LERNEN LEICHTER GEMACHT



Java

für dummies®

Alles-in-einem-Band



- Die Grundlagen des Programmierens kennenlernen
- Fenster- und Grafikprogrammierung verstehen
- Sich an den wichtigsten Werkzeugen keinen Bruch heben

Arnold Willemer

Java für Dummies. Alles-in-einem-Band

Schummelseite

WICHTIGE TASTENKOMBINATIONEN VON ECLIPSE

- Strg + A markiert den kompletten Text.
- Strg + C kopiert den markierten Text in den Hintergrundspeicher.
- Strg + V fügt an die Stelle, an der der Cursor steht, den Text ein, der im Hintergrundspeicher steht.
- Strg + X löscht den markierten Text und schiebt ihn in den Hintergrundspeicher.
- Strg + ⇧ + F formatiert den markierten Bereich nach den Vorgaben des Java Style Guides.
- Strg + I erneuert die Einrückung in dem markierten Bereich.
- Strg + ⇧ + R: Der markierte Bezeichner wird geändert, und zwar nicht nur an Ort und Stelle, sondern überall, wo er noch verwendet wird.
- Strg + ⇧ + O verwaltet die Importliste. Fehlende Importe werden hinzugefügt und nicht mehr verwendete Importe entfernt.
- Strg + Mausklick auf eine Methode oder Variable führt zu der Stelle ihrer Definition.
- Alt + ← kehrt an die zuletzt bearbeitete Stelle zurück.

STEUERUNG DES DEBUGGERS UNTER ECLIPSE

- F11 startet den Debugger.
- F8 lässt den Debugger bis zum nächsten Breakpoint laufen.
- F2 stoppt das Programm.
- F5 geht einen Schritt weiter und springt gegebenenfalls in die Methode.
- F6 geht einen Schritt weiter, aber überspringt die Methode.

MODIFIZIERER FüR VARIABLEN, METHODEN UND KLASSEN

- ✔ `private`: Reine Privatsache. Nur die eigene Klasse darf dies sehen.
- ✔ `protected`: Familienangelegenheit. Nur die Klasse selbst und alle abgeleiteten Kinder dürfen dies sehen.
- ✔ `public`: Öffentlichkeitswirksam: Das hier ist so toll. Das darf jeder sehen!
- ✔ `final`: Diese Variable darf auch mit Gewalt nach der Initialisierung nicht mehr geändert werden.
- ✔ `static`: Diese Variable oder Methode existiert exakt einmal. Es kann also nicht mehrere Exemplare geben, existiert aber auch ohne Geburtshilfe durch `new`.

PRIMITIVE VARIABLENTYPEN

- ✔ `int`: Eine ganze Zahl.
- ✔ `long`: Eine ganze Zahl mit besonders hoher Kapazität.
- ✔ `short`: Eine zu kurz gekommene ganze Zahl.
- ✔ `byte`: Eine richtig kleine ganze Zahl, nur acht Bits lang.
- ✔ `float`: Eine kleine Fließkommazahl, inzwischen nicht mehr üblich.
- ✔ `double`: Dies verwendet man heute als Fließkommazahl.
- ✔ `char`: Ein Buchstabe, kann aber auch als Zahl betrachtet werden. Buchstabenliterale haben Hochkommata, wie etwa ‘A’.
- ✔ `boolean`: Wahrheitswert, enthält `true` oder `false`.
- ✔ `String`: ist nicht richtig primitv, aber fast – enthält Zeichenketten, deren Literale in Anführungszeichen stehen.
- ✔ `void`: gibt es eigentlich gar nicht, wird aber bei Methoden verwendet, um zu kennzeichnen, dass sie nichts hergeben.

NICHTPRIMITIVE VARIABLEN

- ✔ Nichtprimitive Variablentypen sind Arrays oder Klassen.
- ✔ Die Variablen nennt man *Referenzen*.

- Vor ihrem Zugriff müssen sie durch `new` erzeugt werden.
- Tritt eine `NullPointerException` auf, hat man das `new` vergessen.

TYPEN DES JAVA COLLECTION FRAMEWORKS

- `ArrayList`: Dynamisch vergrößerbares Array. Darauf kann über einen Index zugegriffen werden.
- `LinkedList`: Verkettete Liste, ideal für Warteschlangen und Stapel.
- `Set`: Die Rache der Mengenlehre: Keine doppelten, keine Ordnung.
- `Map`: Assoziativer Speicher, auf den über Schlüssel zugegriffen wird. Die Datenbank des kleinen Mannes im Speicher.



