung steht. Manchmal ist es jedoch aus gestalterischen oder funktionalen Gründen notwendig, Aktionen über einen Hyperlink auszulösen.

Im Querystring können nur maximal 2.000 Zeichen übertragen werden. Da der Umfang der Daten schwanken kann und nur schwer einschätzbar ist, wäre die Übertragung des Viewstate im URL sehr unsicher. Deshalb setzt ASP.NET konsequent und zwangsweise auf Formulare.

JavaScript

Die Lösung ist ebenso trickreich wie banal. Tatsächlich wird hier intern JavaScript verwendet. In der HTML-Programmierung ist dies ein bekanntes und verbreitetes Verfahren. Im Link wird als Ziel eine JavaScript-Funktion angegeben:

```
<a href="javascript:mycall()">Klick mich!</a>
```

Die Funktion `mycall()` wird nun folgendermaßen definiert:

```
mycall()
{
   document.form.action = "formularziel.aspx";
   document.form.submit();
}
```

Der Klick auf den Link führt so zum Senden des Formulars. Damit arbeitet auch ASP.NET. Der einzige Unterschied: Sie müssen JavaScript nicht kennen und brauchen auch keine Funktionen definieren. Um Hyperlinks in Formularen behandeln zu können, müssen Sie diese nur innerhalb der `<form>`-Tags platzieren und zum Server-Steuerelement erklären. Das folgende Beispiel zeigt eine alternative Form der Datenverarbeitung im Praxisprojekt:

```
<script language="C#" runat="Server">
void Page_Load()
{
    string[,] ProduktArr = new string[3,3];
    // Produktdefinition als Array

    ControlCollection c = myform.Controls;
    int k = 0;
    for (int i = 0; i < c.Count; i++)
    {
        Control cc = c[i];
        if (cc is HtmlAnchor)
```

```
        {
            HtmlAnchor anchor = (HtmlAnchor) cc;
            if (anchor.ID == ProduktArr[k, ProdID])
            {
                anchor.InnerText = ProduktArr[k, ProdName]
                                   + " (" + ProduktArr[k, ProdPreis] + ")";
            }
            k++;
        }
    }
}

void checkClick (object sender, EventArgs e)
{
    HtmlAnchor Link = (HtmlAnchor) sender;
    Bestaetigung.InnerHtml= "Artikel " + Link.InnerText + " mit ID "
                            + Link.ID + " wurde gew&auml;hlt.";

}

</script>

<html lang="de">
    <head>
        <title>Shop</title>
    </head>
    <body>
        <h1> Shop </h1>
        Unsere Produkte:
        <form runat="server" id="myform">
        <ul>
            <li><a runat="server" id="A0001" onServerClick="checkClick">
                </a>
            </li>
            <li><a runat="server" id="A0002" onServerClick="checkClick">
                </a>
            </li>
            <li><a runat="server" id="A0003" onServerClick="checkClick">
                </a>
            </li>
        </ul>
        </form>
        <div id="Bestaetigung" runat="server"/>
    </body>
</html>
```

Listing 4.6: Reaktion auf Klickereignisse (anchorstate.aspx)

Wie es funktioniert

Das Programm basiert auf den mit `runat="server"` zu HTML Server-Steuerelemente konvertierten `<a>`-Tags. Diese enthalten jeweils ein spezielles Attribut, `onServerClick`. Der Parameter bezeichnet eine Methode, mit der Sie auf das Klickereignis reagieren möchten:

```
<a runat="server" id="A0003" onServerClick="checkClick"></a>
```

Der Link ist vorerst leer, er wird im Beispiel erst zur Laufzeit mit Daten gefüllt. Dazu ist ein Zugriff auf die Kollektion der HTML Server-Steuerelemente erforderlich. Da es sich um ein Formular handelt, interessieren hier nur die Steuerelemente, die im Formular sind:

```
ControlCollection c = myform.Controls;
```

Alle Steuerelemente werden mit einer Schleife durchlaufen:

```
for (int i = 0; i < c.Count; i++)
```

Jedes einzelne `Control`-Objekt wird dann entnommen:

```
Control cc = c[i];
```

Nun ist festzustellen, ob es sich um ein `<a>`-Tag handelt, Typ `HtmlAnchor`:

```
if (cc is HtmlAnchor)
```

Zum Zugriff muss eine Typumwandlung vorgenommen werden:

```
HtmlAnchor anchor = (HtmlAnchor) cc;
```

Da dem passenden Tag die Bezeichung zugeordnet werden soll, wird nun die ID des Tags mit der ID des Array verglichen:

```
if (anchor.ID == ProduktArr[k, ProdID])
```

Stimmen beide überein, wird der Text zugewiesen:

```
anchor.InnerText = ProduktArr[k, ProdName] + " (" + ProduktArr[k,
ProdPreis] + ")";
```

Den Index auf das Array bestimmt die Hilfsvariable `k`, die mit jeder erfolgreichen Zuweisung um eins erhöht wird:

```
k++;
```

Nun ist die Seite vollständig und wird an den Browser gesendet. Jeder Klick auf einen Link führt zum Aufruf der Methode `checkClick`. Diese müssen Sie selbst definieren. Sie hat, wie alle anderen Ereignisbehandlungsmethoden auch, die optionalen Parameter Senderobjekt und Ereignisargument:

```
void checkClick (object sender, EventArgs e)
```

Die Variable `sender` enthält in diesem Fall das Objekt des Steuerelements, das das Ereignis ausgelöst hat. Es wird nun explizit in ein `HtmlAnchor` konvertiert:

```
HtmlAnchor Link = (HtmlAnchor) sender;
```

Nun kann auf die Eigenschaften und Methoden zugriffen werden, die dieses Objekt enthält. Die folgende Zuweisung erzeugt eine Bestätigung des Links im HTML Server-Steuerelement `Bestaetigung`, das am Ende als `<div>`-Tag definiert wurde:

```
Bestaetigung.InnerHtml= "Artikel " + Link.InnerText + " mit ID "
                        + Link.ID + " wurde gewählt."; }
```

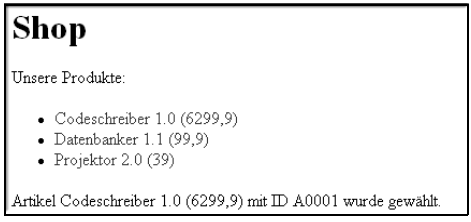


Abbildung 4.6: Nutzung des Viewstate mit Links (anchorstate.aspx)

Zum Verständnis ist ein Blick in den Quelltext der HTML-Seite sinnvoll. Dort finden Sie das am Anfang erwähnte Verfahren mit JavaScript in der Anwendung:

```
<body>
<h1> Shop </h1>
Unsere Produkte:
<form name="myform" method="post" action="anchorstate.aspx" id="myform">
<input type="hidden" name="__EVENTTARGET" value="" />
<input type="hidden" name="__EVENTARGUMENT" value="" />
<input type="hidden" name="__VIEWSTATE"
value="dDwyNzk3ODAxMzc7dDw7bDxpPDI+O2k8ND47PjtsPHQ8O2w8aTwxPjtpPDM+O2k8NT47Pj
tsPHQ8cDxsPGlubmVyaHRtbDs+O2w8Q29kZXNjaHJlaWJlciAxLjAgKDYyOTksOSk7Pj47Oz47dDx
wPGw8aW5uZXJodG1sO2Z47bDxEYXRlbmJhbmtlciAxLjEgKDk5LDkpO2z4+Ozs+O3Q8cDxsPGlubmVy
aHRtbDs+O2w8UHJvamVrdG9yIDIuMCAoMzkpO2z4+Ozs+Oz4+O3Q8cDxsPGlubmVyaHRtbDs+O2w8Q
XJ0aWtlbCBDb2Rlc2NocmVpYmVyIDEuMCAoNjI5OSw5KSBtaXQgSUQgQTAwMDEgd3VyZGUgZ2V3Jm
F1bWxcO2hsdC47Pj47Oz47Pj47PjAKTq5lsgZNtWqHKelpadMrhEVH" />
<script language="javascript">
<!--
function __doPostBack(eventTarget, eventArgument) {
  var theform = document.myform;
    theform.__EVENTTARGET.value = eventTarget;
    theform.__EVENTARGUMENT.value = eventArgument;
    theform.submit();
}
```

```
// -->
</script>
   <ul>
     <li><a id="A0001" href="javascript:__doPostBack('A0001','')">
           Codeschreiber 1.0 (6299,9)</a></li>
     <li><a id="A0002" href="javascript:__doPostBack('A0002','')">
           Datenbanker 1.1 (99,9)</a></li>
     <li><a id="A0003" href="javascript:__doPostBack('A0003','')">
           Projektor 2.0 (39)</a></li>
   </ul>
 </form>
 <div id="Bestaetigung">Artikel Codeschreiber 1.0 (6299,9) mit ID A0001
         wurde gew&auml;hlt.
 </div>
</body>
```

Zum einen finden Sie den Aufruf der JavaScript-Funktion, die immer den Namen `__doPostback` trägt:

```
<a id="A0003" href="javascript:__doPostBack('A0003','')">
```

Die Funktionsdefinition selbst enthält keine Besonderheiten. Das Attribut `action` wurde bereits im `<form>`-Tag definiert, verbleibt also nur die Aufgabe, das Formular zu versenden:

```
theform.submit();
```

onServerChange
onServerClick

Das Objekt vom Typ `HtmlAnchor` kann ein Ereignis `onServerClick` verarbeiten. Dies können Sie auch für `<button>` und `<input type="button">` einsetzen. Ein zweites Ereignis ist `onServerChange`, dass auftritt, wenn der Inhalt eines Objekts geändert wurde. Anwendbar ist dies bei folgenden HTML Server-Steuerelementen:

- `<input type="text">`
- `<input type="password">`
- `<input type="checkbox">`
- `<input type="radio">`
- `<input type="image">`
- `<input type="hidden">`
- `<textarea>`
- `<select>`

Das Ereignis wird ausgelöst, wenn der Inhalt eines Feldes des vorhergehenden mit dem aktuellen Formular nicht übereinstimmt. Auch die Methode, die `onServerChange` bedient, ist vom Typ `void` und kann zwei Parameter vom Typ `object` und `EventArgs` verarbeiten.

4.2.3 Gestalterische Elemente

Auf die Möglichkeit, Attribute von HTML Server-Steuerelementen programmatisch zuverändern, wurde bereits hingewiesen. Das ist im Detail nicht immer einfach, wenn es sich um Attribute handelt, die selbst wieder in viele kleinere Einheiten aufgelöst werden können. Dies trifft für Stylesheets zu. Das folgende Steuerelement enthält mehrere Style-Attribute (mit `runat="server" id="btn"` würde es zum Server-Steuerelement werden):

```
<input type="button" style="font-weight:bold; font-size:24pt;color:green"/>
```

Wenn Sie nun auf das Attribut `style` zugreifen, erhalten Sie eine Zeichenkette, die mühevoll zerlegt werden muss. Das erledigt ASP.NET für Sie jedoch automatisch. Das folgende Beispiel zeigt, wie eine Schaltfläche einfach gestaltet werden kann:

```
<script language="C#" runat="Server">
void Page_Load()
{
    btn.Value = "Ich bin ein schicker Button";
    btn.Style["font-family"] = "Arial";
    btn.Style["font-weight"] = "bold";
    btn.Style["color"] = "green";
    btn.Style["font-size"] = "24pt";
}

void clickBtn (object sender, EventArgs e)
{
    HtmlInputButton btn = (HtmlInputButton) sender;
    Bestaetigung.InnerHtml = "Meine Gestaltung basiert auf:<br/>";
    IEnumerator keys = btn.Style.Keys.GetEnumerator();
    while (keys.MoveNext())
    {
        String key = (String) keys.Current;
        Bestaetigung.InnerHtml += key + "='" + btn.Style[key] + "'<br/>";
    }
}

</script>

<html lang="de">
    <head>
        <title>Schaltfl&auml;che gestalten</title>
    </head>
    <body>
```

```
        <h1>Schaltfl&auml;che gestalten:</h1>
        <form runat="server" >
            <input type="button" id="btn" runat="server"
                    onServerClick="clickBtn"/>
        </form>
        <div id="Bestaetigung" runat="server"/>
    </body>
</html>
```

Listing 4.7: Gestaltung einer Schaltfäche mit CSS (buttonstyle.aspx)

Hier wird die Zuweisung der CSS-Attribute einzeln innerhalb von `Page_Load` erledigt. Die entsprechende Eigenschaft heißt `Style`.

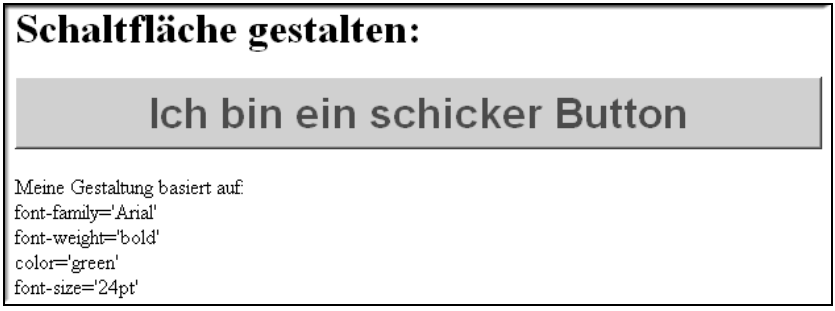


Abbildung 4.7: Gestaltung von HTML Server-Steuerelemente (button_style.aspx)

Wie es funktioniert

Das Klickereignis übergibt die Schaltfläche als Objekt. Damit es weiterverarbeitet werden kann, ist eine Typkonvertierung erforderlich:

```
HtmlInputButton btn = (HtmlInputButton) sender;
```

Verwechseln Sie dies nicht mit `HtmlButton`; diese Klasse bedient `<button>`-Tags, im Sprachumfang von HTML 4 definiert sind und nur selten zum Einsatz kommen.

Style-Attribute verarbeiten

Die `Style`-Eigenschaft ist eine Aufzählung. Innerhalb der Methode `clickBtn`, die ein Anklicken dieser Schaltfläche verarbeitet, werden zur Demonstration alle Elemente ausgelesen. Das geht am einfachsten, indem zuvor die Schlüssel der Kollektion zum Typ `IEnumerator` konvertiert werden:

```
IEnumerator keys = btn.Style.Keys.GetEnumerator();
```

Die Aufzählung wird mit einer `while`-Schleife durchlaufen, wobei bei jedem Durchlauf das nächste Element angewählt wird:

```
while (keys.MoveNext())
```

Dann wird der aktuelle Inhalt des Elements ermittelt:

```
String key = (String) keys.Current;
```

Nun wird dem Ausgabe-Tag der Name und der Wert des `Style`-Attributes zugewiesen:

```
Bestaetigung.InnerHtml += key + "='" + btn.Style[key] + "'<br/>"
```

Das Ergebnis war bereits in der Abbildung zu sehen. Ausführliche Informationen zum Umgang mit Aufzählungen und Kollektionen finden Sie in „Aufzählungen und Kollektionen“ auf Seite 288.

4.2.4 Erweiterung des Praxisprojekts

Mit den nun verfügbaren Informationen kann das Praxisprojekt erweitert werden. Geändert hat sich vorerst nur die Datei *shop.aspx*. Diesmal werden die Tags, die die Produkte zur Anzeige bringen, vollständig programmatisch erzeugt:

```
<script language="C#" runat="Server">
void Page_Load()
{
    string[,] ProduktArr = new string[3,3];
    const int ProdName = 0;
    const int ProdPreis = 1;
    const int ProdID = 2;
    ProduktArr[0,ProdName] = "Codeschreiber 1.0";
    ProduktArr[0,ProdPreis] = "6299,9";
    ProduktArr[0,ProdID] = "A0001";
    ProduktArr[1,ProdName] = "Datenbanker 1.1";
    ProduktArr[1,ProdPreis] = "99,9";
    ProduktArr[1,ProdID] = "A0002";
    ProduktArr[2,ProdName] = "Projektor 2.0";
    ProduktArr[2,ProdPreis] = "39";
    ProduktArr[2,ProdID] = "A0003";
    for (int i=0; i < ProduktArr.GetLength(0); i++)
    {
        HtmlGenericControl tempLIControl = new HtmlGenericControl("li");
        HtmlGenericControl tempAControl = new HtmlGenericControl("a");
        tempAControl.InnerText = ProduktArr[i, ProdName] + " ("
                               + string.Format("{0:N2}",
                               Convert.ToDouble(ProduktArr[i, ProdPreis]))
                               + ")";
        tempAControl.Attributes["href"] = "order.aspx?prodid="
                                        + ProduktArr[i,ProdID]
                                        + "&prodname="
```

```
                                        + ProduktArr[i,ProdName]
                                        + "&prodpreis="
                                        + ProduktArr[i,ProdPreis];
        tempLIControl.Controls.Add (tempAControl);
        Produkte.Controls.Add(tempLIControl);
    }
}
</script>
<html lang="de">
    <head>
        <title>Shop</title>
    </head>
    <body>
        <h1> Shop </h1>
        Unsere Produkte:
        <ul id="Produkte" runat="Server"></ul>
    </body>
</html>
```

Listing 4.8: Dynamische Erzeugung von HTML aus Daten (shop.aspx)

Wie es funktioniert

Die Definition der Arrays wurde unverändert aus dem ersten Schritt des Projekts übernommen. Allein die Erzeugung der Anzeige ist nun vollständig im Programmcode zu finden. Für jedes Produkt wird ein `<li>`- und ein `<a>`-Tag erzeugt:

```
HtmlGenericControl tempLIControl = new HtmlGenericControl("li");
HtmlGenericControl tempAControl = new HtmlGenericControl("a");
```

Dem Link wird dann der Text zugewiesen, der angezeigt werden soll. Dies umfasst den Produktnamen und den Preis. Der Preis wird folgendermaßen formatiert:

```
tempAControl.InnerText = ProduktArr[i, ProdName] + " ("
+ string.Format("{0:N2}",
Convert.ToDouble(ProduktArr[i, ProdPreis])) + ")";
```

Dann wird der Link dem Attribut `href` zugewiesen:

```
tempAControl.Attributes["href"] = "order.aspx?prodid="+ ProduktArr[i,ProdID]
+ "&prodname=" + ProduktArr[i,ProdName] + "&prodpreis="
+ ProduktArr[i,ProdPreis];
```

Hier werden alle drei Produktinformationen an den Querystring angehängt, damit sie auf der nächsten Seite verfügbar sind. Dieses Verfahren wird solange beibehalten, bis eine Technik zur Verfügung steht, die Daten zentral zu speichern; beispielsweise eine Textdatei, XML oder eine Datenbank.

Dann wird das <a>-Tag dem -Tag hinzugefügt, denn es soll ja in diesem enthalten sein:

```
tempLIControl.Controls.Add (tempAControl);
```

Zuletzt wird das schon recht komplexe -Tag in das umgebende <ul> eingefügt:

```
Produkte.Controls.Add (tempLIControl);
```

Da dies in der Schleife abläuft, wird so für jedes Produkt ein Aufzählungs-Tag <li> erzeugt. Richtig sinnvoll ist diese Vorgehensweise natürlich nur dann, wenn die Anzahl der Produkte öfters schwankt. Es ist immer clever, ein Programm auf viele zukünftige Änderungen vorzubereiten, denn dadurch wird der Wartungsaufwand verringert.

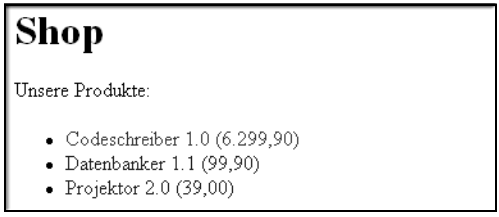


Abbildung 4.8: Zweite Version des Shops – alle Elemente sind vom Programm erzeugt worden

Mit den HTML Server-Steuerelementen werden auch Formulare aufgebaut. Es ist deshalb mit denselben Mitteln möglich, auch das Formular zu Erfassung der Kundendaten aufzubauen. Die Datei *order.aspx* enthält den entsprechenden Code.

```
<script language="C#" runat="Server">
void Page_Load()
{
    if (IsPostBack)
    {
        Bestaetigung.InnerHtml  = Request.Form["ProdNameTag"];
        double Summe = Convert.ToDouble(Request.Form["ProdPreisTag"]) *
          Convert.ToDouble(Request.Form["KundMenge"]);
        Bestaetigung.InnerHtml += "<br/>Der Gesamtwert der Bestellung
                                   betr&auml;gt: &euro;<b>"
                                + string.Format("{0:N2}", Summe) + "</b>";
    } else {
        BestellIDTag.InnerText = ProdIDTag.Attributes["Value"] =
                                   Request.QueryString["prodid"];
        BestellNameTag.InnerText = ProdNameTag.Attributes["Value"] =
                                     Request.QueryString["prodname"];
        string S = string.Format("{0:N2}",
                         Convert.ToDouble(Request.QueryString["prodpreis"]));
```

```
        BestellPreisTag.InnerText = ProdPreisTag.Attributes["Value"] = S;
    }
}
</script>
<html lang="de">
    <head>
        <title>Bestellung</title>
    </head>
    <body>
        <h1>Bestellung</h1>
        <b>Ihre Bestellung:</b>  ID:<span id="BestellIDTag" runat="Server">
                                    </span>
        Artikel: <span id="BestellNameTag" runat="Server"></span><br/>
        Unser Preis pro St&uuml;ck: <span id="BestellPreisTag"
                                        runat="Server"></span> &euro;
        <form runat="Server">
            <input type="hidden" id="ProdNameTag" runat="Server"/>
            <input type="hidden" id="ProdIDTag" runat="Server"/>
            <input type="hidden" id="ProdPreisTag" runat="Server"/>
            Name: <input type="text" runat="Server" id="KundName"/><br/>
            Stra&szlig;e: <input type="text" runat="Server" id="KundStr"/>
            <br/>
            Ort: <input type="text" runat="Server" id="KundOrt"/><br/>
            Plz: <input type="text" runat="Server" id="KundPlz"/><br/>
            E-Mail: <input type="text" runat="Server" id="KundEMail"/><br/>
            Menge: <input type="text" runat="Server" id="KundMenge"/><br/>
            <br/>
            <input type="submit" value="Jetzt bestellen"/>
        </form>
        Sie haben bestellt: <span runat="Server" id="Bestaetigung"></span>
    </body>
</html>
```

Listing 4.9: Verarbeiten der Bestellung und Abfragen der Kundendaten (order.aspx)

Wie es funktioniert

Das HTML-Formular weist kaum Besonderheiten auf. Es wird die eingebaute Formularautomatik von ASP.NET verwendet, weshalb das `<form>`-Tag außer `runat="server"` keine weiteren Attribute benötigt. Die Funktionsweise des Ablaufs bedingt, dass versteckte Felder eingesetzt werden. Der erste Aufruf der Seite übergibt die Artikelinformationen als Querystring, also per GET. Da sich diese Seite nun selbst aufruft und dabei ein Formular sendet (POST), müssen die Daten erhalten bleiben. Es gibt für solche Fälle mehrere Lösungen: Cookies, versteckte Felder, Datenbanken oder mit XML persistent gemachte Daten. Die letzten beiden Methoden kennen Sie noch nicht, von den ersten beiden erscheinen Cookies unsympatischer. Denn wenn der Benutzer Cookies abschaltet, funktioniert der Shop nicht mehr. Da es sich nicht mehr nur um

Session-Cookies handelt, sondern sensible Daten übertragen werden, besteht eine erhebliche Wahrscheinlichkeit, dass einige Benutzer nicht bedient werden können. Versteckte Felder sind deshalb die beste Wahl. Sie werden definiert, indem für `<input>` das Attribut `type="hidden"` verwendet wird. Ansonsten unterscheiden sie sich nicht von anderen Feldern.

Es gilt nun, beim ersten Aufruf der Seite die Felder mit Werten zu versehen. Der erste Aufruf kann von allen folgenden unterschieden werden, indem die Eigenschaft `IsPostBack` des Objekts `Page` geprüft wird. Ist der Wert `false`, handelt es sich um den ersten Aufruf. Im Beispiel wird dieser Teil im `else`-Zweig verarbeitet. Der folgenden Aufruf liest den übergebenen Wert aus dem Querystring und schreibt ihn in zwei HTML Server-Steuerelemente:

```
BestellIDTag.InnerText = ProdIDTag.Attributes["Value"] =
Request.QueryString["prodid"];
```

Das Objekt `BestellIDTag` dient lediglich einer Kontrollanzeige des Benutzers. Es handelt sich um ein `<span>`-Tag, dass hier Text bekommt. `ProdID Tag` ist dagegen das Objekt des versteckten Feldes. Diese enthalten keinen Text, sondern tragen ein Attribut `value`. Entsprechend erfolgt ein Zugriff auf die `Attributes`-Kollektion.

Mit den anderen Daten wird entsprechend verfahren. Beim Preis wird wieder die bereits auf der ersten Seite verwendete Formatierung angewendet. Dann kann das Formular angezeigt werden. Der Benutzer kann es nun ausfüllen. Sendet er das Formular ab, ist `IsPostBack` gleich `true` und der erste Zweig in der Methode `Page_Load` wird ausgeführt:

```
Bestaetigung.InnerHtml = Request.Form["ProdNameTag"];
```

Zuerst wird der Wert des Formularfeldes mit `Request.Form` der `Form`-Kollektion entnommen.

```
double Summe = Convert.ToDouble(Request.Form["ProdPreisTag"]) *
Convert.ToDouble(Request.Form["KundMenge"]);
```

Dann werden die Werte für Preis und Menge geholt. Da Formularfelder immer Zeichenketten enthalten, ist die explizite Konvertierung in `double` notwendig. Damit kann die Multiplikation ausgeführt werden. Der berechnete Wert wird mit etwas erklärendem Text dann an das HTML Server-Steuerelement `Bestaetigung` gesendet:

```
Bestaetigung.InnerHtml += "<br/>Der Gesamtwert der Bestellung betr&auml;gt:
&euro; <b>" + string.Format("{0:N2}", Summe) + "</b>";
```

Damit ist der Bestellvorgang in diesem Stadium des Projekts beendet. Damit eine Weiterverarbeitung erfolgen kann, ist weiteres Wissen not-

wendig. Die Ablage der Bestellungen kann in einer Datenbank, einer Textdatei oder in XML erfolgen. Sie können die Angaben auch oder zusätzlich per E-Mail versenden.

4.3 Web Server-Steuerelemente (Web Server Controls)

Web Server-Steuerelemente sind ebenso wie die HTML Server-Steuerelemente für das Erzeugen von HTML-Code zuständig. Es gibt jedoch wesentliche Unterschiede, die den Einsatz der einen oder anderen Version sinnvoll erscheinen lassen.

Hohe Abstraktion

Web Server-Steuerelemente sind teilweise komplexer und erlauben eine echte Abstraktion von der HTML-Ebene. Das heißt, die Definition erfolgt generell mit eigenen Tags und unabhängig von Objektmodell von HTML. Dadurch kann ein Web Server-Steuerelement mehr als ein HTML-Tag generieren bzw. verwalten. Das Objektmodell dahinter ist insgesamt reichhaltiger und strenger typisiert. HTML kennt per se keine Datentypen. Das führt dazu, dass Attribute bei HTML Server-Steuerelementen immer als Zeichenkette betrachtet werden. Tatsächlich sind aber bestimmte Attribute durchaus an bestimmte Datentypen bindbar, wie beispielsweise die Breite einer Tabellenzelle (`width="43"`). Hier sind entweder ganze Zahlen oder Prozentwerte angebbar, aber mit Sicherheit nie eine Zeichenkette. Web Server-Steuerelemente enthalten für alle in HTML zulässigen Attribute einzelne Eigenschaften. Insofern ist die Programmierung damit strenger am Objekt- und Typisierungskonzept von C# und dem .Net-Framework ausgerichtet. Von Vorteil ist auch die Verfügbarkeit komplexer Objekte, wie ganzer Kalender, die intern aus Tabellen und Links bestehen.

Browser-Erkennung

Bei den HTML Server-Steuerelementen wurde gezeigt, dass Standard-HTML-Tags erzeugt werden. Das funktioniert bei den HTML 3.2 entstammenden Tags mit jedem Browser. Komplexere Objekte sind dagegen nicht so leicht allein mit HTML 3.2 zu erstellen. Mancher Effekt ist nicht in jedem Browser gleichartig implementiert. Außerdem kommt manchmal JavaScript zum Einsatz, was auch nicht problemlos in allen Browsern funktioniert. Web Server-Steuerelemente enthalten deshalb eine automatische Browser-Erkennung. Je nach Typ wird dann automatisch unterschiedlicher Code erzeugt. Dies ist sehr vorteilhaft, denn Sie müssen sich keine Gedanken darüber machen, dass durch den Einsatz von ASP.NET Netscape-Benutzer eventuell ausgeschlossen wären.

4.3.1 Übersicht

Dieser Abschnitt zeigt eine Übersicht der verfügbaren Steuerelemente und deren Einordnung im .Net-Framework. Dies erleichtert das Auffinden der Methoden und Eigenschaften in der Referenz. Allerdings ist dies keineswegs so einfach wie bei den HTML Server-Steuerelemente. Zwischen den Klassen des Namensraumes `System.Web.UI.WebControls` und dem Rest des Frameworks bestehen vielfältige

Abhängigkeiten und Beziehungen. Die Anzahl der verfügbaren Elemente ist auch deutlich höher. Es ist also etwas Systematik erforderlich, um der schieren Menge an Möglichkeiten Herr zu werden. Einteilen kann man die Web Server-Steuerelemente nach der Komplexität und Aufgabe, die sie erfüllen:

- **Webformular-Steuerelemente** (Web Form Controls) – Diese Gruppe enthält alle Steuerelemente, die in Formularen zum Einsatz kommen. Diese Gruppe ist den bereits im letzten Abschnitt vorgestellt HTML Server-Steuerelemente ähnlich. Oft werden dieselben Tags erzeugt. Die Gruppe lässt sich feiner unterteilen:
  - **Web Server-Steuerelemente** (Web Controls) – Dies sind alle normalen Eingabe-Tags, Label und Tabellen-Tags.
  - **Text-Steuerelemente** (Text Controls) – Hiermit erfolgt die Ausgabe von Text.
  - **Listen-Steuerelemente** (List Controls) – Listen-Tags sind deshalb von besonderer Bedeutung, weil sie wiederholende Teile enthalten; ideal zum Erzeugen der Ergebnisse von Datenbankabfragen.
  - **Daten-Steuerelemente** (Data Controls) – Direkt zur Ausgabe von Datenmengen aus Datenbanken sind diese Steuerelemente geeignet.
- **Kontroll-Steuerelemente** (Validation Controls) – Daten aus Formularen müssen vielfältig geprüft werden. Mit den Steuerelementen dieser Gruppe ist das besonders einfach zu erledigen.
- **Komplexe Steuerelemente** (Rich Controls) – Diese Gruppe umfasst komplexere Steuerelemente, die beispielsweise Banner verwalten oder Kalender darstellen.

Die folgenden Tabellen zeigen die Mitglieder der einzelnen Gruppen.

Name des Steuerelements	Korrespondierende HTML-Tags
`<asp:hyperlink>`	`<a>`
`<asp:linkbutton>`	`<a><img/></a>`
`<asp:image>`	`<img>`

Tabelle 4.4: Übersicht Web Server-Steuerelemente

Name des Steuerelements	Korrespondierende HTML-Tags
<asp:button>	<input type="button">
<asp:textbox>	In Abhängigkeit von Attributen: <input type="text">, <input type="password">, oder <textarea>
<asp:checkbox>	<input type="checkbox">
<asp:radiobutton>	<input type="radio">
<asp:imagebutton>	<input type="image">
<asp:table>	<table>
<asp:tablerow>	<tr>
<asp:tablecell>	<td>

Tabelle 4.4: Übersicht Web Server-Steuerelemente (Forts.)

Tabellen

Auf die Elemente zur dynamischen Erzeugung von Tabellen kann hier mangels Platz nicht eingegangen werden. Sie finden jedoch eine Anwendung im Abschnitt „Zugriff über Aufzählungen“ auf Seite 295. Dort wird zwar ein anderes Thema behandelt, aber das Prinzip der Verwendung von <asp:table> wird ebenfalls kurz vorgestellt.

Text-Steuerelemente	Korrespondierende HTML-Tags
<asp:panel>	
<asp:label>	
<asp:literal>	Einfacher Text, der verändert werden kann
<asp:placeholder>	An dieser Stelle kann neuer Text erzeugt werden

Tabelle 4.5: Übersicht Text-Steuerelemente

Listen-Steuerelemente	Korrespondierende HTML-Tags
<asp:dropdownlist>	<select size="1">
<asp:listbox>	<select>
<asp:checkboxlist>	Mehrere Elemente vom Typ <input type="checkbox">
<asp:radionbuttonlist>	Mehrere Elemente vom Typ <input type="radio">
<asp:listitem>	Expliziter Zugriff auf <option>
<asp:repeater>	Wiederholt das enthaltene Element

Tabelle 4.6: Übersicht Listen-Steuerelemente

Die Daten-Steuerelemente dienen der Ausgabe von Datensätzen, wie sie Datenbanken entnommen werden können. Dieser Teil wird zusammen mit dem Datenbankzugriff in Kapitel 6 beschrieben.

Name des Steuerelements	Bedeutung
<asp:RequiredFieldValidator>	Prüft, ob ein Feld ausgefüllt worden ist
<asp:RangeValidator>	Prüft, ob der erfasste Werte innerhalb eines Wertebereiches ist
<asp:CompareValidator>	Vergleicht zwei Felder
<asp:RegularExpressionValidator>	Prüft den Inhalt mit Hilfe eines regulären Ausdrucks
<asp:CustomValidator>	Prüft den Inhalt mit einer selbstdefinierten Methode
<asp:ValidationSummary>	Zeigt alle Fehlermeldungen auf Grund der Feldprüfungen an

Tabelle 4.7: Übersicht Kontroll-Steuerelemente

Komplexe Steuerelemente	Beschreibung
<asp:adrotator>	Dient der Verwaltung von Bannern
<asp:calender>	Ein sehr umfangreiches Control zur Anzeige von Kalendern

Tabelle 4.8: Übersicht Komplexe Steuerelemente

Alle Steuerelemente stammen aus dem Namensraum `System.Web.UI.Control`. Da die vielen Klassen unzählige Eigenschaften und Methoden enthalten, ist ein Blick auf die Hierarchie sehr hilfreich, denn die Definitionen in höheren Klassen wiederholen sich in allen tiefer liegenden. Wenn Sie sich eine Basisklasse anschauen, kennen Sie damit schon eine Großteil der Leistungen der konkreten Klassen.

Abstrakte Basisklassen, auf die Sie im Allgemeinen nicht direkt zugreifen, sind fett gekennzeichnet; alle übrigen, die bei der Programmierung eine Rolle spielen, sind kursiv. Die in der Abbildung gezeigten Klassen repräsentieren alle eigenständige Steuerelemente. Viele weitere Klassen werden zur Ableitung von Attributen verwendet. Diese stammen aber nicht aus dem Namensraum `System.Web.UI.Control`.

4.3.2 Einsatzprinzipien und Basiseigenschaften

Die Vielfalt der Elemente systematisch darzustellen, ist ein mühsames und wenig hilfreiches Unterfangen. Ähnlich wie bei den HTML Server-Steuerelementen ist es entscheidend, das Prinzip zu verstehen. Mit Hilfe der Online-Referenz können Sie sich dann jedes gerade benötigte Element leicht erschließen. In der Übersicht der Klassenhierarchie haben Sie bereits die Namen der Klassen kennen gelernt. Die Elemente, die im HTML-Teil der Seite eingesetzt werden können, entsprechen diesen

Namen. So implementieren Sie ein Objekt der Klasse `Label` folgendermaßen:

```
<asp:label id="TextAusgabe" runat="server" />
```

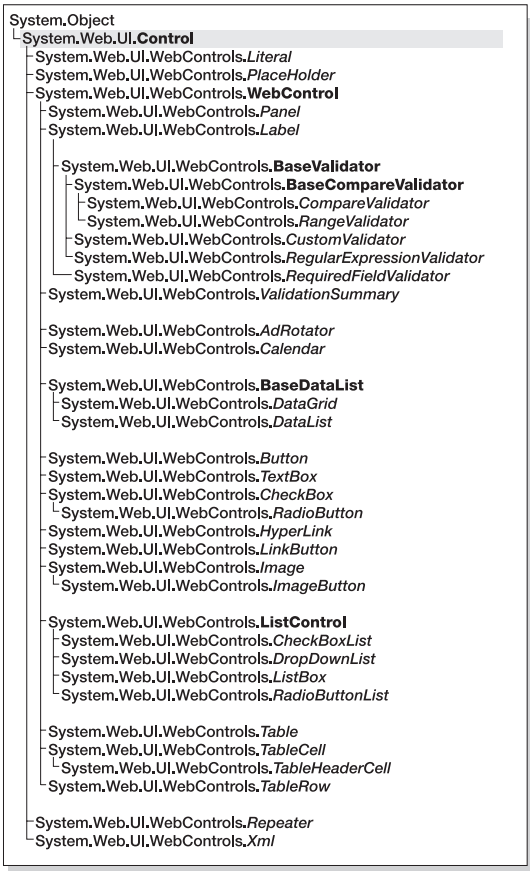


Abbildung 4.9: Hierarchie der Klassen im Namensraum System.Web.UI.Control

Diese Syntax stammt aus der XML-Welt. Der konkrete Name setzt sich aus dem XML-Namensraum „asp“ und dem Namen der Klasse „label“ zusammen. Wie schon bei den HTML Server-Steuerelemente wird das Objekt dann im Programmteil instanziiert, wenn es mit dem Attribut `runat="server"` versehen wurde. Das Attribut `id` bestimmt, unter welchem Namen das Objekt entsteht. Der Zugriff ist, wie nachfolgend gezeigt, möglich:

```
TextAusgabe.Text = "Dieser Text wird angezeigt";
```

Wenn sie alle Eigenschaften eingestellt haben und der Seitenerstellungsprozess abgeschlossen ist, wird das fertige HTML an den Browser gesendet. Im Beispiel würde dies folgendermaßen aussehen:

```
<span id="TextAusgabe">Dieser Text wird angezeigt</span>
```

Web Server-Steuerelemente haben noch einige Basiseigenschaften, die die bei HTML Server-Steuerelementen beschriebenen Möglichkeiten erweitern.

Zugriff auf die Feldwerte

Die Nutzung der `Form`-Kollektion (`Request.Form`) ist zwar weiter möglich, aber nicht unbedingt empfehlenswert, weil die Webserver-Klassen außerhalb der Steuerelemente-Klassen stehen. Je nach Typ des Web Server-Steuerelementes werden verschiedene Eigenschaften für den Zugriff auf die Feldwerte eingesetzt.

Der Zugriff auf die Feldwerte erfolgt bei Textfeldern über die Eigenschaft `Text`. Listen, also `DropDownList` oder `ListBox`, werden über die Listenmitglieder angesprochen. Der erste in der Liste ausgewählte Index wird in der Eigenschaft `SelectedIndex` übergeben, das Listenobjekt selbst in `SelectedItem`. Bei Kontrollkästchen (`CheckBox`) und Optionsfeldern (`RadioButton`) ist die Eigenschaft `Checked` gleich `true`, wenn das Element ausgewählt wurde.

Automatisches Senden von Formularen

AutoPostBack

Im weitesten Sinne geht es bei fast allen Web Server-Steuerelementen um Formulare. Bislang war es immer notwendig, nach dem Ausfüllen eines Formulars dieses mit einer Sendeschaltfläche abzusenden. Manchmal ist das lästig und unnötig, beispielsweise wenn jede Änderung an einer Option sofort zu einem Ereignis führen soll. Die Funktion, die das erledigt, heißt *AutoPostBack*. Dabei kommt JavaScript zum Einsatz, was Sie aber nicht selbst implementieren müssen. Dies erledigt die ASP.NET-Komponente für Sie. Standardmäßig steht dieses Verhalten bei folgenden Objekten zur Verfügung: `CheckBox` (Kontrollkästchen), `RadioButton` (Optionsschaltfläche) und `TextBox` (Eingabefelder). Die Funktion basiert auf den in HTML definierten `onClick`- bzw. `onChange`-Ereignissen.

Denken Sie bei der Implementierung der AutoPostBack-Funktion daran, dass die Ereignisverarbeitung auf dem Server erfolgt und das jeder Klick des Benutzers zum Absenden des Formulars führt. Unter Umständen, beispielsweise bei langsamen Verbindungen oder großen Seiten, ist dies lästig und erschwert die Bedienung mehr, als dass es nutzt.

Ob ein Element dieses Verhalten aufweisen soll, ist einfach über das Attribut `autopostback` festzulegen, bzw. programmatisch über die Eigenschaft `AutoPostBack`. Das folgende Beispiel zeigt das Prinzip:

```
<script language="C#" runat="Server">
void Page_Load()
{
   Bestaetigung.Text = "Keine Auswahl getroffen";
}

void clickBtn (object sender, EventArgs e)
{
    RadioButton currentRadio = (RadioButton) sender;
    Bestaetigung.Text = currentRadio.Text;
}

</script>

<html lang="de">
    <head>
        <title>RadioButton</title>
    </head>
    <body>
        <h1>Was ist die beste Programmierumgebung f&uuml;r Webserver?</h1>
        <form runat="server" >
            <asp:radiobutton autopostback="true" OnCheckedChanged="clickBtn"
                             id="Wahl1" text="ASP.NET mit C#"
                             groupname="Wahl" runat="server"/><br/>
            <asp:radiobutton autopostback="true" OnCheckedChanged="clickBtn"
                             id="Wahl2" text="Klassik ASP mit VBScript"
                             groupname="Wahl" runat="server"/><br/>
            <asp:radiobutton autopostback="true" OnCheckedChanged="clickBtn"
                             id="Wahl3" text="PHP"
                             groupname="Wahl" runat="server"/><br/>
            <asp:radiobutton autopostback="true" OnCheckedChanged="clickBtn"
                             id="Wahl4" text="Perl/CGI"
                             groupname="Wahl" runat="server"/><br/>
            <asp:radiobutton autopostback="true" OnCheckedChanged="clickBtn"
                             id="Wahl5" text="Java"
                             groupname="Wahl" runat="server"/>
        </form>
        <asp:label id="Bestaetigung" runat="server"/>
    </body>
</html>
```

Listing 4.10: Optionsfelder mit AutoPostBack abfragen (autopostback.aspx)

Wie es funktioniert

Im HTML-Teil wird eine Gruppe von Optionsfeldern definiert. Die Gruppierung findet über das Attribut `groupname` statt. Die Funktion `clickBtn`, die auf ein Klickereignis reagieren soll, wird mit `onCheckedChanged` festgelegt. Außerdem wird natürlich mit `autopostback="true"` sichergestellt, dass eine Änderung der Option zum Senden des Formulars führt. Viel Code ist hier ohnehin nicht notwendig. Die Funktion `clickBtn`, die zur Reaktion auf das Ereignis führt, empfängt das Element als Objekt. Da bekannt ist, dass es sich um Optionsfelder handelt, wird eine explizite Typkonvertierung ausgeführt:

```
RadioButton currentRadio = (RadioButton) sender;
```

Zur Bestätigung des Vorgangs soll auf die Eigenschaft `Text` zugegriffen werden:

```
Bestaetigung.Text = currentRadio.Text;
```

Hier kommt ein Steuerelement vom Typ `Label` zum Einsatz, dass als HTML zu `<span>` mutiert. Ein Blick auf den HTML-Code ist an dieser Stelle ohnehin spannender. Zuerst aber das Bild der fertigen Seite:

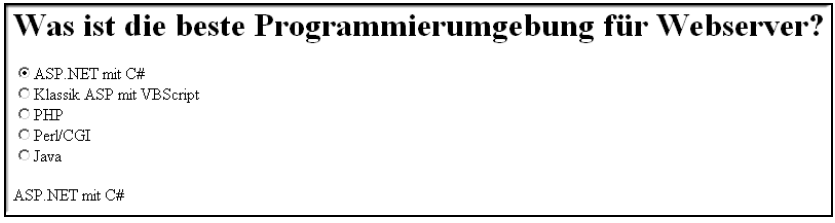


Abbildung 4.10: Auswahl ohne Sendeschaltfläche mit AutoPostBack

Im HTML-Code finden Sie die schon bekannte Vorgehensweise in ASP.NET. Das Formular sendet sich an dieselbe Seite zurück:

```
<form name="_ctl0" method="post" action="autopostback.aspx" id="_ctl0">
```

Der Auslöser für das Absenden des Formulars ist der Aufruf einer JavaScript-Funktion beim Auftreten des Ereignisses `onClick`. Dies führt zu folgendem Eingabefeld:

```
<input id="Wahl1" type="radio" name="Wahl" value="Wahl1" checked="checked"
onClick="__doPostBack('Wahl1','')" language="javascript" />
```

Interessant ist noch die Darstellung des Textes der Optionsschaltfläche, der an das Objekt gebunden ist und nicht – wie bei HTML sonst üblich – losgelöst auf der Seite steht:

```
<label for="Wahl1">ASP.NET mit C#</label>
```

Soweit zum Prinzip. Nachfolgend werden exemplarisch einige Klassen näher vorgestellt.

4.3.3 Text auf der Seite steuern

Die Ausgabe von Textelementen, die vorerst keine Interaktion zulassen, mag kein sinnvoller Einsatz für Web Server-Steuerelemente sein. Denken Sie aber an eine gute Benutzerführung, sollten Webseiten auf Aktionen jederzeit reagieren. Es kann also notwendig sein, Textelemente auszutauschen, gestalterisch zu verändern oder überhaupt erst erscheinen zu lassen, um den Benutzer über bestimmte Vorgänge oder Ereignisse zu informieren.

Diese Möglichkeiten werden von folgenden Klassen geboten:

- `Panel` – Dies ist ein Container, der eine definierte Fläche belegt und mit statischem Inhalt oder weiteren Elementen gefüllt werden kann. Das korrespondierende HTML-Element ist `<div>`. `Panel` erzeugt immer Absätze.
- `Label` – Dies ist ein Container, der mehrere Zeichen umfasst und entweder allein im Fließtext oder als Kind eines Panels auftritt.
- `PlaceHolder` – Diese Klasse wird direkt von `Control` abgeleitet, nicht von `WebControl`, wie die anderen. Die Objekte sind etwas primitiver und haben weniger Methoden und Eigenschaften. `PlaceHolder` dient der Einbettung anderer `Controls` oder der direkten Ausgabe von Text. Es erzeugt kein eigenes HTML-Tag. Natürlich fehlen mangels umschließendem Tag sämtliche Formatiermöglichkeiten. Enthaltene Steuerelemente können wiederum alles enthalten, was mit Web Server-Steuerelementen möglich ist.
- `Literal` – Auch dieses Steuerelement erzeugt kein HTML selbst, sondern dient der Ausgabe von einfachem Text. Auch hier sind keine Formatiermöglichkeiten gegeben.

Das folgende Beispiel zeigt die Anwendung von `Panel` und `Label`. Zu `Label` finden Sie außerdem unzählige Verwendungen in den Beispielen dieses Buches.

```
<script runat="server">
void Page_Load(Object sender, EventArgs e)
{
    Label TextLabel = new Label();
    Literal TextLit = new Literal();
    TextLit.Text = "<hr/>";
    TextLabel.Text = "Dieser Text ist ein Label";
    TextLabel.ID = "PanelLabel";
    GroupPanel.Controls.Add(TextLit);
    GroupPanel.Controls.Add(TextLabel);}
```

```
void checkVisibility(object sender, EventArgs e)
{
    if (CheckVisibility.Checked)
    {
        GroupPanel.Visible=false;
        CheckVisibility.Text = "Panel wieder anzeigen";
    } else {
        GroupPanel.Visible=true;
        CheckVisibility.Text = "Panel verstecken";
    }
}
</script>
<html>
 <head>
   <title>Panele und Label</title>
   <style>
   #PanelLabel {font-size:12pt; color:red; font-weight:bold}
   </style>
 </head>
 <body>
    <h3>Panele und Label</h3>
    <form runat=server>
        <asp:panel id="GroupPanel" runat="server"
                       BackColor="#eeeeee"
                       Height="50px" Width="250px">
                       <b>Einige Label:</b><br/>
        </asp:panel>
        <p>
        <asp:checkbox id="CheckVisibility"
                      Text="Panel verstecken"
                      autopostback="true"
                      onCheckedChanged="checkVisibility"
                      runat="server"/>
        <p/>
    </form>
 </body>
 </html>
```

Listing 4.11: Erzeugen eines Panels und Einbau von Text (panellabel.aspx)

Wie es funktioniert

Die Definition des `Panel`-Objekts erfolgt im HTML-Teil mit `<asp:panel>`. Enthalten ist bereits ein statischer Text. Der Zugriff auf den Inhalt erfolgt nicht mit der Eigenschaft `Text`, sondern – und dies ist der Unterschied zu `Label` – mit einer `Control`-Kollektion. Im Beispiel werden zwei neue Steuerelemente hinzugefügt, einmal ein `Label` mit dem Namen *TextLabel* und ein Stück HTML-Code, der als `Literal` (einfacher Text) ausgegeben wird. Literale verändern den empfangenen Text niemals, sie

unterscheiden sich insofern von den Eigenschaften `InnerText` und `InnerHtml`, die bei den HTML Server-Steuerelementen zur Ausgabe von Text zum Einsatz kamen. Das Hinzufügen zur `Control`-Kollektion erfolgt dagegen wieder ganz konservativ mit `Add`:

```
GroupPanel.Controls.Add(TextLit);
```

Was genau passiert, verrät ein Blick in den HTML-Code, den dieses Programm erzeugt:

```
div id="GroupPanel" style="background-color:#EEEEEE;height:50px;
                           width:250px;">
<b>Einige Label:</b><br/>
<span id="PanelLabel">Dieser Text ist ein Label</span><hr/>
</div>
```

Erwartungsgemäß ist ein `<div>`-Tag zu finden, das optimal mit CSS gestaltet wurde. Es enthält neben dem statischen Text – der intern als `Literal` gespeichert wird – das Label in Form eines `<span>`-Tags und das zusätzliche HTML-Tag `<hr/>`.

Abbildung 4.11: Textfelder als Panel mit gestalteten Elementen

Die Gestaltung des Steuerelements `Label` rührt von einem zusätzlichen Style her, dass als `<style>`-Anweisung direkt im HTML-Teil eingebettet wurde. Alle Elemente, die ein `id`-Attribut haben, lassen sich auch über CSS ansprechen. Das hat zwar nichts mit ASP.NET zu tun, ist aber eine interessante Verquickung beider Technologien, die die Gestaltung von Seiten vereinfachen kann.

Ohne das dies nochmals erläutert werden soll, enthält auch dieses Beispiel eine Anwendung der `AutoPostBack`-Funktion, die in diesem Fall das `Panel`-Objekt ein- und ausschaltet. Dies erfolgt mit der Eigenschaft `Visible`.

Wenn ein Element unsichtbar gemacht wird, erledigt dies das Style-Attribut `display:none`. Nichtsdestotrotz bleibt der HTML-Code im Quelltext der Seite erhalten und ist für neugierige Benutzer sichtbar. Zum Verstecken von Informationen ist dieser Weg nicht geeignet.

4.3.4 Texteingabefelder erzeugen und auswerten

Formulare bestehen fast immer aus Texteingabefeldern. ASP.NET verfolgt mit den Web Server-Steuerelementen hier einen interessanten Ansatz, denn die logisch ähnlichen Texteingabefelder `<input type="text">` und `<textarea>` sind unter einer Klasse erreichbar. Dass ein einzeiliges Feld so völlig anders benannt und behandelt wird als ein mehrzeiliges, bereitet HTML-Anfängern oft Schwierigkeiten. Damit ist nun Schluss, denn das Web Server-Steuerelement `TextBox` definiert alle Arten von Eingabefeldern. Dank CSS auch mit einheitlichem Aussehen bei vergleichbaren Attributen. Bei HTML allein ist das nicht der Fall, denn die Angabe `size="10"` im Tag `<input type="text">` führt nicht zu derselben Breite des Feldes wie die Angabe `columns="10"` im Tag `<textarea>`, was man eigentlich erwarten könnte.

Trotzdem wird ASP.NET in jedem Fall das passende HTML-Tag erzeugen und senden und damit die Kompatiblität mit allen Browsern sicherstellen. Ein Beispiel zeigt, wie dieses Steuerelement eingesetzt werden kann:

```
<script runat="server">
void Page_Load(Object sender, EventArgs e)
{
        TextEingabe.Width = 300;
        TextEingabe.BorderWidth = 1;
        TextEingabe.Text = "Ihre Eingabe";
}

void setBox(object sender, EventArgs e)
{
    RadioButton zeilenzahl = (RadioButton) sender;
    int z = Convert.ToInt16(Request.Form["Zeilen"]);
    if (z > 1)
    {
        TextEingabe.TextMode = TextBoxMode.MultiLine;
    } else {
        TextEingabe.TextMode = TextBoxMode.SingleLine;
    }
    TextEingabe.Rows = z;
}
</script>
<html>
 <head>
   <title>Textbox</title>
 </head>
```

```
<body>
   <h3>Textbox</h3>
   <form runat=server>
       <asp:radiobutton id="Zeile1" groupname="Zeilen" runat="server"
                        autopostback="true" onCheckedChanged="setBox"
                        text="1 Zeile" value="1" /><br/>
       <asp:radiobutton id="Zeile3" groupname="Zeilen" runat="server"
                        autopostback="true" onCheckedChanged="setBox"
                        text="3 Zeilen" value="3"/><br/>
       <asp:radiobutton id="Zeile7" groupname="Zeilen" runat="server"
                        autopostback="true" onCheckedChanged="setBox"
                        text="7 Zeilen" value="7" /><br/>
       <asp:textbox id="TextEingabe" runat="server"/>
   </form>
</body>
</html>
```

Listing 4.12: Dynamisch verändertes Texteingabefeld (textbox.aspx)

Wie es funktioniert

Das Formular enthält drei Optionsfelder, die die Auswahl zwischen einer, drei oder sieben Zeilen für das Textfeld erlauben. Auch hier wird wieder `AutoPostBack` verwendet. Die Größe wird in der Methode `Page_Load` voreingestellt. Außerdem wird hier ein Text vorbelegt. Wird eine der Optionen angeklickt, sendet der Browser das Formular an den Server. Die Methode `setBox` wird zur Bearbeitung des Ereignisses ausgelöst. Hier wird zuerst wieder das passende Objekt konvertiert:

```
RadioButton zeilenzahl = (RadioButton) sender;
```

Dann wird etwas getrickst. Die Technik der Web Server-Steuerelemente beruht darauf, mit den vorhandenen Eigenschaften zu arbeiten. Optionsfelder sollen danach über die ID erkannt werden. Wenn Sie aber direkt Werte verarbeiten möchten, bietet sich eigentlich das Attribut `value` an. Erstaunlicherweise gibt es in der Klasse `RadioButton` dazu keine Eigenschaft. Das Attribut kann aber dennoch verwendet werden, indem es direkt dem Tag `<asp:radiobutton>` hinzugefügt wird. Alle Attribute, die nicht direkt als Eigenschaft zur Verfügung stehen, lassen sich über die Kollektion `Attributes` erreichen. Hier wird zur Abwechslung mal direkt auf die Form-Daten zugegriffen, mit dem „alten“ `Request.Form`:

```
int z = Convert.ToInt16(Request.Form["Zeilen"]);
```

Optionsfelder bilden Gruppen über gleiche Namen, das Attribut `groupname` taucht in HTML als `name` auf. Das ist der Name des Feldes, das gesendet wird.

Jetzt wird die `TextBox` entsprechend der Angabe eingerichtet. Dies erfolgt über die Eigenschaft `TextMode`. Zulässig sind drei Eigenschaften, die aus einer Aufzählung vom Typ `TextBoxMode` entnommen werden können:

- `TextBoxMode.MultiLine` – Mehrzeiliges Feld, erzeugt `<textarea>`.
- `TextBoxMode.SingleLine` – Einzeiliges Feld, erzeugt immer `<input type="text">`.
- `TextBoxMode.Password` – Einzeiliges Feld, erzeugt immer `<input type="password">`.

Allein mit der Angabe der Zeilenanzahl ändert sich die Darstellung nicht. Wenn die Zeilenzahl größer als 1 ist, wird der Modus geändert:

```
TextEingabe.TextMode = TextBoxMode.MultiLine;
```

Dann wird die Anzahl der Zeilen zugewiesen:

```
TextEingabe.Rows = z;
```

Das Ergebnis ist, gemessen am Aufwand, durchaus überzeugend und vor allem sehr einfach erweiterbar und mit vielfältigen Formatieroptionen zu ergänzen.

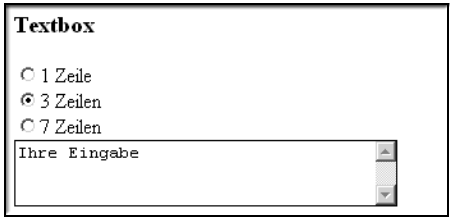


Abbildung 4.12: Steuerung der Zeilenzahl einer TextBox (textbox.aspx)

Was jetzt noch fehlt, ist eine Möglichkeit, das Formular zu senden. Dazu sind Schaltflächen nötig.

4.3.5 Schaltflächen erzeugen und verwenden

Der Umgang mit Schaltflächen beschränkt sich in HTML auf Sendeschaltflächen (Submit) und einige Tricks mit JavaScript. Letzteres ist unumgänglich, verliert aber Dank ASP.NET seine Tücken und wird zu einem gemeinsamen Steuerelement: `Button`.

Eine typische Aufgabe war bislang immer wieder die Reaktion auf mehr als eine Schaltfläche. Dies ist mit so genannten Kommandoschaltflächen möglich. Einheitlich ist dagegen die Darstellung des Steuerelements:

```
<asp:button id="name" runat="server"/>
```

In Vorbereitung auf die Erweiterung des Praxisprojekts wäre es sicher sinnvoll, über mehr als nur einen Bestellknopf zu verfügen. Der Benutzer sollte auch abbrechen können. In ASP.NET lösen Sie dies über Kom-

mandos, die dem Steuerelement mitgegeben werden. Die Methode, die das Klickereignis empfängt, kann diese Kommandos auswerten. Das folgende Beispiel zeigt die Funktionsweise:

```
<script language="C#" runat="Server">
void Page_Load()
{
    b1.Text = "Wähle 1";
    b2.Text = "Wähle 2";
    b3.Text = "Wähle 3";
    b4.Text = "Wähle 4";
    b5.Text = "Wähle 5";
    bCancel.Text = "Abbrechen";
}
void checkVote(object sender, CommandEventArgs e)
{
    if (e.CommandName == "vote")
    {
        Result.Text = "Ihre Wahl war: " + e.CommandArgument;
    } else {
        Result.Text = "Schade...";
    }
}
</script>
<html lang="de">
    <head>
        <title>Buttons</title>
    </head>
    <body>
    <h3>Ihre Bewertung bitte:</h3>
    <form runat="server">
        <asp:button id="b1" runat="server"
                        commandname="vote"
                        commandargument="1"
                        oncommand="checkVote"/>
        <asp:button id="b2" runat="server"
                        commandname="vote"
                        commandargument="2"
                        oncommand="checkVote"/>
        <asp:button id="b3" runat="server"
                        commandname="vote"
                        commandargument="3"
                        oncommand="checkVote"/>
        <asp:button id="b4" runat="server"
                        commandname="vote"
                        commandargument="4"
                        oncommand="checkVote"/>
```

```
        <asp:button id="b5" runat="server"
                        commandname="vote"
                        commandargument="5"
                        oncommand="checkVote"/>
        <asp:button id="bCancel" runat="server"
                        commandname="cancel"
                        oncommand="checkVote"/>
    </form>
    <asp:label id="Result" runat="server"/>
    </body>
</html>
```

Listing 4.13: Schaltflächen mit Kommandos identifizieren (button_command.aspx)

Wie es funktioniert

Die Definition der Steuerelemente unterscheidet sich nur wenig von allen anderen. Neu sind lediglich die Attribute `commandname` und `commandargument`. Zur Definition der Ereignisbehandlung wird diesmal `oncommand` eingesetzt. Interessant ist hier, wie die Kommandos im Programm verarbeitet werden. Die Methode `checkVote` übernimmt im Beispiel diese Aufgabe. Dabei werden die Argumente leicht modifiziert. Bislang wurde das zweite Argument, Typ `EventArgs`, nie verwendet. Diesmal kommt es zum Einsatz, und zwar in einer überladenen Form, `CommandEventArgs`. Nur so haben Sie Zugriff auf die Ereignisse der Kommandoschaltflächen:

```
void checkVote(object sender, CommandEventArgs e)
```

Die Variable `e` enthält nun die Kommandos, die mit den Eigenschaften `CommandName` und `CommandArgument` abgefragt werden können:

```
if (e.CommandName == "vote")
```

Das ist eigentlich alles, was speziell zu dieser Art Schaltfläche zu sagen ist. Die übrigen Eigenschaften betreffen vor allem die Gestaltung.

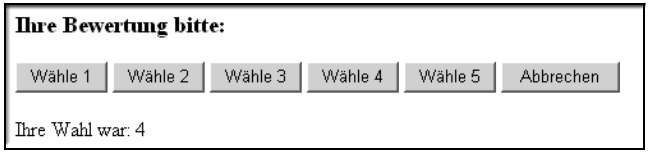


Abbildung 4.13: Mehrere Schaltflächen in einem Formular

Beschriftung der Schaltfläche

Die Kommandos müssen natürlich nicht verwendet werden, wenn es nur eine einfache Sendeschaltfläche gibt. Das Formular gelangt dann auf dem normalen Weg zum Server, wo mit `IsPostBack` festgestellt werden kann, ob die Seite das erste Mal oder nachfolgend aufgerufen wurde. Wollen Sie die Sendeschaltfläche im Programm gestalten, ist es dennoch notwendig, sie als Steuerelement zu deklarieren. Das folgende

Beispiel zeigt, wie die Beschriftung in Abhängigkeit von der Situation geändert werden kann.

```
<script language="C#" runat="Server">
void Page_Load()
{
    if (!IsPostBack)
    {
        senden.Text = "Bitte senden Sie das Formular";
    } else {
        senden.Text = "Formular erneut senden";
    }
}
</script>
<html lang="de">
    <head>
        <title>Sendeschaltfl&auml;che</title>
    </head>
    <body>
    <h1>Sendeschaltfl&auml;che</h1>
    <form runat="server">
        <asp:button id="senden" runat="server"/>
    </form>
    </body>
</html>
```

Listing 4.14: Beschriftung der Sendeschaltfläche dynamisch ändern (button_submit.aspx)

Als einzige Besonderheit ist das Fehlen besonderer Techniken anzumerken. Die Eigenschaft `Text` beeinflusst die Beschriftung der Schaltfläche.

Nach dem das Senden der Formulare mit den gezeigten Methoden möglich ist, sollte auch die Qualität des Inhalts geprüft werden. Eine spezielle Gruppe von Web Server-Steuerelementen ist dafür geeignet – die Kontroll-Steuerelemente (Validation Controls). Beschrieben werden sie in „Kontroll-Steuerelemente (Validation Controls)“ auf Seite 240.

4.3.6 Listen erzeugen und verwalten

Besondere Schwierigkeiten bereiten immer wieder Listen, deren Umfang von dynamischen Daten abhängig ist. ASP.NET kennt dafür mehrere Elemente. Ähnlich wie bei Texteingabefeldern ist die Darstellung logischer und der Windows-Bedienung angepasster. Bei den Textfeldern existierten in HTML zwei Tags, die logisch zu einem zusammengefasst wurden. Für die Listen, die in zwei verschiedenen Darstellungen auftreten, gibt es dagegen nur ein Tag.

Eine weitere Anwendung von Listen finden Sie im Abschnitt „Zugriff über Aufzählungen“ auf Seite 295. Dort wird die Liste mit den Werten eines Arrays gefüllt.

Einfache Listboxen

Bei den Listen gibt es in HTML nur ein Tag, das von zwei Web Server-Steuerelementen erzeugt wird:

- `DropDownList` – Dies sind Listen, die immer einzeilig erscheinen und beim Anklicken herunter klappen (engl. *drop down*, daher der Name). Sie werden in HTML mit `<select size="1">` erzeugt.
- `ListBox` – Hier werden alle Elemente der Liste gleichzeitig angezeigt. Es können ein oder auch mehrere Elemente ausgewählt werden. In HTML wird dazu ebenso `<select>` verwendet, mit dem Attribut `size` und einem Parameter größer als 1. Die Mehrfachauswahl kann mit dem Attribut `multiple` gestattet werden.

Die Elemente der Listen lassen sich über eigene Steuerelemente erzeugen und ändern, `ListItem` genannt. Die Darstellung erfolgt als `<option>`-Tag, das in HTML nur innerhalb einer Liste vorkommen darf. ASP.NET bringt bei Listboxen deutliche Fortschritte gegenüber der herkömmlichen Programmierung. Es war nicht einfach, den Zustand der Liste, also den Status der gewählten Elemente, zu erhalten. Mit Web Server-Steuerelementen ist dies kein Problem mehr. Das folgende Beispiel zeigt, wie die Elemente einer `Listbox` aus einem Array erzeugt werden können:

```
<script language="C#" runat="Server">
void Page_Load()
{
    if (IsPostBack)
    {
        auswahl.Text = "Ihre Wahl war:";
    } else {
        string[] versandarten = {"Express-Paket",
                                 "Over-Night",
                                 "Standard 48 Stunden"};
        int i = 0;
        versand.Items.Clear();
        foreach (string versandart in versandarten)
        {
           ListItem versandoption = new ListItem();
           versandoption.Text = versandart;
           versandoption.Value = i++.ToString();
           versand.Items.Add(versandoption);
        }
```

```
    }
}
</script>
<html lang="de">
    <head>
        <title>Select</title>
    </head>
    <body>
    <h1>W&auml;hlen Sie die Versandart</h1>
    <form runat="server">
        <asp:button id="auswahl" runat="server" text="Bitte wählen"/>
        <asp:dropdownlist id="versand" runat="server"/>
    </form>
    </body>
</html>
```

Listing 4.15: Listbox dynamisch füllen (select_ddlist.aspx)

Wie es funktioniert

Im Formular selbst wird nur das Listen-Steuerelement selbst definiert:

```
<asp:dropdownlist id="versand" runat="server"/>
```

Beim Aufruf der Seite wird ein Array erzeugt, dass die Einträge enthält. Dieses Array wird in einer Schleife durchlaufen:

```
foreach (string versandart in versandarten)
```

Für jedes Element wird nun ein Objekt vom Typ `ListItem` erzeugt:

```
ListItem versandoption = new ListItem();
```

Dann werden diesem Element zwei Attribute zugewiesen, `Text` für den Teil, der angezeigt wird und `Value` für den zu übertragenden Wert. Der zu übertragende Wert ist eine Zahl, die meist einfacher weiterverarbeitet werden kann. Diese Zahl wird mit jedem Durchlauf um eins erhöht (Operator ++) und muss für die Zuweisung noch in eine Zeichenkette konvertiert werden:

```
versandoption.Value = i++.ToString();
```

Zum Schluss wird das `ListItem`-Objekt der Liste hinzugefügt:

```
versand.Items.Add(versandoption);
```

ListItemCollection

Die Eigenschaft `Items` spielt beim Umgang mit Listen eine herausragende Rolle. Objekte, die hier enthalten sind, sind vom Typ `ListItem Collection`.

Auch diese Klasse ist Bestandteil des Namensraumes `System.Web.UI.Web Control`. Da Elemente der Kollektion nicht allein existieren können, son-

dern nur als Bestandteil von `Items`, waren sie in der Übersicht am Anfang des Kapitels nicht enthalten. Wichtigen Eigenschaften sind:

- `Capacity` – Maximale Anzahl Elemente, die erlaubt sind
- `Count` – Aktuelle Anzahl der Elemente
- `Item` – Zugriff auf ein Element. Außerdem ist dies der Indexer, sodass der Zugriff mit der Schreibweise `MeineListe["ListElement"]` erfolgen kann.

Die Methoden entsprechen denen anderer Listen. `Add` fügt ein Element am Ende der Liste hinzu, `Insert` dagegen an einer bestimmten Position. `Clear` löscht alle Elemente, `Remove` dagegen nur ein bestimmtes. Interessant sind zwei spezielle Methoden dieser Listenart: `FindByText` sucht ein Element mit einem bestimmten Inhalt der Eigenschaft `Text`; `FindByValue` entsprechend für `Value`.

Umfangreichere Datenmengen stammen meist aus Datenbanken. Die Zuweisung über einzelne `ListItem`-Objekte wäre umständlich. Weitere Eigenschaften dienen deshalb dem direkten Zugriff auf Datenquellen. Dies wird im Kapitel 6 gezeigt.

Listen aus Optionsfeldern und Kontrollkästchen

Optionsfelder

Auswahlmöglichkeiten in Form von Listen sind nicht immer optimal für die Benutzerführung. Oft sind Kontrollkästchen oder Optionsfelder besser geeignet. Die Erzeugung solcher Gruppen kann mit Objekten vom Typ `CheckBoxList` oder `RadioButtonList` vereinfacht werden. Das Beispiel zeigt, wie dies aussehen kann:

```
<script language="C#" runat="Server">
void Page_Load()
{
    if (IsPostBack)
    {
        auswahl.Text = "Ihre Wahl war:";
    } else {
        string[] versandarten = {"Express-Paket",
                                 "Over-Night",
                                 "Standard 48 Stunden"};
        int i = 0;
        foreach (string versandart in versandarten)
        {
            ListItem versandoption = new ListItem();
            versandoption.Text = versandart;
            versandoption.Value = i++.ToString();
            versand.Items.Add(versandoption);
```

```
        }
    }
}
</script>
<html lang="de">
    <head>
        <title>Select</title>
    </head>
    <body>
    <h1>W&auml;hlen Sie die Versandart</h1>
    <form runat="server">
        <asp:button id="auswahl" runat="server" text="Bitte wählen"/>
        <asp:radiobuttonlist id="versand" runat="server"/>
    </form>
    </body>
</html>
```

Listing 4.16: Dynamisch erzeugte Gruppe von Optionsfeldern (radiobutton_ddlist.aspx)

Tatsächlich besteht der einzige Unterschied gegenüber dem letzten Beispiel im Austausch des Tags im HTML-Teil; hier erscheint nun `<asp:radio- buttonlist>`. Interessanter als eine erneute Analyse ist ein Blick in den erzeugten HTML-Code:

```
<form name="_ctl0" method="post"
      action="radiobutton_ddlist.aspx" id="_ctl0">
<input type="hidden" name="__VIEWSTATE"
value="dDwtNTcwNzM1MzA7dDw7bDxpPDI+Oz47bDx0PDtsPGk8MT47aTwzPjs+O2w8dDxwPHA8bD
xUZXh0Oz47bDxJaHJlIFdhaGwgd2FyOjs+Pjs+Ozs+O3Q8dDw7cDxsPGk8MD47aTwxPjtpPDI+Oz4
7bDxwPEV4cHJlc3MtUGFrZXQ7MD47cDxPdmVyLU5pZ2h0OzE+O3A8U3RhbmRhcmQgNDggU3R1bmRl
bjsyPjs+Pjs+Ozs+Oz4+Oz5UqZgPBlvoAhn2FSJdtw11aq0uSw==" />
<input type="submit" name="auswahl"
       value="Ihre Wahl war:" id="auswahl" />
<table id="versand" border="0">
<tr>
<td><input id="versand_0" type="radio" name="versand"
              value="0" checked="checked" />
       <label for="versand_0">Express-Paket</label></td>
</tr><tr>
<td><input id="versand_1" type="radio" name="versand"
              value="1" />
       <label for="versand_1">Over-Night</label></td>
</tr><tr>
<td><input id="versand_2" type="radio" name="versand"
              value="2" />
       <label for="versand_2">Standard 48 Stunden</label></td>
</tr></table></form>
```

Formatierung von Listen

Interessant ist, dass eine Tabelle zur Formatierung eingesetzt wird. Tatsächlich sind die Formatieroptionen, die sich daraus ableiten lassen, sehr umfangreich. Wenn Sie sehr viele Elemente haben, wie es beispielsweise bei Umfrageseiten der Fall ist, helfen diese Optionen enorm. Hervorzuheben sind folgende Eigenschaften:

- `RepeatColumns` – Anzahl der Spalten, in denen die Anzeige gruppiert wird.
- `RepeatDirection` – Richtung, in der gruppiert wird. Die beiden Tabellen zeigen die Folge der Zellen.

1	3
2	4

Tabelle 4.9: RepeatDirection.Vertical

1	2
3	4

Tabelle 4.10: RepeatDirection.Horizontal

In beiden Fällen ist die Anzahl der Spalten zwei.

- `RepeatLayout` – Diese Eigenschaft bestimmt die Art der Darstellung. Zulässig sind folgende Werte der Aufzählung `RepeatLayout`:
  - `RepeatLayout.Table` – Darstellung als Tabelle (Standardwert)
  - `RepeatLayout.Flow` – Darstellung ohne Tabellenstruktur

Kontrollkästchen

Bei Kontrollkästchen gilt das bereits Gezeigte fast unverändert. Das folgende Beispiel zeigt eine andere Art der Produktpräsentation aus dem Praxisprojekt:

```
<script language="C#" runat="Server">
void Page_Load()
{
    if (!IsPostBack)
    {
        string[,] ProduktArr = new string[3,3];
        const int ProdName = 0;
        const int ProdPreis = 1;
        const int ProdID = 2;
        ProduktArr[0,ProdName] = "Codeschreiber 1.0";
        ProduktArr[0,ProdPreis] = "6299,9";
        ProduktArr[0,ProdID] = "A0001";
        ProduktArr[1,ProdName] = "Datenbanker 1.1";
        ProduktArr[1,ProdPreis] = "99,9";
        ProduktArr[1,ProdID] = "A0002";
        ProduktArr[2,ProdName] = "Projektor 2.0";
```

```
        ProduktArr[2,ProdPreis] = "39";
        ProduktArr[2,ProdID] = "A0003";
        string produkttext, produktid, produktpreis;
        for (int i=0; i < ProduktArr.GetLength(0); i++)
        {
            produkttext = ProduktArr[i, ProdName] + " ("
                                + string.Format("{0:N2}",
                                Convert.ToDouble(ProduktArr[i, ProdPreis]))
                                + ")";
            ListItem produktoption = new ListItem();
            produktoption.Text = produkttext;
            produktoption.Value = ProduktArr[i, ProdID];
            produkte.Items.Add(produktoption);
        }
    }
}
</script>
<html lang="de">
    <head>
        <title>Shop</title>
    </head>
    <body>
        <h1> Shop </h1>
        Unsere Produkte:
        <form runat="server">
            <asp:checkboxlist id="produkte" runat="server"/>
            <asp:button id="bestellen" runat="server"
                        text="Jetzt Bestellen"/>
        </form>
    </body>
</html>
```

Listing 4.17: Mehrfachauswahl mit Kontrollkästchen (shop3.aspx)

Im Programm hat sich gegenüber den letzten Beispielen nur wenig geändert. Als Tag kommt nun `CheckBoxList` zum Einsatz:

```
<asp:checkboxlist id="produkte" runat="server"/>
```

Unverändert ist das Befüllen der Werte, wo erneut die Eigenschaften `Text` und `Value` der `ListItem`-Objekte zum Einsatz kommen. Das Ergebnis ist durchaus überzeugend:

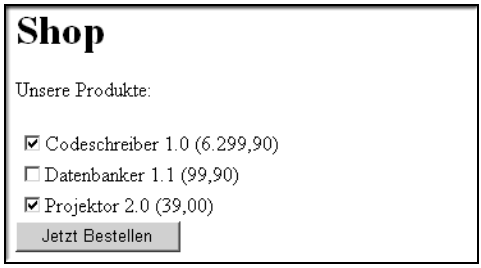


Abbildung 4.14: Produktauswahl mit einem einzigen Tag

Auch hier zeigt ein Blick in den HTML-Code, dass ASP.NET sehr konventionell vorgeht und `<input type="checkbox">`-Tags in einer Tabelle platziert. Die `Repeater`-Eigenschaften stehen auch hier zur Verfügung, falls größere Datenmengen anfallen.

ViewState AutoPostBack

Dank *Viewstate* bleiben in allen Fällen die Werte erhalten, wenn die Formularseite an sich selbst gesendet wird. Auch die Verwendung von *AutoPostBack* ist möglich, damit allein die Aktivierung oder Deaktivierung einer Option das Formular sendet.

4.4 Kontroll-Steuerelemente (Validation Controls)

In den vorangegangenen Abschnitten wurden Formularelemente, bestehend aus HTML Server- oder Web Server-Steuerelementen bereits intensiv genutzt. Es liegt in der Natur der Sache, dass der von Benutzern erfasste Inhalt nicht immer kritiklos angenommen werden kann. Angefangen von einfachen Tippfehlern bis hin zu mutwilligen Falschangaben wird ein Programm viele falsche Daten verkraften müssen. Natürlich ist es möglich, die gesendeten Daten zu überprüfen und dann entsprechend zu reagieren. In der Praxis enthalten aber Formulare viele Felder, die teilweise miteinander in Beziehung stehen. Denken Sie an eine vergleichsweise einfache Kennwortänderung. Hier müssen Sie zum einen die Bedingungen des Kennworts selbst kontrollieren, beispielsweise eine Mindestanzahl Zeichen. Außerdem müssen das Kennwortfeld und ein Kontrollfeld miteinander verglichen werden. Das kann in einer größeren Applikation einigen Programmieraufwand verursachen.

4.4.1 Prinzipien der Feldkontrolle

In ASP.NET werden zur Überprüfung von Feldinhalten so genannte Kontroll-Steuerelemente eingesetzt. Diese stammen aus der Gruppe der Web Server-Steuerelemente und sind vergleichsweise einfach einsetzbar.

Einige Grundlagen sollten Sie vorher kennen lernen, denn dies erleichtert die Entscheidung für die eine oder andere Option bei der Programmierung.

Prüfung: Client oder Server?

Prinzipiell werden in der Formularprogrammierung zwei Prüfformen unterschieden:

- Clientseitige Prüfung mit JavaScript
- Serverseitige Prüfung

Die clientseitige Prüfung nutzt JavaScript und verhindert üblicherweise, dass nicht korrekt ausgefüllte Formulare gar nicht erst abgesendet werden. Dies verringert die Netzlast auf dem Server und beschleunigt die Abarbeitung, denn das Senden an den Server und die Verarbeitung der Antwort entfällt. Nachteilig ist, dass die Präsentation der Fehler im Formular mit JavaScript und DHTML aufwändig zu programmieren ist. Außerdem funktioniert die Applikation möglicherweise überhaupt nicht mehr, wenn der Benutzer aus Sicherheitsgründen das Scripting im Browser grundsätzlich unterbindet.

Bei der serverseitigen Prüfung werden die Formulardaten empfangen, ausgewertet und dann wird eine entsprechende Reaktion an den Browser gesendet, beispielsweise ein neues Formular mit detaillierten Ausfüllhinweisen.

Auswahl der Feldkontrolle

Normalerweise würden sich an dieser Stelle endlose Debatten um die Vorzüge von JavaScript und DHTML anschließen, bzw. um handfeste Gründe, warum der Einsatz unbedingt vermieden werden muss. ASP.NET nimmt Ihnen die Entscheidung eigentlich ab – wenngleich Sie dennoch die letzte Kontrolle darüber behalten. Prinzipiell realisieren die Kontroll-Steuerelemente immer serverseitige Funktionen. Wenn jedoch ein Browser DHTML unterstützt, werden zusätzlich clientseitige Prüfungen eingebaut. Damit wird sichergestellt, dass Ihre Applikation in jeder Umgebung und unter allen Bedingungen funktioniert. Wenn es möglich ist, Bandbreite einzusparen, wird dies auch getan. Allerdings kann die Verwendung von DHTML explizit unterdrückt werden. Dazu wird einfach die Direktive `@Page` um ein entsprechendes Attribut erweitert:

```
<% @Page ClientTarget="Downlevel" %>
```

Die serverseitige Kontrolle kann dagegen nicht unterdrückt werden. Es mag in Anbetracht einer funktionierenden clientseitigen Überprüfung der Felder unnötig erscheinen. Allerdings gehört für einen halbwegs be-

gabten Hacker nicht viel dazu, das Formular auszulesen, den JavaScript-Code zu analysieren, zu verändern und dann das Formular mit fehlerhaften Daten an Ihren Server zu senden. Fatal, wenn dann keine zweite Kontrollmauer existiert, die für Hacker definitiv nicht erreichbar ist.

Ablauf der Prüfung

Im Objekt `Page` gibt es eine Eigenschaft `IsValid`, die anzeigt, ob Fehler im Formular auftraten (`false`) oder nicht (`true`). Dabei spielt es keine Rolle, ob ein oder mehrere Prüfungen nicht bestanden wurden, um das Formular ungültig werden zu lassen. Wie Sie auf den Ausgang reagieren, können Sie selbst entscheiden, an der Bereitstellung der Formulardaten in der `Form`-Kollektion oder als Eigenschaft der Server-Steuerelemente ändert sich nichts.

4.4.2 Typische Kontroll-Steuerelemente

Die häufigste Prüfung besteht in der Feststellung, ob ein Feld überhaupt ausgefüllt wurde. Dazu wird das Steuerelement `RequiredFieldValidator` eingesetzt. Das folgende Listing zeigt, wie es verwendet wird. Realisiert wird der erste Teil eines Eingabeformulars für den Shop, wo Kunden ihre Adressen eingeben können. Verständlich ist sicher, dass bestimmte Angaben, wie Name und Strasse, unbedingt erforderlich sind.

```
<% @Page Language="C#" debug="true" %>
<script runat="server">
void Page_Load()
{

}
</script>
<html>
    <head>
        <title>Validation Control - 1</title>
    </head>
    <body>
        Bitte geben Sie die Lieferadresse an:<br/>
        <form runat="server">
        <table border="1">
            <tr>
                <td>Name</td>
                <td>
                    <asp:textbox runat="server" id="f_name" />
                </td>
                <td>
```

```
                <asp:requiredfieldvalidator runat="server" id="v_name"
                     controltovalidate="f_name" display="dynamic">
                    Geben Sie einen Namen ein!
                </asp:requiredfieldvalidator>
            </td>
        </tr>
        <tr>
            <td>Strasse</td>
            <td>
                <asp:textbox runat="server" id="f_street" />
            </td>
            <td>
                <asp:requiredfieldvalidator runat="server" id="v_street"
                     controltovalidate="f_street" display="dynamic">
                    Geben Sie eine Stra&szlig;e an!
                </asp:requiredfieldvalidator>
            </td>
        </tr>
    </table>
    <asp:button runat="server" id="send" text="Bestellen"/>
    </form>
  </body>
</html>
```

Listing 4.18: Formular mit zwei kontrollierten Eingabefeldern (ValidationControls/vc_required.aspx)

Wie es funktioniert

Dieses Formular verfügt – vor allem in Anbetracht der leeren `Page_Load`-Methode – bereits über beträchtliche Funktionen. Beide Felder werden auf den Zustand „Feld ausgefüllt" hin überprüft. Bleibt ein Feld leer, wird rechts daneben eine Fehlermeldung in roter Schrift eingeblendet. Wenn Sie Ihren Browser genau beobachten, werden Sie außerdem feststellen, dass das Formular nicht gesendet wurde. Das erstaunliche Ergebnis basiert auf einer clientseitigen Prüfung. Die gesamte Funktionalität steckt im folgenden Kontroll-Steuerelement:

```
<asp:requiredfieldvalidator runat="server" id="v_street"
    controltovalidate="f_street" display="dynamic">Geben Sie
eine Stra&szlig;e an!</asp:requiredfieldvalidator>
```

Der Name `asp:requiredfieldvalidator` bestimmt die Art der Prüfung: Das Feld muss mindestens ein Zeichen enthalten. Dann wird mit dem Attribut `controltovalidate` festgelegt, welches Feld überhaupt überwacht werden soll. Damit sind die unbedingt notwendigen Angaben bereits erfolgt – abgesehen von den obligatorischen Attributen `id` und `runat`.

Steuerung der Anzeige

Das Attribut `display` ist eine nähere Betrachtung Wert. Die Ausgabe von Fehlermeldungen inmitten eines Formulars ist bei sorgfältig gestalteten

Seiten keine triviale Aufgabe. Schließlich soll die zusätzlich erscheinende Meldung das Layout nicht zerstören. Andererseits soll die am Anfang und hoffentlich auch später fehlende Fehlernachricht nicht weiße Flecken hinterlassen. Kreative Webdesigner werden sicher einen vernünftigen Weg finden; deren Vorgabe umzusetzen, ist Aufgabe des Attributes `display`:

Wert des Attributes display	Verhalten der Fehlernachricht
`Dynamic`	Der Text erscheint dynamisch, wenn erforderlich. Fehlt die Nachricht, wird kein Platz in Anspruch genommen
`Static`	Der Text erscheint dynamisch, wenn erforderlich. Der dafür erforderliche Platz wird immer in Anspruch genommen
`None`	Der Text erscheint niemals.

Tabelle 4.11: Wirkungsweise des Attributes `display`

Die folgende Abbildung zeigt, wie sich das Kontroll-Steuerelement mit dem Attribut `display="Dynamic"` verhält:

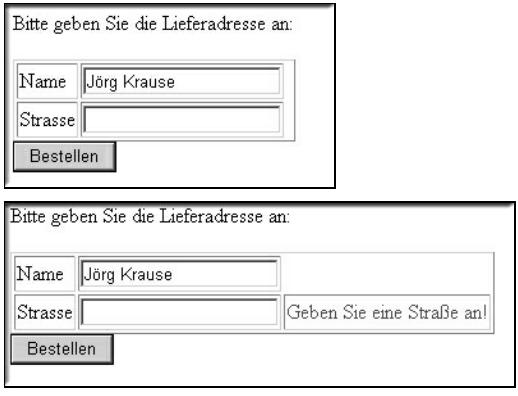


Abbildung 4.15: Dynamische Anzeige einer Fehlermeldung

Die Tabelle wird durch das in der rechten Spalte platzierte Kontroll-Steuerelement dynamisch erweitert. Das kann, wenn die Tabelle ein wichtiges Gestaltungselement der Seite ist, den gesamten Aufbau zerstören. Würde als Attribut `display="static"` verwendet werden, sähe die Tabelle folgendermaßen aus: siehe Abbildung 4.16

Interessant ist natürlich, was ASP.NET eigentlich an den Browser gesendet hat. Ein Blick in den Quelltext lohnt sich auch hier.

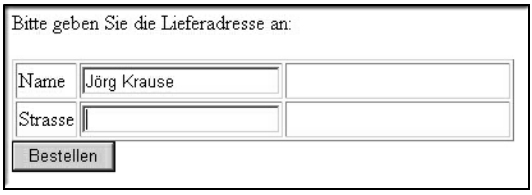


Abbildung 4.16: Tabelle mit reserviertem Platz für Meldungen

Wie die clientseitige Prüfung funktioniert

Die clientseitige Prüfung basiert auf JavaScript. Dazu stellt ASP.NET Skriptbibliotheken zur Verfügung, die im IIS unter dem virtuellen Verzeichnis */aspnet_client* zu finden sind. Das Absenden des Formulars wird durch einen entsprechenden Eintrag kontrolliert:

```
<form ... action="vc_required.aspx" onsubmit="ValidatorOnSubmit();">
```

Diese Funktion führt nach einer Versionsprüfung in die Bibliothek *WebUIValidation.js*. Von dort werden die Fehlermeldungen auf der Seite kontrolliert. Diese werden in `<span>`-Tags platziert, die dann über DHTML sichtbar oder unsichtbar gemacht werden können:

```
<span id="v_name" controltovalidate="f_name"
      evaluationfunction="RequiredFieldValidatorEvaluateIsValid"
      initialvalue="" style="color:Red;visibility:hidden;">
Geben Sie einen Namen ein!
</span>
```

Mit `visibility:hidden` wird die Anzeige am Anfang unterdrückt, der vom Text in Anspruch genommene Platz ist jedoch reserviert. Es wurde also `display="static"` verwendet. Wenn dagegen `display="dynamic"` verwendet wird, entsteht folgendes Tag:

```
<span id="v_name" controltovalidate="f_name" display="Dynamic"
      evaluationfunction="RequiredFieldValidatorEvaluateIsValid"
      initialvalue="" style="color:Red;display:none;">
Geben Sie einen Namen ein!
</span>
```

Mit dem CSS-Attribut `display:none` wird die Anzeige also völlig unterdrückt.

Diese Aussagen treffen für alle Kontroll-Steuerelemente zu, denn die Art der Fehlerausgabe ist für alle identisch.

Serverseitige Prüfungen verwenden

Die Einzelprüfung ist zwar praktikabel und zuverlässig, eine serverseitige Prüfung kann jedoch noch mehr bieten. Im Sinne einer guten Benutzerführung wäre es empfehlenswert, dem Benutzer an zentraler Stelle gut sichtbar einen allgemeinen Hinweis auf das Problem zu geben. Das folgende Beispiel zeigt ein solches Programm. Hier wird ausschließlich serverseitig geprüft und eine zusätzliche Meldung bei Bedarf angezeigt:

```
<% @Page ClientTarget="Downlevel" Language="C#" debug="true" %>
<script runat="server">
void checkform(object sender, EventArgs e)
{
    if (IsValid)
    {
        error.Visible = false;
    } else {
        error.Visible = true;
    }
}
void Page_Load()
{
    error.Visible = false;
}
</script>
<html>
    <head>
        <title>Validation Control - 1</title>
    </head>
    <body>
        Bitte geben Sie die Lieferadresse an:<br/>
        <form runat="server">
        <div style="background-color:red; color:white" id="error"
                    runat="server">
        Es traten Fehler beim Ausf&uuml;llen des Formulars auf.
        </div>
... Rest des Listings entspricht dem vorherigen
```

Listing 4.19: Serverseitige Prüfung mit zentraler Fehlermeldung (ValidationControls/ca_required_isvalid.aspx)

Wie es funktioniert Zuerst wird hier die clientseitige Prüfung mit `ClientTarget="Downlevel"` unterdrückt. Dann wird ein Klickereignis definiert, dass das Absenden des Formulars aufnimmt. Hier wird die Eigenschaft `IsValid` geprüft und ein HTML Server-Steuerelement, dass als `<div>`-Tag vorliegt, wird entsprechend zur Anzeige gebracht, wenn Fehler auftraten. Beachten Sie,

dass Sie diesen Teil nicht in `Page_Load` schreiben können, denn zum Zeitpunkt, wo die Seite geladen wurde, hat die Prüfung noch nicht stattgefunden. Die Kontroll-Steuerelemente müssen in der Lage sein, auf die Seitengestaltung Rücksicht zu nehmen bzw. alle Elemente der Seite kennen. Deshalb wird die Seite erst soweit abgearbeitet, dass `Page_Load` ausgeführt werden kann und dann erfolgt die Prüfung.

4.4.3 Weitere wichtige Kontroll-Steuerelemente

Dieser Abschnitt stellt anhand einfacher Beispiele die wichtigsten Kontroll-Steuerelemente und deren Anwendung vor.

Vergleichs-Steuerelement (Compare Validator)

Das Vergleichs-Steuerelement (Compare Validator) vergleicht ein Feld mit einem konstanten Wert, einem Datentyp oder einem anderen Feld. Da es möglich ist, mehr als ein Kontroll-Steuerelement an ein Feld zu binden, können damit auch Bereiche eingeschränkt werden. Die Kombination mit dem Steuerelement `RequiredFieldValidator` ist erforderlich, denn das Kriterium das Vorhandensein eines Wertes erzwingt es nicht. Das folgende Beispiel erweitert die Adressabfrage und enthält zusätzlich die Festlegung eines Kennwortes.

```
<tr>
    <td>Kennwort</td>
    <td>
        <asp:textbox runat="server" id="f_password1" textmode="password"/>
        <br/>
        <asp:textbox runat="server" id="f_password2" textmode="password"/>
    </td>
    <td>
        <asp:comparevalidator id="Compare1" type="string" runat="server"
                              controltovalidate="f_password1"
                              controltocompare="f_password2">
        Die Kennwortfelder stimmen nicht überein
        </asp:comparevalidator>
    </td>
<tr>
```

Listing 4.20: Mehrere Prüfungen mit dem `CompareValidator` Steuerelement (Ausschnitt aus ValidationControls/vc_compare.aspx)

Das Steuerelement `CompareValidator` verfügt über eine ganze Reihe von Attributen. Dadurch ist der Einsatz sehr oft möglich. Neben den obligatorischen Attributen `runat` und `id` ist der Typ der Daten anzugeben, die

das geprüfte Feld enthalten soll. Dazu dient das Attribut `type`. Es kann folgende Parameter enthalten:

Parameter für type	Beschreibung
`String`	Eine Zeichenkette wird erwartet
`Integer`	Ein ganzzahliger Wert wird akzeptiert
`Double`	Das Feld muss eine Zahl enthalten
`Date`	Ein Datumswert entsprechend dem
`DateTime`	wird erwartet
`Currency`	Eine Währungsangabe – Zahl mit Währungssymbol – wird erwartet

Tabelle 4.12: Datentypprüfung für `CompareValidator`

Soll lediglich der Datentyp eines Feldes überwacht werden, ist nur das Attribut `operator` mit dem Parameter `DataTypeCheck` erforderlich. Folgende Kombination nutzt dies:

```
... type="Integer" operator="DataTypeCheck" ...
```

Zwei Felder vergleichen

Sollen dagegen zwei Felder verglichen werden, sind zwei andere Attribute beteiligt:

`controltovalidate` und `controltocompare`:

```
... controltovalidate="f_password1" controltocompare="f_password2" ...
```

Die Kombination mit `type` ist möglich, dann wird neben der Gleichheit der Felder auch die Übereinstimmung des Datentyps geprüft. Das Attribut `operator` ist dann optional. Für alle anderen Fälle, in denen ein Vergleich erfolgen soll, können Sie `operator` mit folgenden Parametern ausstatten:

Parameter für operator	Beschreibung
`DataTypeCheck`	Datentyp
`Equal`	Gleichheit (=)
`GreaterThan`	Größer als (>)
`LessThan`	Kleiner als (<)
`GreaterThanEqual`	Größer als oder gleich (>=)
`LessThanEqual`	Kleiner als oder gleich (<=)
`NotEqual`	Ungleich (<>)

Tabelle 4.13: Vergleichsopratoren für `CompareValidator`

Der Vergleich kann nicht nur zwischen Feldern, sondern auch mit einem konstanten Wert erfolgen. Dazu wird statt `controltocompare` das Attribut `valuetocompare` verwendet. Für den Vergleich von Zahlen ist das

Attribut `type` erforderlich; `operator` mit einem der genannten Parameter sowieso:

```
<asp:textbox runat="server" id="f_age" />
<asp:comparevalidator id="v_age" runat="server"
                      controltovalidate="f_age"
                      valuetocompare="18"
                      type="Integer"
                      operator="GreaterThan"
                      ErrorMessage="Das Alter muss mindestens 18 sein"/>
```

Der Operator ist so zu gestalten, dass er die Bedingung für eine gültige Prüfung ergibt. Im Beispiel wird die Bedingung „f_age > 18“ geprüft. Ist dieser Vergleich nicht erfüllt, wird die Meldung ausgegeben.

Ausgabe der Fehlermeldung

Die Ausgabe der Fehlermeldung kann auf zwei Wegen erfolgen: Entweder nutzen Sie das Attribut `ErrorMessage` und schließen das Steuerelement sofort ab:

```
<asp:comparevalidator ErrorMessage="Feld falsch" .../>
```

Oder Sie bauen ein Container-Tag, dass die Meldung enthält:

```
<asp:comparevalidator ... >Feld falsch</asp:comparevalidator>
```

Bereichskontroll-Steuerelement (RangeValidator Control)

Manchmal reicht der Vergleich von Feldern mit Festwerten nicht aus. Der `RangeValidator` prüft, ob Werte in einem bestimmten Bereich liegen. So könnte die Prüfung der Postleitzahlen im letzten Beispiel verbessert werden, indem statt der Prüfung des Datentyps eine Einschränkung auf den Bereich zwischen 1000 und 99.999 erfolgt. Der `RangeValidator` arbeitet ähnlich dem `CompareValidator`, verwendet aber zur Kontrolle zwei spezielle Attribute, die den Wertebereich eingrenzen: `MinimumValue` und `MaximumValue`.

```
<tr>
    <td>PLZ</td>
    <td>
        <asp:textbox runat="server" id="f_zipcode" width="5"/>
    </td>
    <td>
        <asp:rangevalidator runat="server" id="v_zipcode" type="Integer"
                            minimumvalue="1000" maximumvalue="99999"
                            controltovalidate="f_zipcode" display="dynamic">
        Postleitzahl im falschen Wertebereich
        </asp:rangevalidator>
```

```
        <asp:requiredfieldvalidator runat="server" id="vr_zipcode"
                            controltovalidate="f_zipcode" display="dynamic">
        Postleitzahl nicht angegeben
        </asp:requiredfieldvalidator>
    </td>
</tr>
```

Listing 4.21: Kontrolle eines Wertebereiches (Ausschnitt aus ValidationControls/vc_compare.aspx)

Regulärer Ausdruck-Steuerelement (Regular Expression Validator)

In allen Fällen, in denen die Prüfung von Eingabewerten nicht mittels Übereinstimmung, Datentyp oder Bereich erfolgen kann, sind reguläre Ausdrücke ein ideales Mittel. Reguläre Ausdrücke bestehen aus einem definierter Satz von Steuerzeichen, mit denen Suchmuster gebildet werden können. Eine Einführung in die Technik und Anwendung finden Sie im Abschnitt „Reguläre Ausdrücke“ auf Seite 329. Dort werden spezielle Klassen des Frameworks verwendet, um Text mit solchen Ausdrücken zu durchsuchen. Hier wird dagegen der `RegularExpressionValidator` verwendet, um Werte zu prüfen.

Ein häufig benötigtes Eingabefeld ist die E-Mail-Adresse. Der dazu benötigte Ausdruck sieht folgendermaßen aus:

```
^
   [_a-zA-Z0-9-]+
   (\.
      [_a-zA-Z0-9-]+
   )*
   @
   (
      [a-zA-Z0-9-]+
      \.
   )+
   (
      [a-zA-Z]{2,3}
)
$
```

Listing 4.22: Regulärer Ausdruck zum Prüfen einer E-Mail-Adresse

Das kryptische Gebilde wurde hier auseinandergezogen, um die einzelnen Elemente kommentieren zu können. In der Praxis schreibt man die Zeichen in eine Zeile. Konsultieren Sie Abschnitt „Reguläre Ausdrücke“ auf Seite 329, um herauszufinden, wie der Ausdruck funktioniert und wie Sie selbst Suchmuster entwerfen können.