Arnold Willemer

Java für dummies®

Alles-in-einem-Band

WILEY

WILEY-VCH Verlag GmbH & Co. KGaA

Java für Dummies. Alles-in-einem-Band

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

1. Auflage 2018

© 2018 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim

All rights reserved including the right of reproduction in whole or in part in any form. This translation published by arrangement with John Wiley and Sons, Inc.

Alle Rechte vorbehalten inklusive des Rechtes auf Reproduktion im Ganzen oder in Teilen und in jeglicher Form. Diese Übersetzung wird mit Genehmigung von John Wiley and Sons, Inc. publiziert.

Wiley, the Wiley logo, Für Dummies, the Dummies Man logo, and related trademarks and trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries. Used by permission.

Wiley, die Bezeichnung »Für Dummies«, das Dummies-Mann-Logo und darauf bezogene Gestaltungen sind Marken oder eingetragene Marken von John Wiley & Sons, Inc., USA, Deutschland und in anderen Ländern.

Das vorliegende Werk wurde sorgfältig erarbeitet. Dennoch übernehmen Autoren und Verlag für die Richtigkeit von Angaben, Hinweisen und Ratschlägen sowie eventuelle Druckfehler keine Haftung.

Coverfoto: nerudol – fotolia.com
Korrektur: Petra Heubach-Erdmann

Print ISBN: 978-3-527-71450-6
ePub ISBN: 978-3-527-81307-0
mobi ISBN: 978-3-527-81306-3

Inhaltsverzeichnis

- Cover
- Titelseite
- Impressum
- Einleitung
  - Über dieses Buch
  - Konventionen in diesem Buch
  - Törichte Annahmen über den Leser
  - Wie dieses Buch aufgebaut ist
  - Symbole, die in diesem Buch verwendet werden
  - Wie es weitergeht
- Teil I: Programmieren lernen
  - Kapitel 1: Ein Loblied auf Java
    - Das Profil von Java
    - Programmiersprache, Interpreter und Compiler
    - Die virtuelle Maschine und das ultimative Wohnmobil
    - Vergleich mit anderen Programmiersprachen
  - Kapitel 2: Entwicklung mit IDEen und ohne
    - Ein kleines Programmbeispiel als Opfer
    - Java-Programmierung mit dem JDK
    - Eclipse
    - NetBeans
    - Aufgabe: Grüßaugust selbst erstellen
  - Kapitel 3: Aufstieg zum Programmierer
    - Java-Programm-Skelett
    - Programmerweiterungen
    - Variable Attribute
    - Kleines Rechenwerk
    - Zeichen

Ordnende Elemente
Aufgaben
Anmerkungen
Kapitel 4: Entscheiden und Wiederholen
Nur unter einer Bedingung
Logeleien
Die Fallunterscheidung – switch case
Endlich lernen wir Schleifen zu binden!
Überblick behalten mit Struktogrammen
Aufgaben
Kapitel 5: Funktionierende Methoden
Vorstellung der Methode main
Methoden gliedern und sparen Arbeit
Salzkartoffeln kochen für Anfänger
Methoden mit Datenfluss
Überladen von Methoden
Rekursionen
Aufgaben
Kapitel 6: Die Strings: Viele, viele Buchstaben
Variablen und Texte
Texteingabe
Textmanipulationen
Typwandler
StringBuffer und StringBuilder
Beispiel Pfadumbau
Aufgaben
Kapitel 7: Arrays
Wir bauen ein Array
Übung: Ein Array für die Monatstage
Übung: Ein Kartenspiel
Das String-Array der Methode main
Aufbruch in neue Dimensionen

Aufgaben
Kapitel 8: Bermuda: Die Schiffe und das Meer
Spielregeln
Arrays für Spielfeld und Schiffe
Der Spielablauf
Das komplette Listing
Teil II: Objektorientiert und modular
Kapitel 9: Klasse Objekte
Ein Beispiel für objektorientierte Programmierung
Eine Klasse im Selbstbau
Selbstverweisende Klassen
Die Vorteile der Privatsphäre
Konstruktionsbedingt
Statische Attribute und Methoden
Wrapper-Klassen für Primitive
Aufzählungstyp mit enum definieren
Aufgaben
Kapitel 10: Klassen erweitern
Noch mal Klasse: Der Mensch als Objekt
Referenzen auf erweiterte Objekte
Konstruktoren in der Vererbung
Die Mensa der Universität Norgaardholz
Der Beruf wird geschützt
Anonyme Klassen und das Lambda
Aufgaben
Kapitel 11: Verteilung der Klassen auf Pakete
Packages
Ausführbare Dateien und Bibliotheken: JAR
Kapitel 12: Bermuda-Klassen
Das Projekt erstellen
Die Klassen
Teil III: Erweitern Sie den Horizont