Die Anwendung ist einfacher. Das Steuerelement verfügt über ein neues Attribut, `ValidationExpression`. Die Prüfung der E-Mail-Adresse sieht dann folgendermaßen aus:

```
<tr>
    <td>E-Mail</td>
    <td>
        <asp:textbox runat="server" id="f_email" />
    </td>
    <td>
        <asp:regularexpressionvalidator runat="server" id="v_email"
             validationexpression="^[_a-zA-Z0-9-]+(\.[_a-zA-Z0-9-]
                                  +)*@([a-zA-Z0-9-]+\.)+([a-zA-Z]{2,3})$"
             controltovalidate="f_email" display="dynamic">
        Diese E-Mail-Adresse ist nicht gültig
        </asp:regularexpressionvalidator>
    </td>
</tr>
```

Listing 4.23: Prüfung einer E-Mail-Adresse (Ausschnitt aus ValidationControls/vc_compare.aspx)

Grenzen regulärer Ausdrücke

Reguläre Ausdrücke können Suchmuster aufbauen. Es gibt zwar prinzipiell die Möglichkeit, Abhängigkeiten zwischen Teilen des Ausdrucks festzulegen, die Werte sind aber statischer Natur. Berechnungen lassen sich nicht anstellen. Ein typisches Beispiel sind ISBN-Nummer, Datumswerte oder Kreditkartennummern. In allen drei Fällen sind die Abhängigkeiten komplizierter. Das Muster einer ISBN-Nummer ist statisch: L-VVV-NNNNN-P. Dabei steht L für das Land, beispielsweise 3 für Deutschland, VVV für den Verlag (dieser Teil kann auch vier- oder fünfstellig sein) und NNNNN ist die eigentliche Buchnummer, wobei die Anzahl so gewählt wird, dass die gesamte Zeichenfolge ohne Bindestriche immer 10 Zeichen umfasst. Bis dahin wäre ein regulärer Ausdruck in der Lage, eine Prüfung vorzunehmen. P ist aber eine Prüfziffer, die auf Basis der anderen Zahlen berechnet werden muss. Dies kann nicht mit einem Suchmuster erfolgen. Sie müssen dazu eine eigene Funktion schreiben, die die Prüfziffer selbst berechnet und vergleicht.

4.4.4 Selbstdefinierte Kontrollelemente (Custom Validation Controls)

Natürlich ist es ohne weiteres möglich, die empfangenen Daten auf dem Server aus der `Form`-Kollektion oder der `Text`-Eigenschaft zu entnehmen und eine Prüffunktion zu schreiben. Im Sinne eines einheitlichen, übersichtlichen und geradlinigen Programmierkonzepts ist aber der Einbau in ein Kontroll-Steuerelement zu empfehlen. Dafür gibt es den

`CustomValidator`. Auch dieses Steuerelement ist mit den bisher vorgestellten vergleichbar. Ein zusätzliches Attribut `onservervalidate` dient der Angabe einer Methode, die sie selbst definieren und die die `IsValid`-Eigenschaft setzt.

Serverseitige Prüfungen

Das folgende Beispiel zeigt die Prüfung von ISBN-Nummern. Zuerst das Listing des HTML-Codes. Die Prüfung ist in eine hinterlegte Code-Datei ausgelagert, die danach vorgestellt wird:

```
<% @Page Language="C#" Inherits="checker" src="checker.cs" %>
<html>
    <head>
        <title>Validation Control - 3</title>
    </head>
    <body>
        Nach welcher ISBN m&ouml;chten Sie suchen?<br/>
        <form runat="server">
            <asp:textbox runat="server" id="f_isbn" />
            <asp:button runat="server" id="send" text="Prüfen"/>
            <br/>
            <asp:customvalidator runat="server" id="v_isbn"
                                 onservervalidate="checkisbn"
                                 display="dynamic">
            ISBN-Nummer ist falsch
            </asp:customvalidator>
            (Gepr&uuml;fte Nummer: <asp:label id="l_isbn" runat="server"/>)
        </form>
    </body>
</html>
```

Listing 4.24: HTML-Teil mit der Definition des benutzerdefinierten Kontroll-Steuerelements (ValidationControls/vc_custom.aspx)

Definiert wird hier neben dem Eingabefeld (`f_isbn`) und der Sendeschaltfläche (`send`) auch ein `Label`, dass die gewählte Nummer oder eine Fehlermeldung anzeigt. Alle Berechnungen werden in der Datei *checker.cs* definiert:

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public class checker : Page
{
    public CustomValidator v_isbn;
    public TextBox f_isbn;
    public Label l_isbn;
```

```
    private int substr(string isbn, int pos1, int pos2)
    {
        return Convert.ToInt32(isbn.Substring(pos1, pos2));
    }

    public void checkisbn(object sender, ServerValidateEventArgs e)
    {
        string isbn = f_isbn.Text;
        string isbn10 = isbn.Replace("-", "");
        l_isbn.Text = isbn10;
        if (isbn10.Length != 10)
        {
            l_isbn.Text = "Anzahl Ziffern stimmt nicht = "
                          + isbn.Length.ToString();
            e.IsValid = false;
        } else {
            int checkdigit = 11 - ( ( 10 * substr(isbn10,0,1) + 9 *
                                        substr(isbn10,1,1) + 8 *
                                        substr(isbn10,2,1) + 7 *
                                        substr(isbn10,3,1) + 6 *
                                        substr(isbn10,4,1) + 5 *
                                        substr(isbn10,5,1) + 4 *
                                        substr(isbn10,6,1) + 3 *
                                        substr(isbn10,7,1) + 2 *
                                    substr(isbn10,8,1) ) % 11);
            int comparedigit = (isbn10.Substring(9,1) == "X") ? 10 :
                               Convert.ToInt32(isbn10.Substring(9,1));
            checkdigit = (checkdigit == 11) ? 0 : checkdigit;
            e.IsValid = (checkdigit == comparedigit);
        }
    }
}
```

Listing 4.25: Klasse zur Prüfung von ISBN-Nummern (checker.cs im Verzeichnis ValidationControls)

Wie es funktioniert

Die Datei beginnt mit dem Zugriff auf die benötigten Namensräume. Die verwendeten Zeichenketten-Methoden aus der Klasse `System.String` sind im Basisnamensraum `System` definiert. Die Klasse `checker` muss außerdem von `Page` erben und natürlich öffentlich zugänglich sein:

```
public class checker : Page
```

Dann werden die Web Server- bzw. Kontroll-Steuerelemente der Seite bekannt gemacht:

```
public CustomValidator v_isbn;
public TextBox f_isbn;
public Label l_isbn;
```

Im Programm wird sehr häufig eine Zeichenkettenoperation eingesetzt, bei der ein Teil einer Zeichenkette als Zahlenwert benötigt wird. Deshalb wurde dafür eine private Methode `substr` definiert. Die eigentliche Arbeit erledigt die Methode `checkisbn`:

```
public void checkisbn(object sender, ServerValidateEventArgs e)
```

Das Argument `e` erlaubt den Zugriff auf das Kontroll-Steuerelement selbst, dass beim Aufruf als Objekt übergeben wird.

Zuerst erfolgt der Zugriff auf das Eingabefeld über die Eigenschaft `Text`:

```
string isbn = f_isbn.Text;
```

Dann werden möglicherweise mit eingegebene Bindestriche entfernt, die ISBN-Nummern nur an den sinntrennenden Stellen auflockern, sonst aber für die Berechnung keine Bedeutung haben. Die Zeichenkettenmethode `Replace` wird hier verwendet:

```
string isbn10 = isbn.Replace("-", "");
```

Dann wird dem Label der Text der ISBN zugewiesen:

```
l_isbn.Text = isbn10;
```

Im nächsten Schritt wird die Länge geprüft; ISBN-Nummern haben immer 10 Ziffern. Die tatsächliche Länge wird mit der Zeichekettenmethode `Length` ermittelt:

```
if (isbn10.Length != 10)
```

Stimmt die Länge nicht, wird eine entsprechende Information an das Label ausgeben und der Status des Kontroll-Steuerelements so gesetzt, dass die Fehlermeldung angezeigt wird. Dazu wird auf die Eigenschaft `IsValid` zugegriffen:

```
l_isbn.Text = "Anzahl Ziffern stimmt nicht = " + isbn.Length.ToString();
e.IsValid = false;
```

Beachten Sie hier, dass Sie nicht nur `IsValid` schreiben, denn dann wird `Page.IsValid` verwendet. Das ist zwar nicht prinzipiell falsch, denn mit der Ungültigkeit eines Kontroll-Steuerelements wird auch der Prüfzustand der gesamten Seite ungültig, aber im Beispiel wird `Page.IsValid` nicht ausgewertet. Um die Fehlermeldung zu provozieren, muss das Objekt `e` angesprochen werden.

Sind alle Prüfungen erfolgt, wird die Berechnung durchgeführt. Die Formel gibt Prüfziffern zwischen 0 und 11 zurück. In der ISBN wird die 11 als „0“ und die 10 als „X“ dargestellt. Die letzte Stelle der ISBN-Nummer muss zum Vergleich entsprechend umgewandelt werden:

```
int comparedigit = (isbn10.Substring(9,1) == "X") ? 10 :
Convert.ToInt32(isbn10.Substring(9,1));
```

Dann folgt der Vergleich. Das Ergebnis wird gleich für `IsValid` verwendet:

```
e.IsValid = (checkdigit == comparedigit);
```

Clientseitige Prüfungen

Es ist durchaus möglich, auch clientseitige Prüfungen zu programmieren. Dazu müssen Sie die entsprechende Funktion in JavaScript schreiben. Als Attribut für das Kontroll-Steuerelement wird `clientvalidationfunction` verwendet. Werden beide Definitionen verwendet, entscheidet ASP.NET anhand der Möglichkeiten des Browsers, welche eingesetzt wird. Das folgende Beispiel zeigt, wie zusätzlich zur bereits gezeigten serverseitigen Kontrolle die Definition der Prüfziffernberechnung in JavaScript erfolgen kann:

```
<% @Page Language="C#" debug="true" Inherits="checker" src="checker.cs" %>
<html>
    <head>
        <title>Validation Control - 3</title>
        <script language="JavaScript">
        function checkisbn(sender, e)
        {
            isbn = document.checkform.f_isbn.value;
            isbn10 = isbn.replace(/-/g, "");
            if (isbn10.length != 10)
            {
                alert("Anzahl Ziffern falsch: " + isbn10.length + " ("
                                                + isbn10 + ")");
                e.IsValid = false;
            } else {
                checkdigit = 11 - ( ( 10 * isbn10.substr(0,1) + 9 *
                                        isbn10.substr(1,1) + 8 *
                                        isbn10.substr(2,1) + 7 *
                                        isbn10.substr(3,1) + 6 *
                                        isbn10.substr(4,1) + 5 *
                                        isbn10.substr(5,1) + 4 *
                                        isbn10.substr(6,1) + 3 *
                                        isbn10.substr(7,1) + 2 *
                                        isbn10.substr(8,1) ) % 11);
                checkdigit = (checkdigit == 11) ? 0 : checkdigit;
```

```
                comparedigit = (isbn10.substr(9,1) == "X") ? 10 :
                                isbn10.substr(9,1);
                e.IsValid = (checkdigit == comparedigit);
            }
        }
        </script>
    </head>
    <body>
        Nach welcher ISBN m&ouml;chten Sie suchen?<br/>
        <form runat="server" id="checkform">
            <asp:textbox runat="server" id="f_isbn" />
            <asp:button runat="server" id="send" text="Prüfen"/>
            <br/>
            <asp:customvalidator runat="server" id="v_isbn"
                                 onservervalidate="checkisbn"
                                 clientvalidationfunction="checkisbn"
                                 display="dynamic">
            ISBN-Nummer ist falsch
            </asp:customvalidator>
            (Gepr&uuml;fte Nummer: <asp:label id="l_isbn" runat="server"/>)
        </form>
    </body>
</html>
```

Listing 4.26: Clientseitige Realisierung einer Prüffunktion (ValidationControls/vc_custom_js.aspx)

Wie es funktioniert

Die Funktion `checkisbn` selbst unterscheidet sich kaum von der bereits in C# gezeigten, die Sprachen sind sich an dieser Stelle sehr ähnlich. Beachtenswert ist der Zugriff auf die Feldwerte, der über das DOM-Modell des Browser erfolgt:

```
isbn = document.checkform.f_isbn.value;
```

Damit das funktioniert, wurde das Formular mit der ID `checkform` versehen, die in HTML auch als Attribut `name` verwendet wird.

Im Beispiel erfolgt die Ausgabe der Fehlermeldung bei falscher Anzahl Ziffern über die `alert`-Funktion von JavaScript. Wenn Sie die Ausgabe über das Label erledigen möchten, müssen Sie den Zugriff auf die HTML-Elemente über das Objektmodell des Browsers kennen. ASP.NET erzeugt für `<asp:label>` ein `<span>`-Element, das über das Attribut `id` erreicht werden kann. Tauschen Sie die Zeile mit `alert` gegen folgende aus, um dynamisch auf HTML zugreifen zu können:

```
document.getElementById("l_isbn").innerHTML = "Anzahl Ziffern falsch:"
+ isbn10.length;
```

Mit `getElementById(id)` kann im Browser jederzeit auf alle Steuerelemente der Seite zugegriffen werden, die das Attribut `id` tragen. Das Attribut `runat="server"` kennt JavaScript nicht, es ist nicht notwendig, stört aber auch nicht.

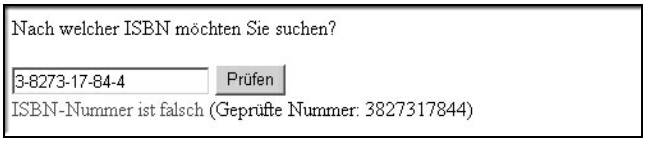


Abbildung 4.17: Reaktion des Programms auf eine falsche ISBN

Im Sinne eines gutes Programmstils sollten Sie immer client- und serverseitige Steuerelemente realisieren. Wenn der Browser JavaScript unterstützt, spart es Bandbreite und die Verarbeitung der Seite wirkt auf den Benutzer schneller. Ist JavaScript nicht einsetzbar, funktioniert es trotzdem.

4.5 Benutzer-Steuerelemente (User Controls)

ASP.NET bietet bei der Programmierung großer Projekte eine starke Basis für den Entwickler. Eines der wichtigen Instrumente zur Modularisierung sind Benutzer-Steuerelemente (User Controls).

4.5.1 Grundlagen

Benutzer-Steuerelemente sind so einfach und flexibel, dass der Einsatz sich auch schon bei kleineren Projekten lohnen kann. Praktisch werden hier Objekte definiert, die ähnlich wie die Web Server-Steuerelemente einen eigenen Namensraum besitzen. Wenn die Erweiterung von HTML über den Namensraum „asp:“ gelingt, sollte es auch möglich sein, eigene Namensräume zu deklarieren. Tatsächlich ist dies kein Trick, sondern vom Systemdesign vorgegeben. Eine solche Erweiterung wird als Benutzer-Steuerelement bezeichnet. Welcher Präfix verwendet wird, können Sie selbst festlegen.

Aufbau und Einsatz

Da Benutzer-Steuerelemente per Definition zur Gruppe der Web Server-Steuerelemente gehören, verhalten sie sich auch ähnlich. Der Inhalt eines Benutzer-Steuerelementes darf nur aus HTML-Tags bestehen, wie er zwischen den `<body>`-Tags stehen kann. `<html>` und `<body>` selbst dürfen nicht auftreten. Ebenso sollte das `<form>`-Tag vermieden werden, da die ASP.NET-Komponente erwartet, das Benutzer-Steuerelemente innerhalb oder außerhalb eines Formulars erscheinen, aber dieses nicht selbst bilden. Benutzer-Steuerelemente sind in ihrem Einsatzspektrum dennoch kaum eingeschränkt. Was immer Sie in Ihrem Projekt mehr als einmal benötigen, können Sie so definieren. Durch die Möglichkeit, auf die

enthaltenen Elemente programmatisch zugreifen zu können, ist die Anwendung sehr flexibel möglich.

Benutzer-Steuerelemente sind kein Allheilmittel. Es gibt in ASP.NET mehr Technologien zur Modularisierung. Einsetzen sollten Sie sie in folgenden Situationen:

- Elemente werden mehrfach benötigt, beispielsweise bei Menüs, Kopf- und Fußzeilen etc.
- Zur Reduzierung von Code in einer Seite, das heißt, zur Erhöhung der Übersichtlichkeit

In anderen Fällen gibt es bessere Möglichkeiten:

- Bereitstellung von Steuerelementen für fremde Applikationen (dafür sind Custom Controls besser geeignet)
- Kapselung von Code für fremde Applikationen (dafür sind compilierte Assemblies gedacht)
- Trennung von Code und Design (dies wird mit Code Behind gemacht)

4.5.2 Wie Benutzer-Steuerelemente entstehen

Erweiterung .ascx

Benutzer-Steuerelemente entstehen in zwei Schritten. Zuerst wird eine neue Datei angelegt, die den HTML-Code aufnimmt. Diese Datei muss die Endung *.ascx* bekommen. Dies ist zwingend vorgeschrieben, damit der Baustein beim Übersetzen der Seite korrekt eingebunden wird. Als Beispiel soll hier eine einfache Navigation aufgebaut werden, die auf jeder Seite der Applikation erscheint. Über den Programmcode soll eine Eigenschaft so verändert werden, dass die aktuelle Seite in den Navigationselementen hervorgehoben wird. Das folgende Beispiel zeigt den Aufbau eines einfachen Benutzer-Steuerelementes:

```
<table border="1">
    <tr>
        <td>
            <a href="home.aspx">Startseite</a>
        </td>
        <td>
            <a href="impressum.aspx">Impressum</a>
        </td>
        <td>
            <a href="produkte.aspx">Produkte</a>
        </td>
    </tr>
</table>
```

Listing 4.27: Einfache Navigation (UserControls/navigation.ascx)

Im zweiten Schritt muss das Benutzer-Steuerelement nun bekannt gemacht werden. Auf jeder Seite der Applikation wird es eingebaut. Bislang fehlte noch die Definition des Namensraumes, der für das XML-Tag verwendet wird. ASP.NET trifft hier keine Vorgabe. Hier soll „uc:“ verwendet werden. Die Anmeldung wird über die Direktive `@Register` vorgenommen:

@Register

```
<% @Register TagPrefix="uc" TagName="navigation" src="navigation.ascx" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="en">
    <head>
        <title>Home</title>
    </head>
    <body>
        <uc:navigation id="nav" runat="server"/>
        <h1>Startseite</h1>
        Das ist die Startseite
    </body>
</html>
```

Listing 4.28: Nutzung des User Controls (UserControls/home.aspx)

Die Direktive bestimmt mit dem Attribut `TagPrefix`, welcher Namensraum verwendet wird. `TagName` definiert, wie das Tag auf dieser Seite heißt. Ein- und derselbe Code kann also nicht nur mehrfach verwendet werden, sondern auch verschieden benannt sein. Natürlich muss noch die Quelle mit dem Attribut `src` benannt werden. Auf der Seite selbst können Sie den Baustein nun folgendermaßen einbauen:
`<uc:navigation id="nav" runat="server"/>` Das Attribut `runat="server"` ist wieder obligatorisch. Da Benutzer-Steuerelemente auch als Objekt zur Verfügung stehen, ist die Benennung mit `id="ucname"` sinnvoll. Aber auch ohne Zugriff über Programmcode wird es nun bereits in der Seite angezeigt.

Präfix und Quelle

Abbildung 4.18: Die Navigationslinks oben entstehen durch ein User Control

Spannender ist freilich der Zugriff über den Programmcode, denn nur so entfalten Benutzer-Steuerelemente ihr gesamtes Leistungsspektrum. Das .Net-Framework besteht aus einer ausgefeilten und komplexen Hierarchie von Klassen. Jedes Objekt stammt in der einen oder anderen

Form von einer der Klassen ab. Das trifft zwangsläufig auch auf Benutzer-Steuerelemente zu. Diese stammen aus der Klasse `Controls`, aus der auch HTML Server-Steuerelemente und Web Server-Steuerelemente abgeleitet wurden. Aus `Controls` erbt auch `TemplateControls`; aus dieser Klasse wird die Basisklasse `UserControl` abgeleitet, aus der letztlich auch die Objekte der Benutzer-Steuerelemente instanziiert werden. Damit stehen Basiseigenschaften wie beispielsweise `Visible` sofort zur Verfügung, ohne dass eine einzige Codezeile erforderlich wäre. Der Name des Objekts wird durch das Attribut `id` festgelegt. Auf die Elemente des Steuerelements können Sie nicht so direkt zugreifen – dazu jedoch später mehr. Der Zugriff auf die Elemente vom Programm der aufrufenden Seite aus, ist natürlich möglich, allerdings nicht direkt. Als Entwickler eines Benutzer-Steuerelements sollten Sie die Kontrolle darüber behalten, welche Details sich beeinflussen lassen und welche nicht. Es ist nahe liegend, dass der Code dazu im Steuerelement selbst untergebracht wird. Wenn nun aber Code geschrieben wird, sind auch Direktiven notwendig. Tatsächlich gibt es kaum Unterschiede zu einer normalen *aspx*-Seite. Statt der Direktive `@Page` wird das Benutzer-Steuerelement mit `@Control` ausgestattet. Sämtliche Attribute und Parameter sind identisch, mit Ausnahme von `trace` – diese Option kann nicht im Steuerelement ein- und ausgeschaltet werden. Stehen nun alle Techniken zur Verfügung, die *aspx*-Seiten bieten, ist es auch leicht möglich, Teile des Benutzer-Steuerelements als Eigenschaften zur Verfügung zu stellen.

Eigenschaften der Bausteine

Im Beispiel wäre es nahe liegend, die Hintergrundfarbe der Tabellenzelle einzufärben, deren Link auf die gerade aktive Seite zeigt. Dies soll als neue Eigenschaft des Benutzer-Steuerelements dem aufrufenden Programm bereit gestellt werden. Das folgende Beispiel enthält das derart erweiterte Steuerelement zur Navigation. Beachten Sie, dass der direkte Zugriff nicht möglich ist, Sie müssen die erforderlichen Eigenschaften über Programmcode zur Verfügung stellen. Definiert werden dazu Eigenschaften der das Steuerelement repräsentierenden Klasse. Im Beispiel wird eine nur schreibbare Eigenschaft benötigt; es genügt also, den `set`-Zweig zu definieren.

```
<% @Control %>
<script runat="server" language="C#">
void Page_Load()
{
    link1.Text = "Startseite";
    link1.NavigateUrl = "home2.aspx";
    link2.Text = "Impressum";
    link2.NavigateUrl = "impressum2.aspx";
    link3.Text = "Produkte";
    link3.NavigateUrl = "produkte2.aspx";
}
```

```
public bool nav_link1
{
    set
    {
        if (value)
        {
            td1.Style["Background-Color"] = "#dddddd";
        } else {
            td1.Style["Background-Color"] = "#ffffff";
        }
    }
}
public bool nav_link2
{
    set
    {
        if (value)
        {
            td2.Style["Background-Color"] = "#dddddd";
        } else {
            td2.Style["Background-Color"] = "#ffffff";
        }
    }
}
public bool nav_link3
{
    set
    {
        if (value)
        {
            td3.Style["Background-Color"] = "#dddddd";
        } else {
            td3.Style["Background-Color"] = "#ffffff";
        }
    }
}

</script>
<table border="1">
    <tr>
        <td id="td1" runat="server">
            <asp:hyperlink id="link1" runat="server"/>
        </td>
        <td id="td2" runat="server">
            <asp:hyperlink id="link2" runat="server"/>
        </td>
```

```
        <td id="td3" runat="server">
            <asp:hyperlink id="link3" runat="server"/>
        </td>
    </tr>
</table>
```

Listing 4.29: Umfangreiches User Control mit Steuerung der Eigenschaften (UserControls/navigation2.ascx)

Der der Eigenschaft zugewiesene Wert ist über das Schlüsselwort `value` erreichbar. `td1` repräsentiert die zum HTML Server-Steuerelement konvertierten Tabellenzellen, in welcher der Link steht.

Das folgende Programm zeigt, wie der Einbau erfolgt:

```
<% @Page language="C#" debug="true" %>
<% @Register TagPrefix="uc" TagName="navigation" src="navigation2.ascx" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<script runat="server" language="c#">
    void Page_Load()
    {
        nav.nav_link1 = true;
    }
</script>
<html lang="en">
    <head>
        <title>Home</title>
    </head>
    <body>
        <uc:navigation id="nav" runat="server"/>
        <h1>Startseite</h1>
        Das ist die Startseite
    </body>
</html>
```

Listing 4.30: Nutzung des komplexeren User Controls (UserControls/home2.aspx)

Über `nav.nav_link1` wird auf die Eigenschaft zugegriffen. Das erscheint auf den ersten Blick umständlich, immerhin wäre es einfacher, beispielsweise `nav.td1.Style["Background-Color"] = "#dddddd"` zu schreiben, um ein Teil des Steuerelements farblich zu verändern. Allerdings wären dann alle Teile öffentlich zugänglich, was im Sinne eines guten Programmierstils nicht gut ist. Sie sollten immer genau die Eigenschaften exakt definieren, die auch tatsächlich benötigt werden. .Net erzwingt das kurzerhand. Das verbesserte Benutzer-Steuerelement ist auch in allen anderen Dateien zu finden, die es anwenden, jeweils mit einer anderen Eigenschaft. Sie können also nicht nur flexibel eingesetzt werden, sondern auch umfangreiche Aktionen ausführen. Dazu gehören auch

Datenbankabfragen, Zugriffe auf das Dateisystem oder Interaktionen mit der aktuellen Seite.

Wenn Sie in ASP.NET programmieren, sollten Sie sich intensiv mit Web Server-Steuerelementen oder Benutzer-Steuerelementen auseinandersetzen. Es sind Kernelemente der Programmierung und ebnen den Weg zu großen und stabilen Anwendungen.

4.5.3 Spezielle Techniken der Benutzer-Steuerelemente

Code der Benutzer-Steuerelemente auslagern

@Control

Die Philosophie, die dahinter steckt, ist grundlegend anders als die im alten ASP vermittelte. Anstatt HTML und Code wild zu vermischen, ist eine strenge Trennung nun möglich und empfehlenswert. Die Auslagerung von Code in externe Dateien mit Code Behind ist Ihnen sicher bereits vertraut. Da die `@Control`-Direktive den gesamten Umfang der `@Page`-Direktive bietet, ist dies auch mit *ascx*-Dateien möglich. Wenn Sie externen Code verwenden, leiten Sie ihre eigenen Klassen normalerweise von der Klasse `Page` ab, um über die Eigenschaften und Methoden der Seite verfügen zu können. Benutzer-Steuerelemente erben auch auf diese Weise, allerdings von der Klasse `UserControl`:

```
void navigationclass : UserControl
```

Das Beispiel mit der Navigation lässt sich mit dieser Technik gut erweitern. So stehen die Zellen der Tabelle im Moment nebeneinander, ideal für eine Navigation am oberen Rand der Seite. Soll dieselbe Navigation links erscheinen, ist es besser, die Zellen untereinander anzuordnen. Da sich der HTML-Code völlig vom bereits gezeigten unterscheidet, ist ein neues Benutzer-Steuerelement zu entwerfen. Der C#-Code, der den Zugriff auf die Eigenschaften gestattet, ist dagegen mit dem bereits gezeigten identisch. Es ist mehr als nur guter Programmierstil, den Code mehrfach zu verwenden. In der Kopfzeile wird dazu folgende Direktive geschrieben: `<% @Control src="" ... %>`.

Der Code entspricht dem im `<script>`-Tag von *home2.aspx* gezeigten; für dieses Beispiel heißt die Datei *home3.aspx*. Auf dieser Basis sind jetzt zwei Benutzer-Steuerelemente verfügbar, die denselben Code verwenden. Der Einbau in einer HTML-Seite kann folgendermaßen aussehen:

```
<% @Page language="C#" %>
<% @Register TagPrefix="uc" TagName="topnavigation" src="navigation_top.ascx"
%>
<% @Register TagPrefix="uc" TagName="dwnnavigation" src="navigation_dwn.ascx"
%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<script runat="server" language="c#">
```

```
    void Page_Load()
    {
        switch (Convert.ToInt32(Request.QueryString["i"]))
        {
            case 1:
            default:
                topnav.nav_link1 = true;
                dwnnav.nav_link1 = true;
                show.Text = "<h1>Startseite</h1>";
                break;
            case 2:
                topnav.nav_link2 = true;
                dwnnav.nav_link2 = true;
                show.Text = "<h1>Impressum</h1>";
                break;
            case 3:
                topnav.nav_link3 = true;
                dwnnav.nav_link3 = true;
                show.Text = "<h1>Produkte</h1>";
                break;
        }
    }
</script>
<html lang="en">
    <head>
        <title>Home</title>
    </head>
    <body>
        <table border="2">
            <tr>
                <td><h1>LOGO</h1></td>
                <td><uc:topnavigation id="topnav" runat="server" /></td>
            </tr>
            <tr>
                <td><uc:dwnnavigation id="dwnnav" runat="server" /></td>
                <td>
                    <asp:label id="show" runat="server"/>
                </td>
            </tr>
        </table>
    </body>
</html>
```

Listing 4.31: Zwei interaktiv gesteuerte User Controls (UserControls/home3.ascx)

Die folgende Abbildung zeigt, wie die fertige Seite aussieht. Die um die Navigation erleichterte Seite ist nicht nur schlanker und übersichtli-

cher, sondern auch einfacher zu warten. Änderungen an der Navigation wirken sich automatisch an allen Stellen aus, an denen sie verwendet wird. Dies kann auch bei kleineren Projekten erheblich Zeit sparen und die Qualität der Seiten verbessern. Durch die Trennung des Codes sind überdies auch die Änderungen selbst einfacher ausführbar.

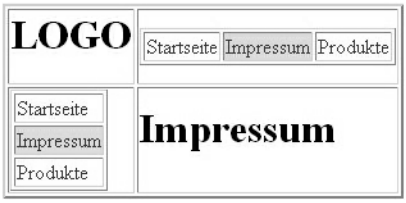


Abbildung 4.19: User Controls, die denselben Code nutzen

Benutzer-Steuerelemente perfektionieren

Aus den mit festen Links versehenen Steuerelementen lassen sich leicht universelle Elemente entwickeln. Die Tags in den bereits gezeigten Beispielen sind fest codiert. Da Sie aus jedem HTML-Element ein HTML Server-Steuerelement machen können und die Links ohnehin bereits Web Server-Steuerelemente sind, bietet sich eine Erweiterung der Liste über Eigenschaften an. Hier kann über ein Formular die Navigation dynamisch erweitert werden, ähnlich den persönlichen Menüs in MS Office. Tatsächlich verursacht der Einbau der erweiterten Benutzer-Steuerelemente kaum Aufwand. Entsprechend sind Änderungen – auch solche tiefgehenden wie bei der Navigation – bei geschickter Planung der Applikation sehr einfach ausführbar. Webseiten sind dynamische Gebilde. Der Erfolg einer Site hängt wesentlich davon ab, wie schnell und flexibel sich diese an ändernde Benutzergewohnheiten anpassen lassen. Eine der aktuellen Entwicklungen sind so genannte My-Sites, bei denen die Gestaltung der Seite und die Anzahl verfügbarer Optionen an die Wünsche eines Benutzers angepasst werden kann. Benutzer-Steuerelemente sind das ideale Werkzeug dafür. Wie beim Navigationsbeispiel gezeigt, können Sie leicht jedes änderbare Teil der Seite damit definieren. Die vom Benutzer wählbaren Eigenschaften werden auch programmatisch als Eigenschaften abgelegt. Es ist nun möglich, jedem Steuerelement die Möglichkeit zu geben, aus der Session-ID den aktuellen Benutzer zu ermitteln und „sich selbst" so einzustellen, wie beispielsweise in einer Datenbank hinterlegt. Der Programmierer der Seite muss dann nicht auf die Belange der Benutzer-Steuerelemente Rücksicht nehmen, sondern kann diese wie Web Server-Steuerelemente einsetzen. Die Sammlung solcher universeller Steuerelemente ist eine ideale Basis für eine Bibliothek. Wie in Bibliotheken üblich, werden nur Teile „ausgeliehen". In den vorherigen Beispielen wurden die dynamisch veränderlichen Navigations-Elemente selbst statisch implementiert. Es wäre im Sinne einer guten Benutzerführung aber durchaus praktisch, wenn die

An- und Abwahl programm- und damit benutzergesteuert erfolgen kann. Die bereits mit HTML Server- und Web Server-Steuerelemente verwendeten Techniken lassen sich mit Benutzer-Steuerelemente gut kombinieren. Die folgende Datei zeigt die Nutzung einer direkt von der Klasse `UserControls` geerbten Eigenschaft – `Visible`. Damit lassen sich die Benutzer-Steuerelemente gezielt ein- und ausschalten. Das Formular dient der Steuerung der Anzeige der Navigation durch den Benutzer:

```
<% @Page language="C#" debug="true" %>
<% @Register TagPrefix="uc" TagName="topnavigation" src="navigation_top.ascx"
%>
<% @Register TagPrefix="uc" TagName="dwnnavigation" src="navigation_dwn.ascx"
%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<script runat="server" language="c#">
    void Page_Load()
    {
        switch (Convert.ToInt32(Request.QueryString["i"]))
        {
            case 1:
            default:
                topnav.nav_link1 = true;
                dwnnav.nav_link1 = true;
                show.Text = "<h1>Startseite</h1>";
                break;
            case 2:
                topnav.nav_link2 = true;
                dwnnav.nav_link2 = true;
                show.Text = "<h1>Impressum</h1>";
                break;
            case 3:
                topnav.nav_link3 = true;
                dwnnav.nav_link3 = true;
                show.Text = "<h1>Produkte</h1>";
                break;
        }
        if (IsPostBack)
        {
            if (settop.Checked)
            {
                topnav.Visible = true;
            } else {
                topnav.Visible = false;
            }
```

```
            if (setdwn.Checked)
            {
                dwnnav.Visible = true;
            } else {
                dwnnav.Visible = false;
            }
        }
    }
</script>
<html lang="en">
    <head>
        <title>Home</title>
    </head>
    <body>
        <table border="2">
            <tr>
                <td><h1>LOGO</h1></td>
                <td><uc:topnavigation id="topnav" runat="server" /></td>
            </tr>
            <tr>
                <td><uc:dwnnavigation id="dwnnav" runat="server" /></td>
                <td>
                    <asp:label id="show" runat="server"/>
                </td>
            </tr>
        </table>
        <form runat="server">
        <asp:checkbox autopostback="true" id="settop"
                      runat="server" text="Top Navigation"
                      groupname="nav_set"/>
        <asp:checkbox autopostback="true" id="setdwn"
                      runat="server" text="Down Navigation"
                      groupname="nav_set"/>
        </form>
    </body>
</html>
```

Listing 4.32: Interaktive Aktivierung und Deaktivierung von Benutzer-Steuerelementen (UserControls/home4.aspx)

In der Methode `Page_Load` wird dann die Veränderung der Einstellung abgefragt und die Sichtbarkeit der Navigationselemente gesteuert. `topnav` und `dwnnav` sind die Instanzen der Benutzer-Steuerelemente, `settop` und `setdwn` die Web Server-Steuerelemente, mit denen die Anzeige gesteuert wird (Kontrollkästchen).

Abbildung 4.20: Formular zum Ein- und Ausschalten von User Controls

4.5.4 Benutzer-Steuerelemente und hinterlegter Code

Code Behind

Das nächste Beispiel zeigt eine zusätzliche Erweiterung, bei der über ein drittes Kontrollkästchen dynamisch ein weiterer Link angezeigt werden kann. Neben der Definition der Tabellenzelle bzw. Tabellenreihe wird der Code aus dem letzten Beispiel um die Steuerung der entsprechenden Eigenschaft dieser Zelle erweitert und als Datei *navigationcontrol_admin.cs* abgelegt:

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;

public class navigationcontrol : UserControl
{

    public HyperLink link1;
    public HyperLink link2;
    public HyperLink link3;
    public HyperLink link4;     // Erweiterung Admin
    public HtmlControl td1;
    public HtmlControl td2;
    public HtmlControl td3;
    public HtmlControl td4;   // Erweiterung Admin

    void Page_Load()
    {
        link1.Text = "Startseite";
        link1.NavigateUrl = "home5.aspx?i=1";
        link2.Text = "Impressum";
        link2.NavigateUrl = "home5.aspx?i=2";
        link3.Text = "Produkte";
        link3.NavigateUrl = "home5.aspx?i=3";
    }
```

```
public bool nav_link1
{
    set
    {
        if (value)
        {
            td1.Style["Background-Color"] = "#dddddd";
        } else {
            td1.Style["Background-Color"] = "#ffffff";
        }
    }
}
public bool nav_link2
{
    set
    {
        if (value)
        {
            td2.Style["Background-Color"] = "#dddddd";
        } else {
            td2.Style["Background-Color"] = "#ffffff";
        }
    }
}
public bool nav_link3
{
    set
    {
        if (value)
        {
            td3.Style["Background-Color"] = "#dddddd";
        } else {
            td3.Style["Background-Color"] = "#ffffff";
        }
    }
}
// Erweiterung Admin
public bool nav_link4
{
    set
    {
        if(value)
        {
            td4.Style["Background-Color"] = "#dddddd";
            td4.Visible = true;
            link4.Text = "Administration";
```

```
                link4.NavigateUrl = "home5.aspx?i=4";
            } else {
                td4.Visible = false;
                link4.Text = "";
                link4.NavigateUrl = "home5.aspx";
            }
        }
    }
}
```

Listing 4.33: Hinterlegter Code für das erweiterte Benutzer-Steuerelemente (UserControls/navigationcontrol_admin.cs)

Mit der Zuweisung von `true` zur Eigenschaft `nav_link4` wird neben der Anzeige auch der URL und die Farbe gesteuert. Statt des Kontrollkästchens könnte auch eine Abfrage der Benutzer-ID erfolgen, sodass die Administration nur bestimmten Personen zugänglich ist. Das letzte Benutzer-Steuerelement erfüllt natürlich auch die Ansprüche der ersten Aufgabe, dadurch können Sie praktisch mit einem komplexeren Gebilde gleich mehrere Aktionen bedienen. Der Einbau des Codes erfolgt folgendermaßen:

```
<% @Control Inherits="navigationcontrol" src="navigationcontrol_admin.cs" %>
<table border="1">
    <tr>
        <td id="td1" runat="server">
            <asp:hyperlink id="link1" runat="server"/>
        </td>
        <td id="td2" runat="server">
            <asp:hyperlink id="link2" runat="server"/>
        </td>
        <td id="td3" runat="server">
            <asp:hyperlink id="link3" runat="server"/>
        </td>
        <td id="td4" runat="server">
            <asp:hyperlink id="link4" runat="server"/>
        </td>
    </tr>
</table>
```

Listing 4.34: Benutzer-Steuerelement, das den Code des letzten Listings nutzt (UserControls/navigation_top_admin.ascx)

Die Darstellung in der HTML-Seite selbst beschränkt sich weiterhin auf zwei XML-Tags und zwei Aufrufe der Direktive `@Register`. Für die Musterapplikation wird allerdings noch die Steuerung des Formulars benötigt:

```
<% @Page language="C#" debug="true" %>
<% @Register TagPrefix="uc" TagName="topnavigation"
src="navigation_top_admin.ascx" %>
<% @Register TagPrefix="uc" TagName="dwnnavigation"
src="navigation_dwn_admin.ascx" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<script runat="server" language="c#">
    void Page_Load()
    {
        switch (Convert.ToInt32(Request.QueryString["i"]))
        {
            case 1:
            default:
                topnav.nav_link1 = true;
                dwnnav.nav_link1 = true;
                show.Text = "<h1>Startseite</h1>";
                break;
            case 2:
                topnav.nav_link2 = true;
                dwnnav.nav_link2 = true;
                show.Text = "<h1>Impressum</h1>";
                break;
            case 3:
                topnav.nav_link3 = true;
                dwnnav.nav_link3 = true;
                show.Text = "<h1>Produkte</h1>";
                break;
            case 4:
                topnav.nav_link4 = true;
                dwnnav.nav_link4 = true;
                show.Text = "<h1>Administration</h1>";
                break;
        }
        if (IsPostBack)
        {
            if (settop.Checked)
            {
                topnav.Visible = true;
            } else {
                topnav.Visible = false;
            }
            if (setdwn.Checked)
            {
                dwnnav.Visible = true;
            } else {
                dwnnav.Visible = false;
```

```
            }
            if (setadm.Checked)
            {
                topnav.nav_link4 = true;
                dwnnav.nav_link4 = true;
            } else {
                topnav.nav_link4 = false;
                dwnnav.nav_link4 = false;
            }
        }
    }
</script>
<html lang="en">
    <head>
        <title>Home</title>
    </head>
    <body>
        <form runat="server">
        <asp:checkbox autopostback="true" id="settop"
                      runat="server" text="Top Navigation"
                      groupname="nav_set"/>
        <asp:checkbox autopostback="true" id="setdwn"
                      runat="server" text="Down Navigation"
                      groupname="nav_set"/>
        <asp:checkbox autopostback="true" id="setadm"
                      runat="server" text="Admin Area" groupname="nav_set"/>
        </form>
        <table border="2">
            <tr>
                <td><h1>LOGO</h1></td>
                <td><uc:topnavigation id="topnav" runat="server" /></td>
            </tr>
            <tr>
                <td><uc:dwnnavigation id="dwnnav" runat="server" /></td>
                <td>
                    <asp:label id="show" runat="server"/>
                </td>
            </tr>
        </table>
    </body>
</html>
```

Listing 4.35: Anwendung des Benutzer-Steuerelements mit hinterlegtem Code und Formularsteuerung (UserControls/home5.aspx)

Die Abbildung zeigt den Effekt, wenn der zusätzliche Link eingeschaltet ist:

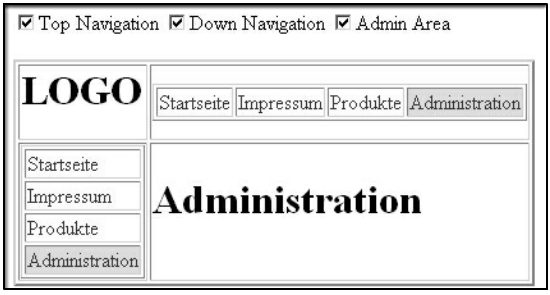


Abbildung 4.21: Komplexes Benutzer-Steuerelement, interaktiv erweiter- und aktivierbar

4.6 Komplexe Steuerelemente (Rich Controls)

ASP.NET liefert nicht nur elementare Steuerelemente, sondern auch außerordentlich komplexe. Eines der interessantesten ist das Kalender-Steuerelement (`Calendar`). Zuerst eine Präsentation:

Abbildung 4.22: Interaktiver Kalender

Wie lange brauchen Sie, um dies mit herkömmlichen Elementen zu programmieren? Immerhin besitzt der Kalender Blätterfunktionen zum Wechsel der Monate. Der ausgewählte Tag wird grau hinterlegt und erscheint bei Mausklick unterhalb des Kalenders. Das Programm selbst – tatsächlich vollständig:

```
<% @ Page language="C#" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<script runat="server" language="C#">
void zeigeDatum(Object sender, EventArgs e)
{
   kalender.TodaysDate = kalender.SelectedDate;
   auswahl.Text = "Ihre Auswahl: " + kalender.TodaysDate.ToShortDateString();
}
```

```
</script>
<html lang="de">
    <head>
        <title>Kalender</title>
        <basefont face="Arial">
    </head>
    <body>
        <h1>Kalender</h1>
        <form runat="server">
        <asp:calendar id="kalender" runat="server"
                      onSelectionChanged="zeigeDatum"/>
        <hr noshade="noshade" width="200px" align="left"/>
        <asp:label id="auswahl" runat="server"/>
        </form>
    </body>
</html>
```

Listing 4.36: Ein interaktiver Kalender (calendar.aspx)

Wie es funktioniert

Im Kern basiert das Programm auf dem Web Server-Steuerelement `Calendar`:

```
<asp:calendar id="kalender" runat="server" onSelectionChanged="zeigeDatum"/>
```

Hier wird nur eines der sehr zahlreichen Attribute verwendet: `onSelectionChanged`. Wann immer die Auswahl geändert wird, ruft ASP.NET die Methode `zeigeDatum` auf.

Hier passiert auch nicht viel. Zuerst der Zugriff auf das ausgewählte Datum, das zugleich zum aktuellen gemacht wird:

```
kalender.TodaysDate = kalender.SelectedDate;
```

Dann wird diese Angabe in ein passendes Format gebracht und an das `Label` übergeben:

```
auswahl.Text = "Ihre Auswahl: " + kalender.TodaysDate.ToShortDateString();
```

Das war's auch schon. In der Referenz finden Sie zahlreiche Eigenschaften zur Gestaltung und Methoden zur Behandlung der ermittelten Datumswerte.

4.7 Fragen und Übungsaufgaben

1. Nennen Sie die Typen von Steuerelementen in ASP.NET.
2. Erklären Sie, warum Formulare sowohl eine client- als auch eine serverseitige Prüfung benötigen, um zuverlässig zu funktionieren. Warum ist eine reine serverseitige Prüfung nicht optimal?

3. Entwerfen Sie ein vollständiges Kontaktformular für Ihre Homepage und lassen Sie sämtliche Felder sinnvoll überprüfen.
4. Entwerfen Sie ein Formular, dass die Auswahl von sechs Artikel über Kontrollkästchen ermöglicht. Zusätzlich sollen die Artikel in drei Gruppen ausgewählt werden können. Dazu sind drei Optionsfelder zu erstellen, die nach dem Anklicken zu einer Vorauswahl der Kontrollkästchen führen. Eine Sendeschaltfläche sendet das Formular und das Ergebnis der Auswahl wird angezeigt.

   Diese Aufgabe ist etwas knifflig, denn Sie müssen unterscheiden, ob ein Optionsfeld gewählt wurde (dies führt lediglich zur Änderung der Vorauswahl der Kontrollkästchen) oder das Formular gesendet wurde, um die Auswahl der Kontrollkästchen zu ermitteln.

5 Basisklassen des .Net-Frameworks

Der praktische Umgang mit ASP.NET verlangt den Zugriff auf viele Klassen des Frameworks. Das trifft fast unverändert auch auf anderen Anwendungen von .Net zu, ist also nicht spezifisch für ASP.NET. Die hier gezeigten Verfahren sind aber in der Praxis unverzichtbar.

5.1 Was Sie in diesem Kapitel lernen

Das .Net-Framework bietet sehr viele Klassen, die allen Programmierumgebungen zur Verfügung stehen. Egal ob Sie ASP.NET- oder Windows-Anwendungen programmieren, es werden dieselben Basisklassen eingesetzt. Auch wenn dieses Wissen eher allgemeiner Natur ist, darf es in einem ASP.NET-Buch nicht fehlen, denn Sie werden sonst schon einfache Aufgaben nicht oder nur umständlich lösen können.

Die Vorgehensweise ist bei den vorgestellten Klassen immer ähnlich, sodass Sie mit diesem Kapitel auch den allgemeinen Umgang mit .Net festigen. Konkret behandelt werden die folgenden Themen:

- *Datum und Zeit* – Zugriff auf das aktuelle Datum, Datumsberechnungen und formatierte Ausgabe von Daten.
- In diesem Zusammenhang wird auch die Lokalisierung – also die Einstellung einer spezifischen Landessprache – gezeigt.
- *Aufzählungen und Kollektionen* – Oft werden mehr komplexe Datentypen benötigt, als mit Arrays und Strukturen zur Verfügung stehen. Bevor Sie es selbst programmieren, schauen Sie auf die Möglichkeiten, die Aufzählungen und Kollektionen wirklich bieten.
- *Zugriff auf das Dateisystem* – Kaum eine Anwendung kann darauf verzichten, Verzeichnisse und Dateien zu lesen und neue Dateien zu erzeugen und zu beschreiben.

- *Dynamische Bilderzeugung* – Von JPEG über GIF bis PNG, in nahezu allen Grafikformaten können mit dem Framework Bilder während der Laufzeit des Programms erzeugt werden.
- *E-Mail versenden* – Das Versenden von E-Mail mit einer Serveranwendung ist eine häufig benötigte Funktion. Lernen Sie alles über diese spannende Technik.
- *Reguläre Ausdrücke* – Kaum ein professionelles Programm, dass diese Methode zum Suchen und Ersetzen nicht hat, kaum ein Anfänger, der damit nicht Schwierigkeiten hat. Eine kompakte Einführung ebnet Ihnen den Weg.

Insgesamt ist dieses Kapitel also eine Sammlung verschiedener Themen, die dennoch zusammen in ASP.NET-Applikationen zu Einsatz kommen können und sicher auch werden.

5.2 Datum und Zeit

Datum und Zeit spielen in vielen Programmen eine große Rolle. Es gibt in jeder Programmiersprache und in jedem Betriebssystem andere Techniken, Datumswerte darzustellen. Vielleicht kennen Sie aus der Unix-Welt die Timestamps (Zeitmarken), die vom 1.1.1970 an zählen und ein 32–Bit-Wort nutzen. Dieser Zeitraum ist leider sehr begrenzt, das Geburtsdatum des Verfassers liegt früher und wäre nicht darstellbar.

Wenn im Folgenden von Datumswerten gesprochen wird, ist damit immer auch ein Zeitwert gemeint, andernfalls wird explizit darauf hingewiesen.

DateTime

Im .Net-Framework ist die Struktur `DateTime` aus dem Namensraum `System` für den Umgang mit Datum und Zeit zuständig. Datumswerte können im Wertebereich vom 1.1.0001 0:00 Uhr (Mitternacht) bis zum 31.12.9999 23:59:59 Uhr verwendet werden. Die typischen Beschränkungen anderer Systeme sind damit endgültig passé. Die Genauigkeit basiert auf so genannten Ticks, dies sind Zeitsprünge von 100 Nanosekunden.

Für Berechnungen mit Datumswerten dient außerdem die `TimeSpan`-Struktur. Objekte daraus enthalten Datumsdifferenzen statt absoluter Daten.

5.2.1 Datumsabfragen

Die Abfrage das Datums und der aktuellen Zeit gelingt mit der statischen Eigenschaft `Now`. Interessiert nur das Datum, ist `ToDay` die bessere Wahl. Die dritte statische Eigenschaft im Bunde ist `UtcNow` – die Darstel-

lung im UTC-Format – früher bekannt als GMT (Greenwich Mean Time) und oft auch Weltzeit genannt. UTC steht für Universal Time Coordinated, ein 24-Stunden-Format, das international verwendet wird.

Die aktuelle Zeit bezieht sich immer auf den Server. Denken Sie daran, wenn Sie die Angabe zur Begrüßung verwenden, dass Benutzer in anderen Zeitzonen leben können.