Kapitel 13: Hoffentlich Exception-versichert
Ausnahmsweise Fehler
Eingabe einer Zahl per Dialogbox
Der catch-Parameter
Ein gelungener Wurf
Der Stammbaum der Exceptions
Die Katzen-Exception
Nochmal clone
Aufgaben
Kapitel 14: Fertige Bausteine zur Selbstbedienung
Wie der Zufall es will
Das Datum, der Kalender und die Neuerungen
Rechnerisches aus der Math-Klasse
Hilfsmethoden für Arrays
Kapitel 15: Datenelemente sammeln und anordnen
Das flexible Array: ArrayList
Ein altes flexibles Array: Der Vector
In Ketten: Die LinkedList
Sammlung der Einzigartigen: Set
Zugriff per Schlüssel: Der assoziative Speicher
Die Interfaces
Die Methoden des Java Collection Frameworks
Wie sortiert man Autos?
Lambda-Ausdrücke und Streams
Hinweis
Teil IV: Fenster- und Grafikprogrammierung
Kapitel 16: Mit Swing Fenster, Mäuse und Grafik programmieren
JFrame: Guck mal durchs Fenster
Zeichnen und Malen
Modernisierung: Graphics2D
Ereignisreichtum
Aufgaben

Kapitel 17: Erstellen eines Layouts
- Alle bitte nebeneinander aufstellen: FlowLayout
- Gleichmäßig vergittert: GridLayout
- Die Mitte wird fett: BorderLayout
- Kombination der Layouts mit Panels
- Das Layout der Dialoge: GridBagLayout
- Das BoxLayout und die Box
- Null-Diät für Layout-Manager
- Aufgaben

Kapitel 18: Kontrollelemente im Swing
- Hochgradiger Simpel: Das Label
- Das Auffangen der Ereignisse
- Druckvolle Buttons
- Ein BMI-Programm als Spielwiese
- Eine doch nicht so einfache Liste
- Der JTable und sein Modell
- Bäume pflanzen mit JTree
- Check den Button
- Menüs und Dateidialoge
- Der Fokus und der Multiliner
- Aufgaben

Kapitel 19: Bermuda im Swing
- Klassenbildung
- Grafik aus Images
- Ein Fensterrahmen für Bermuda
- Das Bermuda-Panel
- Erstellen einer Bermuda-JAR-Datei

Kapitel 20: Java FX, das neue Swing?
- Mit Eclipse zur FX-Anwendung
- Bühnendekoration
- Mit NetBeans zur Java-FX-Anwendung
- Ein Button als Kontrollelement

Mit dem Pane um Fassung ringen
Es kommt Leben in die Bude: EventHandler
Ein paar Kontrollelemente
Standarddialoge
Schickmachen mit CSS
Komponieren in FXML im SceneBuilder
Aufgaben
Kapitel 21: Bermuda im FX-Kleid
Das Hauptprogramm
Das Canvas für die Grafikdarstellung
Kapitel 22: Android
Alles ist ein wenig anders
Arbeitsumgebung schaffen
Gestaltung einer App
Java macht die Activity erst aktiv
Ereignisse der Kontrollelemente
Texte aus dem Java-Programm lesen
Bermuda auch auf Android
Und nun?
Teil V: Daten längerfristig speichern
Kapitel 23: Dateien halten die Daten frisch
Daten aus Textdateien lesen
Daten in einer Textdatei
Serialisierte Klassen
Spielstandsicherung für Bermuda
Properties als Konfigurationsdatei
Auf Strings über Strings zugreifen
Properties als Konfigurationsdatei
Random-Access-Dateizugriffe
Umgang mit dem Dateisystem
Verzeichnisse auslesen
Aufgaben