Das folgende Programm verwendet drei Zeitdarstellungen:

```
<% @Page Language="C#" debug="true" %>
<script runat="server" language="C#">
void Page_load()
{
    ausgabe.InnerHtml  = "Ausgabe von Datumswerten:";
    ausgabe.InnerHtml += "<br/>Es ist jetzt: " + DateTime.Now;
    ausgabe.InnerHtml += "<br/>Es ist heute: " + DateTime.Today;
    ausgabe.InnerHtml += "<br/>UTC: " + DateTime.UtcNow;
}
</script>
<html>
    <head>
        <title>Datum und Zeit</title>
    </head>
    <body>
        <h1>Datum und Zeit</h1>
        <p id="ausgabe" runat="server"/>
    </body>
</html>
```

Listing 5.1: Ausgabemöglichkeiten für das aktuelle Datum (dt_now.aspx)

Da die drei Eigenschaften statisch sind, muss kein Objekt instanziiert werden.

Abbildung 5.1: Ausgabe des aktuellen Datums des Servers

Sollen Datum und Zeit getrennt werden, verwenden Sie die Eigenschaften `Date` bzw. `TimeOfDay`. Dazu muss jedoch eine Instanz gebildet werden. Die folgenden Beispiele nutzen denselben HTML-Teil wie das erste, er wird deshalb nicht erneut wiederholt.

```
<script runat="server" language="C#">
void Page_load()
{
    ausgabe.InnerHtml  = "Ausgabe von Datumswerten:";
    ausgabe.InnerHtml += "<br/>Es ist jetzt: " + DateTime.Now;
    DateTime heute = DateTime.Now;
    ausgabe.InnerHtml += "<br/>Datum: " + heute.Date;
    ausgabe.InnerHtml += "<br/>Uhrzeit: " + heute.TimeOfDay;
}
</script>
```

Listing 5.2: Datum und Uhrzeit (dt_datetime.aspx)

Hier wird zuerst eine Instanz des aktuellen Datums gebildet:

```
DateTime heute = DateTime.Now;
```

Auf das Objekt `heute` werden die Eigenschaften dann angewendet.

Abbildung 5.2: Ausgabe von Datum und Zeit

Die Zeit enthält den Anteil in der Genauigkeit, die die Ticks bieten (100 Nanosekunden). Das wird nicht oft benötigt, deshalb ist eine Abtrennung mit Zeichenketten-Methoden denkbar:

```
String zeit = heute.TimeOfDay.ToString();
zeit = zeit.Substring(0, zeit.IndexOf("."));
ausgabe.InnerHtml += "<br/>Uhrzeit: " + zeit;
```

Die Trennung erfolgt am Punkt, der in der Zeitdarstellung nur ein Mal auftritt.

Datumswerte

Um auf die Bestandteile von Datumswerten zugreifen zu können, gibt es natürlich auch die passenden Eigenschaften:

```
<script runat="server" language="C#">
void Page_load()
{
    ausgabe.InnerHtml  = "Ausgabe von Datumswerten:";
    ausgabe.InnerHtml += "<br/>Es ist jetzt: " + DateTime.Now;
    DateTime heute = DateTime.Now;
    ausgabe.InnerHtml += "<br/>Tag: " + heute.Day;
    ausgabe.InnerHtml += "<br/>Monat: " + heute.Month;
```

```
    ausgabe.InnerHtml += "<br/>Jahr: " + heute.Year;
    ausgabe.InnerHtml += "<br/>Wochentag: " + heute.DayOfWeek;
    ausgabe.InnerHtml += "<br/>Tag im Jahr: " + heute.DayOfYear;

}
</script>
```

Die Eigenschaften dürften selbsterklärend sein. Sie geben alle numerische Werte zurück, mit Ausnahme der Methode `DayOfWeek`, die den ausgeschriebenen englischen Wochentagsnamen ausgibt.

Abbildung 5.3: Zerlegen eines Datums

Schaltjahr

Um zu ermitteln, ob ein Jahr ein Schaltjahr ist, verwenden Sie die statische Methode `IsLeapYear`, die als Argument die Jahreszahl erwartet.

Zeitwerte

Analog dazu werden für die Zeiten die Eigenschaften `Hour`, `Minute`, `Second` und `Millisecond` verwendet. Die Ticks werden mit der gleichnamigen Eigenschaft `Ticks` ermittelt. Der Datentyp, der zurückgegeben wird, ist `Long`.

5.2.2 Datumsberechnungen

Berechnungen mit Datumswerten sind im .Net-Framework sehr einfach. Dazu müssen Sie natürlich auch über Objekte verfügen, die nicht das aktuelle, sondern ein ganz spezifisches Datum enthalten. Dies ist möglich, indem Sie beim Instanziieren des Objekts `DateTime` die entsprechenden Angaben machen:

```
DateTime datum = new DateTime(2002, 5, 26);
```

Stunde, Minute und Sekunden können analog als weitere Parameter angehängt werden. Berechnungen führen Sie nun aus, indem die Methoden `AddDays`, `AddMonth`, `AddYears` bzw. `AddHours`, `AddMinutes` und `AddSeconds` verwendet werden. Ist der übergebene Parameter negativ, erfolgt eine Subtraktion.

```
<script runat="server" language="C#">
void Page_load()
{
    ausgabe.InnerHtml  = "Berechnungen von Datumswerten:";
    DateTime datum = new DateTime(2002, 5, 26);
    ausgabe.InnerHtml += "<br/>Datum: " + datum.ToShortDateString();
    ausgabe.InnerHtml += "<br/>Wochentag: " + datum.DayOfWeek;
    DateTime naechster = datum.AddYears(1);
    ausgabe.InnerHtml += "<br/>Nächstes Jahr: " + naechster.DayOfWeek;

}
</script>
```

Listing 5.3: Berechnungen mit Datumswerten (dt_add.aspx)

Die Ausgabe zeigt die Wochentage des erzeugten Datums (26.5.2002) für das aktuelle Jahr (2002) und das folgende (2003):

Abbildung 5.4: Ergebnis einer Datumsberechnung

Operatoren Vergleiche zwischen Daten können mit den üblichen Operatoren (<, >, >=, ==, <=, !=) ausgeführt werden, die für diesen Zweck überladen wurden.

TimeSpan Reichen diese direkten Berechnungen nicht aus, können Sie mit Zeitspannen arbeiten. Diese werden aus der Struktur `TimeSpan` abgeleitet. Die Struktur kann als Datumswert in der folgenden Form dargestellt werden:

```
[-]d.hh:mm:ss:ff
```

Dabei ist das Vorzeichen optional – bei Berechnungen mit negativen Werten wird eine Subtraktion ausgeführt. `d` bestimmt eine Anzahl Tage, `ff` den Teil einer Sekunde.

TicksPerHours, .. Add, Subtract. Der Konstruktor erwartet Ticks (ein Parameter), Zeiten (drei Parameter für Stunde, Minute und Sekunde) oder vier (Tage plus Zeit). Für Berechnungen sind auch ein paar Konstanten interessant – in der `TimeSpan`-Struktur als Felder definiert – die die Ticks pro Zeiteinheit enthalten: `TicksPerHour` für 1 Stunde, `TicksPerMinute` für eine Minute usw. bis hin zu `Zero` für 0. Instanzen von `TimeSpan` können direkt berechnet werden, dazu dienen die Methoden `Add` und `Subtract`.

Das folgende Beispiel berechnet einen Kursplan:

```
<script runat="server" language="C#">
void Page_load()
{
    ausgabe.InnerHtml  = "Eine Kurstafel:<br/>";
    DateTime kurs = new DateTime(2002, 5, 26, 9, 0, 0);
    TimeSpan pause = new TimeSpan(0, 30, 0);
    TimeSpan mittag = new TimeSpan(1, 15, 0);
    TimeSpan lektion = new TimeSpan(0, 90, 0);
    ausgabe.InnerHtml += "<br/>1. Lektion: " + kurs.ToShortTimeString();
    kurs = kurs.Add(lektion);
    ausgabe.InnerHtml += "<br/>Pause: " + kurs.ToShortTimeString();
    kurs = kurs.Add(pause);
    ausgabe.InnerHtml += "<br/>2. Lektion: " + kurs.ToShortTimeString();
    kurs = kurs.Add(lektion);
    ausgabe.InnerHtml += "<br/>Mittag: " + kurs.ToShortTimeString();
    kurs = kurs.Add(mittag);
    ausgabe.InnerHtml += "<br/>3. Lektion: " + kurs.ToShortTimeString();
    kurs = kurs.Add(lektion);
    ausgabe.InnerHtml += "<br/>Pause: " + kurs.ToShortTimeString();
    kurs = kurs.Add(pause);
    ausgabe.InnerHtml += "<br/>4. Lektion: " + kurs.ToShortTimeString();
    kurs = kurs.Add(lektion);
    ausgabe.InnerHtml += "<br/>Ende: " + kurs.ToShortTimeString();
}
</script>
```

Wie es funktioniert

Hier wird am Anfang das Startdatum festgelegt:

```
DateTime kurs = new DateTime(2002, 5, 26, 9, 0, 0);
```

Dann werden die Zeitspannen programmiert, die zur Berechnung eingesetzt werden sollen. Eine Pause soll 30 Minuten dauern:

```
TimeSpan pause = new TimeSpan(0, 30, 0);
```

Dann wird noch die Zeitspanne der Mittagspause festgelegt (eine Stunden und 15 Minuten):

```
TimeSpan mittag = new TimeSpan(1, 15, 0);
```

Zuletzt wird die Dauer einer Unterrichtseinheit auf 90 Minuten gesetzt:

```
TimeSpan lektion = new TimeSpan(0, 90, 0);
```

Mit diesen Angaben erfolgt nun die Berechnung. Das Ende der ersten Lektion ist leicht zu ermitteln:

```
kurs = kurs.Add(lektion);
```

Die folgenden Berechnungen laufen analog ab.

Abbildung 5.5: Eine Kurstafel, mit TimeSpan-Objekten berechnet

Negate

Alternativ zu Verwendung von `Subtract` zum Abziehen von Zeiten können Sie das `TimeSpan`-Objekt auch mit der Methode `Negate` negieren und `Add` verwenden.

5.2.3 Datumswerte formatieren

Kurze und lange Datumsformate

Eine Formatierung wurden bereits verwendet: `ToShortDateString`. Diese Methode verkürzt das Datum auf eine Darstellung ohne den Zeitanteil. Entsprechend können Sie auch `ToShortTimeString` verwenden (Format „hh:mm“). Nebenbei findet in beiden Fällen eine Typkonvertierung nach `String` statt. Alternativ kann auch `ToLongDateString` und `ToLongTimeString` verwendet werden. Das ist meist nicht ausreichend, denn für gut gestaltete Webseiten werden sehr individuelle Datumsangaben verlangt. Dazu gehören auch sprachliche Versionen, also deutsche Wochentags- und Monatsnamen auf deutschen Seiten. Falls Ihre Seite mehrsprachig ist, müssen Sie darüberhinaus auch an andere Sprachen denken. Mit dem .Net-Framework sind Sie für viele Sprachen bestens gerüstet.

Individuelle Datumsformate

Der einfachste Zugriff auf die Formatierung erfolgt mit der Methode `ToString`, die als Parameter eine Formatieranweisung bekommt:

```
heute.ToString("dd.MM.yyyy");
```

Die Sache ist trickreich, wenn Sie komplizierte Formatierungen vorhaben. Es gibt nämlich zwei Formatiermöglichkeiten. Zum einen können Sie vorgefertigte Kombinationen auswählen, beispielsweise stellt der Code `t` die Zeit in der Form „HH:mm“ dar. Andererseits steht auch jede einzelne Zeitinformation zur Verfügung, um in der oben gezeigten Form das Ergebnis zusammenzusetzen. Dabei gelten folgende Regeln:

- Steht eine einzelne Formatanweisung allein, wird eine Kombination angenommen.
- Werden mehrere Formate angegeben, handelt es sich um eine Einzelformatierung
- Wollen Sie dennoch eine Einzelformatierung, setzen Sie ein %-Zeichen davor.
- Unbekannte Zeichen werden unverändert ausgegeben. Sollen andere Zeichen, die sonst eine Bedeutung haben, ausgegeben werden, schreiben Sie ein Backslash \ davor. Damit die Backslashs C# nicht stören, müssen Sie fast immer ein @-Zeichen vor die Zeichenkette stellen.

Kombinationsformate

Die folgende Tabelle zeigt eine Auswahl Kombinationsformate:

Format	Darstellung des Datums: 26.5.2002 9:00:00
D	Sonntag, 26. Mai 2002
f	Sonntag, 26. Mai 2002 09:00 Hinweis: Lokale Zeit
F	Sonntag, 26. Mai 2002 09:00:00 Hinweis: Lokale Zeit
m, M	26 Mai Hinweis: Ohne Punkt
t	09:00
T	09:00:00
U	Sonntag, 26. Mai 2002 07:00:00 Hinweis: Die Zeit wird in UTC ausgegeben

Tabelle 5.1: Kombinationsformate (Auswahl)

Einzelformate

Die Liste der Einzelformate ist umfangreicher und erlaubt beliebige Kombinationen der einzelnen Bestandteile von Datum und Zeit.

Format	Beschreibung
d, dd	Tag ohne und mit führender Null
ddd	Kurzform des Wochentagsnamens (Fr)
dddd	Langform des Wochentagsnamens (Freitag)
M, MM	Monat ohne und mit führender Null
MM	Kurzform des Monatsnamens (Feb)
MMM	Langform des Monatsnamens (Februar)
yy	Jahr, zweistellig
yyyy	Jahr, vierstellig

Tabelle 5.2: Auswahl wichtiger Formatieranweisungen für Datum und Zeit

Format	Beschreibung
H, HH h, hh	Stunden im 24–Stunden-Format ohne und mit führender Null, die kleinen Buchstaben erzeugen 12–Stunden-Format
m, mm	Minute ohne und mit führender Null
s, ss	Sekunde ohne und mit führender Null
tt	AM oder PM
:	Standardtrennzeichen für Zeitwert
/	Standardtrennzeichen für Datum, der Schrägstrich wird auf einem deutschen Windows als Punkt erscheinen.

Tabelle 5.2: Auswahl wichtiger Formatieranweisungen für Datum und Zeit (Forts.)

Das folgende Beispiel zeigt einige Formatierungen:

```
<script runat="server" language="C#">
void Page_load()
{
    ausgabe.InnerHtml  = "Kursinfo:<br/>";
    DateTime kurs = new DateTime(2002, 5, 26, 9, 0, 0);
    ausgabe.InnerHtml += kurs.ToString(@"dd.MM.yyyy \u\m H \U\h\r");
    ausgabe.InnerHtml += "<br/>";
    ausgabe.InnerHtml += kurs.ToString("dd/MM (ddd)");
    ausgabe.InnerHtml += "<br/>";
    ausgabe.InnerHtml += kurs.ToString("hh:mm");
}
</script>
```

Listing 5.4: Datums- und Zeitformatierungen (format21.aspx)

Beachten Sie das @-Zeichen zur Unterdrückung der Escape-Zeichen, mit denen freier Text in die Formatierung eingebaut wird:

```
ausgabe.InnerHtml += kurs.ToString(@"dd.MM.yyyy \u\m H \U\h\r");
```

Erwähnenswert ist auch die Umwandlung des /-Zeichens in einen Punkt bei der folgenden Formatierung:

```
ausgabe.InnerHtml += kurs.ToString("dd/MM (ddd)");
```

Die Abbildung zeigt den Effekt der Formatierungen:

Abbildung 5.6: Auswirkung von Datumsformatierungen

Globalisierungseinstellungen

Die formatierten Ausgaben folgen der aktuellen Spracheinstellung. Die Ausgabe folgt auf einem deutschen Windows den Regeln der deutschen Sprache. Wenn das nicht gewünscht ist oder auch dann sicher gestellt werden soll, wenn die Sprachversion des Betriebssystems nicht garantiert werden kann, müssen Sie sich mit den Globalisierungseinstellungen beschäftigen. Der dafür benötigte Namensraum `System.Globalization` muss zusätzlich eingebunden werden:

```
<% @Import Namespace="System.Globalization" %>
```

Aus diesem Namensraum können Sie die Klasse `CultureInfo` verwenden, deren Konstruktor eine Zeichenkette der Form „sprachcode-land“ erwartet. Beispielsweise werden die deutschsprachigen Länder mit „de-DE“, „de-AT“, „de-CH“ usw. bezeichnet. Ist die Unterscheidung nicht wichtig, reicht auch die Angabe „de“ aus.

```
<% @Import Namespace="System.Globalization" %>
<script runat="server" language="C#">
void Page_load()
{
    ausgabe.InnerHtml  = "Wochentage, mehrsprachig:<br/>";
    DateTime kurs = new DateTime(2002, 5, 26, 9, 0, 0);
    CultureInfo deutsch = new CultureInfo("de-DE");
    CultureInfo france = new CultureInfo("fr-FR");
    CultureInfo rossija = new CultureInfo("ru-RU");
    CultureInfo chinese = new CultureInfo("zh-CN");
    ausgabe.InnerHtml += "<br>" + kurs.ToString("dd/MM (ddddd)", deutsch);
    ausgabe.InnerHtml += "<br>" + kurs.ToString("dd/MM (ddddd)", france);
    ausgabe.InnerHtml += "<br>" + kurs.ToString("dd/MM (ddddd)", rossija);
    ausgabe.InnerHtml += "<br>" + kurs.ToString("dd/MM (ddddd)", chinese);
}
</script>
```

Listing 5.5: Ausgabe von Datumsangaben in Abhängigkeit von Sprache und Ländercode (dt_formatcult.aspx)

Im Beispiel werden vier Sprachobjekte abgeleitet, die für die Ausgabe der Daten in Deutsch, Französich sowie Russisch und Chinesisch sorgen – unabhängig von der Sprache des Webserver-Betriebssystems. Das Ergebnis zeigt die folgende Abbildung:

Abbildung 5.7: Mehrsprachige Ausgabe von Wochentagen

Beachten Sie, dass für die Darstellung von asiatischen Schriftzeichen die entsprechenden Sprachpakete im Browser installiert sein müssen.

5.3 Aufzählungen und Kollektionen

Der Umgang mit Arrays in C# wurde bereits in der Spracheinführung gezeigt. Tatsächlich ist dies kein Sprachmerkmal, sondern die Abbildung der Klasse `System.Array` des Frameworks. Arrays werden sehr häufig eingesetzt und die direkte Verfügbarkeit erleichtert den Umgang damit.

Die Speicherung von zusammengehörenden Werten kann jedoch mit mehr Verfahren erfolgen. Für alle denkbaren Variationen bietet das .Net-Framework die nötige Unterstützung. Dafür steht sogar ein eigener Namensraum mit dem Namen `System.Collections` zur Verfügung. Im weitesten Sinne handelt es sich um Klassen, mit denen Wertegruppen, Felder, Schlüssel-/Wertepaare und ähnliche Datensammlungen manipuliert werden können.

5.3.1 Einführung in die Welt der Kollektionen

Viele Programmiersprachen bieten neben einfachen Arrays auch so genannte Hashes. Dies sind Arrays mit nichtnumerischen Indizes. .Net bietet gleich fünf Varianten solcher Gebilde, mit denen sehr elegant programmiert werden kann.

Nachfolgend finden Sie Liste der Basisklassen des Namensraumes `System.Collections`.

- `ArrayList` – Eine Werte-Liste, die ähnlich wie ein Array verwendet werden kann. Die Indizes sind numerisch.
- `Hashtable` – Die Implementierung des klassischen Hashes; die Indizes (Schlüssel) sind Zeichenketten.
- `SortedList` – Diese Klasse bietet beide Indexvarianten, Zahlen und Zeichenketten, realisiert also ein Array mit dem Verhalten eines Hashes.

- `Queue` – Arrays und Hashes erlauben den wahlfreien Zugriff auf die Elemente. Bei Objekten der Klasse `Queue` ist dies anders. Hier erfolgt der Zugriff nach dem FIFO-Prinzip (FIFO = First In, First Out). Elemente, die zuerst abgelegt wurden, müssen auch zuerst wieder entnommen werden.
- `Stack` – Wie bei der Klasse `Queue` ist der wahlfreie Zugriff nicht möglich, das Speicherprinzip ist hier allerdings FILO (FILO = First In, Last Out). Realisiert wird ein Stapel – das zuletzt abgelegte Elemente muss zuerst wieder entnommen werden.

Allen Elementen ist gemeinsam, dass die Einträge mit `foreach` sequenziell ausgelesen werden können. Die wichtigsten Kollektionen `ArrayList`, `Hashtable` und `SortedList` werden nachfolgend vorgestellt.

Zugriff auf den Namensraum

Namensraum aktivieren

Der Zugriff auf den Namensraum ist in ASP.NET standardmäßig nicht aktiviert. Sie müssen deshalb Ihren Programmen folgende Direktive voranstellen:

```
<% @Import Namespace="System.Collections" %>
```

Wenn Sie mit hinterlegtem Code arbeiten, ist diese Anweisung im Kopf des Programms zu ergänzen:

```
using System.Collections;
```

Beispiele

Alle folgenden Beispiele dieses Abschnitts verwenden Code Behind. Der Abdruck der Listings beschränkt sich deshalb auf die *cs*-Dateien. Die Ausgabe erfolgt einheitlich mit einer kleinen *aspx*-Datei:

```
<% @Page Language="C#" Inherits="class" src="class.cs"%>
<html>
    <head>
        <title>Kollektionen</title>
    </head>
    <body>
        <asp:label runat="server" id="ausgabe"/>
    </body>
</html>
```

Listing 5.6: Ausgabe der in den Beispielen erzeugten Daten

Die Attribute `Inherits` und `src` ändern sich jeweils, diese Angabe finden Sie bei den Listingunterschriften.

5.3.2 ArrayList

Methoden

Eine `ArrayList` ist eine Sammlung beliebiger Objekte. Die Ablage erfolgt mit numerischen Indizes. Die Objekte können anhand des Indizes oder des Objekts selbst entnommen werden. Zum Hinzufügen wird die Methode `Add` eingesetzt. Entfernt werden Objekte unter Angabe des Index mit `RemoveAt`, bei Angabe des Objekts mit `Remove`. An einer bestimmten Stelle – mit Indexnummer – kann ein Element mit `Insert` eingefügt werden. `Clear` entfernt alle Elemente.

Eigenschaften

Der Zugriff auf die Elemente erfolgt in C# über einen Indexer: `element[3]`. Die Anzahl kann mit `Count` ermittelt werden. Der aktuell reservierte Platz wird mit `Capacity` festgelegt. Der Standardwert ist 16; werden mehr Elemente hinzugefügt, wird das Array bei Überschreiten der Grenze verdoppelt. `Capacity` kann nie kleiner als `Count` sein.

Ein Beispiel zeigt die Verwendung:

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using System.Collections;

public class floyd : Page
{
    public Label ausgabe;

    void Page_Load()
    {
        ausgabe.Text = "<h3>Aus ArrayList erzeugt.</h3>";
        ArrayList album = new ArrayList();
        ausgabe.Text += "Erlaubt sind derzeit " + album.Capacity + " Werte";
        album.Add("The Wall");
        album.Add("Animals");
        album.Add("Ummelgummel");   // Fehler
        album.Add("Atom Heart Mother");
        album.Add("Meddle");
        album.Add("Wish You Were Here");
        album.Add("The Final Cut");
        album.Add("The Division Bell");
        album.Add("The Dark Side Of The Moon");
        album.Remove("Ummelgummel");
        album.Insert(2, "Ummagumma");
```

```
        foreach (String titel in album)
        {
            ausgabe.Text += "<br/>" + titel;
        }
        ausgabe.Text += "<br/>Es waren: " + album.Count + " Werte";
    }

}
```

Listing 5.7: Verwendung von `ArrayList` (coll_arraylist.aspx und floyd.cs)

Wie es funktioniert

Zuerst wird hier ein Objekt vom Typ `ArrayList` erzeugt:

```
ArrayList album = new ArrayList();
```

Dann werden mehrere Werte hinzugefügt:

```
album.Add("The Wall");
```

Ein falscher Wert wird wieder entfernt:

```
album.Remove("Ummelgummel");
```

An seiner Stelle wird das korrekte Element eingefügt:

```
album.Insert(2, "Ummagumma");
```

Die Indizierung ist Nullbasiert. Wenn ein Element mit dem Index 2 eingefügt wird, verschieben sich die alten Elemente ab Index 2; 2 wird also 3 usw.

Die Ausgabe mit `foreach` ist so einfach wie möglich gehalten. Falls Ihre Liste verschiedene Objekte enthält, müssen Sie hier natürlich eine entsprechende Sonderbehandlung einfügen.

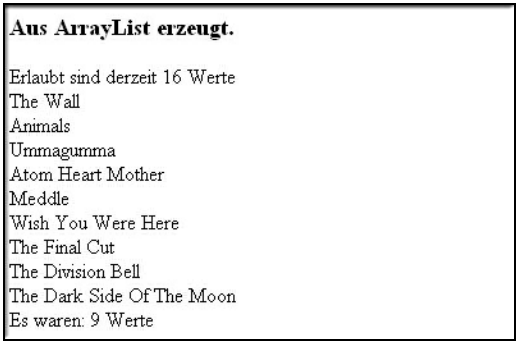


Abbildung 5.8: Ausgabe von Daten aus einer `ArrayList`

Methoden Weitere Methoden, die Sie verwenden können, sind unter anderem:

- `Reverse` – Hiermit wird die Reihenfolge der Elemente gedreht.
- `Sort` – Sortiert den Inhalt. Beachten Sie, dass Sie eigene Sortieralgorithmen mit der Implementierung der Schnittstelle `IComparer` entwickeln können. Wie das geht, wurde bereits in Abschnitt „Definition eigener Suchalgorithmen“ auf Seite 116 gezeigt.
- `Contains` – Die Methode ermittelt, ob ein bestimmtes Element in der Liste ist.
- `BinarySearch` – Hiermit können Sie Objekte ebenfalls suchen. Für kompliziertere Gebilde kann die `IComparer`-Schnittstelle zur Implementierung eigener Suchmethoden genutzt werden.
- `AddRange` – Fügen Sie mit dieser Methode mehrere Elemente hinzu. Der Parameter soll vom Typ `ICollection` sein; eine Schnittstelle, die viele andere Datenlisten im .Net-Framework auch verwenden und die sehr gut die Kompatibilität sicherstellt, beispielsweise zu `DataView` (siehe Kapitel 6).

5.3.3 Hashtable

Falls Sie das alte ASP kennen, kann man diese Implementierung mit *Scripting.Dictionary* vergleichen. Allerdings bietet .Net – wie nicht anders zu erwarten war – mehr als dies.

Methoden Eine `Hashtable` ist eine Sammlung beliebiger Objekte. Die Ablage erfolgt mit eigenen, alphanumerischen Schlüsseln. Objekte können anhand des Schlüssels oder des Objekts selbst entnommen werden. Zum Hinzufügen wird die Methode `Add` eingesetzt. Entfernt werden Objekte unter Angabe des Index mit `RemoveAt`, bei Angabe des Objekts mit `Remove`. An einer bestimmten Stelle – mit Indexnummer – kann mit `Insert` eingefügt werden. `Clear` entfernt alle Elemente.

Das folgende Beispiel zeigt den Umgang mit `Hashtable`. Gespeichert werden hier als Werte nicht Zeichenketten, sondern Strukturen:

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using System.Collections;

public class books : Page
{
    public Label ausgabe;
```

```
    public struct  bookdata
    {
        public string autor;
        public double  preis;
        public int seiten;
    }

    void Page_Load()
    {
        ausgabe.Text = "<h3>Aus Hashtable erzeugt.</h3>";
        Hashtable buchliste = new Hashtable();
        bookdata buch = new bookdata();
        buch.autor = "Jörg Krause";
        buch.preis = 24.95;
        buch.seiten = 360;
        buchliste.Add("ASP.NET lernen", buch);
        buch.autor = "Jörg Krause, Uwe Bünning";
        buch.preis = 59.95;
        buch.seiten = 960;
        buchliste.Add("ASP.NET mit C#", buch);
        ArrayList abuch = new ArrayList();
        ArrayList atitel = new ArrayList();
        abuch.AddRange(buchliste.Values);
        atitel.AddRange(buchliste.Keys);
        for(int i = 0; i < atitel.Count; i++)
        {
            ausgabe.Text += "<br/><b>Titel: " + atitel[i] + "</b>";
            bookdata bd = (bookdata) abuch[i];
            ausgabe.Text += "<br/>Autor: " + bd.autor;
            ausgabe.Text += "<br/>Preis: " + bd.preis;
            ausgabe.Text += "<br/>Seiten: " + bd.seiten;
        }
    }

}
```

Listing 5.8: Verwendung von Hashtable (coll_books.aspx und books.cs)

Wie es funktioniert

Das `Hashtable`-Objekt soll Bücher speichern. Dazu wird eine Struktur definiert, die die Daten aufnimmt:

```
private struct bookdata
```

Diese enthält drei Variablen, in denen der Autor, der Preis und die Anzahl Seiten gespeichert werden.

Im Hauptprogramm wird zuerst das `Hashtable`-Objekt erzeugt:

```
Hashtable buchliste = new Hashtable();
```

Dann wird eine Instanz der Struktur benötigt:

```
bookdata buch = new bookdata();
```

Dieser werden dann die Werte zugewiesen:

```
buch.autor = "Jörg Krause";
buch.preis = 24.95;
buch.seiten = 360;
```

Das fertige Objekt wird dann der `Hashtable` hinzugefügt:

```
buchliste.Add("ASP.NET lernen", buch);
```

Mit einem weiteren Objekt soll jetzt der Zugriff auf die Elemente erfolgen. Auf direktem Wege ist das leider nicht möglich, deswegen werden Schlüssel und Werte einzeln in Instanzen von `ArrayList` abgelegt. Lesen Sie in Abschnitt „Zugriff über Aufzählungen“ auf Seite 295, wie dies mit Aufzählungen eleganter erfolgen kann. Hier zuerst die beiden `ArrayList`-Objekte:

```
ArrayList abuch = new ArrayList();
ArrayList atitel = new ArrayList();
```

Dann werden die Titel zugewiesen; dies sind die Schlüssel der `Hashtable`:

```
atitel.AddRange(buchliste.Keys);
```

Mit der Eigenschaft `Values` erfolgt der Zugriff auf die Strukturen:

```
abuch.AddRange(buchliste.Values);
```

Das Auslesen der Elemente erfolgt in einer `for`-Schleife, der Umfang des Arrays wird mit der Eigenschaft `Count` bestimmt:

```
for(int i = 0; i < atitel.Count; i++)
```

Auf den Titel kann direkt über den Indexer zugegriffen werden:

```
ausgabe.Text += "<br/><b>Titel: " + atitel[i] + "</b>";
```

Zum Auslesen der Struktur muss diese erst wieder hergestellt werden:

```
bookdata bd = (bookdata) abuch[i];
```

Auf die Instanzen erfolgt nun der Zugriff in gewohnter Weise:

```
ausgabe.Text += "<br/>Autor: " + bd.autor;
```

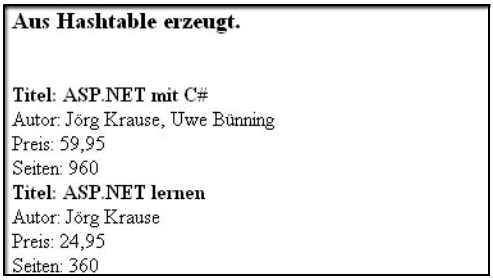


Abbildung 5.9: Ausgabe der in einer Hashtable gespeicherten Daten

Interner Aufbau der Hashtable

Der Ansatz, die Werte und Schlüssel über die `ICollection`-Schnittstelle zu entnehmen, ist zwar direkt und oft sinnvoll, kann aber vereinfacht werden. Intern werden nämlich nicht diese Werte und Schlüssel einzeln gespeichert, sondern in Objekten vom Typ `DictionaryEntry`.

Ein Durchlaufen der Liste kann auch mit `foreach` erfolgen, was folgendermaßen aussieht:

```
foreach(DictionaryEntry dasbuch in buchliste)
{
   ausgabe.Text += "<br/><b>Titel: " + dasbuch.Key + "</b>";
   bookdata bd = (bookdata) dasbuch.Value;
   ausgabe.Text += "<br/>Autor: " + bd.autor;
   ausgabe.Text += "<br/>Preis: " + bd.preis;
   ausgabe.Text += "<br/>Seiten: " + bd.seiten;
}
```

Listing 5.9: Vereinfachte Ausgaben (die cs-Datei dazu finden Sie unter dem Namen books2.cs)

Objekte vom Typ `DictionaryEntry` kennen zwei Eigenschaften: `Key` und `Value`, mit denen dann auf Schlüssel und Werte zugegriffen werden kann.

5.3.4 Zugriff über Aufzählungen

Vielfach geben Eigenschaften Aufzählungen zurück oder Objekte können ihren Inhalt alternativ als Kollektion exportieren. Viele Klassen verfügen über eine Methode mit dem Namen `GetEnumerator`. Zurückgegeben wird eine Aufzählung, deren Definition auf der Schnittstelle `System.Collection.IEnumerator` basiert.

Weit über 100 Klassen implementieren diese Schnittstelle. Der Umgang mit `IEnumerator` ist deshalb elementares Handwerkszeug.

Anwendungsbeispiel

Tabellen manipulieren

Das folgende Beispiel zeigt den Einsatz – der Zugriff erfolgt auf eine Kollektion von Tabellenzellen einer HTML-Tabelle:

```
<%@ Page Language="C#" %>
<script runat="server">
void Page_Load(Object sender, EventArgs e)
{
  int zahlReihen = 4;
  int zahlZellen = 8;
  Random rnd = new Random(100);
  for (int j = 0; j < zahlReihen; j++)
  {
    TableRow reihe = new TableRow();
    for (int i = 0; i < zahlZellen; i++)
    {
        TableCell zelle = new TableCell();
        zelle.Text = rnd.Next(1000, 9999).ToString();
        reihe.Cells.Add(zelle);
    }
    tabelle.Rows.Add(reihe);
  }
  if (!IsPostBack)
  {
    ArrayList arrayReihe = new ArrayList();
    for (int k = 1; k <= zahlReihen; k++)
    {
       arrayReihe.Add(k.ToString());
    }
    liste.DataSource = arrayReihe;
    liste.DataBind();
  }
}

void holeReihe(object sender, EventArgs e)
{
   int reihe = liste.SelectedIndex;
   TableCell aktuelleZelle;
   IEnumerator aufzaehlung = tabelle.Rows[reihe].Cells.GetEnumerator();
   ausgabe.Text = "Folgende Reihe wurde ausgew&auml;hlt: <br/>";
   while (aufzaehlung.MoveNext())
   {
      aktuelleZelle = (TableCell) aufzaehlung.Current;
      ausgabe.Text += " " + aktuelleZelle.Text;
   }
}
```

```
</script>

<html>
    <head>
        <title>GetEnumrator</title>
    </head>
    <body>
    <h3>Aufz&auml;hlungen verarbeiten</h3>
    <form runat="server">
        <asp:Table id="tabelle"  runat="server"/>
        Auswahl einer Reihe:
        <asp:DropDownList id="liste"  runat="server"/>
        <asp:Button id="senden" Text="Reihe ermitteln"
                    onclick="holeReihe" runat="server"/>
        <hr noshade="noshade" width="300px" align="left"/>
        <asp:Label id="ausgabe" runat="server"/>
    </form>
    </body>
</html>
```

Listing 5.10: Dynamisch erzeugte Tabelle und Auswahl einer Zellenreihe mit GetEnumerator (getenum.aspx)

Bevor die Funktion im Detail behandelt wird, soll ein Blick auf die Ausgabe das Ergebnis zeigen:

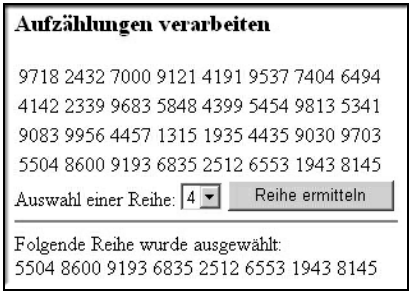


Abbildung 5.10: Tabellarische Ausgabe und Zugriff auf Tabelleninhalte

Wie es funktioniert

Das Programm zeigt neben der Nutzung der Aufzählung noch zwei besondere Techniken, die anderswo im Buch nicht angesprochen werden konnten:

- Dynamisches Erzeugen einer Tabelle
- Binden einer Datenquelle an eine `DropDownList`.

Tabelle aufbauen

Betrachten Sie zuerst den HTML-Teil. Dort wird eine Tabelle ganz allgemein definiert – noch ohne irgendwelchen Inhalt:

```
<asp:table id="tabelle" runat="server"/>
```

Dann wird noch ein `DropDownList`-Steuerelement erzeugt, auch ohne Inhalt:

```
<asp:DropDownList id="liste" runat="server"/>
```

Das Formular wird komplettiert durch eine Sendeschaltfläche:

```
<asp:Button id="senden" Text="Reihe ermitteln"
onclick="holeReihe" runat="server"/>
```

Damit eine Ausgabe zu sehen ist, folgt noch ein `Label`-Steuerelement:

```
<asp:Label id="ausgabe" runat="server"/>
```

Die eigentliche Arbeit wird wieder über ein Programm erledigt. Dieses Programm beginnt mit `Page_Load`. Zuerst werden mehrere Variablen definiert, unter anderem auch ein Zufallsgenerator, dessen Startwert festgelegt ist:

```
Random rnd = new Random(100);
```

Es folgt eine `for`-Schleife, die die Reihen der Tabellen erzeugt:

```
for (int j = 0; j < zahlReihen; j++)
```

Dazu wird jeweils ein neues Reihen-Objekt erzeugt:

```
TableRow reihe = new TableRow();
```

Innerhalb der Reihe sind nun die Zellen zu erzeugen:

```
for (int i = 0; i < zahlZellen; i++)
```

Auch hier wird für jede Zelle ein entsprechendes Objekt erzeugt:

```
TableCell zelle = new TableCell();
```

Jede Zelle wird nun mit einer Zufallszahl gefüllt. Die Datenquelle kann natürlich auch anders gestaltet werden:

```
zelle.Text = rnd.Next(1000, 9999).ToString();
```

Dann wird die fertige Zelle der aktuellen Reihe hinzugefügt:

```
reihe.Cells.Add(zelle);
```

Ist eine Reihe fertig, wird auch diese an die Tabelle angefügt:

```
tabelle.Rows.Add(reihe);
```

Liste aufbauen

Jetzt ist noch das Formular auszuwerten. Der Vorgang besteht aus zwei Teilen. Zuerst muss noch die Liste aufgebaut werden, allerdings nur beim ersten Aufruf des Programms (`IsPostBack` gleich `false`). Dazu wird ein `ArrayList`-Objekt instanziiert:

```
ArrayList arrayReihe = new ArrayList();
```

Dieses wird mit so vielen Zahlen gefüllt, wie Reihen in der Tabelle existieren:

```
for (int k = 1; k <= zahlReihen; k++)
{
   arrayReihe.Add(k.ToString());
}
```

Dieses Array wird nun als Datenquelle für die Liste verwendet:

```
liste.DataSource = arrayReihe;
```

Mit der Bindung erscheinen die Listeneinträge auf der Seite:

```
liste.DataBind();
```

Zugriff auf die Aufzählung

Die Aufzählung wird nun benötigt, um eine Zeile der Tabelle auszulesen. Dies passiert jedoch erst, wenn das Formular gesendet wurde, also in der Methode `holeReihe`, die das Ereignis `onClick` serverseitig verarbeitet:

```
void holeReihe(object sender, EventArgs e)
```

Zuerst wird der Index ermittelt, der in der Liste ausgewählt wurde:

```
int reihe = liste.SelectedIndex;
```

Dann wird ein Zellenobjekt deklariert:

```
TableCell aktuelleZelle;
```

Nun kann auf die Reihe einer Tabelle zugegriffen werden. Die Methode `GetEnumarator` liefert die komplette Reihe als Objekt vom Typ `IEnumerator`:

```
IEnumerator aufzaehlung = tabelle.Rows[reihe].Cells.GetEnumerator();
```

Nach einer Textausgabe für den Benutzer wird diese Aufzählung nun duchlaufen. Dabei gibt die Methode `MoveNext` solange `true` zurück, wie noch Elemente vorhanden sind. Gleichzeitig wird mit jedem Aufruf ein interner Zeiger der Aufzählung ein Element weiter gesetzt:

```
while (aufzaehlung.MoveNext())
```

Das aktuelle Element wird nun mit der Eigenschaft `Current` gelesen:

```
aktuelleZelle = (TableCell) aufzaehlung.Current;
```

Der Rest beschäftigt sich mit der Ausgabe, die zu dem bereits gezeigten Ergebnis führt:

```
ausgabe.Text += " " + aktuelleZelle.Text;
```

Zusammenfassung

Die Schnittstelle `IEnumerator` ist sehr einfach. Hier eine Zusammenfassung:

- `Current` – Die einzige Eigenschaft gibt das aktuelle Element zurück.
- `MoveNext` – Mit dieser Methode wird die Aufzählung durchlaufen. Jeder Aufruf schaltet ein Element weiter. Sind keine Elemente mehr vorhanden, wird `false` zurückgegeben, sonst `true`.
- `Reset` – Mit dieser Methode wird der Zeiger wieder auf das erste Element zurück gesetzt.

Beide Methoden haben keinen Parameter.

5.4 Zugriff auf das Dateisystem

Für den Zugriff auf das Dateisystem via Web gibt es viele sinnvolle Anwendungen. Das .Net-Framework bietet eine reichhaltige Unterstützung dafür – nicht nur in speziellen Klassen, sondern auch durch darauf abgestimmte Methoden und Eigenschaften anderer Klassen, beispielsweise zur Analyse von Dateidaten.

5.4.1 Einführung

Einige Einsatzbeispiele, wofür Zugriffe auf das Dateisystem erforderlich sein können, sind:

- *Protokolldateien* – Protokollieren Sie Vorgänge in Ihrer Applikation in eine Textdatei.
- *Ablage von Formulardaten* – Legen Sie den Inhalt eines Formulars als Textdatei ab.
- *News und Tipp des Tages* – Zeigen Sie auf Ihrer Website News an, die Sie als Text- oder HTML-Datei auf dem Server ablegen.

Zugriff auf den Namensraum

Namensraum aktivieren

Der Zugriff auf den Namensraum ist in ASP.NET standardmäßig nicht aktiviert. Sie müssen deshalb Ihren Programmen folgende Direktive voranstellen:

```
<% @Import Namespace="System.IO" %>
```

Wenn Sie mit hinterlegtem Code arbeiten, ist diese Anweisung im Kopf des Programms zu ergänzen:

```
using System.IO;
```

Dieser Namensraum enthält eine große Anzahl Klassen, die hier nur ansatzweise vorgestellt werden können. Wesentlich sind:

- `Directory` – Zugriff auf Verzeichnisse
- `File` – Zugriff auf Dateien
- `StreamReader`, `StreamWriter` – Lesen und Schreiben von Textdateien

Beispiele

Alle folgenden Beispiele dieses Abschnitts verwenden hinterlegten Code. Der Abdruck der Beispiele beschränkt sich deshalb auf die *cs*-Dateien. Die Ausgabe erfolgt einheitlich mit einer kleinen *aspx*-Datei:

```
<% @Page Language="C#" Inherits="class" src="class.cs"%>
<html>
    <head>
        <title>Kollektionen</title>
    </head>
    <body>
        <asp:label runat="server" id="ausgabe"/>
    </body>
</html>
```

Listing 5.11: Ausgabe der in den Beispielen erzeugten Daten

Die Attribute `Inherits` und `src` ändern sich jeweils, diese Angabe finden Sie bei den Listingunterschriften.

5.4.2 Zugriff auf Verzeichnisse und Dateien

An erster Stelle soll der Zugriff auf Verzeichnisse gezeigt werden. Vorab soll an die Behandlung des Backslashs erinnert werden. Da dieser in Pfadangaben vorkommt und zugleich als Markierungs-Zeichen dient, wird fast immer das @-Zeichen vor die Zeichenkette gestellt, um die Auswertung zu unterdrücken. Ohne dieses schreiben Sie statt „c:\pfad“ dann „c:\\pfad“.

Das folgende Beispiel zeigt ein Formular, in das ein Pfad eingegeben werden kann. Ist der Pfad vorhanden, wird eine Liste der Verzeichnisse und Dateien darin angezeigt:

```
<% @Page Language="C#" Inherits="dir" src="readdir.cs"%>
<html>
    <head>
        <title>Kollektionen</title>
    </head>
    <body>
        <form runat="server">
            <asp:textbox runat="server" id="dirname"/>
            <asp:button runat="server"  id="senden" text="Anzeigen"/>
        </form>
        <asp:label runat="server" id="ausgabe"/>
    </body>
</html>
```

Listing 5.12: Formular zur Anzeige von Verzeichnisinformationen (file_readdir.aspx)

Die Klasse `dir` erledigt dann die eigentliche Arbeit:

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using System.IO;

public class dir : Page
{
    public Label ausgabe;
    public TextBox dirname;

    void Page_Load()
    {
        if (!Page.IsPostBack)
        {
            ausgabe.Text = "Geben Sie ein Verzeichnis an";
        } else {
            ausgabe.Text = "Verzeichnisinhalt für <b>"
                           + dirname.Text + "</b>";
            try
            {
                DirectoryInfo dinfo = new DirectoryInfo(dirname.Text);
                ausgabe.Text += "<br/>Erzeugt am : "
                             + dinfo.CreationTime.ToShortDateString();
                ausgabe.Text += "<br/><u>Unterverzeichnisse:</u>";
                foreach (DirectoryInfo subdir in dinfo.GetDirectories())
                {
                    ausgabe.Text += "<br/>" + subdir.FullName;
                }
```

```
                ausgabe.Text += "<br/><u>Dateien:</u>";
                foreach (string files in Directory.GetFiles(dirname.Text))
                {
                    FileInfo finfo = new FileInfo(files);
                    ausgabe.Text += "<br/>" + finfo.Name;
                    ausgabe.Text += " [" + finfo.CreationTime  + "] ";
                    ausgabe.Text += finfo.Length + " Byte";
                }
            }
            catch (DirectoryNotFoundException nichtgefunden)
            {
                ausgabe.Text  = "<span style=\"color:red\">
                                Verzeichnis nicht gefunden: ";
                ausgabe.Text += nichtgefunden.Message + "</span>";
            }
        }

    }
}
```

Listing 5.13: Lesen eines Verzeichnisses (readdir.cs)

Wie es funktioniert

Beim Zugriff auf Verzeichnisse tritt oft das Problem auf, dass der Pfad nicht gefunden werden kann oder die Zugriffsrechte nicht ausreichend sind. Um robuste Applikationen zu schreiben, sollten die Ausnahmen abgefangen werden, die die Klassen erzeugen. Ideal ist dafür die `try catch`-Anweisung. Der `try`-Zweig wird zuerst ausgeführt, bis eine Ausnahmebedingung auftritt.

Verzeichnisinformationen

Der Zugriff auf Verzeichnisinformationen erfolgt mit der Klasse `DirectoryInfo`. In `dirname.Text` steht der Inhalt des Eingabefeldes des Formulars:

```
DirectoryInfo dinfo = new DirectoryInfo(dirname.Text);
```

Die Unterverzeichnisse können nun über eine Kollektion ermittelt werden, die `GetDirectories` zurückgibt. Für jedes Unterverzeichnis wird wiederum ein Objekt vom Typ `DirectoryInfo` erzeugt:

```
foreach (DirectoryInfo subdir in dinfo.GetDirectories())
```

Die Objekte haben viele Eigenschaften und Methoden, von denen hier nur die Eigenschaft `FullName` verwendet wird:

```
ausgabe.Text += "<br/>" + subdir.FullName;
```

Nun folgt der Zugriff auf die Dateien des Hauptverzeichnisses. Auf direktem Wege kann dazu die statische Methode `GetFiles` verwendet werden, die als Parameter eine Pfadangabe erwartet:

```
foreach (string files in Directory.GetFiles(dirname.Text))
```

Dateiinformationen

Die Methode gibt eine Liste von Verzeichnisnamen zurück – als Zeichenkette. Sollen Dateiinformationen ermittelt werden, die über den Namen hinausgehen, muss ein `FileInfo`-Objekt erzeugt werden:

```
FileInfo finfo = new FileInfo(files);
```

Auch dieses Objekt ist reichhaltig mit Eigenschaften und Methoden ausgestattet. Zuerst erfolgt der Zugriff auf den Namen (die Variable `files` selbst enthält auch den Pfad):

```
ausgabe.Text += "<br/>" + finfo.Name;
```

Im Beispiel wird auch das Dateidatum – der Zeitpunkt der Erzeugung – ermittelt:

```
ausgabe.Text += " [" + finfo.CreationTime  + "] ";
```

Auch die Größe der Datei ist direkt feststellbar:

```
ausgabe.Text += finfo.Length + " Byte";
```

Falls die Anzeige des Verzeichnisses misslingt, wird eine Ausnahme vom Typ `DirectoryNotFoundException` erzeugt. Mit `catch` können Sie diese abfangen:

```
catch (DirectoryNotFoundException nichtgefunden)
```

Interessant ist eventuell die Fehlermeldung, die die Klasse hier erzeugt:

```
ausgabe.Text += nichtgefunden.Message;
```

Damit sind alle Basisfunktionen für den Verzeichnis- und Dateizugriff bereits im Einsatz gewesen.

Abbildung 5.11: Ausgabe des Inhalts eines Verzeichnisses

Methoden und Eigenschaften für den Verzeichniszugriff

Die wichtigsten Methoden der Klasse `Directory` zeigt die folgende Tabelle. Alle Methoden sind statisch, werden also in der Form `Directory.Methode()` verwendet.

Eigenschaft	Bedeutung
`GetDirectories`	Gibt eine Kollektion der enthaltenen Verzeichnisse zurück: • `GetDirectories(pfad);` • `GetDirectories(pfad,platzhalter)`
`GetCreationTime`	Datum der Erzeugung lesen und schreiben
`GetFiles`	Gibt eine Kollektion der Dateien zurück: • `GetFiles(pfad);` • `GetFiles(pfad,platzhalter)`
`CreateDirectory`	Erzeugt ein Verzeichnis
`SetDirectory`	Setzt das aktuelle Verzeichnis
`Move`	Verschiebt ein Verzeichnis mit dem gesamten Inhalt: • `Move(pfad_quelle, pfad_ziel);`
`Delete`	Löscht ein Verzeichnis und – wenn *subdirs* gleich `true` ist – auch den Inhalt: • `Delete(pfad [, subdirs]);`
`Exists`	Prüft, ob ein Verzeichnis existiert

Tabelle 5.3: Methoden der Klasse `Directory`

Die Kollektionen sind vom Typ `ICollection` und können direkt an `ArrayList`, `Hashtable` und `Array` übergeben werden.

Der Zugriff auf die Informationen über ein Verzeichnis erfolgt mit `DirectoryInfo`. Im Gegensatz zu `Directory` sind die Methoden nicht statisch und es gibt viele Eigenschaften. Die Anwendung erfolgt durch direkte Instanziierung des Objekts:

```
DirectoryInfo dinfo = new DirectoryInfo("pfadangabe");
```

Es stehen hier ähnliche Methoden zur Verfügung wie bei der Klasse `Directory`. Hinzu kommen Eigenschaften, die Auskunft über Attribute und Namen geben. Die wichtigsten finden Sie nachfolgend:

Eigenschaft	Beschreibung
`Attributes`	Attribute
`CreationTime`	Datum der Erzeugung
`FullName`	Pfad und Name
`Parent`	Übergeordnetes Verzeichnis
`Root`	Stammverzeichnis

Tabelle 5.4: Wichtige Eigenschaften der Klasse `DirectoryInfo`

Dateiattribute

Die Attribute sind eine Aufzählung mit dem Namen `FileAttributes`. Diese Aufzählung ist intern ein Bitfeld. Die Erkennung eines spezifischen Attributes kann durch die Formel `Attribut & FileAttributes.AttributName` erfolgen. Das im letzten Beispiel gezeigte Programm kann leicht um eine Auswertung der Attribute erweitert werden. Dazu wird folgende Methode der Klasse `dir` hinzugefügt:

```
private string ShowAttributes(FileAttributes fa)
{
    string sa = "";
    if ((fa & FileAttributes.Archive) > 0) sa += "A";
    if ((fa & FileAttributes.Compressed) > 0) sa += "C";
    if ((fa & FileAttributes.Directory) > 0) sa += "D";
    if ((fa & FileAttributes.Hidden) > 0) sa += "H";
    if ((fa & FileAttributes.System) > 0) sa += "S";
    if ((fa & FileAttributes.ReadOnly) > 0) sa += "R";
    return sa;
}
```

Listing 5.14: Methode zur Ermittlung der Verzeichnis- und Dateiattribute (file_readdir_attr.aspx und readdir_attr.cs)

Wie es funktioniert

Die Methode untersucht, ob ein Bit im Bitfeld `fa` gesetzt ist und fügt dann eine entsprechende Information an den Rückgabewert an. Der Aufruf erfolgt für ein Verzeichnis folgendermaßen:

```
ausgabe.Text += ShowAttributes(subdir.Attributes);
```

Dateien werden fast identisch angesprochen:

```
ausgabe.Text += ShowAttributes(finfo.Attributes);
```

A = Archiv
D = Directory
H = Hidden
S = System
R = Read Only
C = Compressed

Die Ausgabe sieht dann – je nach Inhalt des Verzeichnisses – folgendermaßen aus:

```
c:\Recycled ==> ADHS
c:\RECYCLER ==> DHS
c:\System Volume Information ==> ADHS
c:\temp ==> AD
c:\tmp ==> D
c:\trials ==> D
c:\W2K ==> AD
c:\WINNT ==> AD
c:\WUTemp ==> DH
Dateien:
10K1AT.EXE [30.7.1999 02:06:00] 312145 Byte ==> AC
10K1AT.LOG [20.9.1999 12:02:32] 7829 Byte ==> AC
ATTRIB.EXE [6.5.1998 20:01:50] 15252 Byte ==> AC
AUTOEXEC.BAT [13.3.2000 15:53:27] 437 Byte ==> ACH
AUTOEXEC.M01 [17.2.2000 14:10:55] 362 Byte ==> AC
boot.ini [24.2.2000 17:35:27] 287 Byte ==> HS
bootfont.bin [15.1.2002 13:36:33] 4952 Byte ==> AHSR
```

Abbildung 5.12: Anzeige von Verzeichnis- und Dateiattributen

Die Methoden der Klasse `DirectoryInfo` erlauben ähnliche Operationen wie bei `Directory`. Eine Auswahl zeigt die folgende Tabelle:

Methode	Beschreibung
`Create`	Verzeichnis erzeugen
`CreateSubDirectory`	Kompletten Pfad erzeugen
`Delete`	Verzeichnis löschen
`MoveTo`	Verzeichnis mit Inhalt verschieben: • `MoveTo(pfad_ziel)`
`GetDirectories`	Kollektion der Verzeichnisse
`GetFiles`	Kollektion der Dateien

Tabelle 5.5: Methoden der Klasse `DirectoryInfo`

Das Prinzip der Anwendung wurde bereits im ersten Beispiel dieses Abschnitts gezeigt. Beachten Sie bei `MoveTo`, dass nur die Angabe des Zieles erforderlich ist. Das `DirectoryInfo`-Objekt enthält bereits eine Pfadinformation der Quelle. Die Methode `Move` der Klasse `Directory` verlangt dagegen die Angabe von Quelle und Ziel.

Der Zugriff auf Dateien basiert auf zwei vergleichbaren Klassen: `File` mit ausschließlich statischen Methoden für den direkten Zugriff auf Dateien und `FileInfo` für den Zugriff über Objekte. Eine Auswahl wichtiger Methoden zeigt die folgende Tabelle:

Methode	Bedeutung
`GetAttributes`	Aufzählung der Dateiattribute
`GetCreationTime`	Datum der Erzeugung lesen und setzen
`Open`	Öffnet eine Datei zum Lesen oder Schreiben

`OpenRead`
`OpenWrite`
`OpenText`
	Zugriff auf eine Datei zum Lesen, Schreiben oder als Textdatei im UTF-8–Format

`Create`	Erzeugt eine Datei
`SetDirectory`	Setzt das aktuelle Verzeichnis
`Move`	Verschiebt eine Datei mit dem gesamten Inhalt: • `Move(quelle, ziel);`
`Delete`	Löscht eine Datei
`Exists`	Prüft, ob eine Datei existiert

Tabelle 5.6: Methoden der Klasse `File`

Im Abschnitt „Dateien erzeugen und schreiben“ auf Seite 310 finden Sie Informationen darüber, wie Dateien erzeugt, geschrieben und gelesen werden können.

Eine Instanz der Klasse `FileInfo` erlaubt den Zugriff auf Dateiinformationen:

```
FileInfo dinfo = new FileInfo("pfad/dateiname");
```

Es stehen hier ähnliche Methoden zur Verfügung wie bei `File`. Hinzu kommen Eigenschaften, die Auskunft über Attribute und Namen geben. Die wichtigsten finden Sie nachfolgend:

Eigenschaft	Beschreibung
`Attributes`	Attribute
`CreationTime`	Datum der Erzeugung
`FullName`	Pfad und Name
`Name`	Dateiname
`Parent`	Übergeordnetes Verzeichnis
`Length`	Größe der Datei in Byte
`Root`	Stammverzeichnis

Tabelle 5.7: Wichtige Eigenschaften der Klasse `FileInfo`

Die Eigenschaft `Attributes` basiert auf derselben Aufzählung wie die bei den Verzeichnissen beschriebene. Das letzte Beispiel verwendete diese bereits für Dateien.

Die Methoden erlauben ähnliche Operationen wie bei `File`. Eine Auswahl zeigt die nächste Tabelle:

Methode	Beschreibung
`Create`	Datei erzeugen
`CreateText`	`StreamWriter` erzeugen
`Delete`	Datei löschen
`MoveTo`	Datei verschieben: • `MoveTo(pfad_ziel)`
`Open`	Universell öffnen zum Lesen oder Schreiben

`OpenRead`
`OpenWrite`
`OpenText`
	Zugriff auf eine Datei zum Lesen, Schreiben oder als Textdatei im UTF-8–Format

`Refresh`	Status erneuern

Tabelle 5.8: Methoden der Klasse `FileInfo`

In Abschnitt „Dateien erzeugen und schreiben“ auf Seite 310 finden Sie Informationen darüber, wie Dateien erzeugt, geschrieben und gelesen werden können.

5.4.3 Ausnahmebehandlung

Dateizugriffsfehler abfangen

Operationen mit Verzeichnissen und Dateien scheitern oft an mangelnden Rechten, falschen Pfadangaben oder Störungen der Speichergeräte. In jedem dieser Fälle erzeugt die .Net-Laufzeitumgebung eine Ausnahme. Am Anfang wurde bereits gezeigt, wie Sie diese mit `try-catch` abfangen können:

```
try
{
  // Versuche Zugriff
}
catch (Ausnahmedefinition)
{
  // Behandle Ausnahme
}
catch (Andere_Ausnahmedefinition)
{
  // Behandle andere Ausnahme
}
finally
{
  // Aktionen, die immer ausgeführt werden sollen
}
```

Der `finally`-Teil ist optional. Hier platzieren Sie Code, der immer ausgeführt werden soll, egal ob eine Ausnahme auftrat oder nicht. Um nun eine spezifische Ausnahme abfangen und behandeln zu können, müssen Sie den oben als *Ausnahmedefinition* bezeichneten Teil entsprechend definieren:

```
catch (DirectoryNotFoundException e)
```

Diese Zeile führt dazu, dass der Fehler „Verzeichnis nicht gefunden" abgefangen wird. Das Fehler-Objekt vom Typ `DirectoryNotFoundException` wird in `e` gespeichert. Bevor Sie jedoch feinkörnige Fehlerbehandlungen schreiben, müssen Sie die im aktuellen Kontext möglichen Ausnahmen kennen:

Ausnahmetyp	Behandelte Ausnahme
`DirectoryNotFoundException`	Verzeichnis nicht gefunden
`FileNotFoundException`	Datei nicht gefunden
`EndOfStreamException`	Ende der Datei erreicht
`PathTooLongException`	Pfad zu lang (maximal sind 248 Zeichen erlaubt)

Tabelle 5.9: Im Zusamenhang mit Verzeichnis- und Dateioperationen auftretende Ausnahmen

Ausnahmetyp	Behandelte Ausnahme
`SecurityException` `UnauthorizedAccessException`	Mangelnde oder völlig fehlende Zugriffsrechte
`ArgumentException` `ArgumentNullException`	Ein Argument ist falsch oder `null`
`IOException`	Allgemeiner IO-Fehler. Tritt meist beim Löschen von Nur-Lese-Objekten auf

Tabelle 5.9: Im Zusamenhang mit Verzeichnis- und Dateioperationen auftretende Ausnahmen (Forts.)

Nicht in allen Fällen können alle Ausnahmen auftreten. Besonders wichtig sind: `DirectoryNotFoundException`, `FileNotFoundException` und `SecurityException`.

Die Klassen `SecurityException` und `UnauthorizedAccessException` stammen aus dem Namensraum `System.Security`. Sie müssen diesen entsprechend einbinden, wenn Sie derartige Ausnahmen verarbeiten möchten:

```
using System.Security;
```

5.4.4 Dateien erzeugen und schreiben

Der Zugriff auf Verzeichnisse und Dateien dient meist nur dem Zweck, lesend und schreibend auf Dateien zuzugreifen. Das folgende Beispiel zeigt, wie mit jedem Aufruf einer Seite Informationen über die Verbindung in eine Datei geschrieben werden – eine Protokolldatei entsteht. Zur Kontrolle wird die Datei gleich angezeigt. Zuerst die *aspx*-Datei zur Ausgabe:

```
<% @Page Language="C#" debug="true"  Inherits="log" src="log.cs"%>
<html>
    <head>
        <title>Protokolldatei</title>
    </head>
    <body>
        <h3>Ausgabe einer Protokolldatei</h3>
        <pre><asp:label runat="server" id="ausgabe"/></pre>
    </body>
</html>
```

Listing 5.15: Ausgabe von Protokollinformationen (file_log.aspx)

Die eigentliche Arbeit erledigt wieder eine *cs*-Datei:

```
using System;
using System.Web;
using System.Web.UI;
```

```
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using System.IO;
using System.Security;

public class log : Page
{
    public Label ausgabe;

    private void ShowFile(string logfile)
    {
        StreamReader test = File.OpenText(logfile);
        ausgabe.Text += test.ReadToEnd();
        test.Close();
    }

    void Page_Load()
    {
        string log = Server.MapPath(@".\logs\log.txt");
        string logeintrag = Request.ServerVariables["REMOTE_ADDR"] + " "
                          + DateTime.Now;
        if (File.Exists(log))
        {
            StreamWriter logs = File.AppendText(log);
            logs.WriteLine(logeintrag);
            logs.Close();
            ShowFile(log);
        } else {
            try
            {
                StreamWriter logs = File.CreateText(log);
                logs.WriteLine(logeintrag);
                logs.Close();
                ShowFile(log);
            }
            catch (DirectoryNotFoundException e)
            {
                ausgabe.Text  = "<span style=\"color:red\">
                                 Verzeichnis nicht vorhanden: ";
                ausgabe.Text += e.Message + "</span>";
            }
            catch (UnauthorizedAccessException e)
            {
                ausgabe.Text  = "<span style=\"color:red\">Kein Zugriff: ";
                ausgabe.Text += e.Message + "</span>";
            }
```

```
        }

    }

}
```

Listing 5.16: Programm zum Schreiben und Lesen einer Protokolldatei (log.cs)

Wie es funktioniert

Das Programm beginnt mit der Festlegung des Pfades. Die Protokolldatei soll innerhalb der Webapplikation gespeichert werden. Aus Sicht des Frameworks ist das Stammverzeichnis *%systemroot%\system32*. Das es sich um eine Webapplikation handelt, ist erstmal nicht bekannt. Schließlich werden dieselben Klassen auch in der Windows-Programmierung verwendet. In Kapitel 3 wurden die Klassen vorgestellt, die in ASP.NET den Zugriff auf den Webserver erlauben. Dazu gehört auch `Server`. Diese Klasse enthält eine Methode `MapPath`, die einen lokalen oder virtuellen Pfad einer Webapplikation in eine physische Pfadangabe umwandelt:

```
string log = Server.MapPath(@".\logs\log.txt");
```

Außerdem wird das Maskierungszeichen `\` mit dem `@` unterdrückt. Dann wird der Eintrag, der das Protokoll füllen soll, erzeugt. Verwendet wird hier die IP-Adresse des Browsers und das aktuelle Datum einschließlich der Zeit:

```
string logeintrag = Request.ServerVariables["REMOTE_ADDR"] + " "
                                                           + DateTime.Now;
```

Falls die Datei schon existiert, kann jetzt der Eintrag hinzugefügt werden:

```
if (File.Exists(log))
```

Dazu wird ein `StreamWriter`-Objekt erzeugt:

```
StreamWriter logs = File.AppendText(log);
```

In dieses wird der Protokolleintrag geschrieben:

```
logs.WriteLine(logeintrag);
```

Dann wird die Datei geschlossen und zur Kontrolle angezeigt. Die Vorstellung der Anzeigefunktion `ShowFile` erfolgt weiter unten.

Sie sollten Dateien immer explizit und so schnell wie möglich schließen. In Mehrbenutzerumgebungen führen offene Dateien zu Fehlern oder Leistungseinbußen. Vertrauen Sie nicht darauf, dass offene Dateien am Ende der Seite sofort automatisch geschlossen werden, auch wenn dies beispielsweise in einer Testumgebung der Fall sein dürfte.

Falls die Datei noch nicht existierte, wird der zweite Teil der `if`-Anweisung ausgeführt. Der einzige Unterschied besteht darin, dass statt `AppendText` zum Anhängen an die Datei mit `CreateText` eine neue Textdatei erzeugt wird:

```
StreamWriter logs = File.CreateText(log);
```

Zwei Ausnahmen werden außerdem verarbeitet, falls sie auftreten. Zuerst wird auf ein fehlendes Verzeichnis reagiert:

```
catch (DirectoryNotFoundException e)
```

Außerdem reichen die Zugriffsrechte möglicherweise nicht aus:

```
catch (UnauthorizedAccessException e)
```

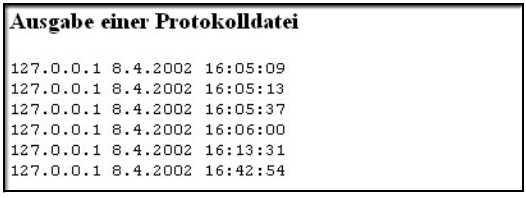


Abbildung 5.13: Darstellung der Protokolldatei nach ein paar Seitenabrufen vom lokalen Client

5.5 Bilder dynamisch erstellen

Bildfunktionen sind sehr beliebte und leistungsfähige Funktionen. Statt viele Bilder im Voraus mit aufwändigen Werkzeugen zu erzeugen, werden diese bei Bedarf in .NET generiert.

5.5.1 Prinzip der Bilderzeugung

Wenn man sich den Vorgang der Übertragung einer Webseite vor Augen hält, erkennt man, dass alle Elemente der Seite auf einzelnen Anforderungen beruhen. Erst wird die aufgerufene Stammseite übertragen, dann wertet der Browser alle enthaltenen Elemente aus und ruft diese einzeln ab. Solange es Bandbreite und Systemleistung zulassen, erfolgt dieser Abruf parallel.

Durch dieses Verfahren kann das Ziel eines Bildaufrufes natürlich von der Adresse der ursprünglichen Seite abweichen. Insbesondere ist die Angabe der Bildquelle primär nicht an eine Bilddatei gebunden, sondern es kann ebenso auch ein ASP.NET-Programm sein. In HTML sieht ein Aufruf dann folgendermaßen aus:

```
<img src="ladebilder.aspx"/>
```

Das Programm *ladebilder.aspx* ist dann dafür verantwortlich, dem Browser die erwarteten Daten zu übertragen. Eine einfache Form wäre der Zugriff auf bereits vorhandene Bilddaten und die Auswahl und Versendung per Programm. Wenn Sie dem Programm Parameter übergeben, kann damit schon eine gewisse Dynamik erzielt werden. Als Aufruf käme folgende Version in Frage:

```
<img src="ladebilder.aspx?bildnummer=3"/>
```

Wenn ein ASP.NET-Programm Bilder versendet, müssen Sie daran denken, dass der Browser zwei Dinge erwartet:

1. Die Angabe des korrekten Headers im Kopf der HTTP-Nachricht:
   ```
   Content-type: image/png
   ```
   Die Angabe des Bildtypes hängt vom erzeugten Format ab.
2. Die binären Bilddaten ohne jedes weitere Zeichen im Körper der HTTP-Nachricht

Am einfachsten lässt sich dies erreichen, indem das entsprechende Attribut der Seitendirektive verwendet wird:

```
<% @Page ContentType="image/png" %>
```

Außerdem müssen Sie sicherstellen, dass wirklich nur Bilddaten übertragen werden – HTML oder auch nur ein übriggebliebener Zeilenumbruch müssen unbedingt vermieden werden.

Dynamische Bildauswahl

Als erstes Beispiel soll eine Gruppe von Bildern dynamisch ausgewählt werden. Ideal ist dies für einen grafischen Hitzähler, der mit vorgefertigten Bilddateien arbeitet. Für jede Ziffer wird eine Bilddatei abgelegt; die Namen lauten *0.gif*, *1.gif* usw.

Der Zähler selbst wird hier nur simuliert – mit einem Zufallsgenerator:

```
<% @ Page language="C#" debug="true" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<script runat="server" language="C#">
void Page_Load()
{
    Random rnd = new Random();
    String zahl = rnd.Next(10000, 99999).ToString();
    d1.ImageUrl = "ladebild.aspx?nummer=" + zahl[0];
```

```
    d2.ImageUrl = "ladebild.aspx?nummer=" + zahl[1];
    d3.ImageUrl = "ladebild.aspx?nummer=" + zahl[2];
    d4.ImageUrl = "ladebild.aspx?nummer=" + zahl[3];
    d5.ImageUrl = "ladebild.aspx?nummer=" + zahl[4];
}
</script>
<html lang="de">
    <head>
        <title>Dynamische Bilder</title>
    </head>
    <body>
        <h1>Dynamische Bilder</h1>
        <div id="zaehler" runat="server">
        <asp:image id="d1" runat="server"/>
        <asp:image id="d2" runat="server"/>
        <asp:image id="d3" runat="server"/>
        <asp:image id="d4" runat="server"/>
        <asp:image id="d5" runat="server"/>
        </div>
    </body>
</html>
```

Listing 5.17: Anzeige eines grafischen Zählers (pix_select.aspx)

Wie es funktioniert

Die Bilder werden mit Web Server-Steuerelementen vom Typ `Image` angezeigt. Nach dem Laden der Seite wird zuerst eine Zufallszahl ermittelt, im Bereich von 10.000 bis 99.999. Damit später ziffernweise zugegriffen werden kann, wird die Zahl gleich in eine Zeichenkette umgewandelt:

```
String zahl = rnd.Next(10000, 99999).ToString();
```

Dann wird den fünf Steuerelementen jeweils einen URL zugewiesen. Als GET-Parameter `nummer` wird die passenden Ziffer übergeben:

```
d1.ImageUrl = "ladebild.aspx?nummer=" + zahl[0];
```

Die eigentliche Darstellung der Bilder erfolgt in *ladebild.aspx*. Im HTML-Code erfolgt der Aufruf folgendermaßen:

```
<img id="d1" src="/aspdotnet/ladebild.aspx?nummer=1" border="0" />
```

Die ganze Arbeit wird also in folgendem Programm erledigt:

```
<% @ Page language="C#" debug="true" ContentType="image/gif" %>
<% @ Import namespace="System.IO" %>
<script runat="server" language="C#">
void Page_Load()
{
    string bildname = Server.MapPath("img/" + Request.QueryString["nummer"]
                                     + ".gif");
```

```
    try
    {
        FileStream fs = new FileStream(bildname, FileMode.Open);
        int groesse = (int) fs.Length;
        byte[] puffer = new byte[groesse];
        fs.Read(puffer, 0, groesse);
        fs.Close();
        Response.ClearContent();
        Response.ContentType = "image/gif";
        Response.BinaryWrite(puffer);
        Response.End();
    }
    catch (Exception e)
    {
        Response.Write ("Fehler: " + e.Message);
    }
}
</script>
```

Listing 5.18: Ausgabe der Bilddaten (ladebild.aspx)

Wie es funktioniert

Das Programm beginnt mit der Festlegung des MIME-Types der Übertragung. Dazu wird die Seitendirektive `@Page` verwendet:

```
<% @Page language="C#" ContentType="image/gif" %>
```

Da die Bilddaten von der Festplatte gelesen werden müssen, muss der Namensraum `System.IO` eingebunden werden:

```
<% @Import namespace="System.IO" %>
```

Innerhalb der Methode `Page_Load` wird zuerst der Pfad zur passenden Bilddatei ermittelt:

```
string bildname = Server.MapPath("img/" + Request.QueryString["nummer"]
+ ".gif");
```

Die eigentliche Abarbeitung wird innerhalb eines `try`-Zweiges vorgenommen, damit Laufzeitfehler beim Dateizugriff nicht stören. Zum Dateizugriff wird ein `FileStream`-Objekt erzeugt:

```
FileStream fs = new FileStream(bildname, FileMode.Open);
```

Dann wird die Größe der Datei ermittelt:

```
int groesse = (int) fs.Length;
```

Nun wird für diese Datei ein Pufferspeicher erzeugt. Da Bilder Binärdaten sind, eignet sich ein Byte-Array dafür:

```
byte[] puffer = new byte[groesse];
```

Nun kann die Datei komplett in den Puffer gelesen werden:

```
fs.Read(puffer, 0, groesse);
```

Anschließend wird die Datei geschlossen:

```
fs.Close()
```

Jetzt können die Bilddaten an den Browser gesendet werden. Zuerst wird der eventuell bereits im Ausgabepuffer – der nichts mit dem Bildpuffer zu tun hat – befindliche Text gelöscht:

```
Response.ClearContent();
```

Dann werden die binären Daten gesendet:

```
Response.BinaryWrite(puffer);
```

Damit ist die Übertragung beendet, das Bild erscheint:

```
Response.End();
```

Der Rest kümmert sich um die Ausnahmebehandlung, wenn erforderlich.

Abbildung 5.14: Ausgabe von Zahlenwerten mit dynamischer Bildzuordnung

5.5.2 Bilder direkt erzeugen

Der erste Abschnitt zeigte das Prinzip der dynamischen Bilderzeugung, indem die Bildquelle ein ASP.NET-Programm ist. Spannender ist die Möglichkeit, gleich die ganze Grafik dynamisch zu erzeugen. Im .Net-Framework gibt es eine große Zahl von Klassen, die der Bilderzeugung dienen. Durch ein einheitliches Schnittstellenkonzept können die erzeugten Bilddaten nicht nur Windows-Applikationen bereitgestellt, sondern direkt an `Stream`-Objekte übertragen werden. Einen solchen Datenstrom kann man speichern oder wie im vorangegangenen Beispiel mit `Response.BinaryWrite` an den Browser senden. Alternativ kann der Ausgabefunktion als Stream-Objekt auch `Response.OutputStream` übergeben werden, eine Repräsentation des Ausgabestroms zum Browser.

Das folgende Beispiel basiert wieder auf einer HTML-Vorlage. Dem Programm, das die Grafik erzeugt, werden zwei Parameter übergeben: `titel` und `text`.

```
<html lang="de">
    <head>
        <title>Dynamische Bilder</title>
    </head>
    <body>
        <h1>Dynamische Bilder</h1>
        <asp:image id="d5" runat="server"
imageurl="ladebild2.aspx?titel=Bilder%20erstellen&text=Hallo, ich bin ein
          dynamisches Bild"/>
    </body>
</html>
```

Listing 5.19: Aufruf des Grafikprogramms mit GET-Parametern (pix_dynamic.aspx)

Angezeigt wird damit folgendes Banner:

Abbildung 5.15: Anzeige eines dynamisch erzeugten Bildes

Das eigentliche Geheimnis steckt im folgenden Programm:

```
<% @Page language="C#" debug="true" ContentType="image/jpeg" %>
<% @Import namespace="System.Drawing" %>
<% @Import namespace="System.Drawing.Imaging" %>
<script runat="server" language="C#">
void Page_Load()
{
    try
    {
        string text = Request.QueryString["text"];
        string titel = Request.QueryString["titel"];
        Bitmap ob = new Bitmap (460, 80);
        Graphics banner = Graphics.FromImage(ob);
        SolidBrush pinselGruen = new SolidBrush(Color.FromArgb(0,80,0));
        SolidBrush pinselSchwarz = new SolidBrush(Color.Black);
        SolidBrush pinselGelb = new SolidBrush(Color.Yellow);
        Font fontTitel = new Font ("Verdana", 20);
        Font fontText = new Font ("Verdana", 14);
        SizeF titelGroesse = banner.MeasureString(titel, fontTitel);
        SizeF textGroesse = banner.MeasureString(text, fontText);
        banner.FillRectangle (pinselGelb, 0, 0, 460, 80);
        banner.DrawString (titel, fontTitel, pinselSchwarz, 230 -
                           (titelGroesse.Width / 2), 8);
```

```
        banner.DrawString (text, fontText, pinselGruen, 230 -
                           (textGroesse.Width / 2), 40);
        Response.ClearContent();
        ob.Save(Response.OutputStream, ImageFormat.Jpeg);
        Response.End();
        banner.Dispose();
        ob.Dispose();
    }
    catch (Exception e)
    {
        Response.Write ("Fehler: " + e.Message);
    }
}
</script>
```

Listing 5.20: Programm zum Erzeugen eines Banners (ladebild2.aspx)

Wie es funktioniert

Das Programm ruft zuerst die GET-Parameter `text` und `titel` ab:

```
string text = Request.QueryString["text"];
string titel = Request.QueryString["titel"];
```

Dann wird zuerst ein `Bitmap`-Objekt erzeugt, dass die Basisgröße des Bildes festlegt:

```
Bitmap ob = new Bitmap (460, 80);
```

Auf der Grundlage dieser Zeichnungsfläche wird ein `Graphics`-Objekt erstellt, auf das nun Zeichenfunktionen angewendet werden können:

```
Graphics banner = Graphics.FromImage(ob);
```

Für die Ausgabe werden nun drei `SolidBrush`-Objekte erstellt, die zum Zeichnen durchgehender Flächen benötigt werden und einen Farbwert zugewiesen bekommen.

```
SolidBrush pinselGruen = new SolidBrush(Color.FromArgb(0,80,0));
SolidBrush pinselSchwarz = new SolidBrush(Color.Black);
SolidBrush pinselGelb = new SolidBrush(Color.Yellow);
```

Als Farbwerte können Farbnamen oder die Methode `FromArgb` für RGB-Farbwerte genutzt werden. `Color` ist eine Struktur, die vielfältige Farbunterstützung bietet, beispielsweise Konvertierungsmethoden. Nach den Farben müssen die Fonts ausgewählt werden. Der Titel soll die Schriftart Verdana in einer Höhe von 20 Punkt bekommen:

```
Font fontTitel = new Font ("Verdana", 20);
```

Der Text wird nur 14 Punkt groß:

```
Font fontText = new Font ("Verdana", 14);
```

Da die Texte mehr oder weniger lang sein können, wird nun die tatsächliche Größe in Abhängigkeit von der gewählten Schriftart ermittelt:

```
SizeF titelGroesse = banner.MeasureString(titel, fontTitel);
SizeF textGroesse = banner.MeasureString(text, fontText);
```

Nun wird der Hintergrund gezeichnet – ein gelbes Rechteck, das die gesamte Fläche ausfüllt:

```
banner.FillRectangle (pinselGelb, 0, 0, 460, 80);
```

Die Methode `DrawString` zeichnet nun den Text auf die Fläche; angegeben werden die Zeichenkette, der Font, die Farbe und die Platzierung der linken oberen Ecke in einem XY-Koordinatensystem:

```
banner.DrawString (titel, fontTitel, pinselSchwarz, 230 -
  (titelGroesse.Width / 2), 8);
banner.DrawString (text, fontText, pinselGruen, 230 -
  (textGroesse.Width / 2), 40);
```

Die Berechnung der X-Koordinate geht von der Hälfte des Banners aus (230 Pixel). Dann wird die Breite des Textes halbiert und abgezogen; damit steht der linke Rand fest. Die Höhe wird fest vorgegeben. Natürlich kann auch dieser Wert leicht in Abhängigkeit von der Schriftgröße berechnet werden, beispielsweise um den Text vertikal zu zentrieren.

Der Rest des Programms dient nur der Ausgabe. Zuerst wird der Ausgabepuffer geleert:

```
Response.ClearContent();
```

Dann wird das Bildobjekt mit der Methode `Save` direkt an den Ausgabedatenstrom übergeben. Die Methode erwartet außerdem die Angabe des Typs der Grafik. Im Beispiel wird JPEG genutzt:

```
ob.Save(Response.OutputStream, ImageFormat.Jpeg);
```

Dann wird die Ausgabe beendet, das Bild erscheint im Browser:

```
Response.End();
```

Zuletzt werden die beiden Grafikobjekt (`Bitmap` und `Graphics`) explizit zerstört. Das erledigt zwar die Garbage Collection irgendwann automatisch, aber Grafiken sind außerordentlich speicherhungrig und die schnelle Entfernung aus dem Speicher entlastet das System deutlich. Ohne diese Maßnahmen wären Anwendungen mit dynamisch erstellten Bildern kaum unter Last zu betreiben:

```
banner.Dispose();
ob.Dispose();
```

Der übrige Teil des Programms dient nur der Fehlerbehandlung.

5.5.3 Hinweise zur Fehlersuche

Problematisch sind die Grafikfunktionen, weil die Ausgabe normalerweise nicht zu einer Fehlermeldung führt – das gilt auch für Compilerfehler. Der Browser erwartet durch den Header immer ein Bild. Er wird also im Zweifelsfall sein Ersatzbild anzeigen, statt der Fehlerausgabe aus dem `catch`-Zweig.

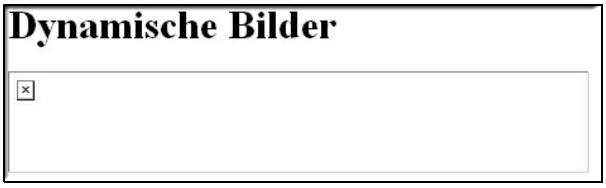


Abbildung 5.16: Offensichtlich funktioniert das Grafikprogramm nicht – dennoch erscheint kein Fehler

In diesem Fall sollten Sie das Grafikprogramm direkt aufrufen, gegebenenfalls mit den notwendigen Parametern. Der Browser kann in jedem Fall ein Bild allein anzeigen, er benötigt das `<img>`-Tag nicht. Handelt es sich um Compilerfehler, werden diese in der üblichen Form angezeigt. Andernfalls wird der Text der Fehlermeldung ausgegeben, wenn Sie die `catch`-Anweisung entsprechend aufbauen.

Tipps

Falls die Gefahr von Laufzeitfehlern sehr groß ist, sollten Sie die Ausgabestrategie etwas ändern. Statt den MIME-Typ global festzulegen, können Sie diesen auch direkt als Eigenschaft des `Response`-Objekts angeben:

```
Response.ClearContent();
Response.ContentType = "image/gif";
ob.Save(Response.OutputStream, ImageFormat.Jpeg);
Response.End();
```

Die Ausgabe im `catch`-Zweig sieht dann die Generierung von HTML vor:

```
catch (Exception e)
{
   Response.ClearContent();
   Response.ContentType = "text/html";
   Response.Write ("Fehler: " + e.Message);
   Response.End();
}
```

Die Fehlermeldung erscheint, wenn der Browser HTML erkennt, in der Standardschrift Times. Wird dagegen Text erkannt, wird meist Courier verwendet. Sie können so prüfen, welcher *Content-type*-Header gesendet worden ist.

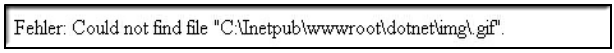


Abbildung 5.17: Fehlermeldung beim Aufruf eines Grafikprogramms

5.6 E-Mail senden

Kontaktformulare können Sie mit dem bisherigen Wissen bereits erstellen. Ebenso ist die Speicherung als Textdatei möglich. Eleganter wäre es natürlich, wenn der Server die Informationen gleich per E-Mail an Sie senden könnte – natürlich ohne das E-Mail-Programm des Benutzers zu verwenden. Die Benutzung des clientseitigen Programms ist eher unglücklich, weil viele Programme drastische Warnungen beim Aufruf aus einer Webseite heraus aussprechen und die Existenz eines solchen Clients nicht sicher festgestellt werden kann. Lassen Sie den E-Mail-Versand serverseitig ablaufen, kann der Benutzer dies nicht beeinflussen und es stört in keiner Weise seine Privatsphäre.

5.6.1 Vorbereitung

Für den Versand von E-Mail sind einige vorbereitende Schritte notwendig. Zuerst müssen Sie wissen, dass der E-Mail-Verkehr im Internet über so genannte SMTP-Server (SMTP = Simple Mail Transfer Protocol) abgewickelt wird. Wenn ASP.NET eine E-Mail versenden soll, muss ein solcher Server zur Verfügung stehen. Wenn Sie derartige Programme auf einem lokalen Entwicklungssystem testen, müssen Sie auch daran denken, dass eine Verbindung zum Internet bestehen muss. SMTP basiert auf einer ständigen Erreichbarkeit des jeweiligen Servers. Im Gegensatz dazu arbeitet das Protokoll zum Abruf von E-Mail durch einen Client – POP3 – nur auf Anforderung. Dies wird oft verwechselt. Sie befinden sich nun auf der „Serverseite“ der E-Mail und müssen sich mit den Gepflogenheiten bei SMTP auseinandersetzen.

Zum Glück macht es ASP.NET recht einfach, die nötigen Einstellungen vorzunehmen. Die Angabe des SMTP-Servers erfolgt einfach durch Setzen einer Eigenschaft des entsprechenden Objekts.

5.6.2 Eine E-Mail-Anwendung programmieren

Das .Net-Framework stellt E-Mail-Funktionen für Webserver im Namensraum `System.Web.Mail` zur Verfügung. Die Bekanntgabe erfolgt entweder in der HTML-Seite mit folgender Direktive:

```
<% @Import namespace="System.Web.Mail" %>
```

Alternativ ergänzen sie eine Datei mit hinterlegtem Code folgendermaßen:

```
using System.Web.Mail;
```

Basis der E-Mail-Anwendung: Ein Kontaktformular

Damit es überhaupt etwas zu versenden gibt, beginnt die Entwicklung der E-Mail-Anwendung mit einem Kontaktformular. Dieses ist bewusst einfach gehalten. Mit den Informationen aus Kapitel 4 können Sie es leicht an Ihre Bedürfnisse anpassen und erweitern.

Der Code zum Versenden des Formularinhalts als E-Mail wird in einer Code-Datei hinterlegt (Code Behind).

```
<% @ Page language="C#" debug="true" Inherits="Mail" src="email_class.cs" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="de">
    <head>
        <title>Kontaktformular</title>
    </head>
    <body>
        <h1>Kontaktformular</h1>
        <form runat="server">
        <table>
            <tr>
                <td>Name</td>
                <td>
                    <asp:textbox runat="server" id="name"
                                 width="250px"/>
                    <asp:requiredfieldvalidator runat="server"
                         id="checkname"
                         controltovalidate="name" display="dynamic">
                         <div style="color:red">
                         Geben Sie einen Namen ein</div>
                    </asp:requiredfieldvalidator>
                </td>
            </tr>
            <tr>
                <td>Anschrift</td>
                <td>
                   <asp:textbox runat="server" id="anschrift" width="250px"/>
                </td>
            </tr>
            <tr>
                <td>Plz / Ort</td>
```

```
                <td>
                    <asp:textbox runat="server" id="plz" width="50px"/>
                    <asp:textbox runat="server" id="ort" width="196px"/>
                </td>
            </tr>
            <tr>
                <td>Telefon</td>
                <td>
                    <asp:textbox runat="server" id="telefon"/>
                    mit Vorwahl
                </td>
            </tr>
            <tr>
                <td>E-Mail</td>
                <td>
                    <asp:textbox runat="server" id="email" width="250px"/>
                </td>
            </tr>
            <tr>
                <td>Anschrift</td>
                <td>
                    <asp:textbox textmode="multiline" runat="server"
                                 id="nachricht" width="250px" height="100px"/>
                </td>
            </tr>
            <tr>
                <td colspan="2" align="right">
                    <asp:button runat="server" id="send" text="Senden"
                                        onclick="SendMail"/>
                </td>
            </tr>
        </table>
        <asp:label runat="server" id="ergebnis"/>
        </form>
    </body>
</html>
```

Listing 5.21: Kontaktformular, das zum Versenden per E-Mail verwendet wird (email_form.aspx)

Das Formular verfügt nur über eine einfache Prüfung des Namens. Die gesamte Arbeit erledigt die Klasse `Mail`.

E-Mail versenden

Das Versenden ist weniger aufwändig als das Erstellen des Formulars:

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.Mail;

public class Mail : Page
{
    public TextBox name, anschrift, plz, ort, telefon, email, nachricht;
    public Label ergebnis;
    private MailMessage eMail;
    private string eMailText;
    const string CR = "\n";

    public void SendMail (object sender, EventArgs ea)
    {
        try
        {
            eMail = new MailMessage();
            eMailText  = "Kontaktformular:\n";
            eMailText += "Name     : " + name.Text + CR;
            eMailText += "Anschrift: " + anschrift.Text + CR;
            eMailText += "Plz/Ort  : " + plz.Text + " " + ort.Text + CR;
            eMailText += "E-Mail   : " + email.Text + CR;
            eMailText += "Telefon  : " + telefon.Text + CR;
            eMailText += "Nachricht: " + CR;
            eMailText += nachricht.Text + CR + CR;
            eMail.From = email.Text;
            SmtpMail.SmtpServer = "smtp.kundenserver.de";
            eMail.To = "webmaster@comzept.de";
            eMail.Subject = "Kontaktformular ASP.NET";
            eMail.Body = eMailText;
            SmtpMail.Send(eMail);
            eMail = null;
            ergebnis.Text = "Nachricht erfolgreich gesendet.";
        }
        catch (Exception e)
        {
            ergebnis.Text = "FEHLER: " + e.Message;
        }
    }
}
```

Listing 5.22: Programm zum Versenden von Formulardaten per E-Mail (email_class.cs)

Wie es funktioniert

Das Programm versendet eine E-Mail in Textform. .Net unterstützt auch HTML-Mails und Anhänge, dazu folgen am Ende noch Hinweise. Die

gesamte Arbeit erledigt die Methode `SendMail`, die das `onClick`-Ereignis der Sendeschaltfläche verarbeitet. Auf `Page_Load` kann damit verzichtet werden:

```
public void SendMail (object sender, EventArgs ea)
```

Die Methode startet mit einem `try`-Zweig. Das Versenden von E-Mail kann an vielen Problemen scheitern; beispielsweise kann der SMTP-Server nicht erreichbar sein, die Nachricht ablehnen, Netzwerkfehler können stören usw. Alle Fehler werden mit einem universellen `catch` abgefangen.

Zuerst wird nun die eigentliche Nachricht zusammengebaut. Basis ist ein Objekt der Klasse `MailMessage`:

```
eMail = new MailMessage();
```

Jetzt wird der Text der Nachricht erstellt:

```
eMailText = "Kontaktformular:" + CR;
```

Als Zeilenendezeichen wird die Konstante `CR` verwendet, die das Escape-Zeichen `\n` enthält. In Text-E-Mails dürfen Sie natürlich nicht `<br/>` verwenden, dies ist HTML vorbehalten. Ohne Zeilenumbruch würde aber der gesamte Text in einer Zeile stehen. Die übrigen Zuweisungen lesen die Formularfelder aus.

Dann sind Absender und Empfänger zu bestimmen. Zuerst der Absender, dieser Wert wird dem Formular entnommen – in der Hoffnung, der Benutzer gibt seine E-Mail-Adresse vernünftig an:

```
eMail.From = email.Text;
```

Dann wird der Empfänger bestimmt. Tragen Sie hier die Adresse ein, an die das Formular gesendet werden soll:

```
eMail.To = "webmaster@domain.de";
```

Der Betreff kann auch festgelegt werden. Es ist sinnvoll, dies nicht dem Benutzer zu überlassen, damit Sie die E-Mails vom Kontaktformular leicht erkennen können:

```
eMail.Subject = "Kontaktformular ASP.NET";
```

Zuletzt wird noch der Text an das Objekt übergeben:

```
eMail.Body = eMailText;
```

Damit der Versand auch funktioniert, ist nun der SMTP-Server zu bestimmen. Ist – unter Windows 2000 Pro, Server oder XP Pro – der SMTP-

Server korrekt installiert, verwenden Sie „localhost“. Andernfalls tragen Sie den STMP-Server Ihres Providers ein:

```
SmtpMail.SmtpServer = "smtp.kundenserver.de";
```

Sollte der SMTP-Server des Providers mit einer Fehlermeldung der Art „Relaying prohibited“ reagieren, wurden Sie nicht als autorisierter Benutzer erkannt. Meist kann dies umgangen werden, indem unmittelbar vorher eine autorisierte Abfrage des POP3-Servers erfolgt, also Post abgeholt wird. Auf der Website zum Buch finden Sie ein ASP.NET-Programm, dass dies erledigt.

Nachdem nun alles vorbereitet wurde, kann der E-Mail-Versand erfolgen:

```
SmtpMail.Send(eMail);
```

Zuletzt wird das `Mail`-Objekt vernichtet und das Programm endet mit einer Erfolgsmeldung.

Abbildung 5.18: Viel Formular und wenig E-Mail – der Versand geht eher still vor sich

Wenn Sie die ersten Versuche an sich selbst senden, sollten Sie im Postfach bald die entsprechenden Nachrichten vorfinden.

Fehlerquellen

Abschließend soll noch auf einen speziellen Fehler hingewiesen werden, der häufig auftritt:

```
Could not access 'CDO.Message' object
```

Schalten Sie das ausführliche Debugging an und fangen den Laufzeitfehler nicht mit `catch` ab, erhalten Sie einen genaueren Hinweis der folgenden Art:

```
Es ist mindestens eines der Felder "Von" oder "Absender" erforderlich, es
wurde jedoch keines der Felder gefunden.
```

Meist wurde vergessen, die Eigenschaft `From` zu füllen. Bauen Sie hier einen `RequiredFieldValidator` ein, um den Fehler zu vermeiden.

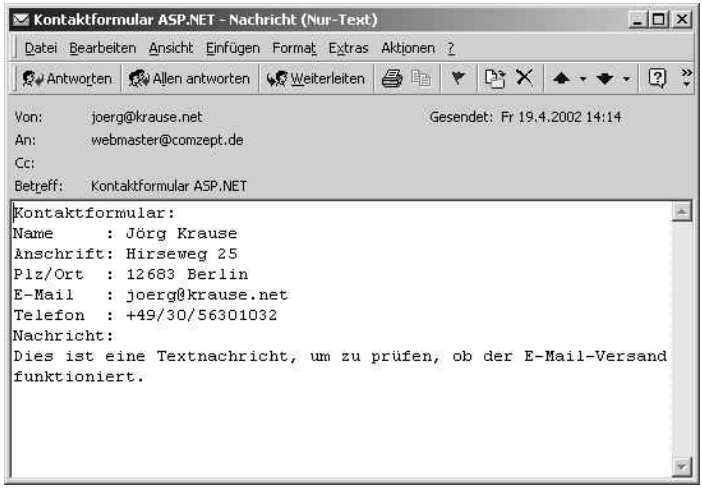


Abbildung 5.19: Nachricht in Outlook 2000, die mit dem Kontaktformular erzeugt wurde

5.6.3 Spezielle E-Mail-Funktionen

Massenpost

Das Versenden von E-Mail in der vorgestellten Form kann leicht erweitert werden. Wollen Sie Massenpost versenden, muss nur der Inhalt der Eigenschaft `To` ausgetauscht werden. Natürlich muss es nicht gleich Massenpost sein. Denken Sie beispielsweise daran, das Kontaktformular mit einer pauschalen Reaktion auch gleich an den Benutzer selbst zu senden. Er weis dann sicher, dass der Versand funktioniert hat. Dies ist gute Benutzerführung und kundenfreundlich.

Beachten Sie, dass das Versenden von Massenpost an Personen, mit denen Sie nicht in einer laufenden Geschäftsbeziehung stehen, strafbar ist. Verwenden Sie Rundschreiben nur, wenn der Benutzer sich vorher damit einverstanden erklärt hat, beispielsweise durch Beitritt zu einer Mailingliste oder einem Newsletter.

Dateianhänge

Eine weitere Funktion betrifft Dateianhänge. Dazu wird die Klasse `MailAttachment` eingesetzt. Es ist sehr einfach, die Anforderung einer Datei per E-Mail mit ASP.NET zu programmieren:

```
MailAttachment mailDatei = new MailAttachment("pfad/datei.pdf");
eMail.Attachments.Add(mailDatei);
```

Die Eigenschaft `Attachments` bildet eine Kollektion; es können deshalb leicht mehrere Dateianhänge hinzugefügt werden.

HTML-Mails

Die letzte Sonderfunktion sind HTML-Mails. Dazu ist nur eine einzige Zeile Code hinzuzufügen:

```
eMail.BodyFormat = Mailformat.Html;
```

`Mailformat` ist eine Aufzählung, die als Aufzählungswerte `Text` oder `Html` enthält. Jetzt müssen Sie natürlich den Text der E-Mail entsprechend aufbereiten:

```
<html>
<body>
<h1>Hallo, willkommen bei unserem HTML-Newsletter</h1>
...
```

5.7 Reguläre Ausdrücke

Reguläre Ausdrücke begegnen dem Programmierer immer wieder. Anfänger empfinden die kryptischen Schlangen voller Sonderzeichen als unlesbar und auch mancher Profi schreibt lieber schnell ein paar Schleifen, als sich Gedanken über ein solches Muster zu machen. Dabei sparen reguläre Ausdrücke viele Zeilen Code und lösen manches Problem auf verblüffend elegante Weise. Dieser Abschnitt führt Sie in die Welt der regulären Ausdrücke ein und zeigt, wie Sie sich ein unglaublich spannendes Kapitel der Programmierkunst erschließen.

5.7.1 Überall reguläre Ausdrücke

Das .Net-Framework verfügt auch über Klassen, die mit regulären Ausdrücken umgehen. Sie sind im Namensraum `System.Text.RegularExpressions` zusammengefasst.

Was aber sind diese Gebilde eigentlich? Reguläre Ausdrücke (aus dem Englischen: regular expressions) beschreiben Suchmuster. Man kann damit nach Zeichen in einer Zeichenkette oder einem Text suchen – nicht mehr, aber auch nicht weniger. Das geht freilich mit einfachen Suchfunktionen auch. Oft fehlt es aber an einer klaren Definition, was eigentlich gesucht werden soll. Wenn Sie in einem längeren Text nach Stellen suchen, die Währungsangaben beinhalten, dann nutzt eine normale Suchfunktion nicht. Viele Suchfunktionen in Editoren, Textverarbeitungen und anderen Programmen, die Textstellen suchen (wie beispielsweise Visual Studio.Net), nutzen deshalb auch reguläre Ausdrücke. Diese erlauben auch die Suche nach allgemeineren Angaben. Die Möglichkeiten sind dabei aber so umfangreich, dass es sich schon wieder

lohnt, ganze Bücher darüber zu schreiben. Die Vielfalt der Funktionen und die Mächtigkeit der Definitionen führt dann auch zu diesen unlesbaren, verwirrenden Konstrukten.

Das Programm in einer Zeile

Wie der Ausdruck intern arbeitet

In einem regulären Ausdruck steckt, wenn man es genau betrachtet, ein komplettes Programm. Wenn man dies akzeptiert und versteht, dann ist der Entwurf und die Nutzung nicht schwieriger als mit jedem anderen Stück Code auch. Sie müssen sich die Zeit nehmen, einen regulären Ausdruck sorgfältig zu entwerfen und zu testen. Niemand – auch Profis nicht – schreiben einen neuen Ausdruck, der vielleicht nur 10 oder 20 Zeichen umfasst, in ein paar Sekunden auf. Intern arbeitet auch nicht ein einfacher Mustervergleich, sondern die Regex-Maschine; ein Programm, das die Ausdrücke auflöst und ausführt.