Kapitel 24: Datenbankzugriff aus Java
Datenbanken und ihre Tabellen
Eine Datenbank für die Bonbonbude
Wenn Datenbanken einmal groß werden
Transaktionen
Datenbankunterstützung durch Eclipse
Aufgaben
Java Persistence API: JPA
Kapitel 25: XML-Dateien
Die Strukturen von XML-Dateien
Einlesen einer XML-Datei
XML-Dateien schreiben
Eine beliebige XML-Datei auslesen
Kapitel 26: JSON
Der JSON-Sprachschatz
JSON in Java
Aufbau der JSON-Struktur
Klassen sorgen selbst für JSON
Frag mal Google Maps
Teil VI: Praktisches
Kapitel 27: Javabohnen als Datenmodell
Serialisierbarkeit
Attributänderungen und Meldung
Einspruch!
Kapitel 28: Parallelverarbeitung mit Threads
Erst einmal Schlaf nachholen
Leichtgewichtige Threads
Kollisionen in der Vasenfabrik
Warten auf Kartons
Wir warten nicht auf Godot: Time-out
Prozessoren rösten
Multitasking in der GUI

Aufgabe
Teil VII: Netzwerke und Internet
Kapitel 29: Netzwerkprogrammierung
Das Liebesgeflüster der Sockets
Mails versenden
Java als Web-Client
Java ruft Java: RMI
Time-out und Parallelität
Kapitel 30: Heidis Berge und die Enterprise Edition
Das Konzept des Application Servers
Vorbereitung und Installation
Glassfish spielt Indianer und Servlet
Java Server Pages
Java Server Faces
Der Gast und die Datenbank
Datenbankanwendung per JPA
Jersey und der ganze REST
SOAP: Ferngespräch in XML
Teil VIII: Werkzeuge
Kapitel 31: Dem Fehler auf der Spur
Alles so schön rot hier
Gelbe Warnungen
Der Eclipse-Debugger
Kapitel 32: Automatisierte Tests: JUnit
Konfiguration von Eclipse
Erstellen eines TestCase
Annotationen
JUnit-Testaufbau bei NetBeans
Kapitel 33: Die Versionskontrolle Git
Ein lokales Repository für Eclipse
Branches
Ein zentrales Repository

Git unter Android Studio
Teil IX: Der Top-Ten-Teil
Kapitel 34: Zehn Dinge
Zehn Hilfen bei der Fehlersuche
Zehn ganz persönliche Dinge
Stichwortverzeichnis
End User License Agreement

Tabellenverzeichnis

Kapitel 3
Tabelle 3.1: Primitive Typen und ihre Dimensionen
Tabelle 3.2: Operatoren in Java
Tabelle 3.3: Backslash-Sequenzen für Steuerzeichen
Kapitel 4
Tabelle 4.1: Vergleichsoperatoren
Tabelle 4.2: Wahrheitswertetabelle für UND
Tabelle 4.3: Wahrheitswertetabelle für ODER
Tabelle 4.4: Wahrheitswertetabelle für exklusives ODER
Tabelle 4.5: Wahrheitswertetabelle für NOT
Tabelle 4.6: Wahrheitswertetabelle zulässigen Fahrens und der Negation
Tabelle 4.7: Wahrheitswertetabelle der Umkehrung mit UND
Tabelle 4.8: Wahrheitswertetabelle der Umkehrung mit ODER
Tabelle 4.9: Notentabelle
Kapitel 6
Tabelle 6.1: Reguläre Ausdrücke
Kapitel 9
Tabelle 9.1: Primitive Typen und ihre Klassen-Wrapper
Kapitel 14
Tabelle 14.1: Datumskonstanten
Kapitel 15
Tabelle 15.1: Die Methoden der Queue

Kapitel 18

Tabelle 18.1: Spannungsabfall an LEDs

Kapitel 23

Tabelle 23.1: Der erste Block einer dBase-Datei

Kapitel 24

Tabelle 24.1: Eine Liste für die Artikel

Tabelle 24.2: Datenbanken und ihre JDBC-Treiberklassen

Kapitel 30

Tabelle 30.1: Entsprechung der HTTP- und SQL-Befehle

Illustrationsverzeichnis

Kapitel 1

Abbildung 1.1: Vom Wohnmobil zum Wohnboot

Kapitel 2

Abbildung 2.1: Eclipse-Installer

Abbildung 2.2: Eclipse nach Anlegen des Projekts

Abbildung 2.3: Ein Fehler im Eclipse-Editor

Abbildung 2.4: NetBeans wird installiert.