Ein paar kleine Beispiele

Konkrete Beispiele sind der beste Weg, reguläre Ausdrücke zu verstehen.

HTML-Tags suchen

Das folgende Muster sucht nach dem HTML-Tag `<h1>`, `<h2>` usw: `<h\d>`

Das ist nun wenig spektakulär, weil diese Aufgabe auch eine normale Suchfunktion erfüllen würde. So ein Suchmuster wird nun von links nach rechts abgearbeitet und „gleitet“ dabei über den Text, der durchsucht werden soll. Wird eine Übereinstimmung gefunden, entsteht ein so genannter Match – ein Treffer. Im Beispiel stehen alle Zeichen für sich selbst, außer `\d` – dies ist ein Steuerzeichen, dass für ein Zahlzeichen steht. Dieser Ausdruck erkennt allerdings auch `<h9>`, was in HTML nicht vorkommt. Reguläre Ausdrücke „denken“ Zeichenweise. Jedes Zeichen kann aber unterschiedlich komplex definiert werden. Was hier benötigt wird, ist eine Zeichenklasse, die nur die Ziffern 1 bis 6 umfasst. Zeichenklassen werden in eckige Klammern gesetzt: `<h[123456]>`. Die gesamte Klasse steht dennoch nur für ein Zeichen. Eleganter ist der folgende Ansatz: `<h[1-6]>`. Innerhalb der Zeichenklasse können also Bereiche definiert werden. Mit Hilfe des `^`-Symbols kann der Inhalt der Zeichenklasse ausgeschlossen werden. Eine Alternative zur letzten Definition sähe dann so aus: `<h[^0789a-z]>`. Es gibt bei regulären Ausdrücken oft viele Wege zum Ziel.

Häufig werden reguläre Ausdrücke eingesetzt, um Eingaben aus Formularen zu überprüfen. Die Prüfung einer E-Mail-Adresse ist dabei schon eine kleine Herausforderung. Mit einem regulären Ausdruck können Sie zumindest die grundsätzliche Form überprüfen.

E-Mail-Adresse

Ein E-Mail-Name wird aus kleinen und großen Buchstaben aufgebaut. Als zusätzliche Zeichen sind das Minuszeichen „-“, der Unterstrich „_“

und der Punkt „.“ erlaubt. Groß- und Kleinschreibung spielt keine Rolle. Nach dem Namen folgt das @-Zeichen und die Domain. Hier dürfen keine Unterstriche auftreten, ansonsten entspricht es dem E-Mail-Namen. Es ist sinnvoll, den Ausdruck Schritt für Schritt zu entwickeln, wie es im Folgenden gezeigt wird:

- *^[_a-z0-9-]+* – Dieser Teil untersucht den Mailnamen auf gültige Zeichen. Erlaubt sind neben Ziffern und Buchstaben noch Minuszeichen und Unterstriche. Außerdem soll der Ausdruck von Beginn an untersucht werden, was mit dem Symbol ^ erreicht wird.
- *^[_a-z0-9-]+ (\.[_a-z0-9-]+)** – Dieser Ausdruck akzeptiert auch den Punkt, allerdings nicht am Anfang oder am Ende des Namens. Die Gruppierung erlaubt die Wiederholung des zweiten Teils, er kann aber auch entfallen. Damit sind Konstruktionen wie „joerg.krause“ oder „joerg“ oder „joerg_krause.csharp“ erlaubt.

  Damit ist der Teil vor den @-Zeichen abgeschlossen.
- *([a-z0-9-]+\.)[a-z]{2,4}$* – Hier wird nur der Domainname untersucht. Unterstriche sind nicht erlaubt, der Punkt ist zwingend und wird von 2 bis 4 Zeichen gefolgt, die wiederum nur Buchstaben sein können. Hier stellt die Gruppierung sicher, dass der Domainname aus mehreren Teilen bestehen darf. Damit sind Konstruktion wie „domain.de“, aber auch „mail.rz.uni-magdeburg.de“ erlaubt. Das abschließende $-Zeichen erzwingt, dass nach dem Domainnamen nichts mehr folgen darf – die zu untersuchende Zeichenkette muss hier enden.

Das Gebilde ist nun schon recht unübersichtlich. Fertig ist es in Kombination aus dem vorderen und dem hinteren Teil, verbunden durch das @-Zeichen:

```
^[_a-zA-Z0-9-]+(\.[_a-zA-Z0-9-]+)*@([a-zA-Z0-9-]+\.)[a-zA-Z]{2,4}$
```

Der Ausdruck ist keineswegs perfekt. Es gibt immer noch Fälle, die unzulässig sind und nicht erkannt werden. Aber er ist schnell und fängt die meisten Tippfehler oder bewusst unsinnige Angaben ab.

Programmierung regulärer Ausdrücke im Framework

Bevor eine tiefgehendere Betrachtung der regulären Ausdrücke folgt, soll eine praktische Nutzung im .Net-Framework und mit C# gezeigt werden. Es gibt zwei Möglichkeiten, die Ausdrücke zu nutzen. Zum einen stehen die Klassen des Namensraumes `System.Text.RegularExpressions` bereit. Zum anderen gibt es ein Kontroll-Steuerelement mit dem Namen `RegularExpressionValidator`, das die Feldinhalte aus Formularen direkt prüft. Reguläre Ausdrücke erlauben nicht nur eine einfache Aus-

sage der Form „passt“ oder „passt nicht“. Innerhalb des Ausdrucks können Klammern zur Bildung von Gruppen genutzt werden. Die Untersuchung der E-Mail-Adresse liefert nicht nur eine Gültigkeitsprüfung, sondern auch gleich die Bestandteile Name, Domain und Toplevel. Dies ist mit dem Kontroll-Steuerelement nicht möglich.

```
<script runat="server" language="C#">
void CheckEmail(object sender, EventArgs e)
{
    Regex rx;
    Match ma;
    String rxmail = @"^([_a-zA-Z0-9-]+(\.[_a-zA-Z0-9-]+)*)(@)
                    ((?>[a-zA-Z0-9-]+\.)+)([a-zA-Z]{2,4})$";
    rx = new Regex(rxmail, RegexOptions.IgnoreCase);
    ma = rx.Match(email.Text);
    if (ma.Success)
    {
        ausgabe.InnerHtml = "<b>Die Adresse ist gültig</b><p/>";
        for (int i = 0; i < ma.Groups.Count; i++)
        {
          ausgabe.InnerHtml += "Treffer: <u>" + ma.Groups[i];
          ausgabe.InnerHtml += "</u> an Position " + ma.Groups[i].Index;
          ausgabe.InnerHtml += " (" + ma.Groups[i].Length + " Zeichen)<br/>";
        }
    } else {
        ausgabe.InnerHtml = "<b>Die Adresse ist ungültig</b>";
    }
}
</script>
<html>
    <head>
        <title>Reguläre Ausdrücke</title>
    </head>
    <body>
        <h1>Reguläre Ausdrücke</h1>
        <form runat="server">
            <asp:textbox runat="server" id="email"/>
            <asp:button runat="server" id="send" text="Prüfen"
                                onclick="CheckEmail"/>
        </form>
        <p id="ausgabe" runat="server"/>
    </body>
</html>
```

Listing 5.23: Praktische Nutzung regulärer Ausdrücke (regex1.aspx)

Der Ausdruck in der Variablen `rxmail` ist hier gegenüber der vorgestellten Version noch um diverse Klammern erweitert worden. Runde Klammern dienen der Gruppierung, gehen also in die Analyse des Suchmusters selbst nicht ein, sondern steuern die Rückgabe von Teilen des Ausdrucks.

Beachten Sie, dass reguläre Ausdrücke viele Sonderzeichen verwenden, die selbst mit Backslashes markiert sind. Diese dürfen von C# nicht aufgelöst werden, deshalb sollten Zeichenketten mit dem @-Zeichen am Anfang versehen werden, das die Analyse unterdrückt.

Wie es funktioniert

Die Analyse basiert auf einer Instanz der `Regex`-Klasse:

```
rx = new Regex(rxmail, RegexOptions.IgnoreCase);
```

Regex

Die Aufzählung `RegexOptions` kann mehrere Steuerungen des Ausdrucks vornehmen, `IgnoreCase` unterbindet die Unterscheidung von Groß- und Kleinschreibung. `rx` ist nun ein `Regex`-Objekt, dass mit dem Ausdruck verbunden ist. „Gegen" diesen Ausdruck können nun Zeichenketten geprüft werden. Dazu dient die Methode `Match`, die ein Objekt der Klasse `Match` zurückgibt:

```
ma = rx.Match(email.Text);
```

Match

In `ma` ist nun das `Match`-Objekt enthalten, der Ausdruck selbst wurde bereits abgearbeitet. `Match` ist sowohl eine Klasse, als auch eine Kollektion. Wenn das Suchmuster im durchsuchten Text mehrmals gefunden wird, enthält `Match` mehr als ein Element. Wenn die Suche allgemein erfolgreich war – also ein oder mehr Treffer vorhanden waren, ist die Eigenschaft `Success` gleich `true`. In diesem Fall wird der Ausdruck weiter untersucht:

```
if (ma.Success)
```

Groups

Jedes Element der Kollektion `Match` enthält eine Eigenschaft `Groups`.

Diese Eigenschaft gibt ein Objekt der Klasse `Group` zurück. Diese Klasse erlaubt den Zugriff auf die durch runde Klammern gebildeten Gruppen. Eine wichtige Eigenschaft ist `Count` – die Anzahl der Gruppen.

```
for (int i = 0; i < ma.Groups.Count; i++)
```

Die Gruppen selbst bilden eine Kollektion, die in C# zugleich ein Indexer sind. Der Zugriff erfolgt deshalb wieder in der üblichen Arrayschreibweise:

```
ausgabe.InnerHtml += "Treffer: "<u> + ma.Groups[i];
```

Jede Gruppe hat wiederum Eigenschaften. `Index` enthält die Position in der durchsuchten Zeichenkette, mit Null beginnend gezählt:

```
ausgabe.InnerHtml += "</u> an Position " + ma.Groups[i].Index;
```

Die Eigenschaft `Length` enthält die Anzahl Zeichen, die in der Gruppe gefunden wurden:

```
ausgabe.InnerHtml += " (" + ma.Groups[i].Length + " Zeichen)<br/>";
```

Alle Zugriffe auf die Ergebnisse regulärerer Ausdrücke funktionieren auf Basis dieser Klassen.

Es gibt noch eine weitere Klasse, `Capture`, die ebenfalls Kollektionen bilden kann. Diese Klasse kann von `Group`-Elementen abgeleitet werden. Beim Aufbau von Ausdrücken mit Gruppen gibt es zwei Vorgehensweisen: ([Zeichenklasse])+ und ([Zeichenklasse]+). Im ersten Fall bildet das Zeichen eine Gruppe, die sich wiederholen darf. Im zweiten Fall werden die Zeichen wiederholt, und dann eine Gruppe gebildet. Beides erkennt dasselbe Suchmuster. Im zweiten Falle wird jedoch der Inhalt der Gruppe aus Fundstellen gebildet. Wenn Sie auf die Fundstellen zugreifen möchten, hilft der Zugriff auf die `Capture`-Kollektion.

Dies soll hier nicht weiter erörtert werden, da es nur selten benötigt wird und ein tiefgehenderes Verständnis der regulären Ausdrücke verlangt.

5.7.2 Grundlagen regulärer Ausdrücke

In der Praxis kommt es weniger auf die Programmierung, sondern auf die Entwicklung der benötigten Ausdrücke an. Nachfolgend werden reguläre Ausdrücke genauer vorgestellt. Zuerst die Zeichen zum Eingrenzen der Position des Suchmusters in der zu durchsuchenden Zeichenfolge:

Bedingung	Beschreibung
^	Beginn des Musters
$	Ende des Musters
\A	Unbedingter Beginn auch bei mehrzeiligen Texten
\Z	Unbedingtes Ende auch bei mehrzeiligen Texten oder vor dem Zeilenumbruchzeichen \n
\z	Unbedingtes Ende auch bei mehrzeiligen Texten

Tabelle 5.10: Positionierungsbedingungen

Es gibt Zeichen, die etwas anderes als sich selbst repräsentieren:

Zeichen	Bedeutung
\uNNNN	Unicode-Zeichen mit dem Code NNNN (Hexadezimale Angabe). ASCII-Zeichen nutzen das hintere Byte, ein Leerzeichen wird also als \u0020 geschrieben
\n	Zeilenumbruch

Tabelle 5.11: Sonderzeichen in Zeichenklassen

Zeichen	Bedeutung
\r	Wagenrücklauf
\0XXX	Oktale Angabe eines Zeichencodes mit ein bis drei Zahlen von 0 bis 7.
\xHH	Hexadezimale Angabe eines Zeichens
\\	Der Backslash selbst

Tabelle 5.11: Sonderzeichen in Zeichenklassen (Forts.)

Der Bildung von Zeichenklassen liegen fast alle regulären Ausdrücke zugrunde. Die Idee dahinter ist eine Beschreibung eines bestimmten Zeichens im Suchmuster. Wenn an einer Stelle eine Zahl zwischen 0 und 4 auftreten darf, schreibt man eine Zeichenklasse: `[0-4]` oder `[01234]`. In jedem Fall repräsentiert diese Klasse nur ein Zeichen.

Zeichen	Bedeutung
.	Der Punkt repräsentiert jedes beliebige Zeichen
[abcd]	Eine Klasse wird durch eckige Klammern gebildet. Darin stehen die Zeichen, die die Klasse bilden oder die Sonderzeichen aus Tabelle 5.11.
[^abcd]	Eine Klasse, in der die aufgeführten Zeichen nicht enthalten sind.
[a-z]	Mit Minuszeichen werden Bereiche von Zeichen definiert.
\p{name}	Vordefinierte Zeichenklasse *name*.
\P{name}	Alles außer dem Inhalt der vordefinierten Zeichenklasse *name*.
\w, \W	Wortzeichen bzw. kein Wortzeichen
\s, \S	Whitespace bzw. kein Whitespace[a]
\d, \D	Zahlzeichen bzw. keine Zahlzeichen
\b, \B	Wortgrenze bzw. keine Wortgrenze

Tabelle 5.12: Zeichenklassen

a. Als Whitespace gelten Leerzeichen, Zeilenumbrüche, Tabulatoren usw.

Wenn Sie mehr als nur ein Zeichen benötigen, müssen Sie einen der Wiederholungsoperatoren angeben.

Zeichen	Bedeutung
*	Kein oder beliebig viele Zeichen
+	Ein oder beliebig viele Zeichen
?	Kein oder genau ein Zeichen
{n}	Genau *n* Zeichen
{n,}	Mindestens *n* Zeichen
{n.m}	Mindestens *m* jedoch höchstens *n* Zeichen

Tabelle 5.13: Wiederholungsoperatoren

Zeichen	Bedeutung
*?	Das nachgestellte Fragezeichen macht den Ausdruck in diesem Bereich „ungierig“. Normalerweise sind reguläre Ausdrücke „gierig“, führen also die Prüfung solange fort, wie es geht, auch wenn die Erfüllung bereits möglich war. Dieses Verhalten kann mit dieser Notation unterdrückt werden.
+?	
??	

Tabelle 5.13: Wiederholungsoperatoren (Forts.)

Modifikatoren verändern das Verhalten des Ausdrucks innerhalb einer Gruppe oder eines Teils des Ausdrucks unabhängig von den globalen Optionen:

Kontrukt	Beschreibung
(?o) (?-o)	Für *o* können Sie eine oder mehrere der folgenden Optionen einsetzen: • i – ignoriert Groß- und Kleinschreibung • m – Mehrzeiliger Modus, ^ und $ sind am Anfang und Ende jeder Zeile gültig • s – Einzeilenmodus, . erkennt auch das Zeilenendezeichen • x – Ignoriere Whitespaces, jedoch nicht in Zeichenklassen
(?#)	Kommentare. Die Zeichen vom # bis zum Ende der Klammer werden ignoriert.

Tabelle 5.14: Modifikatoren und Kommentare in regulären Ausdrücken

Direkter Zugriff auf Gruppen

Es besteht auch die Möglichkeit, auf die gefundenen Zeichen der Gruppen direkt im selben Ausdruck zuzugreifen. Der folgende Ausdruck sucht in einem Text Wörter, die doppelt hintereinander vorkommen: `\b([a-z]+)(\s)+(\1\b)`.

Der ersten Teil beginnt mit einer Wortgrenze `\b`. Danach wird als Wort erkannt, was nur aus Buchstaben besteht: `([a-z]+)`. Selbstverständlich würde man in der Praxis noch den `(?i`-Operator einfügen, um Groß- und Kleinschreibung zu gestatten. Dann folgen Leer- oder Trennzeichen; mindestens eins oder beliebig viele `(\s)+`. Der zweite Teil ist der eigentliche Clou an der Sache, hier wird nämlich mit dem speziellen Operator `\1` auf die erste gefundene Referenz verwiesen. Welche Ziffern man einsetzen muss, kann anhand der Anzahl der öffnenden Klammern ausgezählt werden. Danach folgt wieder eine Wortgrenze `\b`. Der Ausdruck findet in Sätzen wie: „Hier kommt kommt das Programm“ die Doppelungen „kommt kommt“ und gibt diese (und nur diese) zurück. Sind mehrere Worte doppelt, wird auch das erkannt. Sind in einem Text mehrere Wörter (hintereinander) doppelt, werden auch mehrere Gruppen zurückgegeben.

Es gibt verschiedene Konstrukte mit Gruppierungen:

Gruppierung	Beschreibung
( )	Einfache Gruppe
(?<name>) (?'name')	Gruppe, deren Inhalt einem Elemente der `Captures`-Kollektion mit dem Index *name* zugewiesen wird
(?:)	Gruppe, die nicht in die Groups- oder `Captures`-Kollektion aufgenommen werden soll.
(?=)	Vorausschauende zutreffende Bedingung
(?!)	Vorausschauende unzutreffende Bedingung
(?<=)	Zurückschauende zutreffende Bedingung
(?<!)	Zurückschauende unzutreffende Bedingung
(?>)	Unterdrückt die Gierigkeit

Tabelle 5.15: Gruppierungskonstrukte

Auf benannte Gruppen kann innerhalb des Ausdrucks mit `\k<name>` zugegriffen werden (die spitzen Klammern gehören dazu). Unbenannte Gruppen werden über die Numerierung erreicht – gezählt werden die öffnenden Klammern –, also beispielsweise `\3` für die dritte Gruppe.

Alternativen

Wenn ein Muster aus mehr als einem Basiswert besteht, sind Alternativen gefragt. Es gibt mehrere Möglichkeiten, alternative Bedingungen zu formulieren:

Konstrukt	Beschreibung
|	Trennt zwei oder mehr Muster: `aa|ab|bb`
(?(ausdruck)y|n)	Führt den Zweig *y* aus, wenn der ausdruck übereinstimmt, sonst *n*.
(?(name)y|n)	Führt den Zweig *y* aus, wenn die benannte Gruppe name übereinstimmt, sonst *n*.

Tabelle 5.16: Formulierung von Alternativen

Der Umgang mit den letzten beiden Optionen ist keineswegs trivial, aber ausgesprochen flexibel. Man kann damit gut einige Zeilen konventionellen Codes sparen.

5.8 Fragen und Übungsaufgaben

1. Schreiben Sie ein kleines Programm, dass mit Hilfe eines regulären Ausdrucks den Inhalt eines Eingabefeldes prüft. Es sollen dabei genau 4 oder 5 Zahlzeichen erlaubt sein.

2. Erstellen Sie eine Applikation, die ein Verzeichnis ausliest und nur die Dateien mit einem wählbaren Dateityp anzeigt. Die Wahl einiger Dateitypen soll über ein Listen-Steuerelement erfolgen.
3. Schreiben Sie ein Programm, dass den Inhalt eines Arrays, das Namen und Zahlenwerte zwischen 1 und 100 enthält, in grafischer Form ausgibt. Erzeugen Sie dazu mit den Grafikklassen aus den Namensräumen `System.Drawing` und `System.Drawing.Imaging` entsprechende Balken mit eingebetteter Schrift.
4. Erstellen Sie ein Formular, dass einen Datumswert aufnimmt und diesen mit Wochentags- und Monatsnamen in drei Sprachen ausgibt.
5. Schreiben Sie eine Applikation, die eine E-Mail mit einem festen Text und einem Dateianhang an mehrere Personen versendet. Die Liste von Personen soll in einer Textdatei stehen, die folgendermaßen aufgebaut ist:

```
mail@adresse.de Name des Empfängers
```

Lesen Sie die Textdatei mit `StreamReader` zeilenweise ein. Verwenden Sie Zeichenkettenmethoden, um die Zeile am ersten Leerzeichen zu trennen.

6 Datenbankzugriff

lernen

In diesem Kapitel geht es um die Nutzung von Datenbanken in ASP.NET-Programmen. Kaum eine Website kommt heute im Internet ohne eine flexible Datenquelle aus. Sehr oft basieren diese auf relationalen Datenbanken. Manche Datenstrukturen liegen jedoch in einer Form vor, die sicher besser hierarchisch darstellen lassen. Dennoch bilden relationale Datenbanken heute immer noch den Schwerpunkt der Datenbankprogrammierung, weswegen diese Technik hier vorgestellt wird.

6.1 Was Sie in diesem Kapitel lernen

Datenbank sind der Kern jeder größeren Webapplikation. Kaum ein Programm kann ohne eine professionelle Datenspeicherung auskommen. Es gibt kaum einen Grund, darauf zu verzichten, denn Datenbankmanagementsysteme stehen fast überall zur Verfügung und können mit standardisierten Methoden benutzt werden.

Dieses Kapitel führt in die Welt der Datenbanken ein. Dies verlangt wieder viel Engagement beim Durcharbeiten der Beispiele, setzt aber kein Wissen voraus. Sie werden mit folgenden Themen Bekanntschaft schließen:

- *Was eine Datenbank ist und wie sei aufgebaut wird* – Dieser Teil bietet die elementarsten Grundlagen.
- *Grundlagen der Datenbanktechnik* – Wie Sie richtig Tabellenstrukturen entwerfen und welche Theorie dahinter steckt, wird kurz angesprochen.
- *Anlegen einer Datenbank* – Von Grund auf sehen Sie, wie eine Datenbank in Microsoft Access erstellt wird.
- *Einführung in SQL* – Die Basis im Umgang mit Datenbanken ist SQL, eine Datenbankabfragesprache. Lernen Sie die elementarsten Grundlagen kennen.

- *ADO.NET* – ADO.NET ist eine Sammelbezeichnung für die Namensräume und Klassen im Framework. Gezeigt wird die Bedienung der Datenbank ebenso wie die Präsentation der Daten in ansprechender Form in einer Website. Dazu werden Sie weitere Web Server-Steuerelemente kennenlernen.

Insgesamt steht dieses Kapitel nicht zufällig am Ende des Buches. Es ist am anspruchsvollsten und baut auf den bereits gezeigten Techniken auf. Gelegentlich werden Sie zurückblättern müssen, um den einen oder anderen Aspekt nochmals nachzulesen. Nichtsdestotrotz werden auch hier alle Schritte sorgfältig präsentiert und sollten genau so nachvollzogen werden.

6.2 Die Basistechnologien

SQL

Gerade am Anfang ist es wichtig, den Überblick zu behalten und die geeignete Methode zum Speichern und Abfragen von Daten auszuwählen. Ist die Wahl gefallen, kann eine systematische Einarbeitung in die speziellen Methoden der gewählten Technologie erfolgen. Relationale Datenbank-Management-Systeme (RDBMS) bestimmen seit langer Zeit den Stand der Datenspeichertechnik. In vielen Fällen lassen sich Daten damit effektiv ablegen und schnell wieder abfragen. Als Abfragesprache kommt meist SQL – Structured Query Language – zum Einsatz.

Diese Sprache liefert einige elementare Befehle, die sowohl das Speichern als auch das Auslesen von Daten in einfacher Form gestatten. Freilich liegt auch hier die Tücke im Detail. SQL kann, wenn die Problemstellung entsprechend komplex ist, sehr anspruchsvoll sein. Für die ersten Schritte im Zusammenhang mit ASP.NET gibt es dennoch einen einfachen Einstieg.

ADO.NET suggeriert, dass die Abfrage von Datenbanken durch geeignete Methoden und Eigenschaften erfolgen kann. Tatsächlich geht es aber mehr darum, die Art und Weise des Zugriffs so einfach wie möglich zu gestalten und gleichzeitig die technischen Ressourcen bestmöglich auszunutzen. Im Endeffekt basiert jedoch auch mit ADO.NET jede Abfrage und jeder Einfügevorgang, der direkt in der Datenbank ausgeführt werden soll, auf SQL. SQL umfasst sehr viele Befehle. Unverzichtbar sind folgende:

- `INSERT` – Damit werden Daten eingefügt
- `DELETE` – Dieser Befehl löscht Daten wieder
- `UPDATE` – Vorhandene Daten werden damit aktualisiert
- `SELECT` – Der sehr mächtige Befehl erlaubt die Abfrage von Daten

Der folgende Abschnitt führt sehr kompakt in die Grundlagen von SQL ein, ohne die Sie ADO.NET nich erfolgreich einsetzen können. Weiterführende Bücher zu SQL kann diese Darstellung natürlich nicht ersetzen.

ADO.NET

Mit der Datenbank auf der einen Seite und einer Abfragesprache gelangen die Daten noch nicht zum Benutzer. Zwischen ASP.NET und der Datenbankschnittstelle stellt das .Net-Framework eine Schicht zur Verfügung, die den Zugriff weitgehend abstrahiert. Diese Schicht wird ADO.NET genannt. Sie bildet sich aus einer ganzen Reihe von Klassen im Namensraum `System.Data`. Weitere Namensräume folgen darunter, was auf den Umfang der Möglichkeiten hindeutet, denn jeder enthält unzählige Klassen. Vom Prinzip her schafft ADO.NET eine lokale Kopie des aktuell gefragten Teils der Datenbank. Diese Kopie wird, wann immer es möglich ist, im Speicher gehalten. Entsprechend effizient können Abfragen erfolgen.

Steuerelemente

Nun steht dem programmatischen Zugriff von ASP.NET nichts mehr im Wege. Was noch fehlt, sind geeignete Web Server-Steuerelemente, die die Darstellung übernehmen. Auch hier kann der Softwareentwickler aus dem Vollen schöpfen. Nahezu jedes Element einer ASP.NET-Seite kann eine Datenquelle an sich binden und so die Darstellung sehr einfach und direkt übernehmen. Die Datenquelle beschränkt sich dabei nicht auf Objekte aus ADO.NET, sondern kann auch Typen wie `Array`, `Hashtable` oder `SortedList` umfassen. Letztere können ihren Inhalt natürlich aus Datenbankabfragen bezogen haben.

6.3 SQL lernen

Am besten lernen Sie SQL kennen, wenn Sie mit den Daten spielen. Die folgenden Beispiele zeigen, wie eine neue Datenbank in Microsoft Access erzeugt und mit Tabellen und Daten gefüllt wird. Dabei werden unabhängig von den sonstigen Möglichkeiten in Access bewusst SQL-Kommandos verwendet. Die Abschnitte über ADO.NET nutzen sowohl diese Tabellenstrukturen als auch vergleichbare Kommandos.

Praktischer Nutzen inklusive

Dieser Abschnitt bereite die Nutzung der Datenbank in den nächsten Schritten vor. Ziel ist es, eine Datenbank zu entwickeln, als Basis eines Shops dienen kann. Im Shop-Beispiel wird die fertige Version dann eingesetzt, um die Bestellinformationen aufzunehmen.

6.3.1 Was ist SQL ?

SQL ist eine Datenbankabfragesprache. Der große Vorteil liegt darin, dass Sie mit der Kenntnis dieser Sprache viele Datenbanken benutzen können, die mit SQL kompatibel sind. Dadurch werden Ihre Daten-

bankanwendungen unabhängig von einer bestimmten Datenbank eines einzelnen Herstellers.

Im deutschen Sprachraum hat es sich übrigens eingebürgert, die drei Buchstaben auch deutsch und einzeln auszusprechen, wenn von SQL die Rede ist.

Technisch sind SQL-Kommandos kleine Befehlszeilen oder Anweisungen, die an den Datenbankserver gesendet werden. Die Befehle sind in englischer Sprache gehalten und sollten den Sinn der Abfrage erkennen lassen. Sie können in SQL auch kleine Programme schreiben. Allerdings ist SQL ganz streng auf die Bedienung von Datenbanken ausgerichtet. SQL alleine bringt wenig. Richtig leistungsfähig wird SQL in Verbindung mit einer Programmierumgebung – in diesem Buch sind dies die Sprache C# und die Bibliotheken unter dem Namen ADO.NET.

6.3.2 Aus was besteht eine Datenbank?

Wenn Sie noch keine klare Vorstellung davon haben, was eine Datenbank eigentlich ist, lesen Sie die folgende kurze Einführung. Ansonsten springen Sie zum nächsten Abschnitt „Daten löschen – DELETE“ auf Seite 353.

Spalte und Datensatz

Datenbanken dienen der Speicherung von Daten. Das können Namen, Adressen, Zahlen, Zeichenketten usw. sein. Daten stehen auch untereinander in Beziehung. So bilden die Teile einer Adresse eine zusammengehörende Einheit. Eine solche Einheit nennt man Datensatz. Die Teile eines Datensatzes bilden die Felder. Alle Felder eines Typs werden als Spalte bezeichnet.

SQL-Datentypen

Jedes Feld hat zwei grundlegende Eigenschaften – einen Namen und einen Datentyp. Die Datentypen sind in SQL denen von C# ähnlich und werden vergleichbar streng überprüft. Die Namen zur Angabe sind jedoch anders und die Einstellmöglichkeiten sind noch feiner. Größere Datenbankmanagementsysteme, wie der Microsoft SQL Server, kennen noch mehr Datentypen und prüfen noch strenger feinste Einstellungen, als dies mit dem .Net-Framework oder einer Programmiersprache möglich wäre. Wenn Sie Adressen speichern, könnten die Felder und Datentypen folgendermaßen aussehen:

```
FIRMA, Varchar(80)
STRASSE, Varchar(80)
ORT, Varchar(50)
PLZ, Integer(5)
```

Die Bedeutung dieser Angaben ist leicht zu verstehen. Das Feld *FIRMA* kann Zeichenketten mit bis zu 80 Zeichen aufnehmen, das Feld *PLZ* (Postleitzahl) kann ganzzahlige Werte mit 5 Stellen aufnehmen usw. Beachten Sie, dass SQL andere Namen für Datentypen als .Net verwendet. So werden Zeichenketten in den Typen `Char` (feste Länge zwischen 1 und 255 Zeichen) oder `Varchar` (variable Länge bis 255 Zeichen) gespeichert, wobei in beiden Fällen die maximale Zeichenanzahl festgelegt ist. `Char` in SQL und `char` in C# sind also sehr verschiedene Datentypen.

Tabelle und Datenbank

Datenbank

Eine Datenbank ist eine Sammlung von Datenbankobjekten, die neben den eigentlichen Daten auch weitere Hilfsinformationen und Anweisungen enthalten kann. Wesentlicher Bestandteil einer Datenbank sind die Tabellen, in denen der Betreiber die Daten ablegt. In der Datenbank werden aber auch Prozeduren, Zugriffsrechte und Verknüpfungen hinterlegt. Man spricht deshalb allgemein von Datenbankobjekten. Diese sollten Sie nicht mit den in .Net verwendeten Objekten verwechseln, auch wenn einige sehr direkt durch entsprechende Klassen abgebildet werden.

Tabellen

Tabellen sind Datenbankobjekte, in denen die Daten abgelegt sind. Tabellen werden durch eine Anordnung von Spalten und Reihen dargestellt. Wenn eine komplette Reihe aus der Tabelle extrahiert wird, um die darin enthaltenen Daten zu betrachten oder zu manipulieren, wird von einem Datensatz gesprochen. Wenn aus dem Datensatz eine bestimmte Spalte ausgewählt wird, nennt man dies ein Feld. Daraus resultieren oft Missverständnisse. Felder können, müssen aber nicht gleich Spalten sein. Und Datensätze können, müssen aber nicht gleich Reihen sein. Die Unterscheidung ist dann von Bedeutung, wenn davon gesprochen wird, ob Befehle „spaltenweise“ oder „reihenweise“ arbeiten. Ebenso können Datentypen nur für eine Spalte und nicht für ein Feld eingestellt werden. Die folgende Abbildung zeigt die Zusammenhänge auf einen Blick.

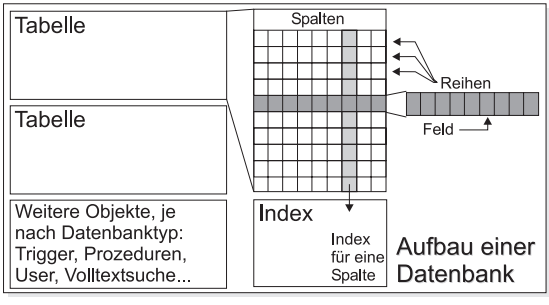


Abbildung 6.1: Prinzipieller Aufbau einer Datenbank

Typische Objekte einer Datenbank sind also in jedem Fall Tabellen, die die Daten enthalten. Definiert werden können auch Indizes, die dem schnelleren Auffinden und Sortieren der Daten dienen. Letztlich ist es noch wichtig zu wissen, dass das Datenbankmanagementsystem die Datenbank selbst als Datei (oder in mehreren Dateien) im Dateisystem ablegt. Bei Microsoft Access wird eine Datei mit der Dateierweiterung *.mdb* verwendet.

Brauchen Sie die Datenbankgrundlagen wirklich? Bei den folgenden Seiten werden Sie sich vielleicht fragen, ob Sie das wirklich brauchen – nur um ein kleines Projekt zu entwickeln. Die Antwort ist ein klares „Ja“: auch kleine Datenbanken sind, wenn ein falsches Modell zugrunde liegt, nicht wirklich beherrschbar. Sie investieren ein Mehrfaches an Zeit, um einfachste Funktionen mit C# und dem Framework umständlich nachzubilden oder können bestimmte Strukturen überhaupt nicht sinnvoll abbilden. Das Verständnis für diese kompakte Einführung ist unbedingt notwendig.

Das relationale Modell und die Datennormalisierung

Das relationale Datenbankmodell verlangt eine ganz bestimmte Form von internen Beziehungen (Relationen) zwischen den Daten. Bei der Wahl der Tabellenstruktur werden Sie damit konfrontiert, Ihre Daten in Tabellen zu speichern. Denken Sie dabei an den bereits mehrfach angesprochenen Web-Shop – sicher eine sehr typische Anwendung. Die folgende Tabelle zeigt eine Reihe eindeutiger Bestellvorgänge, die einige Personen ausgelöst haben:

id	name	artikel	aid	preis
1	Clemens Krause	Windows XP Pro	1325	199,99
2	Clemens Krause	MSDN Abo Professional	6668	2850,00
3	Lukas Werlich	Windows XP Home	1326	139,99
4	Lukas Werlich	Big Color Keyboard	7738	99,00
5	Carolin Scholz	Windows XP Home	1326	139,99
6	Carolin Scholz	Big Color Keyboard	7738	99,00

Tabelle 6.1: Eine nicht normalisierte Tabelle mit Bestelldaten

Die Tabelle enthält alle Informationen, die für die Abwicklung benötigt werden, sowohl die Namen als auch die Daten der Artikel und die Bestellinformationen. Stellen Sie sich außerdem vor, das weitere nicht gezeigten Spalten für die Adresse und die Artikelbeschreibung usw. in der Praxis hinzukommen. Diese Tabelle ist offensichtlich sehr unübersichtlich und – was noch schwerer wiegt, – sie enthält redundante

Daten. So tauchen sowohl die Namen als auch Artikelnummern und Preise mehrfach auf. Die Pflege einer solchen Tabelle ist sehr aufwändig. Die Änderung an einem Datensatz – beispielsweise einem Benutzernamen – würde dazu führen, dass mit der unmittelbaren Operation nicht im Zusammenhang stehende Änderungen an anderen Datensätzen erforderlich sind.

Normalisierungsregeln

Um die Tabelle in eine für relationale Datenbanken passende Form zu bringen, werden die Daten „normalisiert". Dazu werden folgende vier Regeln angewandt:

- **Eliminierung sich wiederholender Gruppen** – Für jede Gruppe von Daten, die sich wiederholen, legen Sie eine eigene Tabelle an. Im Beispiel haben Sie drei Gruppen: Adressen, Bestellungen und Artikel. Also werden drei Tabellen angelegt. Jede Tabelle bekommt einen eindeutigen Schlüssel, das heißt eine Spalte, anhand der die Datensätze eindeutig unterschieden werden können.
- **Eliminierung redundanter Daten** – Wenn eine Eigenschaft mehrere Werte annehmen kann, bringen Sie diese in eine eigene Tabelle. Wenn Sie zum Beispiel feststellen, dass jeder Artikel mehrere Preise haben kann, legen Sie zusätzlich eine Tabelle für die Preise an.
- **Eliminieren von Spalten, die von keinem Schlüssel abhängen** – Wenn Eigenschaften keinen Zusammenhang mit dem Schlüsselfeld haben, also nicht ebenso eindeutig zugeordnet werden können, übertragen Sie diese Eigenschaften in eine eigene Tabelle. Wenn Sie im Beispiel zwei Adressen pro Kunden haben (Lieferanschrift und Rechnungsanschrift), können Sie die Adresse nicht mehr eindeutig einem Schlüsselfeld Kunden-ID zuordnen. Trennen Sie die Anschriften des Kunden in weitere Tabellen ab.
- **Isolierung unabhängiger Beziehungen** – Keine Tabelle darf zwei oder mehr Beziehungen haben, die nicht direkt abhängig sind.

Die oben gezeigte Tabelle sollten Sie also in mindestens drei Tabellen splitten. Die erste enthält alle Adressinformationen. Die gezeigte Auswahl ist nur ein Teil dieser Tabelle, damit hier die Übersicht nicht verloren geht:

id	name	strasse	plz	ort
1	Clemens Krause	Dornenweg 33	12345	Berlin
2	Lukas Werlich	Spielallee 11	89487	München
3	Carolin Scholz	Eichenwall 4	40748	Düsseldorf

Tabelle 6.2: Eine Tabelle, die ausschließlich Adressdaten speichert (Adressen)

Die zweite Tabelle enthält alle Artikelinformationen. Auch hier können Sie sich beliebig viele weitere eindeutige Informationen in weiteren Spalten vorstellen. Redundante Informationen – beispielsweise Eigenschaftsklassen – könnten zu weiteren Tabellen führen. Hier ein Ausschnitt:

id	name	preis	nummer
1	Windows XP Pro	199,99	1325
2	MSDN Abo Professional	2850,00	6668
3	Windows XP Home	139,99	1326
4	Big Color Keyboard	99,00	7738

Tabelle 6.3: Die Artikeltabelle (Artikel)

Die eigentlichen Bestellungen enthalten weit weniger Stammdaten und dafür mehr Verweise auf andere Tabellen.

id	k-id	a-id	menge	summe
1	1	1	1	199,99
2	1	2	1	2850,00
3	2	3	1	139,99
4	2	4	1	99,00
5	3	3	1	139,99
6	3	4	1	99,00

Tabelle 6.4: Die Bestelltabelle – ohne redunandante Informationen (Bestellungen)

Stamm- und Betriebsdaten

Der Gesamtpreis wurde mit aufgenommen, weil der Preis der Artikeltabelle sich ändern kann, während der Preis einmal verkaufter Artikel sich nicht mehr ändert. Beim Normalisieren muss also immer beachtet werden, wo sich Daten ändern können und welche Auswirkungen die Verknüpfungen haben und ob dies auch beabsichtigt ist. Man muss also sorgfältig zwischen „Stammdaten“ und „Betriebsdaten“ unterscheiden. Im Beispiel ist der Preis in der Tabelle *Produkte* vom Typ Stammdaten. Alle Operationen, die Werte dieser Art übernehmen, entnehmen sie aus einer Stammdatentabelle. Die Bestelltabelle zeichnet dagegen einen Momentanzustand auf. Dieser muss erhalten bleiben, auch wenn sich Stammdaten ändern. Es ist Aufgabe der Programmlogik, zwischen beiden Zuständen eine Beziehung herzustellen, wenn dies erforderlich sein sollte.

Beziehungen = Relationen

Unabhängig vom Typ der Daten werden Beziehungen hergestellt. Die folgende Abbildung zeigt die möglichen Verbindungen. Genutzt werden hier so genannte 1:n-Beziehungen, das heißt auf der einen Seite sind die Schlüssel eindeutig, auf der anderen können dagegen ein oder

auch mehrere Datensätze zugeordnet werden. Es gibt auch 1:1- und n:n-Beziehungen.

Diese Beziehungen – englisch Relations – sind der Grund für den Namen dieser Art Datenspeicher: „Relationale“ Datenbankmanagementsysteme. Im Gegensatz dazu werden Daten in XML beispielsweise hierarchisch gespeichert.

Man kann derartige Beziehungen auch gut grafisch darstellen. Die folgende Abbildung zeigt, wie die Informationen in der Tabelle *Bestellungen* mit denen in *Adressen* und *Artikel* verknüpft sind.

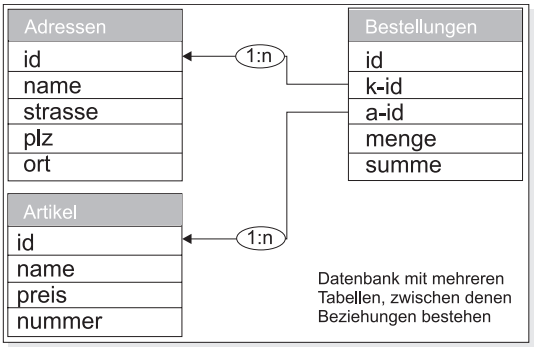


Abbildung 6.2: Beziehungen zwischen denTabellen

Lassen Sie sich an dieser Stelle nicht davon abbringen, sich intensiv mit Datenbanken auseinanderzusetzen, auch wenn dies kompliziert erscheint. Es gibt einfache Anwendungsfälle, die keine Normalisierung benötigen und die nur aus einer einzigen Tabelle bestehen. Es ist aber notwendig zu wissen, was auf Sie zukommt, wenn komplexere Zusammenhänge datentechnisch abgebildet werden müssen. Es ist der Anspruch dieses Buches, dieses Wissen zu vermitteln.

6.3.3 Praktische Arbeit mit Datenbanken

Nach all der Theorie sollten Sie jetzt Access starten und mit der praktischen Arbeit beginnen. Für die ersten Schritte werden drei Tabelle erzeugt und mit ein paar Musterdaten gefüllt. Für das Erzeugen wird Access und der Assistent für neue Tabellen verwendet.

Freilich nutzen Sie danach auf dem Weg zum Profi nicht die Assistenten von Access dafür, sondern SQL, um Daten einzufügen, zu löschen oder zu ändern. Denn exakt dieses Wissen wird danach in der ASP.NET/ADO.NET-Programmierung eingesetzt, um die Datenbank über ein Programm zu bedienen.

Der Datenbankzugriff mit Access

Neue Datenbank anlegen

Starten Sie Access und legen Sie mit DATEI/NEUE DATENBANK ANLEGEN... eine neue Datenbank an. Sie können einen Assistenten oder eine der mitgelieferten Vorlagen benutzen. Für die meisten Webprojekte dürfte das nicht sehr sinnvoll sein. Legen Sie für die Beispiele eine leere Datenbank mit dem *Shop* an. Sie wird in allen restlichen Abschnitten dieses Kapitels verwendet. Speichern Sie diese im Verzeichnis */shop* unterhalb des Übungsverzeichnisses, beispielsweise unter *c:/inetpub/wwwroot/dotnet*.

Für Eilige ist die Tabelle auf der CD zum Buch zu finden. Sie sollten aber die Gelegenheit nutzen, die Schritte selbst anhand der folgenden Schritte nachzuvollziehen.

Mit dem Assistenten ERSTELLT EINE TABELLE IN DER ENTWURFSANSICHT... können Sie nun eine neue Tabelle anlegen. Geben Sie die Informationen wie folgt ein:

Aufbau der Tabellen Adressen

1. Feldname *id*, Felddatentyp AUTOWERT. Klicken Sie außerdem mit der rechten Maustaste auf die äußerst linke Spalte und wählen Sie dann aus dem Kontextmenü PRIMÄRSCHLÜSSEL.
2. Feldname *name*, Felddatentyp TEXT. Im unteren Teil des Assistenten wählen Sie als Feldgröße 80.
3. Feldname *strasse*, Felddatentyp TEXT. Im unteren Teil des Assistenten wählen Sie als Feldgröße 80.
4. Feldname *plz*, Felddatentyp TEXT. Im unteren Teil des Assistenten wählen Sie als Feldgröße 5. Text wird hier verwendet, weil die Postleitzahl auch mit 0 beginnen kann. Diese führende Null würde bei numerischen Feldern verloren gehen.
5. Feldname *ort*, Felddatentyp TEXT. Im unteren Teil des Assistenten wählen Sie als Feldgröße 50.

Klicken Sie dann auf SPEICHERN (oder Menü DATEI/SPEICHERN).

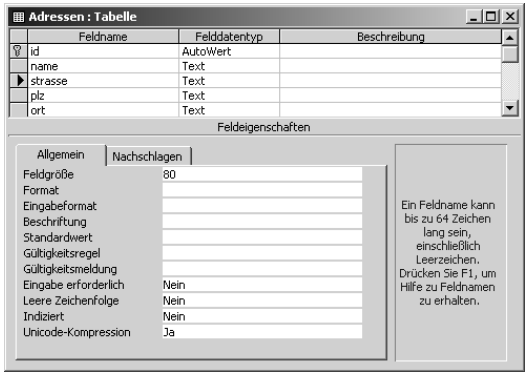


Abbildung 6.3: Die Tabelle Adressen in Access

Schließen Sie nun den Assistenten. Starten Sie ihn erneut, um die Tabellen *Artikel* zu erzeugen. Auch diese Tabelle wird für einige der folgenden Übungen benötigt.

Aufbau der Tabellen Artikel

- Feldname *id*, Felddatentyp AUTOWERT. Klicken Sie außerdem mit der rechten Maustaste auf die äußerst linke Spalte und wählen Sie dann aus dem Kontextmenü PRIMÄRSCHLÜSSEL.
- Feldname *name*, Felddatentyp TEXT. Im unteren Teil des Assistenten wählen Sie als Feldgröße 50.
- Feldname *preis*, Felddatentyp ZAHL. Im unteren Teil des Assistenten wählen Sie als Feldgröße DEZIMAL. In der Zeile DEZIMALSTELLEN wählen Sie den Wert 2.
- Feldname *nummer*, Felddatentyp ZAHL. Im unteren Teil des Assistenten wählen Sie als Feldgröße LONG INTEGER.

Klicken Sie dann auf SPEICHERN (oder Menü DATEI/SPEICHERN) und vergeben Sie den Namen *Artikel*.

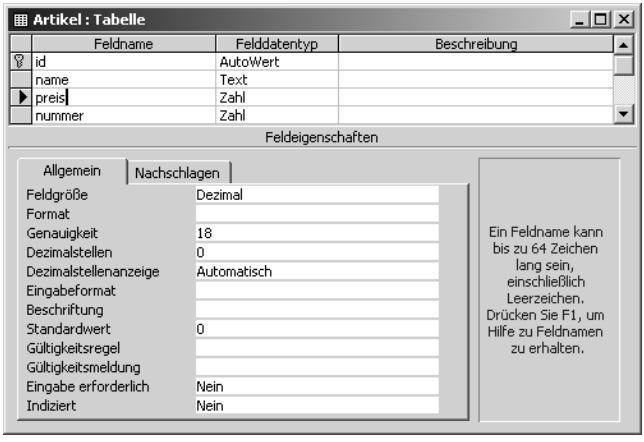


Abbildung 6.4: Die Tabelle Artikel in Access

In einer weiteren Tabelle sollen die Bestellungen des Shops gespeichert werden. Wie in der Übersicht bereits angedeutet, handelt es sich teilweise nur um Relationen:

1. Feldname *id*, Felddatentyp AUTOWERT. Klicken Sie außerdem mit der rechten Maustaste auf die äußerst linke Spalte und wählen Sie dann aus dem Kontextmenü PRIMÄRSCHLÜSSEL.
2. Feldname *k-id*, Felddatentyp ZAHL. Dieses Feld soll die Verknüpfung zu *id* der Tabelle *Adressen* herstellen.
3. Feldname *a-id* , Felddatentyp ZAHL. Dieses Feld soll die Verknüpfung zu *id* der Tabelle *Artikel* herstellen.

4. Feldname *menge*, Felddatentyp ZAHL, Feldgröße SINGLE.
5. Feldname *summe*, Felddatentyp ZAHL. Im unteren Teil des Assistenten wählen Sie als Feldgröße DEZIMAL. In der Zeile DEZIMALSTELLEN wählen Sie den Wert 2.

Klicken Sie dann auf SPEICHERN (oder Menü DATEI/SPEICHERN) und vergeben Sie den Namen *Bestellungen*.

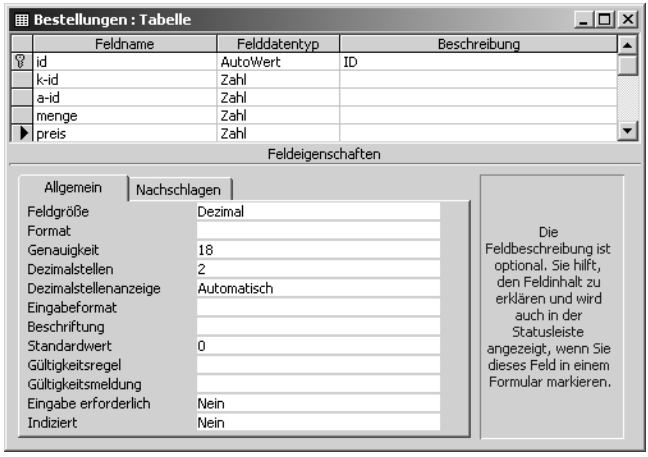


Abbildung 6.5: Die Tabelle Bestellungen in Access

Wenn Sie in Access Spaltennamen verwenden, die Leerzeichen oder Minuszeichen enthalten, müssen Sie bei der späteren Verwendung in Ausdrücken diese Namen in eckige Klammern setzen: `[a-id]`. In den entsprechenden Beispielen wird diese Syntax verwendet, wenn es notwendig ist.

Wo Sie SQL-Kommandos eingeben

Mancher eingefleischte Access-Nutzer wird großartige Projekte erstellt haben und dabei nie mit SQL in Berührung gekommen sein. Es gibt tatsächlich eine Vielfalt von Assistenten und Hilfen, die auch ohne SQL-Kenntnisse ansprechende Ergebnisse hervorbringen. Diese nützen Ihnen nur von ADO.NET aus recht wenig.

Um zum SQL-Eingabefenster zu gelangen, erstellen Sie eine neue Abfrage. Klicken Sie in der Datenbank unter OBJEKTE auf ABFRAGEN. Starten Sie den Assistenten ERSTELLT EINE NEUE ABFRAGE IN DER ENTWURFSANSICHT.

Danach erscheint ein Fenster mit der Auswahl der Tabellen – wählen Sie die eben erzeugten an und klicken Sie dann auf HINZUFÜGEN und dann auf SCHLIESSEN. Klicken Sie nun mit der rechten Maustaste in einen

freien Bereich und wählen Sie aus dem Kontextmenü den Eintrag SQL-ANSICHT.

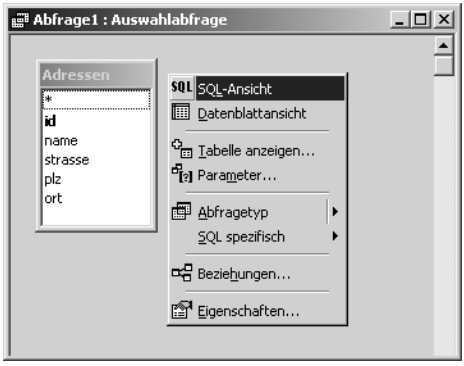


Abbildung 6.6: Umschalten in die SQL-Ansicht in Access

Im danach erscheinenden Editor können Sie SQL-Kommandos eingeben und mit Hilfe des entsprechenden Icons ausführen (siehe Randspalte). Entfernen Sie den vom Assistenten vorgegebenen Text. Alle folgenden Beispiele, die sich um SQL-Kommandos ranken, werden in dieses Fenster eingegeben.

6.3.4 Daten speichern – INSERT

INSERT INTO

Die SQL-Befehle zum Füllen mit Werten sind ein recht umständlicher Weg, Daten einzugeben. Auch hier gilt jedoch, dass Sie im Umgang mit diesen Befehlen sicher sein müssen, um für den Einsatz in ASP.NET vorbereitet zu sein. Was Sie hier mühevoll eintippen, kann dort freilich automatisiert werden.

Mit `INSERT INTO` fügen Sie Daten hinzu. Der Befehl wird folgendermaßen eingesetzt:

```
INSERT INTO adressen (name, strasse, plz, ort)
       VALUES ('Clemens Krause', 'Dornenweg 33', '12345', 'Berlin')
```

Der mehrzeilige Aufbau verbessert nur die Lesbarkeit, SQL erwartet Kommandos in einer Zeile, wird jedoch durch Zeilenumbrüche nicht gestört. Ebenso spielt bei den Befehlen Groß- und Kleinschreibung keine Rolle.

Die Spaltennamen müssen nur angegeben werden, wenn nicht alle Spalten angesprochen werden. Die Reihenfolge der Namen spielt keine Rolle, muss also nicht der Definition entsprechen. Die Werte im `VALUE`-Teil müssen dagegen exakt der Reihenfolge der Spaltenliste folgen. Zeichenketten müssen in Anführungszeichen stehen. Sie können hier – im

Gegensatz zu C# – einfache Anführungszeichen verwenden. Dies hat den Vorteil, dass sie nicht mit den doppelten kollidieren, wenn Sie SQL und C# später kombinieren.

Die Syntax des Befehls `INSERT` ist einfach. Zu beachten ist, dass die Anzahl der Spalten mit den Werten übereinstimmen muss. Ebenso ist die Reihenfolge entscheidend. Zeichenketten müssen in Anführungszeichen stehen. Es gibt auch eine verkürzte Form, bei der hinter `VALUES` exakt die Anzahl der definierten Spalten bedient werden muss:

```
INSERT INTO tabelle VALUES (wert, wert,...);
```

Die Spalte *id* wurde im Beispiel weggelassen, weil sie durch das Feldattribut *AutoWert* automatisch mit fortlaufenden Nummern gefüllt wird.

Geben Sie nun noch einige weitere Datensätze ein, um ein wenig mit den Daten spielen zu können. Es erscheint übrigens vor jeder Einfügeoperation ein Warnhinweis in Access, bei dem Sie die Eingabe bestätigen müssen. Beim Absenden von `INSERT` über ein Programm wird diese Barriere nicht mehr stören.

Mit einem Doppelklick auf den Namen der Tabelle können Sie sehen, ob der Vorgang erfolgreich war und alle Daten erfasst wurden. Wiederholen Sie die letzte Eingabe noch mit einem fehlerhaften Wert – in der folgenden Abbildung steckt der Fehler in Zeile 2. Die fehlerhafte Zeile wird im folgenden Abschnitt mit einem neuen Kommando korrigiert.

Adressen : Tabelle

	id	name	strasse	plz	ort
▶	1	Clemens Kraus	Dornenweg 33	12345	Berlin
	2	Lukass Werlich	Spielallee 11	89487	München
	3	Lukas Werlich	Spielallee 11	89487	München
	4	Carolin Scholz	Eichenwall 4	40748	Düsseldorf
*	(AutoWert)				

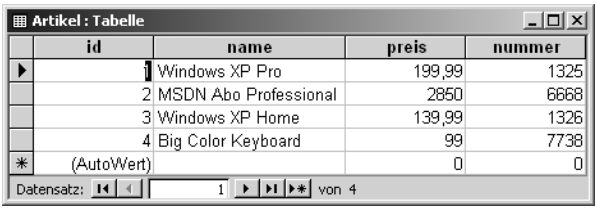
Datensatz: 1 von 4

Abbildung 6.7: Tabellen Adressen mit einigen Musterwerten

Für die nächsten Aufgaben werden alle drei Tabellen benötigt. Füllen Sie auch die Tabellen *Artikel* und *Bestellung* mit ein paar Artikelinformationen:

```
INSERT INTO Artikel (name, nummer, preis)
       VALUES ('Windows XP Home', 1326, 139.99);
```

Beachten Sie, dass Zahlen nicht in Anführungszeichen gesetzt werden und für Dezimalzahlen mit Kommastellen als Trennzeichen der Punkt geschrieben wird, auch wenn Access in der Anzeige die deutsche Darstellung wählt.



Artikel : Tabelle

	id	name	preis	nummer
▶	1	Windows XP Pro	199,99	1325
	2	MSDN Abo Professional	2850	6668
	3	Windows XP Home	139,99	1326
	4	Big Color Keyboard	99	7738
*	(AutoWert)		0	0

Datensatz: 1 von 4

Abbildung 6.8: Tabelle Artikel mit Musterdaten

Ebenso sollten Sie ein paar Bestellungen in die Bestelltabelle eingeben, damit Musterdaten für die Beispiele vorhanden sind:

```
INSERT INTO Bestellungen ( [k-id], [a-id], menge, summe )
       VALUES (3, 4, 1, 99);
```

Verwenden Sie am besten die Musterdaten aus Tabelle 6.4.

Mit diesen Daten wird der kleine SQL-Kurs nun fortgesetzt.

6.3.5 Daten löschen – DELETE

DELETE

Neben dem Einfügen von Datensätzen ist auch das gezielte Löschen möglich. Dazu wird das Kommando `DELETE` eingesetzt. Die Selektion der zu löschenden Datensätze erfolgt mit einer `WHERE`-Bedingung, die ebenso wie bei `SELECT` beschrieben eingesetzt wird:

```
DELETE FROM Adressen WHERE id = 2
```

Lassen Sie die `WHERE`-Bedingung weg, werden alle Datensätze gelöscht. Die Tabelle selbst bleibt aber erhalten. Mehr zu `WHERE` finden Sie im Abschnitt „Daten abfragen – SELECT“ auf Seite 354. Die hier verwendete Bedingung löscht den Datensatz, in dem das Feld *id* den Wert 2 hat (id = 2). SQL kennt ähnliche Operatoren wie C#, der Vergleich wird mit einem statt mit zwei Gleichheitszeichen gebildet.

Eine andere Möglichkeit, den Fehler aus dem letzten Abschnitt zu beseitigen, wäre die Veränderung der Daten.

6.3.6 Daten ändern – UPDATE

UPDATE

Neben dem Einfügen und Löschen können Sie Datensätze auch ändern. Dazu wird `UPDATE` eingesetzt. Sie müssen die zu ändernden Spalten und den neuen Wert angeben. Der Wert kann auch durch eine Formel definiert werden. Welche Datensätze geändert werden, entscheidet wieder die `WHERE`-Bedingung:

```
UPDATE Adressen SET plz = '10999' WHERE id = 1
```

Auch hier erscheint in Access wie schon bei INSERT und DELETE der Warnhinweis. Ohne WHERE werden alle Datensätze geändert – was meist fatale Folgen hat. Mehrere Spalten ändern Sie gleichzeitig, in dem die entsprechenden Anweisungen der Form „Spalte = Neuer Wert“ als Liste, durch Kommata getrennt, aufgeführt werden:

```
UPDATE Adressen SET name = 'Jörg Krause', plz = '12683' WHERE id = 1
```

6.3.7 Daten abfragen – SELECT

SELECT

Häufiger werden Daten abfragt, wozu wird SELECT verwendet wird. Es ist empfehlenswert, sich intensiv mit SELECT auseinanderzusetzen, denn die sorgfältige Auswahl der benötigten Daten durch das Datenbankmanagementsystem ist meist schneller, als wenn Sie alle Daten lesen und dies anschließend programmtechnisch mit C# erledigen.

Die einfachste Abfrage ist die Ausgabe aller Datensätze einer Tabelle; mit dem folgenden Befehl werden alle Name aus der Tabelle *Adressen* angezeigt:

```
SELECT * FROM Adressen
```

Es erscheint nun die Ansicht dieser Abfrage:

Abfrage1 : Auswahlabfrage

	id	name	strasse	plz	ort
▶	1	Jörg Krause	Dornenweg 33	12683	Berlin
	3	Lukas Werlich	Spielallee 11	89487	München
	4	Carolin Scholz	Eichenwall 4	40748	Düsseldorf
*	(AutoWert)				

Datensatz: 1 von 3

Abbildung 6.9: Ergebnis der Abfrage der Tabelle mit SELECT

Nun besteht die Kunst im Umgang mit SQL nicht darin, ganze Tabellen abzurufen. Oft wird ein ganz bestimmter Datensatz benötigt oder es ist sogar ein Zusammenhang zwischen mehreren Tabellen herzustellen. Letzterer Fall resultiert vor allem aus der Normalisierung der Daten, die zur „Auftrennung“ redundanter Informationen führt.

SELECT verstehen

Das „*“ in der letzten Abfrage ist eine Kurzform für „alle Felder“. Der Aufbau des Kommandos orientiert sich ansonsten an folgendem Muster:

```
SELECT spalte, spalte, ... FROM tabelle, tabelle, ... WHERE bedingung
```

Dabei ist mindestens eine Spalte und eine Tabelle anzugeben, weitere Angaben sind optional. Der Befehl besteht aus zwei Schlüsselworten. Das Schlüsselwort SELECT (deutsch Auswahl) leitet den Befehl ein, FROM

(deutsch aus) wählt die Tabelle und WHERE (deutsch wo, wobei) ist die Bedingung.

SQL kennt keine Zeilennummern, nach denen Daten ausgewählt werden könnten. Sie müssen als Alleinstellungsmerkmal selbst eine Spalte dafür einrichten. In den gezeigten Tabellen übernimmt diese Aufgabe jeweils die Spalte *id*. Das die Zahlen darin am Anfang fortlaufend mit 1 beginnend zählen, ist gewollt, aber kein typisches Merkmal dieser Spalten.

Anstatt eines Spaltennamens kann auch ein Ausdruck genutzt werden, beispielsweise `spalte * 1.16`. SQL verfügt hier über ein reiches Spektrum an Funktionen und Operatoren. Das Schlüsselwort WHERE und die nachfolgenden Bedingungen sind optional. Mit ihr kann die Abfrage eingeschränkt werden.

Datenbankabfragen oder, allgemein, Abfragen (engl. query), kennen Sie sicher schon. Jede Anfrage an die Suchmaschinen wie AltaVista oder Yahoo löst eine Datenbankabfrage aus. Viele dieser Abfragefelder kennen Boolesche (logische) Operatoren, UND (engl. and), ODER (engl. or), NICHT (engl. not) usw. So können Sie in AltaVista nach „'C#' AND 'SQL'" suchen. Nur jene Datensätze werden ausgegeben, die „C#" UND „SQL" im Suchtext hatten (beachten Sie die einfachen Anführungszeichen zum Kombinieren der Wörter zu einer Phrase).

Nach einem SELECT-Kommando erscheint das Ergebnis der Abfrage im Fenster und die SQL-Ansicht ist verschwunden. Klicken Sie mit der rechten Maustaste auf die Kopfzeile des Fensters und wählen Sie im Kontextmenü erneut die Option SQL-ANSICHT.

In SQL schreiben Sie die Abfrage, bezogen auf die Mustertabelle *Adressen*, folgendermaßen:

```
SELECT name, ort FROM Adressen WHERE name = 'Jörg Krause'
```

Abfragen gestalten

Sehen Sie sich den folgenden Zugriff auf die Tabelle *Artikel* an:

```
SELECT name, preis FROM Artikel
```

Es entsteht dieses Ergebnis:

name	preis
Windows XP Pro	199,99
MSDN Abo Professional	2850
Windows XP Home	139,99
Big Color Keyboard	99

Abbildung 6.10: Ergebnis der Abfrage

Sie können sich nun alle Preise mit und ohne Mehrwertsteuer ansehen:

```
SELECT name, preis AS netto, preis * 1.16 AS brutto FROM Artikel
```

AS Mit Hilfe des Operators AS werden neue Spaltennamen für die Ausgabe erzeugt. AS steht für Alias. Außerdem werden Berechnungen ausgeführt, ebenso wie Sie dies aus C# bereits kennen. Hier können Sie alle Operationen ausführen, die zu kompletten Ausdrücken führen. Als Werte können Spaltennamen oder Konstanten eingesetzt werden.

name	netto	brutto
Windows XP Pro	199,99	231,9884
MSDN Abo Professional	2850	3306
Windows XP Home	139,99	162,3884
Big Color Keyboard	99	114,84

Abbildung 6.11: Ergebnis der Abfrage

ORDER BY ... DESC|ASC Für eine bestmögliche Übersicht können Sie die Ausgabe sortieren lassen:

```
SELECT name, preis AS netto, preis * 1.16 AS brutto FROM Artikel
       ORDER BY preis DESC
```

Mit DESC (von engl. descend) wird absteigend sortiert und mit ASC (von engl. ascend) können Sie auch aufsteigend sortieren. Sortiert wird nach dem ersten oder dem angegebenen Feld, hier also nach *preis*. Zwei Dinge sollten Sie beachten: Das Sortieren großer Datenbanken beansprucht den Server stark. Sie können außerdem nur nach den Spalten sortieren, die auch ausgegeben werden. Normalerweise können Sie aber davon ausgehen, dass die Sortierung in der Datenbank schneller ist, als mit C# und dem Framework. Voraussetzung ist jedoch eine entsprechende Indizierung der Datenbank.

> Was ist ein Index?
>
> Ein Index ist eine zusätzliche Option zur Verwaltung von Tabellen in Datenbanken. Der Index bildet eine Zugriffstabelle auf Basis einer Spalte einer normalen Tabelle ab. Das Durchsuchen oder Sortieren funktioniert erheblich schneller, als wenn die gesamte Tabelle behandelt werden müsste. Spalten, die häufig als Kriterium für solche Vorgänge herangezogen werden, sollten indiziert werden. In Microsoft Access wählen Sie bei der Tabellendefinition (Entwurfsansicht) die entsprechende Spalte und dann in Eigenschaftsliste, Zeile INDIZIERT, die Option JA.

Die Verwendung des Index passiert intern, sodass Sie sich bei der Gestaltung der SELECT-Kommandos darauf nicht mehr konzentrieren müssen. Indizes wirken übrigens auch positiv auf die Geschwindigkeit beim Einfügen mit INSERT, wenn bereits sehr viele Daten vorhanden sind.

Der bereits angesprochene Primärschlüssel führt übrigens automatisch zu einem Index nach der Spalte *id*. Das ist kein Merkmal von Access, sondern gilt für alle Datenbankmanagementsysteme. Dies impliziert auch, dass jede Tabelle mehrere Indizes besitzen kann.

DISTINCT

Manchmal sind Datensätze oder Teile davon doppelt. So gibt es in der Tabelle *Bestellungen* viele Artikel mehrfach, wenn entsprechende Bestellungen erfolgten. Wenn Sie einfach nur wissen möchten, welche Artikel bereits bestellt wurden, hilft DISTINCT weiter:

```
SELECT DISTINCT [a-id] FROM Bestellungen
```

Dieser Befehl braucht mehr Leistung von der Datenbankmaschine. Setzen Sie ihn nicht in häufig generierte Abfragen ein.

LIKE

In den meisten Fällen werden Zeichenketten verarbeitet. SQL unterstützt die Arbeit mit Zeichenketten deshalb sehr komfortabel. So ist es leicht möglich, eine Teilzeichenkette zur Selektierung anzugeben, die in allen Feldern an allen Positionen getestet wird. Das Schlüsselwort dafür ist LIKE. Der folgende Befehl sucht alle Personen, in deren Namen das Wort „Krause“ steckt:

```
SELECT * FROM Adressen WHERE name LIKE '*Krause*'
```

Access verwendet hier als Platzhalter das bekannte Sternchen bzw. für ein einzelnes Zeichen das Fragezeichen. Wenn Sie mit SQL Server oder einer anderen Datenbank arbeiten, werden üblicherweise die in SQL definierten Platzhalterzeichen „%“ und „_“ verwendet.

Bedingungen mit WHERE definieren

Sie können mit WHERE logische Bedingungen angeben. Wollen Sie alle Artikel in einem bestimmten Preisfenster wissen, können Sie folgendes definieren:

```
SELECT *  FROM Artikel WHERE preis > 100 AND preis < 200
```

id	name	preis	nummer
3	Windows XP Home	139,99	1326
1	Windows XP Pro	199,99	1325

Abbildung 6.12: Resultat der Abfrage

BETWEEN Solche Kombinationen aus zwei komplementären Bedingungen lassen sich auch – und oft besser lesbar – mit BETWEEN ausdrücken:

```
SELECT *  FROM Artikel WHERE preis BETWEEN 100 AND  200
```

Die Schreibweise mit BETWEEN (dt. zwischen) ist zur vorher genutzten Form äquivalent; das Ergebnis ist identisch, wenn Sie beachten, dass BETWEEN auf Gleichheit der Grenzwerte prüft (dies entspräche <= bzw. >= im vorhergehenden Beispiel).

SQL-Funktionen Als Nächstes sollen Artikel anhand des Namens ausgewählt werden. Die einfache Abfrage lautet:

```
SELECT nummer, name
       FROM Artikel
       WHERE LCASE (name) LIKE '*pro*'
```

Hier wird zusätzlich eine Funktion eingesetzt: LCASE. Die überführt alle Zeichen in Kleinbuchstaben. Dadurch erfüllt die Suche nach „pro“ auch den Namen, der „Professional“ enthält. Es gibt in SQL sehr viele solcher Funktionen; welche Sie einsetzen können, ist in der entsprechenden Programm-Hilfe zu finden. Leider ist hier SQL nicht gleich SQL, jedes Datenbankmanagementsystem verwendet teilweise eine unterschiedliche Syntax für diese Funktionen. Access verwendet keine SQL-spezifischen Funktionen, sondern das VBA-Modul (VBA = Visual Basic for Applications). Damit stehen über 100 Funktionen bereit.

IN Alle Reihen sollen ermittelt werden, die in der Spalte *plz*

die Zeichenkette „12683“ oder „89487“ haben. Dies erledigt zuverlässig der Operator IN:

```
SELECT *  FROM Adressen WHERE plz IN ('12683', '89487')
```

NOT Um die Auswahl zu negieren, ist das Schlüsselwort NOT erlaubt. Bei logischen Bedingungen mit AND und OR kennen Sie das schon vom C#-Befehl if. Auch die Schlüsselworte BETWEEN und IN lassen sich mit NOT kombinieren.

Abfragen mehrerer Tabellen

Wenn Daten aus mehreren Tabellen benötigt werden, sind die Abfragen schon etwas komplizierter. Andererseits kann hier SQL seine Vorteile voll ausspielen – es gibt fast nichts, was man nicht auf direkten Weg abfragen könnte.

Am Anfang wurde angenommen, das neben Adressen auch Bestelldaten und Artikel gespeichert werden. Nun sind die Daten aber – wegen der angesprochenen Normalisierung – auf drei Tabellen verteilt. Der Abruf

einer bestimmte Bestellung oder auch aller Daten erfordert, dass alle drei Tabellen gleichzeitig angesprochen werden. Probieren Sie zuerst folgendes aus:

```
SELECT Adressen.name, Artikel.name, Bestellungen.id FROM Adressen, Artikel,
Bestellungen
```

Die Abbildung zeigt, dass das Kommando zwar ausgeführt wurde, allerdings nicht unbedingt sinnvolle Daten lieferte:

Adressen.nam	Artikel.name	id
Jörg Krause	Windows XP Pro	1
Lukas Werlich	Windows XP Pro	1
Carolin Scholz	Windows XP Pro	1
Jörg Krause	MSDN Abo Professi	1
Lukas Werlich	MSDN Abo Professi	1
Carolin Scholz	MSDN Abo Professi	1
Jörg Krause	Windows XP Home	1
Lukas Werlich	Windows XP Home	1
Carolin Scholz	Windows XP Home	1
Jörg Krause	Big Color Keyboard	1
Lukas Werlich	Big Color Keyboard	1
Carolin Scholz	Big Color Keyboard	1
Jörg Krause	Windows XP Pro	2
Lukas Werlich	Windows XP Pro	2
Carolin Scholz	Windows XP Pro	2
Jörg Krause	MSDN Abo Professi	2
Lukas Werlich	MSDN Abo Professi	2
Carolin Scholz	MSDN Abo Professi	2
Jörg Krause	Windows XP Home	2
Lukas Werlich	Windows XP Home	2
Carolin Scholz	Windows XP Home	2
Jörg Krause	Big Color Keyboard	2
Lukas Werlich	Big Color Keyboard	2
Carolin Scholz	Big Color Keyboard	2
Jörg Krause	Windows XP Pro	3
Lukas Werlich	Windows XP Pro	3
Carolin Scholz	Windows XP Pro	3
Jörg Krause	MSDN Abo Professi	3
Lukas Werlich	MSDN Abo Professi	3
Carolin Scholz	MSDN Abo Professi	3
Jörg Krause	Windows XP Home	3
Lukas Werlich	Windows XP Home	3
Carolin Scholz	Windows XP Home	3
Jörg Krause	Big Color Keyboard	3
Lukas Werlich	Big Color Keyboard	3
Carolin Scholz	Big Color Keyboard	3
Jörg Krause	Windows XP Pro	4
Lukas Werlich	Windows XP Pro	4
Carolin Scholz	Windows XP Pro	4
Jörg Krause	MSDN Abo Professi	4
Lukas Werlich	MSDN Abo Professi	4
Carolin Scholz	MSDN Abo Professi	4

Abbildung 6.13: Mehrfachabfrage: So sollte das Ergebnis vermutlich nicht aussehen

Das ist sicher nicht das erwartete Ergebnis, denn Access hat hier einfach alle Tabellen miteinander vermischt. Die Beziehungen, die anfangs ja nur gedanklich existierten, sind nicht angegeben worden. Sie können nicht davon ausgehen, dass nur die Benennung mit „id“ zu brauchba-

ren Ergebnissen führt. Letztlich steht Ihnen die Namenswahl sowieso frei. Also müssen die Beziehungen angegeben werden. Dazu dient wiederum die WHERE-Bedingung:

```
SELECT a.name, p.name, b.id AS BestellID
  FROM Adressen a, Artikel p, Bestellungen b
  WHERE a.id = b.[k-id]
    AND p.id = b.[a-id]
```

Aliase

Neben der Angabe hinter WHERE wurden hier auch noch so genannte Alias-Namen verwendet (a für Adressen, p für Artikel und b für Bestellungen). Diese dienen nur der Verkürzung der Schreibweise. Schauen Sie sich zum Vergleich mit dem letzten Ergebnis die verwendete Tabelle *Bestellungen* an. Das Ergebnis der Abfrage spiegelt diese Tabelle wieder, allerdings sind neben den Bestell-IDs auch die Namen der Käufer und Artikel zu finden:

a.name	p.name	BestellID
Jörg Krause	Windows XP Pro	1
Jörg Krause	MSDN Abo Professional	2
Lukas Werlich	Windows XP Home	3
Lukas Werlich	Big Color Keyboard	4
Carolin Scholz	Windows XP Home	5
Carolin Scholz	Big Color Keyboard	6

Abbildung 6.14: Korrekte Abfrage der Bestellungen mit Zuordnungen aus anderen Tabellen

INNER JOIN

Letztlich basiert das ganze Verfahren nur auf der Verbindung der ID-Spalten: a.id = b.[k-id] usw. Die Verknüpfung zweier Tabellen kann in SQL auch mit dem Operator INNER JOIN ausgedrückt werden. Das Ergebnis erscheint immer dann, wenn zwei Felder zweier Tabellen übereinstimmen:

```
SELECT a.name, b.id AS BestellID
  FROM Bestellungen b INNER JOIN Adressen a
    ON  a.id = b.[k-id]
```

Dieses Kommando zeigt alle Namen aus der *Adressen*-Tabelle an, zu denen Bestellungen vorliegen.

Dieser Abschnitt konnte lediglich eine kompakte Einführung in SQL bieten, gerade genug, um ADO.NET nun erfolgreich einsetzen zu können. Mehr Informationen über SQL finden Sie in einschlägiger Literatur.

6.4 Einführung in ADO.NET

ADO.NET unterscheidet derzeit zwei Zugriffsmethoden auf Datenbanken. Eine Methode ist unter dem Namen OLEDB schon länger bekannt.

OLEDB ist eine definierte Schnittstelle, die auf einem so genannten Datenbankprovider basiert. Der Anbieter der Datenbank muss diesen – ein Stück Software, vergleichbar mit einem Gerätetreiber – zur Verfügung stellen. Die andere Methode nutzt als Datenquelle den MS SQL Server, legt also die Kernkomponente direkt fest. Abgesehen von der durch die direkte Kopplung möglichen Geschwindigkeitssteigerung beim Zugriff gibt es keine Unterschiede. Wollen Sie eine Applikation mit MS Access entwickeln und im Produktionsrechner SQL Server einsetzen, können Sie OLEDB durchaus verwenden.

In diesem Buch wird als Datenquelle Microsoft Access verwendet. Es kommt deshalb ausnahmslos die OLEDB-Schnittstelle zum Einsatz. Durch die Ähnlichkeit der Schnittstellen sind die Informationen aber recht einfach auf den SQL Server übertragbar.

Basisklassen

ADO.NET kennt für den direkten Kontakt mit dem RDBMS drei Basisklassen:

- `OleDbConnection` – Diese Klasse stellt die physische Verbindung her.
- `OleDbCommand` – Hiermit werden Kommandos an die Datenbank gesendet und Daten von dort aufgenommen.
- `OleDbDataReader` – Diese Klasse liest Daten in einen lokalen Puffer, wo sie zum direkten und schnellen Lesen zur Verfügung stehen. Änderungen an der Datenbank können damit nicht erfolgen.

Für den SQL-Server stehen vergleichbare Klassen unter den Namen `SqlConnection`, `SqlCommand` und `SqlDataReader` zur Verfügung.

Am Anfang wurde erwähnt, dass Datenfragmente lokal vorgehalten werden, um sehr schnelle Zugriffe zu ermöglichen. Da Daten nicht einfach aus Selbstzweck existieren, sondern Teil eines komplexen Ganzen sind, müssen neben der Ablage auch Verknüpfungen und Sichten verarbeitet werden können. Diese Schicht umfasst deshalb auch dafür entsprechende Klassen:

- `OleDbDataAdapter` – Diese Klasse stellt die Verbindung zu den anderen OLEDB-Klassen zur Verfügung.
- `DataSet` – Mit dieser Klasse werden zusammenhängende Daten im Speicher verwaltet. Die Informationen müssen nicht zwangsläufig in derselben Tabelle liegen.
- `DataTable` – Diese Klasse repräsentiert genau eine Tabelle.
- `DataRelation` – Relationale Datenbanken definieren Beziehungen zwischen Daten. Diese Klasse dient der Definition, Speicherung und Kontrolle solcher Beziehungen (Relationen).
- `DataView` – Die Definition von Sichten erleichtert komplexe Abfragen. Eine spezielle Klasse dient der Verwaltung.

6.4.1 Zugriff auf Daten mit ADO.NET

Das Einbinden von ADO.NET gehört nicht mehr zu den Standardaufgaben von ASP.NET. Ohnehin erreichen die Anwendungen sehr schnell eine Größenordnung, die die Nutzung von hinterlegtem Code (Code Behind) sinnvoll erscheinen lässt. Sie sollten sich deshalb von vornherein ganz auf diese Technik konzentrieren.

Vorbereitung und Verbindungsherstellung

Namensräume

Das Einbinden der entsprechenden Namensräume kann folgendermaßen geschehen:

```
using System.Data;
using System.Data.OleDb;
```

Verbindungszeichenfolge

Im vorhergehenden Abschnitt über SQL wurde bereits eine Access-Datenbank entwickelt und angelegt. Der Provider nutzt eine so genannte Verbindungszeichenfolge, die im Fall von MS Access folgenden Aufbau hat:

```
Microsoft.Jet.OLEDB.4.0
```

Mit diesen Angaben kann bereits eine Verbindung zur Datenbank aufgebaut werden. Der folgende Ablauf gilt für jede Art Datenbankzugriff:

- Verbindung öffnen
- Abfragen ausführen oder Daten aktualisieren
- Verbindung schließen

Sicherer Code mit try-catch

Das Öffnen der Verbindung steht also immer an erster Stelle und wird hier zuerst behandelt. Sie sollten sich auch bei kleinen Projekten zu einem guten und sicheren Programmierstil zwingen und die sprachlichen Möglichkeiten von C# nutzen. Deshalb wird gleich eine `try-catch`-Kombination eingesetzt. Datenbankverbindungen können aus vielfältigen Gründen nicht funktionieren. Das Abfangen von Laufzeitfehlern ist gerade hier notwendig und sorgt für nutzerfreundlichere Programme, die nicht völlig zusammenbrechen, wenn mal etwas nicht funktioniert.

```
<%@ Page Language="C#" Inherits="connopen" src="ado_connection.cs" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="de">
    <head>
        <title>ADO.NET - Connection</title>
    </head>
    <body>
```

```
    <h1>ADO.NET - Connection</h1>
    <p id="isOpen" runat="server"/>
    </body>
</html>
```

Listing 6.1: Sichere Verbindungseröffnung (ado_trycatch.aspx)

Selbstverständlich wird Code Behind verwendet, die eigentliche Funktion steckt deshalb in der Datei *ado_connection.cs*. Die *aspx*-Datei selbst dient lediglich der Anzeige des Ergebnis im HTML Server-Steuerelement mit dem Namen `isOpen`.

```
using System;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Data;
using System.Data.OleDb;

public class connopen : Page
{
   public HtmlGenericControl isOpen;
   void Page_Load()
   {
        string database = Server.MapPath("../shop/shop.mdb");
        string provider = "PROVIDER= Microsoft.Jet.OLEDB.4.0; Data Source="
                          + database;
        OleDbConnection oConn = new OleDbConnection(provider);
        try
        {
            oConn.Open();
            isOpen.InnerText = "Verbindung erfolgreich";
            oConn.Close();
        }
        catch (OleDbException e)
        {
            isOpen.InnerText = "Verbindungsfehler: " + e.Message;
        }
    }
}
```

Listing 6.2: Code Behind-Datei ado_connection.cs

Wie es funktioniert

Die Klasse `connopen`, die hier definiert wird, erbt wie üblich von `Page`, damit der Zugriff auf die HTML Server-Steuerelemente der Seite gelingt. Dann wird die Ausgabe verfügbar gemacht:

```
public HtmlGenericControl isOpen;
```

Die eigentliche Funktion steckt in der Methode `Page_Load`. Zuerst wird der Pfad zur Datenbank ermittelt. Dies ist nur bei Access-Datenbanken notwendig, die über einen Dateinamen adressiert werden. Der Provider des SQL Servers verlangt nur die Angabe des Katalogs. Da sich die Access-Datenbank in der Regie des Webservers befindet, wird die Methode `MapPath` des ASP.NET-Objekts `Server` verwendet:

```
string database = Server.MapPath("../shop/shop.mdb");
```

Nun kann die Verbindungszeichenfolge festgelegt werden:

```
string provider = "PROVIDER=Microsoft.Jet.OLEDB.4.0; Data Source="
                  + database;
```

Als erstes ist jetzt ein Verbindungsobjekt der Klasse `OleDbConnection` erforderlich, deren Konstruktor die Verbindungszeichenfolge übernimmt:

```
OleDbConnection oConn = new OleDbConnection(provider);
```

Damit ist die Verbindung zwischen Objekt und Datenbank definiert, aber noch nicht eröffnet. Bis hier können auch keine Fehler auftreten, denn es werden lediglich interne Variablen gesetzt. Es folgt nun die Eröffnung der Verbindung. Dieser Teil kann misslingen, wenn der Pfad falsch ist, der Datenbankserver nicht verfügbar ist oder ein anderer Fehler auftritt. Deshalb erfolgt der Einbau in einen `try`-Block, gefolgt vom Versuch, die Verbindung zu öffnen:

```
try
{
   oConn.Open();
```

Es erfolgt in diesem kleinen Programm nur eine Kontrollausgabe – dann wird die Datenbankverbindung wieder geschlossen:

```
oConn.Close();
```

Falls die Eröffnung der Verbindung misslingt, wird die Ausführung im passenden `catch`-Zweig fortgesetzt. Auftreten kann hier die Ausnahme `OleDbException`, die für fast alle Fälle allgemein genug ist. Entprechend sieht dieser Teil folgendermaßen aus:

```
catch (OleDbException e)
```

Das Objekt `e` enthält nun die Daten der Ausnahme; die Eigenschaft `Message` enthält den Fehlertext:

```
isOpen.InnerText = "Verbindungsfehler: " + e.Message;
```

Wenn nun ein Fehler auftritt oder Sie diesen provozieren, beispielsweise durch einen falschen Dateinamen, erhalten Sie folgende Ausgabe:

Abbildung 6.15: Fehlerausgabe bei einem falschen Dateinamen

Ohne die `try-catch`-Kombination würde an dieser Stelle ein Laufzeitfehler erscheinen, was sicher nicht einer guten Benutzerführung entspricht.

Sie sollten Datenbankverbindungen immer schließen, wenn sie nicht mehr benötigt werden, um wertvolle Ressourcen sofort freizugeben – nicht erst am Ende des Programms.

Mit der geöffneten Verbindung erfolgt nun der Zugriff auf die Datenbank.

OleDbConnection
OleDbCommand

In der Shop-Applikation kommt es unter anderem darauf an, Bestellungen aufzunehmen. Dazu ist die Erfassung von Adressen notwendig. Das folgende Beispiel zeigt, wie die Daten eines Formulars erfasst und mit `INSERT` gespeichert werden können.

```
<%@ Page Language="C#" Inherits="getdata" src="ado_getdata.cs" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="de">
    <head>
        <title>Kundenregistrierung</title>
    </head>
    <body>
        <h1>Kundenregistrierung</h1>
        <form runat="Server">
            Name: <input type="text" runat="Server" id="KundName"/><br/>
            Stra&szlig;e: <input type="text" runat="Server" id="KundStr"/>
            <br/>
            Ort: <input type="text" runat="Server" id="KundOrt"/><br/>
            Plz: <input type="text" runat="Server" id="KundPlz"/><br/>
            E-Mail: <input type="text" runat="Server" id="KundEMail"/><br/>
            Menge: <input type="text" runat="Server" id="KundMenge"/><br/>
            <br/>
            <input type="submit" value="Jetzt registrieren"/>
        <p id="isOpen" runat="server">Innern</p>
        </form>
    </body>
</html>
```

Listing 6.3: Daten aus einem Formular speichern (ado/ado_getdata.aspx)

Die eigentliche Arbeit wird in der Code-Datei *ado_getdata.cs* und dort in der Klasse `getdata` erledigt:

```
using System;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using System.Data;
using System.Data.OleDb;

public class getdata : Page
{
   public HtmlGenericControl isOpen;
   public HtmlInputText KundName, KundStr, KundPlz, KundOrt;

   void Page_Load()
   {
        if (Page.IsPostBack)
        {
            string database = Server.MapPath("../shop/shop.mdb");
            string provider = @"PROVIDER=Microsoft.Jet.OLEDB.4.0;
                               Data Source=" + database;
            OleDbConnection oConn = new OleDbConnection(provider);
            try
            {
                oConn.Open();
                string abfrage;
                abfrage  = "INSERT INTO Adressen (name, strasse, plz, ort)";
                abfrage += "VALUES (";
                abfrage += "'" + KundName.Value + "',";
                abfrage += "'" + KundStr.Value + "',";
                abfrage += "'" + KundPlz.Value + "',";
                abfrage += "'" + KundOrt.Value + "')";
                OleDbCommand oComm = new OleDbCommand(abfrage, oConn);
                oComm.ExecuteNonQuery();
                oConn.Close();
            }
            catch (OleDbException e)
            {
                isOpen.InnerText = "Verbindungsfehler: " + e.Message;
            }
        }
   }
}
```

Listing 6.4: Speichern von Formulardaten mit `INSERT` (ado/ado_getdata.cs)

Wie es funktioniert

Das Speichern von Daten erfolgt also im Prinzip mit genau dem SQL-Kommando, das in der Einleitung vorgestellt wurde. Den Transport zur Datenbank erledigt die Klasse `OleDbCommand`, deren Konstruktor das SQL-Kommando als Parameter aufnimmt:

```
OleDbCommand oComm = new OleDbCommand("INSERT INTO ...")
```

ExecuteNonQuery

`oComm` enthält nun ein Kommando-Objekt, das noch ausgeführt werden muss. Erst diese Ausführung sendet das SQL-Kommando an den Datenbankserver. Dazu dient bei SQL-Kommandos, die keine Daten produzieren, die Methode `ExecuteNonQuery`:

```
oComm.ExecuteNonQuery();
```

Außer für `INSERT` findet diese Methode auch bei `DELETE` und `UPDATE` Verwendung. Die Kommandos liefern zwar keine Daten, aber durchaus Informationen über den Erfolg der Aktion. So kann nach `DELETE` festgestellt werden, wie viele Datensätze gelöscht worden sind. Dazu muss lediglich der Rückgabewert der Methode `ExecuteNonQuery` ausgewertet werden, der ein `Int32`-Wert sein kann:

```
iAnzahl = oCommand.ExecuteNonQuery();
```

Der beschriebene Weg ist der direkteste, wenn es darum geht, SQL-Kommandos zu senden. Mit Hilfe von `DataSet`-Objekten ist die Bedienung einer Datenbank noch komfortabler. Mehr dazu finden Sie in „Komfortabler Datenzugriff“ auf Seite 377.

6.5 Daten aus der Datenbank abfragen

Bislang wurden Daten lediglich in die Datenbank geschrieben. Es liegt in der Natur von Webseiten, das häufiger Daten gelesen als geschrieben werden. Abfragen können kompliziert sein, weil es entscheidend darauf ankommt, die Daten in benutzergerechter Form zu präsentieren. Relationale Datenbanken halten die Informationen in der Regel nicht exakt so vor, wie sie für die Ausgabe benötigt werden.

6.5.1 Einfacher Datenbankzugriff

Kern aller Abfragen bildet das SQL-Kommando `SELECT`. Der Ablauf ist auch hier bei allen Applikationen identisch:

- Datenbankverbindung öffnen
- Abfrage ausführen
- Daten in internen Datenspeicherbereich kopieren
- Daten lesen und darstellen
- Datenbankverbindung schließen

OleDbDataReader
ExecuteReader

Da einfache Lesevorgänge häufig sind, gibt es speziell dafür eine optimierte Klasse: `OleDbDataReader`. Aus dieser Klasse stammt das Objekt `DataReader`. Es erlaubt die Ausführung der Abfrage und das sequenzielle Durchlaufen aller Datensätze, falls die Abfrage mehr als einen Datensatz produzierte. Dieses Verfahren ist äußerst effizient, weil es auf jede überflüssige Steuerung verzichtet. Sie können weder rückwärts durch die Datensätze gehen noch mehrere gleichzeitig lesen. Das Objekt bietet immer den aktuellen Datensatz an, die Methode `Read` springt zum nächsten. Das `DataReader`-Objekt entsteht, wenn das Kommando statt mit `ExecuteNonQuery` mit der Methode `ExecuteReader` gesendet wird:

```
OleDbDataReader rs = oComm.ExecuteReader();
```

Direktes Ausgeben der Daten einer Abfrage

Bei einem sequenziellen Ablauf ist es nahe liegend, eine `while`-Schleife zur Darstellung zu verwenden. Das folgende Programm liest die Artikeltabelle aus und zeigt, wie HTML-Tags dynamisch in Abhängigkeit der Daten erzeugt werden können:

```
<%@ Page Language="C#" debug="true" Inherits="shop" src="ado_shop.cs"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="de">
    <head>
        <title>Shop</title>
    </head>
    <body>
        <h1> Shop </h1>
        Unsere Produkte:
        <ul id="produkte" runat="server"/>
        <br/>
        <p id="isOpen" runat="server"/>
    </body>
</html>
```

Listing 6.5: Sequenzielles Auslesen von Daten (shop/chap6/shop.aspx)

Die eigentliche Arbeit erledigt wieder das Programm in einer Code-Datei:

```
using System;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using System.Data;
using System.Data.OleDb;
```

```
public class shop : Page
{
    public HtmlGenericControl produkte;
    public HtmlGenericControl isOpen;

    public void Page_Load()
    {
        string database = Server.MapPath("../shop.mdb");
        string provider = "PROVIDER=Microsoft.Jet.OLEDB.4.0; Data Source="
                          + database;
        OleDbConnection oConn = new OleDbConnection(provider);
        try
        {
            oConn.Open();
            string abfrage = "SELECT * FROM Artikel";
            OleDbCommand oComm = new OleDbCommand(abfrage, oConn);
            OleDbDataReader oData = oComm.ExecuteReader();
            while (oData.Read())
            {
                HtmlGenericControl produkt = new HtmlGenericControl("li");
                HtmlAnchor plink = new HtmlAnchor();
                plink.HRef = "order.aspx?produkt=" + oData["id"].ToString();
                plink.InnerText = oData["name"].ToString();
                produkt.Controls.Add(plink);
                produkte.Controls.Add(produkt);
            }
            oConn.Close();
        }
        catch (OleDbException e)
        {
            isOpen.InnerText = "Verbindungsfehler: " + e.Message;
        }
    }
}
```

Listing 6.6: Auslesen von Daten und dynamische Erzeugung von HTML-Tags (shop/chap6/ado_shop.cs)

Wie es funktioniert

Bis zur Definition der SQL-Abfrage entspricht der Code dem bereits gezeigten – es handelt sich lediglich um die Öffnung der Datenbankverbindung. Die Abfrage selbst ist sehr einfach gehalten:

```
string abfrage = "SELECT * FROM Artikel";
```

Dann wird ein Kommando-Objekt der Klasse `OleDbCommand` erzeugt. Als Parameter werden dem Konstruktor die Abfrage und das Verbindungs-Objekt übergeben:

```
OleDbCommand oComm = new OleDbCommand(abfrage, oConn);
```

Nun ist bekannt, welche Abfrage an welche Verbindung gesendet werden soll. Es fehlt nur der Auslöser dafür. Da hier lediglich lesend zugegriffen, reicht das `OleDbDataReader`-Objekt aus. Die Methode `ExecuteReader` erzeugt ein solches Objekt, ruft die Daten aus der Datenbank ab und weist sie dem Objekt zu. An dieser Stelle sind die Daten bereits in den von ASP.NET verwalteten Speicherbereichen:

```
OleDbDataReader oData = oComm.ExecuteReader();
```

Das Objekt `oData` spielt nun die Hauptrolle. Das Durchlaufen der Datensätze ist lediglich vorwärts und vom ersten zum letzten Datensatz möglich. Gezielte Zugriffe auf einzelne Zeilen oder andere Richtungen können Sie hier nicht verwenden. Dafür ist es sehr schnell und für die Musteranwendung ideal. Den einfachsten Weg, nacheinander alle Datensätze lesen zu können, bietet eine `while`-Schleife. Die Methode `Read` schaltet einen Datensatz weiter und liefert, wenn dies möglich war, `true` zurück. Liegen keine Daten mehr bereit, wird `false` zum Abbruch der Schleife führen:

```
while (oData.Read())
```

In der *aspx*–Datei wurde lediglich ein `<ul>`-Tag geschrieben. Da dieses mit `<li>`-Tags und natürlich den Links zum Bestellen gefüllt werden soll, wird auf entsprechende Methoden der Klasse `HtmlGenericControl` zurückgegriffen. Zuerst wird für jeden Durchlauf ein neues `<li>`-Tag erzeugt:

```
HtmlGenericControl produkt = new HtmlGenericControl("li");
```

Das Objekt erhält den Namen `produkt`. Außerdem wird ein `<a>`-Tag für den Link erzeugt; dieses Objekt trägt den Namen `plink`:

```
HtmlAnchor plink = new HtmlAnchor();
```

Dann wird der Link selbst aufgebaut. Dazu wird auf die Spalte *id* der Tabelle *Artikel* zugegriffen. Der Zugriff auf die Spalten ist in C# besonders einfach, weil die dafür zuständige Methode `Item` den Indexer der Klasse `OleDbDataReader` bildet:

```
plink.Href = "order.aspx?produkt=" + oData["id"].ToString();
```

Der Link hat nun die Form „order.aspx?produkt=1“ usw. Nun fehlt noch der Text, der als Link angezeigt werden soll. Dies ist natürlich der Name des Artikels; also wird die Spalte *name* verwendet:

```
plink.InnerText = oData["name"].ToString();
```

Nun wird dem `<li>`-Tag das `<a>`-Tag hinzugefügt:

```
produkt.Controls.Add(plink);
```

Anschließend wird die ganze Kombination `<li><a>` in `<ul>` eingebaut:

```
produkte.Controls.Add(produkt);
```

Das Ergebnis entspricht den Erwartungen. Beachten Sie in der folgenden Abbildung die Statuszeile, die den ausgewählten Hyperlink anzeigt.

Abbildung 6.16: Dynamisch erzeugte Produktliste

Ausgabe skalarer Werte

Wenn nur ein einzelner Wert einer Tabelle von Interesse ist, gibt es einen direkten Weg, dieses zu ermitteln – ohne `OleDbDataReader`. Die Datei *shop_s.aspx* verwendet diese Methode; sie entspricht jedoch ansonsten der zuletzt vorgestellten *shop.aspx* und verwendet nur eine andere Code-Datei:

```
string abfrage = "SELECT COUNT(*) FROM Artikel";
OleDbCommand oComm2 = new OleDbCommand(anzahl, oConn);
string AnzahlArtikel =  oComm2.ExecuteScalar().ToString();
produkte.InnerText = "Es wurden " + AnzahlArtikel + " Artikel gefunden";
```

Listing 6.7: Ausschnitt aus ado_shop_s.cs (der übrige Teil entspricht)

ExecuteScalar

Das Geheimnis steckt in der Methode `ExecuteScalar`. Damit wird lediglich der erste Wert der Abfrage abgerufen. Es ist natürlich sinnvoll, wenn die Abfrage auch nur einen Wert zurückgibt. Im Beispiel wird die Anzahl der Datensätze mit der SQL-Funktion `COUNT(*)` ermittelt.

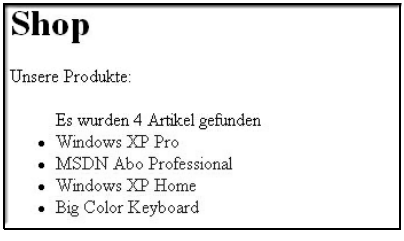


Abbildung 6.17: Ausgabe der Artikelanzahl über eine zweite Abfrage eines skalaren Wertes

Einfache Variablen, die nur einen Datenwert enthalten, werden allgemein als Skalare bezeichnet. Dazu gehören Zahlen, Zeichenketten und Zeichen, nicht jedoch Arrays oder Strukturen.

Nutzung spezialisierter Ausgabemethoden

Die Erzeugung von HTML-Tags in Abhängigkeit von Daten ist ein direkter und sehr universeller Weg der Ausgabe. Listen und Tabellen werden jedoch besonders häufig benötigt. Dafür besitzt ASP.NET einige spezialisierte Steuerelemente, die das so genannten Binden einer Datenquelle erlauben. Sie müssen dann nicht jedes Unterelement selbst erzeugen, sondern können dies dem Steuerelement selbst überlassen.