Abbildung 2.5: Die IDE NetBeans

Kapitel 3

Abbildung 3.1: Start über das grüne Dreieck oder dessen Menü

Abbildung 3.2: Eingabe einer Zahl

Kapitel 4

Abbildung 4.1: Eine Weiche bringt eine Verzweigung des Ablaufs.

Abbildung 4.2: Eine Weiche führt zur Wiederholung.

Abbildung 4.3: Eine Weiche am Ende der Schleife

Abbildung 4.4: Struktogramm für `if`

Abbildung 4.5: Struktogramm für `while`

Abbildung 4.6: Struktogramm für `do`

Abbildung 4.7: Struktogramm für die ggT-Berechnung

Abbildung 4.8: Struktogramm für ggT im Java-Code

Kapitel 5

Abbildung 5.1: Aus Parametern wird der Rückgabewert.

Abbildung 5.2: Türme von Hanoi

Abbildung 5.3: Lösung für die Türme von Hanoi

Kapitel 6

Abbildung 6.1: James Bond in seiner Rolle als String

Abbildung 6.2: Die Wirkung von `substring(26, 30)`

Kapitel 7

Abbildung 7.1: Gleichartige Objekte dicht an dicht und nummeriert

Abbildung 7.2: Ein zweidimensionales Array ist ein Array von Array-Referenzen.

Kapitel 8

Abbildung 8.1: Das Spielfeld von Bermuda

Abbildung 8.2: Richtungen abfragen in Bermuda

Kapitel 9

Abbildung 9.1: Eclipse zeigt die Attribute und Methoden eines Objekts.

Abbildung 9.2: Eclipse-Hilfe mit Javadoc.

Abbildung 9.3: Das `next`-Attribut verkettet die Elemente der Liste.

Abbildung 9.4: Das `next`-Attribut wird auf das übernächste Element gelenkt.

Abbildung 9.5: Das `next`-Attribut wird umgebogen.

Abbildung 9.6: Das letzte `next`-Attribut baut den Ring.

Abbildung 9.7: Zwei Referenzen ziehen einen binären Baum auf.

Abbildung 9.8: Eclipse generiert Getter- und Setter-Methoden.

Abbildung 9.9: Eine Kiste hat drei Dimensionen.

Kapitel 11

Abbildung 11.1: Das Projekt mit seinen Packages

Abbildung 11.2: Automatische Importsuche

Abbildung 11.3: Hinzufügen von JARs zu den Libraries im Build Path

Kapitel 13

Abbildung 13.1: Eingabedialog

Abbildung 13.2: Anzeige der Methodeninformation

Kapitel 15

Abbildung 15.1: Die `ArrayList` in Bauklotzform

Abbildung 15.2: Die `LinkedList` in Bauklotzform

Abbildung 15.3: Iteratoren zeigen auf Container-Elemente.

Abbildung 15.4: Möglichkeiten einer Queue

Abbildung 15.5: Die Map in Bauklotzform

Kapitel 16

Abbildung 16.1: Ein Swing-Programm

Abbildung 16.2: Terminate-Button in Eclipse

Abbildung 16.3: Eine kleine Grafik

Abbildung 16.4: Eckig und geradlinig

Abbildung 16.5: Abstrakte Kunst?

Abbildung 16.6: Fett ist gelb, Zucker ist blau, und Eiweiß ist magenta.

Abbildung 16.7: Ein kleiner Bildbetrachter

Abbildung 16.8: Bildausschnitt kopieren

Abbildung 16.9: Clippingbereich setzen

Abbildung 16.10: Eine `Area` für ein Wappen

Abbildung 16.11: Farbübergang

Abbildung 16.12: Sommersprossen selbst gemacht

Abbildung 16.13: Mit der Maus unterwegs

Kapitel 17

Abbildung 17.1: FlowLayout

Abbildung 17.2: `GridLayout`

Abbildung 17.3: `BorderLayout`

Abbildung 17.4: Ein Beispieldialog

Abbildung 17.5: Dialog-Layout durch Kombination mehrerer Layouts

Abbildung 17.6: `JSplit` erzeugt einen verschiebbaren Balken

Abbildung 17.7: `JScrollPane` erzeugt einen Rollbalken als Rahmen.

Abbildung 17.8: `JTabbedPane`

Abbildung 17.9: `GridBagLayout` für eine einfache Eingabemaske

Abbildung 17.10: `BoxLayout`

Abbildung 17.11: `Box`

Kapitel 18

Abbildung 18.1: Zweimal `JLabel`. Ein Bild und ein Text.