DataList

Für Tabellen setzen Sie am Anfang am Besten die Web Server-Steuerelemente `DataList` und `DataGrid` ein. Beide erwarten eine Liste von Daten, wie sie der Schnittstelle `IEnumerable` entsprechen. Solche Listen können auch mit Datenbankabfragen erstellt werden, sind aber auch mit `ArrayList` und `Hashtable` erzeugbar. Entsprechend müssen drei Schritte zur Nutzung ausgeführt werden:

1. Definition des Steuerelements und seiner Gestaltung
2. Festlegen der Datensenke (wo die Daten angezeigt werden)
3. Auswahl der Datenquelle (wo die Daten herkommen)

Die Gestaltung kann sehr umfassend erfolgen. Im einfachsten Fall genügt jedoch die Angabe des Steuerelements in HTML. Zuerst soll eine Umsetzung der `<ul>`-Liste aus dem letzten Beispiel mit `DataList` erfolgen, wobei folgende Definition erwartet wird:

```
<asp:datalist id="produkte" runat="server" />
```

HTML-Tabellen für Datenausgabe

Es entsteht eine einfache, nicht weiter gestaltete HTML-Tabelle. Eine Tabelle hat einen sehr strukturierten Aufbau. Sie kann folgende Elemente enthalten:

- Eine Kopfzeile
- Ein oder mehrere Zeilen mit den Daten
- Trennzeilen zwischen den Daten
- Eine Fußzeile

Sie können deshalb für jeden Teil der Ausgabe eine so genannte Vorlage (engl. Template) schreiben, die ihrerseits natürlich HTML enthält. Dazu dienen spezielle Tags, die nur innerhalb von `DataList` auftreten dürfen. Das folgende Beispiel zeigt eine Vorlage für die Produktliste:

```
<%@ Page Language="C#" debug="true" Inherits="shop" src="ado_shop_list.cs"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="de">
    <head>
        <title>Shop</title>
    </head>
    <body>
        <h1> Shop </h1>
        <asp:datalist id="produkte" runat="server">
            <headertemplate>
                Unsere Produktauswahl
            </headertemplate>
            <itemtemplate>
                <a href="orders.aspx?produkte=
                     <%# DataBinder.Eval(Container.DataItem, "id") %>">
                <%# DataBinder.Eval(Container.DataItem, "name") %>
                </a>
            </itemtemplate>
        </asp:datalist>
        <br/>
        <p id="isOpen" runat="server"/>
    </body>
</html>
```

Listing 6.8: Vorlage für die Ausgabe einer Produktliste

Wie es funktioniert

Die Gestaltung der Ausgabe wird durch die Tags `<headertemplate>` und `<itemtemplate>` gesteuert. Das erste legt die Ausgabe der Kopfzeile fest, das zweite findet für jeden Datensatz Anwendung. Darin steht – ganz einfach – HTML und – etwas komplizierter – der Abruf der Daten. Dazu verwendet ASP.NET eine besondere Syntax:

```
<%# Datenquelle %>
```

DataBinder.Eval Container.Item

Zwischen den ersten Zeichen `%` und `#` darf kein Leerzeichen stehen. Die Datenquelle kann allgemein immer über die statische Methode `Eval` der Klasse `DataBinder` erreicht werden. Diese Methode erwartet zwei oder drei Parameter. Der erste ist immer das Datenobjekt, das immer `Container.DataItem` heißt. Der zweite ist eine Zeichenkette, die den Spaltennamen aus der Tabelle repräsentiert. Der dritte, optionale Parameter kann eine Formatieranweisung enthalten, wie sie bereits in Tabelle 2.9 für Zahlen beschrieben wurde. Der Name des aktuellen Feldes wird also folgendermaßen in der Vorlage eingetragen:

```
<%# DataBinder.Eval(Container.DataItem, "name") %>
```

Die Änderungen in der Code-Datei, die den Abruf der Daten aus der Datenbank übernimmt, sind gegenüber den letzten Beispielen nur gering.

Der Aufbau ist insgesamt einfacher, da die Erzeugung einzelner HTML-Tags nun Dank der Vorlagen nicht mehr erforderlich ist.

```
using System;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using System.Data;
using System.Data.OleDb;

public class shop : Page
{
    public DataList produkte;
    public HtmlGenericControl isOpen;

    public void Page_Load()
    {
        string database = Server.MapPath("../shop.mdb");
        string provider = "PROVIDER=Microsoft.Jet.OLEDB.4.0; Data Source="
                          + database;
        OleDbConnection oConn = new OleDbConnection(provider);
        try
        {
            oConn.Open();
            string abfrage = "SELECT * FROM Artikel";
            OleDbCommand oComm = new OleDbCommand(abfrage, oConn);
            produkte.DataSource = oComm.ExecuteReader();
            produkte.DataBind();
            oConn.Close();
        }
        catch (OleDbException e)
        {
            isOpen.InnerText = "Verbindungsfehler: " + e.Message;
        }
    }
}
```

Listing 6.9: Abruf der Artikeldaten und Übergabe an DataList (shop/chap6/ado_shop_list.cs)

Wie es funktioniert

Die eigentliche Arbeit erledigen zwei Zeilen. Die Daten, die `ExecuteReader` erzeugt, werden direkt an die Eigenschaft `DataSource` des Web Server-Steuerelementes `DataList` übergeben:

```
produkte.DataSource = oComm.ExecuteReader();
```

Dann erfolgt noch die Datenbindung, die zur Anzeige aller Datensätze führt:

```
produkte.DataBind();
```

Nach der Ausführung wird die Liste angezeigt, wie gehabt mit den Hyperlinks. Erzeugt wird übrigens tatsächlich eine HTML-Tabelle, die vielfältig gestaltet werden kann:

```
<table id="produkte" cellspacing="0" border="0"
          style="border-collapse:collapse;">
<tr>
<td>
      Unsere Produktauswahl
    </td>
</tr><tr>
<td>
      <a href="orders.aspx?produkte=1">
      Windows XP Pro
      </a>
    </td>
</tr><tr>
<td>
      <a href="orders.aspx?produkte=2">
      MSDN Abo Professional
      </a>
    </td>
</tr><tr>
<td>
      <a href="orders.aspx?produkte=3">
      Windows XP Home
      </a>
    </td>
</tr><tr>
<td>
      <a href="orders.aspx?produkte=4">
      Big Color Keyboard
      </a>
    </td>
</tr>
</table>
```

Listing 6.10: Automatisch generierte HTML-Tabelle

Gestaltung der Ausgabe

Die Gestaltung kann mit einer ganzen Palette von Eigenschaften und Attributen erfolgen. Eine etwas komplexere Einleitung des Tags könnte folgendermaßen aussehen:

```
<asp:datalist id="produkte" runat="server"
              font-size="12pt" font-name="verdana"
              borderwidth="1">
<headerstyle backcolor="blue" forecolor="white"/>
...
```

Listing 6.11: Einfache Formatierung mit Attributen

Neben der direkten Gestaltung des gesamten Elements kann auch jeder Bereich der Ausgabe modifiziert werden. Das Tag `<headerstyle>` gestaltet den Kopfbereich, `<itemstyle>` die Elemente usw.

Die folgende Tabelle zeigt die Vorlagen und deren Gestaltungs-Tags auf einen Blick. Die Reihenfolge in der linken Spalte ist dann wichtig, wenn mehrere Gestaltungstags gleiche Attribute verwenden. Dann „gewinnt" die Anweisung, die in der Liste weiter oben steht; die Formatierung von `<EditItemStyle>` überschreibt also beispielsweise die von `<ItemStyle>`.

Priorität	Vorlage	Gestaltung
1	`<EditItemTemplate>`	`<EditItemStyle>`
2	`<SelectedItemTemplate>`	`<SelectedItemStyle>`
3	`<AlternatingItemTemplate>`	`<AlternatingItemStyle>`
4	`<ItemTemplate>`	`<ItemStyle>`
5	`<ControlTemplate>`	`<ControlStyle>`
Ohne	`<HeaderTemplate>`	`<HeaderStyle>`
Ohne	`<FooterTemplate>`	`<FooterStyle>`

Als letzter Schritt zu einer perfekten Gestaltung der Datenlisten folgt die Angabe der wichtigsten Attribute, die zugleich auch als Eigenschaften des entsprechenden Objekts zur Verfügung stehen:

Attribut bzw. Eigenschaft	Verwendungszweck
`BackColor`	Hintergrundfarbe
`BorderColor`	Randfarbe
`BorderStyle`	Randtyp, beispielsweise `solid` oder `dotted` (Punktlinie)
`BorderWidth`	Randbreite
`CssClass`	Name einer in CSS definierten Klasse
`Font`	Schriftart
`ForeColor`	Vordergrund- oder Schriftfarbe
`Height`	Höhe
`Width`	Breite
`HorizontalAlign`	Horizontale Ausrichtung
`VerticalAlign`	Vertikale Ausrichtung
`Wrap`	Behandlung von Zeilenumbrüchen.

Tabelle 6.5: Wichtige Attribute und Eigenschaften zur Gestaltung von Datenlisten

Die Parameter entsprechen denen in HTML üblichen Formen. Die Werte werden direkt nach HTML übertragen.

Abbildung 6.18: Gestaltete Ausgabe einer Datenliste

Für die in der Abbildung gezeigten Formatierungen wurden die im Listing 6.11 verwendeten Attribute eingesetzt.

6.5.2 Komfortabler Datenzugriff

Der sequenzielle Zugriff ist zwar häufig, reicht aber nicht immer aus. Die zweite Variante des Datenzugriffs erlaubt nicht einfach nur mehr Funktionen, sondern bietet gleich den vollen Zugriff. Das erscheint auf den ersten Blick einfacher, als komplizierte SQL-Kommandos zu entwerfen. Tatsächlich transportieren Sie aber nur die Struktur der Datenbank auf den Webserver, was eine bessere Integration in die Applikation erlaubt, oft schneller ist und flexibler sein kann. Ganz bestimmt ist es aber nicht einfacher. Deshalb gilt die Regel, dass Sie immer erst SQL in Kombination mit `DataReader` völlig ausreizen sollten, ehe andere Techniken zum Einsatz kommen. Kern dieser Techniken bilden so genannte `DataSet`-Objekte. Dies sind Kopien einer Datenbank, mit Tabellen, Beziehungen und Sichten. Verständlich, dass dies nicht einfacher ist, als SQL-Kommandos zu verwenden. Allerdings muss in einem `DataSet`-Objekt nicht zwangsläufig die gesamten Datenbank enthalten sein. Sie können selbst bestimmen, welcher Teil kopiert wird, was in der Regel auch eine gute Idee. Zusammen mit dem komplexeren Web Server-Steuerelement `DataGrid` können jedoch mit wenigen Zeilen Code komplexe Programme geschrieben werden. Dieser Abschnitt zeigt die Komplettierung der Shopanwendung mit diesen Klassen.

Transparenter Zugriff auf die Datenbank

Wie am Anfang bereits erwähnt, ist es mit ADO.NET möglich, ein Abbild der gesamten Datenbank oder eines bestimmten Teils daraus in Objekten vorzuhalten. Dies hat den Vorteil, dass der Datenverkehr zwischen ASP.NET und dem Datenbankmanagementsystem reduziert werden kann. Dieser Weg ist immer dann besser als der mit `DataReader` beschriebene, wenn die Daten umfassend bearbeitet werden müssen. Die folgende Abbildung zeigt grob und vereinfacht die Zusammenhänge anhand der Beispieldaten. Selbstverständlich ist die Möglichkeit,

mehrere Tabellen zu verwenden, eine Option. Außerdem fehlen die Objekte zur Definition von Beziehungen und einige aus Hilfsklassen abgeleitete, die vor allem der Definition neuer Elemente dienen (was Sie bisher mit Access direkt erledigt haben).

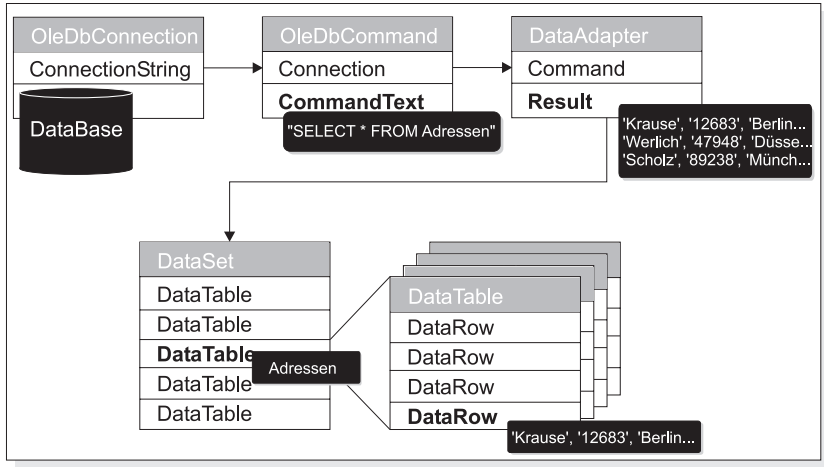


Abbildung 6.19: Vereinfachtes Diagramm der Abhängigkeiten der Objekte

Für die ersten Schritte mit Datenbanken reicht es jedoch aus, über `DataSet`, `DataTable` und `DataRow` zu verfügen.

Komfortables Bearbeiten von Tabellen

Das folgende Beispiel realisiert einen kleinen Warenkorb – der letzte Schritt zum vollständigen Shop. In der Vergangenheit war die Programmierung eines solchen Programms sehr aufwändig, wenn man bedenkt, das jederzeit Artikel hinzugefügt, entfernt oder in der Menge geändert werden können. Verwendet wird zum Datenbankzugriff `DataSet`, zur Darstellung `DataList` und `DataGrid` und zur Steuerung der Abfragen SQL. Der Quelltext wird im Anschluss an die Übersicht ausführlich diskutiert.

```
<%@ Page Language="C#" debug="true" Inherits="shop" src="ado_basket2.cs" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="de">
    <head>
        <title>Shop</title>
    </head>
    <body>
        <h1> Shop </h1>
        <asp:datalist id="produkte" runat="server" font-size="12pt"
              font-name="verdana" borderwidth="1">
            <headerstyle backcolor="blue" forecolor="white"/>
```

```
    <headertemplate>
        Unsere Produktauswahl:
    </headertemplate>
    <itemtemplate>
        <a id="produktlink" runat="server"
        href='<%# "shop_basket.aspx?order="
              + DataBinder.Eval(Container.DataItem, "id")
              + "&amp;price=" + DataBinder.Eval(Container.DataItem,
                  "preis") %>'
        innertext='<%# DataBinder.Eval(Container.DataItem,
                      "name") %>' />
    </itemtemplate>
</asp:datalist>
<br/>
Klicken Sie auf ein Produkt, um es in den Warenkorb zu legen
<h3 id="isOpen" runat="server"/>
<form runat="server">
<asp:datagrid id="warenkorb" runat="server"
              autogeneratecolumns="false"
              cellpadding="2"
              datakeyfield="BestellID"
              oneditcommand="bearbeiteKorb"
              edititemstyle-backcolor="yellow"
              ondeletecommand="loescheKorb"
              oncancelcommand="abbrecheKorb"
              onupdatecommand="aktualisiereKorb">
    <columns>
        <asp:boundcolumn headertext="ID"
                         datafield="BestellID" readonly="true"/>
        <asp:templatecolumn headertext="Name">
            <headerstyle backcolor="silver"/>
            <itemtemplate>
                <%# DataBinder.Eval(Container.DataItem, "name") %>
            </itemtemplate>
        </asp:templatecolumn>
        <asp:templatecolumn headertext="Preis">
            <headerstyle backcolor="silver"/>
            <itemstyle horizontalalign="right"/>
            <itemtemplate >
                <%# String.Format("{0:N}",
                                  DataBinder.Eval(Container.DataItem,
                                                  "Preis")) %>
            </itemtemplate>
        </asp:templatecolumn>
        <asp:templatecolumn headertext="Menge">
            <headerstyle backcolor="silver"/>
```

```
                    <itemtemplate>
                        <%# DataBinder.Eval(Container.DataItem, "Menge") %>
                    </itemtemplate>
                    <edititemtemplate>
                        <asp:textbox id="Menge" width="30px" runat="server"
                                text='<%# DataBinder.Eval(Container.DataItem,
                                      "Menge") %>'/>
                        <asp:comparevalidator id="PruefMenge" type="Integer"
                                    runat="server" display="dynamic"
                                    controltovalidate="Menge"
                                    valuetocompare="1"
                                    operator="GreaterThanEqual"
                                errormessage="<br/>Menge unzul&auml;ssig"/>
                        <asp:requiredfieldvalidator id="IstMenge"
                                    runat="server" display="dynamic"
                                    controltovalidate="Menge"
                                errormessage="<br/>Feld muss ausgef&uuml;llt
                                                werden"/>
                    </edititemtemplate>
                </asp:templatecolumn>
                <asp:templatecolumn headertext="Gesamt">
                    <headerstyle backcolor="silver"/>
                    <itemstyle horizontalalign="right"/>
                    <itemtemplate>
                        <%# String.Format("{0:N}",
                                    DataBinder.Eval(Container.DataItem,
                                    "Summe")) %>
                    </itemtemplate>
                </asp:templatecolumn>
                <asp:editcommandcolumn
                        headertext="Aktion"
                        edittext="Bearbeiten"
                        canceltext="Abbrechen"
                        updatetext="Aktualisieren" />
                <asp:buttoncolumn
                        ButtonType="LinkButton"
                        Text="L&ouml;schen"
                        CommandName="Delete"/>
            </columns>
        </asp:datagrid>
        </form>
    </body>
</html>
```

Listing 6.12: HTML-Vorlage für den Warenkorb mit Artikelauswahl (shop/chap6/shop_basket.aspx)

Wie es funktioniert

Hier gibt es eine ganze Reihe von Besonderheiten, die verwendet werden. Am Anfang steht jedoch das bereits vorgestellte `DataList`-Control. Erweitert wurde die Übertragung der Parameter per Querystring. Hier gibt es mehrere Lösungswege, von denen GET-Parameter der direkteste ist. Durch einen Klick auf einen Link soll der betreffende Artikel der Bestell-Tabelle hinzugefügt werden. Entsprechend wird das Attribut `href` des `<a>`-Tags an die passenden Daten gebunden:

```
href='<%# "shop_basket.aspx?order=" +
      DataBinder.Eval(Container.DataItem, "id") +
      "&price=" +
      DataBinder.Eval(Container.DataItem, "preis") %>'
```

Daraus entstehen dann die Links in der folgenden Form:

```
<a href="/aspdotnet/shop/chap6/shop_basket.aspx?order=1&price=199,99"
id="produkte__ctl1_produktlink">Windows XP Pro</a>
```

DataGrid

Im nächsten Abschnitt wird der Warenkorb angezeigt. Dazu wird ein `DataGrid` eingesetzt. Dieses Steuerelement ist sehr komplex und leistungsfähig. Das Beispiel kann nur das Prinzip demonstrieren; wenn Sie `DataGrid` selbst einsetzen, ziehen Sie die Referenz zu Rate, die alle möglichen Eigenschaften und Methoden enthält. Das Tag selbst kann mehrere Attribute haben, die das Verhalten steuern. Zuerst die Standardattribute:

```
<asp:datagrid id="warenkorb" runat="server"
```

Dann wird festgelegt, dass Spalten nicht automatisch generiert werden. Standardmäßig (Parameter gleich `true`) werden diese in Abhängigkeit von der Datenquelle erzeugt:

```
autogeneratecolumns="false"
```

Dann wird – als Beispiel für die vielfältigen Gestaltungsattribute – der innere Abstand des Textes zum Zellenrand auf 2 Pixel festgelegt, außerdem soll sich die Hintergrundfarbe ändern, wenn ein Datensatz bearbeitet wird:

```
cellpadding="2"
edititemstyle-backcolor="yellow"
```

Weitere Attribute zur Gestaltung können Sie Tabelle 6.5 entnehmen.

Damit die Auswahl von Zeilen gelingt, ist ein Schlüsselwert erforderlich. Das folgende Attribut verknüpft eine der Spalten des Abfrageergebnisses der Datenquelle mit der Schlüsselspalte; hier mit `BestellID`:

```
datakeyfield="BestellID"
```

Nun werden die Methoden bestimmt, die bestimmte Aktionen ausführen sollen. Die Programmierung selbst erfolgt im Code-Teil. Zuerst das Kommando zum Bearbeiten der Daten:

```
oneditcommand="bearbeiteKorb"
```

Erst wenn „Bearbeiten“ gewählt wurde, erscheinen die beiden folgenden Optionen. Dann folgt die Definition der Verarbeitung eines Abbruchwunsches:

```
oncancelcommand="abbrecheKorb"
```

Selbstverständlich lassen sich Einträge auch löschen:

```
ondeletecommand="loescheKorb"
```

Zuletzt die Aktualisierung der Werte im Bearbeitungsmodus:

```
onupdatecommand="aktualisiereKorb"
```

Jetzt werden die Spalten definiert:

```
<columns>
```

Zuerst eine einfache Spalte vom Typ `boundcolumn`. Hier stehen keine Vorlagen zur Verfügung; ASP.NET wird deshalb versuchen, die Anzeige in allen Situationen selbst zu gestalten. Das ist bestenfalls für solch eine einfache ID-Spalte vertretbar:

```
<asp:boundcolumn headertext="ID" datafield="BestellID" readonly="true"/>
```

Gebunden wird die Spalte an das Feld *BestellID*; der Bearbeitungsmodus wird mit `readonly` unterdrückt.

Nun eine einfache Vorlage vom Typ `templatecolumn`. Das Attribut `headertext` bestimmt die Spaltenüberschrift:

```
<asp:templatecolumn headertext="Name">
```

Innerhalb der Definition einer Spalte kann – ähnlich wie bei `DataList` – jeweils eine Vorlage für jeden Zustand der Spalten definiert werden. So wird zwischen der normalen Darstellung und der Form zum Bearbeiten unterschieden, oder auch nur der ausgewählte Zustand hervorgehoben. Die ersten Spalte enthält nur eine Darstellung des Inhalts – ohne Variationen. Deshalb kommt nur `<itemtemplate>` zum Einsatz. Als Gestaltungsbeispiel für die Spalte wird die Hintergrundfarbe des Kopffeldes verändert:

```
<headerstyle backcolor="silver"/>
```

Als einziger Inhalt wird die Datenbindung an die entsprechende Spalte der SQL-Abfrage vorgegeben. `DataGrid` benötigt keine weiteren Angaben,

weil automatisch eine HTML-Tabelle erstellt wird, die Daten also zwischen <td>-Tags platziert werden:

```
<itemtemplate>
     <%# DataBinder.Eval(Container.DataItem, "name") %>
</itemtemplate>
```

Die folgende Spalte für den Preis wird ebenso definiert. Umfangreicher ist die Mengenangabe, denn diesen Wert kann der Benutzer nachträglich ändern. Die Darstellung im Standardmodus gleicht den anderen Spalten:

```
<itemtemplate>
     <%# DataBinder.Eval(Container.DataItem, "Menge") %>
</itemtemplate>
```

Zusätzlich wird jedoch eine weitere Vorlage für den Bearbeitungszustand benötigt:

```
<edititemtemplate>
```

Hier kann alles eingebaut werden, was für ein Formularfeld erforderlich ist. Am sichersten sind Web Server-Steuerelemente in Kombination mit Kontroll-Steuerelementen, damit gleich eine Prüfung des Feldinhaltes stattfinden kann. Zuerst das Eingabefeld für die Menge:

```
<asp:textbox id="Menge" width="30px" runat="server"
```

Der Inhalt wird durch die Datenbindung bestimmt:

```
text='<%# DataBinder.Eval(Container.DataItem, "Menge") %>'/>
```

Es schließen sich die beiden Kontroll-Steuerelemente an, die sich nicht von den bereits in „Kontroll-Steuerelemente (Validation Controls)“ auf Seite 240 beschriebenen unterscheiden.

Umgangreicher ist die Klasse, die die den programmtechnischen Teil übernimmt:

```
using System;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using System.Data;
using System.Data.OleDb;

public class shop : Page
{
    public DataList produkte;
    public HtmlGenericControl isOpen;
```

```
public HtmlAnchor produktlink;
public TextBox NeueMenge, Menge;
public DataGrid warenkorb;
private string abfrage;
private OleDbCommand oComm;
private OleDbConnection oConn;
private DataSet oDataSet;
private OleDbDataAdapter oAdapterKorb, oAdapterArtikel, oBestellAdapter;
private DataView oZeigeKorb, oZeigeArtikel;

public shop()
{
    string database = Server.MapPath("../shop.mdb");
    string provider = "PROVIDER=Microsoft.Jet.OLEDB.4.0; Data Source="
                      + database;
    oConn = new OleDbConnection(provider);
    oDataSet = new DataSet();
    // Artikelliste
    abfrage = "SELECT * FROM Artikel";
    oAdapterArtikel = new OleDbDataAdapter(abfrage, oConn);
    oAdapterArtikel.Fill(oDataSet, "ArtikelListe");
}

private void zeigeKorb()
{
    // Warenkorb
    abfrage  = @"SELECT Artikel.name AS Name ,
                        Artikel.preis AS Preis,
                        Bestellungen.id AS BestellID, ";
    abfrage += "Bestellungen.summe AS Summe,
                Bestellungen.menge AS Menge ";
    abfrage += "FROM Bestellungen
                INNER JOIN Artikel ON Bestellungen.[a-id]=Artikel.id ";
    abfrage += "WHERE Bestellungen.[k-id] = 1";
    oAdapterKorb = new OleDbDataAdapter(abfrage, oConn);
    oAdapterKorb.Fill(oDataSet, "Warenkorb");
    oZeigeKorb = new DataView(oDataSet.Tables["Warenkorb"]);
    warenkorb.DataSource = oZeigeKorb;
    warenkorb.DataBind();
}

public void Page_Load()
{
    try
    {
        oConn.Open();
```

```
        oZeigeArtikel = new DataView(oDataSet.Tables["ArtikelListe"]);
        oZeigeArtikel.Sort = "id";
        produkte.DataSource = oZeigeArtikel;
        produkte.DataBind();
        if (!Page.IsPostBack)
        {
           // Artikel hinzufuegen und Warenkorb anzeigen
           int ArtikelID = Convert.ToInt32(Request.QueryString["order"]);
           if (ArtikelID > 0)
           {
                int i = oZeigeArtikel.Find(ArtikelID);
                double ArtikelPreis =
                   Convert.ToDouble(oZeigeArtikel[i].Row["preis"]);
                isOpen.InnerHtml = @"HINWEIS: F&uuml;ge Artikel "
                                 + ArtikelID.ToString() + " hinzu.";
                abfrage  = "INSERT INTO Bestellungen ([k-id], [a-id],
                           menge, summe) ";
                abfrage += "VALUES (?, ?, ?, ?)";
                oComm = new OleDbCommand();
                oComm.CommandText = abfrage;
                oComm.Connection = oConn;
                oComm.Parameters.Add(new OleDbParameter("kid",
                                     OleDbType.Integer));
                oComm.Parameters.Add(new OleDbParameter("aid",
                                     OleDbType.Integer));
                oComm.Parameters.Add(new OleDbParameter("menge",
                                     OleDbType.Double));
                oComm.Parameters.Add(new OleDbParameter("summe",
                                     OleDbType.Double));
                oComm.Parameters["kid"].Value = 1;
                oComm.Parameters["aid"].Value = ArtikelID;
                oComm.Parameters["menge"].Value = 1;
                oComm.Parameters["summe"].Value = ArtikelPreis;
                oComm.ExecuteNonQuery();
            }
            zeigeKorb();
        }
        oConn.Close();
    }
    catch (OleDbException e)
    {
        isOpen.InnerText = "FEHLER: " + e.Message;
        isOpen.InnerHtml += " " + oComm.CommandText;
    }
}
```

```
public void bearbeiteKorb(object Sender, DataGridCommandEventArgs e)
{
    isOpen.InnerText = @"HINWEIS: Bearbeite Artikel Nr. "
                     + warenkorb.DataKeys[e.Item.ItemIndex].ToString();
    warenkorb.EditItemIndex = e.Item.ItemIndex;
    zeigeKorb();
}

public void aktualisiereKorb(object Sender, DataGridCommandEventArgs e)
{
    if (Page.IsValid)
    {
        NeueMenge = (TextBox) e.Item.FindControl("Menge");
        int index = e.Item.ItemIndex;
        isOpen.InnerText = @"HINWEIS: Aktualisiere "
                         + warenkorb.DataKeys[index] + " mit Menge "
                         + NeueMenge.Text;
        abfrage  = "UPDATE Bestellungen SET ";
        abfrage += "menge = " + NeueMenge.Text + ", ";
        abfrage += "summe = summe/menge * " + NeueMenge.Text + " ";
        abfrage += "WHERE id = " + warenkorb.DataKeys[index];
        oConn.Open();
        oComm = new OleDbCommand(abfrage, oConn);
        oComm.CommandText = abfrage;
        oComm.ExecuteNonQuery();
        oConn.Close();
    } else {
        isOpen.InnerText = "FEHLER: Aktualisieren geht nicht,
                            Felder falsch";
    }
    warenkorb.EditItemIndex = -1;
    zeigeKorb();
}

public void loescheKorb(object Sender, DataGridCommandEventArgs e)
{
    int index = e.Item.ItemIndex;
    abfrage = "DELETE FROM Bestellungen WHERE id = "
              + warenkorb.DataKeys[index];
    isOpen.InnerText = @"HINWEIS: Bestellung "
                     + warenkorb.DataKeys[index].ToString()
                     + " wurde entfernt";
    oComm = new OleDbCommand(abfrage, oConn);
    oConn.Open();
    oComm.ExecuteNonQuery();
    oConn.Close();
```

```
        warenkorb.EditItemIndex = -1;
        zeigeKorb();
    }

     public void abbrecheKorb(object Sender, DataGridCommandEventArgs e)
    {
        isOpen.InnerHtml = @"HINWEIS: &Auml;nderung an "
                           + warenkorb.DataKeys[e.Item.ItemIndex].ToString()
                           + " abgebrochen.";
        warenkorb.EditItemIndex = -1;
        zeigeKorb();
    }

}
```

Listing 6.13: Cod Behind-Klasse für den Warenkorb (shop/chap6/ado_basket.cs)

Eine systematische Betrachtung der Funktionsweise hilft, die Techniken leicht auf eigene Projekte zu übertragen. Der einleitende Teil der Klasse, das Erben von `Page` und die Deklaration öffentlicher und interner Variablen weist keine Besonderheiten auf.

Ausgabe der Artikelliste

Wie es funktioniert

Der Start der Abarbeitung erfolgt diesmal mit einem Konstruktor, der noch vor `Page_Load` ausgeführt wird:

```
public shop()
```

Konstruktoren entstehen in C#, wenn eine Methode den Namen der Klasse trägt. Hier werden die Datenbankverbindung vorbereitet und die Basistabellen abgefragt. Zuerst die Festlegung der Datenbank:

```
string database = Server.MapPath("../shop.mdb");
```

Auf dem physischen Pfad aufbauend wird die Verbindungszeichenkette definiert:

```
string provider = "PROVIDER=Microsoft.Jet.OLEDB.4.0;
Data Source=" + database;
```

Dann wird ein Verbindungs-Objekt `oConn` erstellt, das im gesamten Programm benutzt wird:

```
oConn = new OleDbConnection(provider);
```

Außerdem wird ein `DataSet`-Objekt erzeugt.

Dieses soll alle Daten während der Verarbeitung der Seite aufnehmen:

```
oDataSet = new DataSet();
```

Die Artikelliste wird immer angezeigt. Da sie von keinen Bedingungen abhängig ist, erfolgt dies gleich im Konstruktor. Zuerst die Abfrage selbst:

```
abfrage = "SELECT * FROM Artikel";
```

Dann wird ein Adapter erstellt, der Abfrage und Verbindung verbindet:

```
oAdapterArtikel = new OleDbDataAdapter(abfrage, oConn);
```

Über diesen Adapter können die Daten nun in das `DataSet`-Objekt transportiert werden. Dies erfolgt mit der Methode `Fill`, die zugleich den internen Namen der Tabelle mit den Abfrageergebnissen festlegt:

```
oAdapterArtikel.Fill(oDataSet, "ArtikelListe");
```

Der Konstruktor ist nun beendet. ASP.NET wird jetzt die HTML-Seite analysieren und das Objektmodell erstellen. Wurde die gesamte Seite geladen, wird `Page_Load` aufgerufen. Hier wird zuerst die vorbereitete Datenbankverbindung geöffnet:

```
oConn.Open();
```

Dann wird eine Datensicht der Daten der Tabelle *ArtikelListe* – die nur im `DataSet` existiert – erzeugt:

```
oZeigeArtikel = new DataView(oDataSet.Tables["ArtikelListe"]);
```

Da später auf Elemente der *Artikeliste* zugegriffen werden soll, erfolgt gleich eine Sortierung. Um innerhalb einer `DataView` suchen zu können, wird eine Sortierung vorausgesetzt:

```
oZeigeArtikel.Sort = "id";
```

Diese Datensicht wird als Datenquelle für das Element *produkte* – Typ `DataList` – verwendet:

```
produkte.DataSource = oZeigeArtikel;
```

Mit dem Binden der Daten erfolgt die Anzeige:

```
produkte.DataBind();
```

An dieser Stelle erscheint schon die Ausgabe auf der HTML-Seite (auch wenn Sie dies einzeln nicht beobachten können):

Abbildung 6.20: Ausgabe der Warenliste

Jeder Klick auf einen Link soll nun dazu führen, dass der entsprechende Artikel in den Warenkorb gelegt wird.

An dieser Stelle sind zwei Zustände zu unterscheiden. Entweder es wurde bereits auf einen Link geklickt, oder eine der Funktionen im Warenkorb wurde ausgelöst. Ist letzteres der Fall, ist `IsPostBack` gleich `true`, denn der Warenkorb wird über ein Formular bedient. Mit der folgenden Abfrage beginnt also die Abarbeitung des Klicks auf einen Artikel:

```
if (!Page.IsPostBack)
```

Hier wird zuerst die Artikelnummer aus dem GET-Parameter `orders` ermittelt und in der Variablen `ArtikelID` gespeichert.

```
int ArtikelID = Convert.ToInt32(Request.QueryString["order"]);
```

Es folgt noch eine Prüfung, ob die Nummer gültig ist:

```
if (ArtikelID > 0)
```

Ist auch das der Fall, kann die Bestellung in die Bestell-Tabelle mit aufgenommen werden. Dazu wird noch der Preis ermittelt, der im selben `DataSet` in der Sicht *oZeigeArtikel* steckt. Diese Vorgehensweise spart eine weitere Abfrage der Daten. Zuerst wird hier der Index anhand der Spalte *id* ermittelt. Das funktioniert, weil zuvor die Sortierung mit `Sort = "id"` erfolgte:

```
int i = oZeigeArtikel.Find(ArtikelID);
```

Mit dem Index wird nun die Zeile *i* und dort die Spalte *preis* ermittelt:

```
double ArtikelPreis = Convert.ToDouble(oZeigeArtikel[i].Row["preis"]);
```

Jetzt schließt sich die Ausgabe eines Hinweises an, der nur für die Demonstration interessant ist. Danach wird das SQL-Kommando zum Einfügen des Datensatzes vorbereitet. Gearbeitet wird hier direkt gegen die Datenbank, nicht mit dem `DataSet`:

```
abfrage  = "INSERT INTO Bestellungen ([k-id], [a-id], menge, summe) ";
abfrage += "VALUES (?, ?, ?, ?)";
```

Interessant ist die Angabe der variablen Werte mit Fragezeichen – dies sind Parameter, die anschließend zugewiesen werden. Freilich können Sie auch die Variablen direkt einbauen, dann entstehen allerdings hässliche Schlangen der Art `"'" + variable + "'"`, usw. Dieses Gemengelage aus Anführungszeichen, Pluszeichen und Namen ist schwer lesbar und führt leicht zu Fehlern.

Parameter

Eleganter ist die Nutzung der `Parameters`-Kollektion des Objekts `OleDbCommand`. Zuvor muss dieses jedoch erzeugt und vorbereitet werden. Dazu ist der Abfragetext und die zu verwendende Verbindung zuzuweisen:

```
oComm = new OleDbCommand();
oComm.CommandText = abfrage;
oComm.Connection = oConn;
```

Jetzt werden die Parameter hinzugefügt. Für die ID der *Adressen*-Tabelle sieht das folgendermaßen aus:

```
oComm.Parameters.Add(new OleDbParameter("kid", OleDbType.Integer));
```

Der Konstruktor von `OleDbParameter` kennt verschiedene Überladungen, die gezeigte Anwendung ist die einfachste – es wird nur der Name und der Datentyp festgelegt. Andere Parameter steuern unter anderem die Ausgaberichtung, was für gespeicherte Prozeduren (eine SQL-Technik) benötigt wird. Nachdem alle Parameter bekannt sind, werden ihnen die Werte zugewiesen:

```
oComm.Parameters["kid"].Value = 1;
```

Der Einfachheit halber wird in dieser Applikation die Kundennummer nicht verwendet, die Anzeige erfolgt durch konstante Werte nur für die Adresse mit einer ID gleich 1. Die Erweiterung ist Bestandteil der Übungen dieses Abschnitts.

Sind alle Parameter mit Werten gefüllt, wird das Kommando ausgeführt:

```
oComm.ExecuteNonQuery();
```

Nun wird noch der Warenkorb angezeigt – natürlich bereits in der aktualisierten Form:

```
zeigeKorb();
```

Der Rest der Methode kümmert sich um das Schließen der Verbindung und gegebenenfalls um die Behandlung von Fehlern im `catch`-Zweig.

Anzeige des Warenkorbs

Die Anzeige des Warenkorbs erfolgt in der Methode `zeigeKorb`. Das SQL-Kommando verknüpft zwei Tabellen miteinander, denn es werden die Artikeldaten zur Anzeige benötigt. Außerdem erfolgt die Auswahl der Bestelldaten nur für den aktuellen Kunden, dessen ID hier unveränderlich mit 1 festgelegt wird. Isoliert betrachtet, sieht das Kommando folgendermaßen aus:

```
SELECT rtikel.name AS Name , Artikel.preis AS Preis,
       Bestellungen.id AS BestellID,
       Bestellungen.summe AS Summe, Bestellungen.menge AS Menge
       FROM Bestellungen
       INNER JOIN Artikel ON Bestellungen.[a-id]=Artikel.id
       WHERE Bestellungen.[k-id] = 1
```

Die Spalten der Tabellen werden direkt über „Tabelle.Spalte“ ausgewählt und mit dem Alias-Operator `AS` unter einem eindeutigen Namen bereitgestellt. Mit dieser Abfrage, die im Code in der Variablen `abfrage` gespeichert wird, kann nun ein neuer Adapter erstellt werden:

```
oAdapterKorb = new OleDbDataAdapter(abfrage, oConn);
```

Dieser füllt das schon existente `DataSet`-Objekt (`oDataSet`) und bekommt dort den Namen *Warenkorb*:

```
oAdapterKorb.Fill(oDataSet, "Warenkorb");
```

Damit der Warenkorb auch angezeigt werden kann, wird ein `DataView`-Objekt erzeugt:

```
oZeigeKorb = new DataView(oDataSet.Tables["Warenkorb"]);
```

Diese Sicht auf die Daten wird dann an das `DataGrid`-Element aus der HTML-Vorlage übergeben:

```
warenkorb.DataSource = oZeigeKorb;
```

Zuletzt erfolgt wieder die Bindung, und die Daten erscheinen:

```
warenkorb.DataBind();
```

Nach einigen Klicks auf die Artikelliste ergibt sich etwa folgendes Bild:

HINWEIS: Füge Artikel 4 hinzu.

ID	Name	Preis	Menge	Gesamt	Aktion	
54	Windows XP Pro	199,99	5	199,99	Bearbeiten	Löschen
67	MSDN Abo Professional	2.850,00	1	2.850,00	Bearbeiten	Löschen
71	MSDN Abo Professional	2.850,00	1	2.850,00	Bearbeiten	Löschen
72	Big Color Keyboard	99,00	1	99,00	Bearbeiten	Löschen

Abbildung 6.21: Warenkorb nach dem Hinzufügen einiger Artikel

Löschen von Artikeln im Warenkorb

Aufgrund der Definition der Vorlage erscheinen bereits einige Funktionen in Form von Links. Am einfachsten ist die Löschfunktion zu implementieren. Im `DataGrid` wurde der entsprechende Link mit der Methode `loescheKorb` verknüpft. Diese Methode wird aufgerufen, wenn Sie auf „Löschen“ klicken. Zuerst die Definition der Methode:

```
public void loescheKorb(object Sender, DataGridCommandEventArgs e)
```

Als Ereignis `e` werden Informationen über den Auslösepunkt in der Tabelle übergeben. Das `DataGrid` zählt die Zeilen der Tabelle, die angezeigt wird, mit Null beginnend durch. Dieser Index wird folgendermaßen ermittelt:

```
int index = e.Item.ItemIndex;
```

Dann wird das SQL-Kommando gebildet, hier wird der Schlüsselwert (die *BestellID*, festgelegt durch `datakeyfield="BestellID"` im `DataGrid`-Tags) benötigt:

```
abfrage = "DELETE FROM Bestellungen WHERE id = " + warenkorb.DataKeys[index];
```

Nach dem demonstrativen Hinweis wird wieder ein Kommando-Objekt erzeugt und eine entsprechende Abfrage an die Datenbank gesendet. Zuletzt wird wieder der Warenkorb aktualisiert.

HINWEIS: Bestellung 71 wurde entfernt

ID	Name	Preis	Menge	Gesamt	Aktion	
54	Windows XP Pro	199,99	5	199,99	Bearbeiten	Löschen
67	MSDN Abo Professional	2.850,00	1	2.850,00	Bearbeiten	Löschen
72	Big Color Keyboard	99,00	1	99,00	Bearbeiten	Löschen

Abbildung 6.22: Reaktion auf das Löschen eines Artikels

Verändern der Bestellmenge eines Artikels

Da mit dem Hinzufügen eines Artikels dieser immer mit der Menge 1 aufgenommen wird, wäre eine Funktion hilfreich, mit der sich die Menge nachträglich ändern lässt. Im `DataGrid` wurde dafür bereits die Spalte *Menge* vorbereitet, indem als `EditItemTemplate` ein Element vom Typ `TextBox` vorgesehen wurde. Zuvor muss jedoch in den entsprechenden Modus umgeschaltet werden, denn standardmäßig ist kein Eingabefeld zu sehen. Klicken Sie auf den Link „Bearbeiten“, wird automatisch das alternative Layout – soweit es definiert wurde – ausgegeben. Programmtechnisch wird lediglich der Hinweis erneuert. Dies dient nur der Demonstration. Wichtiger ist die Übergabe des gewählten Index an

das Element `DataGrid`. In der Methode, die den Klick auf „Bearbeiten“ verarbeitet, wird dazu der Index übertragen:

```
warenkorb.EditItemIndex = e.Item.ItemIndex;
```

Dann wird der Warenkorb neu angezeigt. Es ergibt sich etwa folgende Ausgabe:

HINWEIS: Bearbeite Artikel Nr. 73

ID	Name	Preis	Menge	Gesamt	Aktion	
73	Windows XP Pro	199,99	1	199,99	Aktualisieren Abbrechen	Löschen
67	MSDN Abo Professional	2.850,00	1	2.850,00	Bearbeiten	Löschen
72	Big Color Keyboard	99,00	1	99,00	Bearbeiten	Löschen

Abbildung 6.23: Bearbeitungsmodus für eine Zeile des Warenkorbs

Eine zweite Methode wird benötigt, um die Änderung selbst zu verarbeiten. Ein Klick auf „Aktualisieren“ ruft die Methode `aktualisiereKorb` auf.

Genaugenommen handelt es sich hier um Ereignisse, deshalb findet man in der Literatur oft auch Schreibweisen wie „feuert die Methode“. Die Formulierung ist aus sprachlicher Sicht aber eher unglücklich.

Die Definition der Methode weist keine Besonderheiten auf:

```
public void aktualisiereKorb(object Sender, DataGridCommandEventArgs e)
```

Innerhalb erfolgt noch eine Abfrage, ob die Kontroll-Steuerelemente tatsächlich erfüllt waren:

```
if (Page.IsValid)
```

FindControl

Ist das der Fall, ist die neue Menge zu ermitteln. Das `TextBox`-Steuerelement wird über die Methode `FindControl` ermittelt. Das funktioniert über Spalten, die durch Vorlagen definiert wurden (`TemplateColumn`). Die einfacheren `BoundColumn`-Spalten lassen sich nur über numerische Indizes ansprechen:

```
NeueMenge = (TextBox) e.Item.FindControl("Menge");
```

Natürlich wird auch hier wieder der allgemeine Index benötigt:

```
int index = e.Item.ItemIndex;
```

Dann erfolgt die Ausgabe des Hinweises und danach die Definition des SQL-Kommandos. Hier wird der Inhalt des Mengen-Feldes und die ID des betroffenen Artikels benötigt:

```
UPDATE Bestellungen SET
       "menge = " + NeueMenge.Text + ", "
       "summe = summe/menge * " + NeueMenge.Text "
       "WHERE id = " + warenkorb.DataKeys[index]
```

HINWEIS: Bearbeite Artikel Nr. 73

ID	Name	Preis	Menge	Gesamt	Aktion	
73	Windows XP Pro	199,99	1	199,99	Aktualisieren Abbrechen	Löschen
67	MSDN Abo Professional	2.850,00	1	2.850,00	Bearbeiten	Löschen
72	Big Color Keyboard	99,00	1	99,00	Bearbeiten	Löschen

Abbildung 6.24: Warenkorb im Bearbeitungs-Modus

Die Abfrage an die Datenbank entspricht den bereits gezeigten Versionen. Zum Schluss ist der Index noch aufzuheben, damit das Bearbeitungsfeld wieder verschwindet und die Standardsicht angezeigt werden kann. Der Wert -1 setzt den Zeiger außerhalb der Tabelle:

```
warenkorb.EditItemIndex = -1;
```

Die Methode endet mit der Anzeige des aktualisierten Warenkorbs.

Bearbeitung abbrechen

Dieselbe Technik – ohne jeden Einsatz von SQL – verwendet die Methode `abbrecheKorb`, die gestartet wird, wenn der Link „Abbrechen“ gewählt wird. Der Hinweis zeigt, dass der Aufruf korrekt erfolgt:

HINWEIS: Änderung an 73 abgebrochen.

ID	Name	Preis	Menge	Gesamt	Aktion	
73	Windows XP Pro	199,99	1	199,99	Bearbeiten	Löschen
67	MSDN Abo Professional	2.850,00	1	2.850,00	Bearbeiten	Löschen
72	Big Color Keyboard	99,00	1	99,00	Bearbeiten	Löschen

Abbildung 6.25: Abbruch der Bearbeitung

Damit ist das Warenkorb-Programm bereits beendet. Auf den ersten Blick erscheint der Einsatz von `DataGrid` und `DataSet` zwar kompliziert, die erreichte Qualität des Programms ist aber enorm. Es fällt letztendlich leicht, professionell zu programmieren, was viele Webseiten leider vermissen lassen. Andere Systeme erlauben einen schnelleren Zugriff, tun sich jedoch schwer, wenn es um komplexere Aufgaben geht, wie der Austausch der gerade bearbeiteten Zeile eines Datengitters.

6.6 Fragen und Übungsaufgaben

1. Welchen Datenbankprovider verwenden Sie, wenn eine Access-Datenbank verwendet wird?
2. Wie ermitteln Sie den physischen Pfad zu einer *mdb*-Datei, die in einem virtuellen Verzeichnis des Webserver liegt?
3. Schreiben Sie eine kleine Datenbankapplikation:
   - Zeigen Sie alle Benutzer aus der Tabelle *Adressen* an
   - Überlassen Sie dem Benutzer die Auswahl einer vorhandenen Adresse oder die Erfassung einer neuen
4. Erweitern Sie das zuletzt vorgestellte Programm mit dem Warenkorb so, dass die Anmeldung aus der in Punkt 3 erstellten Adressverwaltung die Zuordnung der Bestellungen im Warenkorb übernimmt.

Verwenden Sie Sessions, um die Zuordnung permanent zur Verfügung zu haben.

Über den Autor

Für Fragen, Anregungen aber auch Kritik und Hinweise steht Ihnen der Autor gern zur Verfügung. Dieses Buch soll Ihnen den schnellen Einstieg in eine komplexe Materie ermöglichen. Zugleich soll es aber auch einen professionellen Anspruch erfüllen, für den Autor und Verlag gleichermaßen stehen. Dies war in keiner Phase des Projekts eine leichte Aufgabe.

Insofern sind Verbesserungsvorschläge und konstruktive Anmerkungen gern gesehen und finden in künftigen Auflagen sicher ihre Entsprechung.

A.1 Aktuelle Informationen finden Sie im Internet

Alle Programme, Bugfixes und Korrekturen finden Sie unter der folgenden Adresse: *http://www.dotnet.comzept.de*

Auf dieser Seite sind auch alle Lösungen zu den Übungsaufgaben zu finden. Sie können dort auch eigene Lösungen veröffentlichen und mit anderen Lesern diskutieren.

Außerdem ist die offizielle Seite von Microsoft immer ein Besuch wert:

http://www.asp.net

Den Autor selbst können Sie auf seiner Homepage näher kennenlernen:

http://www.joerg.krause.net

Hilfe finden Sie beim Autor, wenn es um eines der folgenden „Probleme“ geht:

- Entwicklung professioneller Websites jeder Größenordnung
- Projektmanagement und Programmierung in ASP.NET/C#, JavaScript, HTML, aber auch PHP und ASP
- Schulungen, Seminare, Programmierkurse und Workshops, darunter auch spezielle Schulungen zu ASP.NET und C# und nach Kundenwunsch
- Fachliche Unterstützung für Start-Ups, Venture Capitalists und Old Economy ;-)

Anfragen senden Sie bitte direkt per E-Mail an:
joerg.krause@addison-wesley.de

Leider schaffe ich es nicht immer, jede E-Mail sofort zu beantworten. Sie können aber sicher sein, dass jede Nachricht gelesen wird. Insofern können Sie dieses Medium jederzeit nutzen, um irgendetwas los zu werden – was auch immer Sie gern mitteilen möchten.

Stichwortverzeichnis

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Symbole

Numerisch