Abbildung 18.2: Einbau eines `ActionListeners`

Abbildung 18.3: Wenn man Druck auf einen Button ausübt

Abbildung 18.4: Body-Mass-Index-Rechner

Abbildung 18.5: Body-Mass-Index-Rechner mit Radiobutton

Abbildung 18.6: Body-Mass-Index-Rechner mit Combobox

Abbildung 18.7: Eine Auswahlliste

Abbildung 18.8: Eine einfache Hitparade mit einem `JTable`

Abbildung 18.9: `JTable` berücksichtigt Typen und ist editierbar.

Abbildung 18.10: `JTree` stellt das Benutzerverzeichnis dar.

Abbildung 18.11: Eine computergesteuerte Dinner-Party

Abbildung 18.12: Aufbau einer Menüleiste in Swing

Abbildung 18.13: Fokuswechsel

Kapitel 19

Abbildung 19.1: Die Bermuda-Images

Abbildung 19.2: Mit Swing sieht Bermuda schon besser aus.

Kapitel 20

Abbildung 20.1: Installation von `e(fx)clipse`

Abbildung 20.2: NetBeans besitzt bereits Projekte für FX.

Abbildung 20.3: Der Button nach Start des NetBeans-Rahmenprogramms

Abbildung 20.4: Buttons in einer `HBox`

Abbildung 20.5: Eingabemaske mit `GridPane`

Abbildung 20.6: `BorderPane` mit Liste und Rändern

Abbildung 20.7: `GridPane`-Eingabemaske im `BorderPane`

Abbildung 20.8: Ein TabPane mit drei Reitern

Abbildung 20.9: Fokuswechselanzeige

Abbildung 20.10: `CheckBox` und `ToggleButton`

Abbildung 20.11: `RadioButtons` für das Fernsehprogramm

Abbildung 20.12: Eine `ComboBox`

Abbildung 20.13: Eisauswahl mit zwei `ListViews`

Abbildung 20.14: Die `TableView` zeigt den Highscore.
Abbildung 20.15: Nachrichtendienst
Abbildung 20.16: Eingabeaufforderung
Abbildung 20.17: Mehrere Buttons für Ja-Nein-Fragen
Abbildung 20.18: Dateiauswahldialog
Abbildung 20.19: Der SceneBuilder mit einem Dialogfenster

Kapitel 21

Abbildung 21.1: Bermuda mit Java FX

Kapitel 22

Abbildung 22.1: Nach dem Download
Abbildung 22.2: Android Studio nach der Installation
Abbildung 22.3: Neues Projekt im Android Studio
Abbildung 22.4: Zielgeräte für die App auswählen
Abbildung 22.5: Activity für die App auswählen
Abbildung 22.6: Android Studio mit dem ersten Projekt
Abbildung 22.7: Gestaltung der Activity
Abbildung 22.8: Die Kontrollelemente auf dem Display
Abbildung 22.9: Die Android-Emulation mit der BMI-App
Abbildung 22.10: Die Activity als XML-Code in Android Studio
Abbildung 22.11: Neue Sprache hinzufügen
Abbildung 22.12: Die leere `BermudaView`
Abbildung 22.13: Bermuda mit Bitmaps unter Android

Kapitel 23

Abbildung 23.1: LibreOffice Calc erzeugt eine dBase-Datei.

Kapitel 24

Abbildung 24.1: Das Datenbank-Management-System
Abbildung 24.2: Eclipse in der Database-Development-Perspektive
Abbildung 24.3: Der Eclipse-Editor für *persistence.xml*

Kapitel 30

Abbildung 30.1: Glassfish-Einrichtung in Eclipse
Abbildung 30.2: Glassfish-Administration per Browser
Abbildung 30.3: Das Eingabeformular

Abbildung 30.4: Eingabe der Gastdaten

Abbildung 30.5: Der REST-Service hat reagiert.

Abbildung 30.6: Einstellen des Web Service

Kapitel 31

Abbildung 31.1: Eclipse in der Debug-Perspektive

Abbildung 31.2: Eclipse und seine Debug-Symbole

Kapitel 32

Abbildung 32.1: Einbinden der JUnit Library

Abbildung 32.2: Test des JUnit-Test-Case

Kapitel 33

Abbildung 33.1: Der erste lokale Commit in Eclipse

Abbildung 33.2: Eclipse-Ansicht über die Branches über Team|Switch TO|Other

Einleitung

Über dieses Buch

In diesem Buch lernen Sie, warum Nashörner kein Bankkonto haben, dass Katzen besser kratzen, als sich zu übergeben, wie man Bauklötze arrangiert, dass Stringtangas gar nicht so erotisch sind, und dann noch die Programmiersprache Java. Sie werden Programme erstellen, die das Übergewicht des Autors quantifizieren, Schiffe im Bermuda-Dreieck verstecken, den Kiosk von Superhelden verwalten und die Almhütten von Heidi vermieten.

Wir fangen also bei null an. Sie werden Variablen, Schleifen und Methoden kennenlernen. Wir werden in die wunderbare Welt der objektorientierten Programmierung eintauchen und Sie können beim nächsten Stammtisch endlich mit Wörtern wie Polymorphie, Interface und Exception protzen. Aber wir werden dabei noch nicht stehen bleiben.

Die Welt der grafischen Oberflächen ist viel zu attraktiv, um sie zu ignorieren. Die Java-Welt befindet sich in einer Umbruchsituation. Die Java-Hersteller erklären, dass in Zukunft alle grafischen Entwicklungen nur noch mit Java-FX arbeiten werden. Das sehen vermutlich sehr viele Unternehmen anders, da sie seit Jahren bereits Swing verwenden. So ist also beides am Markt. Sicherheitshalber habe ich beide Technologien behandelt. Suchen Sie sich etwas aus! Und da es für Android noch einmal eine eigene grafische Umgebung gibt, werden wir auch da einmal hineinschauen.

Nun kommen wir schon in die Regionen für fortgeschrittene Programmierer. Wir sehen uns die Datencontainer des Java Collection Frameworks an, schreiben und lesen Dateien und Datenbanken, schauen, wie Programme parallel laufen können, und werden die Anwendungen zur Netzwerkkommunikation anregen.

Wenn wir so weit gekommen sind, können wir auch die Stufe zur Client-Server-Programmierung mit der Java Enterprise Edition erklimmen. Sie werden sehen, wie man Software as a Service erstellt, und Client-Server-Architekturen wie SOAP und REST kennenlernen.

Vielleicht sollte ich Sie warnen: Die Leute vom Verlag haben mir keine Vorschriften zu diesem Buch gemacht. Alles war erlaubt. Einfach alles. Es gab zwei Bedingungen: Ich darf nicht gegen geltendes Recht verstoßen und der Leser muss Java lernen. Es ist aber nicht so schlimm, wie Sie jetzt vielleicht fürchten. In weiten Teilen ist es wirklich ein seriöses Fachbuch.

Dieses Buch basiert auf meinem Java-Buch, das 2013 veröffentlicht wurde. Den Inhalt habe ich aktualisiert und noch all die Dinge hineingeschrieben, die ich

wichtig fand. Es ist nun etwa doppelt so umfangreich und in der Reihe *für Dummies* erschienen.

In der ersten Auflage hatte Christine Siedle als Lektorin die Weichen gestellt. Bei dieser Auflage hat mich Marcel Ferner unterstützt. Beiden habe ich und vor allem das Buch viel zu verdanken. Frau Heubach-Erdmann begradigte meine Verbrechen an der deutschen Sprache und viele Leute mit Gefühl für Design bauten aus der Vorlage ein Buch. Und wenn nun noch die Abteilung Marketing richtig aktiv wird, werde ich vielleicht doch noch reich. Ansonsten versuche ich es weiter mit arbeiten.

Das Buch hat sehr davon profitiert, dass ich seit einiger Zeit bei der Hochschule Flensburg angestellt bin. Die Studenten der Veranstaltungen »Strukturierte Programmierung«, »Objektorientierte Programmierung« und »Client-Server-Programmierung mit der Java Enterprise Edition« hatten so einen erheblichen Einfluss auf die Gestaltung und die Verständlichkeit dieser Auflage. Mein Dank gilt auch den zahlreichen Kollegen, die mir viel Vertrauen entgegenbrachten. Die Atmosphäre dort ist toll. Wenn Sie also Informatik oder Medien-Informatik studieren wollen, kommen Sie in den Norden!

Auch meine Familie hat indirekt an diesem Buch mitgewirkt. Sie haben mich mit Nahrungsmitteln versorgt und mir den nötigen Freiraum verschafft. Selbst die Katze hatte ihren Anteil, wie Sie am Kapitel über Exceptions feststellen werden. Allerdings ist sie inzwischen verstorben. Aber ich bin unschuldig. Ehrlich!

Konventionen in diesem Buch

Der Setzer hatte noch ein paar Schriftarten übrig. Daraus haben wir etwas gemacht:

- *Neue Fachbegriffe* werden hervorgehoben.
- Oft haben Sie es mit *Dateien* oder *Pfaden* zu tun.
- Manchmal finden Sie `www.informationen.de/im/internet`.
- Menü- oder Buttonbeschriftungen in grafischen Oberflächen werden auf diese Weise hervorgehoben.
- Für `Befehle` und `Schlüsselworte` verwende ich diese Darstellung.
- `Funktionen` und `Methoden` werden auf diese Weise hervorgehoben. Und dann gibt es noch `Bezeichner` und `Variablen`.

Törichte Annahmen über den Leser

Ich habe eine ganze Weile überlegt, wer Sie sind, der Sie dieses Buch lesen. Ich gehe mal davon aus, dass Sie vielleicht noch nicht oft oder noch nie programmiert haben. Aber pfiffig sollten Sie schon sein. Das sind nämlich alle Programmierer –

fragen Sie doch mal einen!

Dann geriet ich in eine solche Schreibekstase, dass Themen dazukamen, die auch erfahrenen Java-Programmierern noch Spaß machen. Die können die ersten Kapitel mit einem wissenden Lächeln überblättern, werden aber bestimmt auch Spaß an dem Buch haben.

Sie werden nicht nur einen Computer benötigen, sondern ihn auch benutzen müssen. Programmieren ist mehr eine Frage der Routine als der Intelligenz. Das bestätigen alle, die mich kennen. Im Anfang ist es noch frustrierend, weil man ständig auf die selbst gemachten Fehler stößt. Darauf kann man unterschiedlich reagieren: Man wirft entnervt den Computer aus dem Fenster, schiebt die Schuld anderen in die Schuhe, beispielsweise Java oder dem doofen Autor, oder man beginnt, über sich selbst zu lachen.

Letzteres wünsche ich Ihnen. Mit der Zeit wird es nämlich besser. Und dann werden Sie feststellen, dass nichts so kreativ ist wie das Programmieren. Und Sie brauchen nicht mehr als Ihren Computer und Ihre Fantasie. Dann können Sie alles programmieren, was sich ein Mensch denken kann. Und das macht Spaß!

Wie dieses Buch aufgebaut ist

Wir haben uns auch in diesem Buch entschlossen, die Buchstaben von links nach rechts zu schreiben. Damit ein wenig Ordnung in die vielen Buchstaben kommt, gibt es eine grobe Gliederung:

- Teil I: Programmieren lernen

  Sie lernen die Werkzeuge kennen, wie man aus Schleifen, Abfragen und Methoden Programme baut. Sie werden Zahlen und Zeichenketten bändigen und miteinander verbinden. Damit kann man schon ein Spiel bauen und darum werden wir das auch tun.

- Teil II: Objektorientiert und modular

  Nun bauen Sie die Welt nach. Sie gestalten Objekte wie etwa Autos, Personen oder auch Mensaspeisen. Das Tolle an der Objektorientierung ist, dass man die Arbeit auf mehrere Leute verteilen kann.

- Teil III: Erweitern Sie den Horizont

  Wir lassen uns von Java in den Ausnahmesituationen unseres Programms mit Exceptions helfen, erfreuen uns an Werkzeugkästen voller Zufallszahlen, Kalendern und mathematischer Formeln. Gut, wenn man nicht alles selbst machen muss.

- Teil IV: Fenster- und Grafikprogrammierung

  Sie erstellen eigene Fenster. Spiele machen gleich viel mehr Spaß, wenn man

sie grafisch untermalen kann. Das gute alte Swing und das neue FX werden gleichermaßen behandelt. Und da sich Android-Smartphones auch in Java programmieren lassen, spielen wir damit ebenfalls herum.

- Teil V: Daten längerfristig speichern

  Bisher war der Computer nur bis zum Ende des Programms schlau. Nun sollen die Programme lernen, ihre Weisheit aufzubewahren. Dazu greifen wir zu allen Mitteln: Dateien und Datenbanken. Und die wichtigsten Datenformate XML und JSON werden Sie auch noch brauchen.

- Teil VI: Praktisches

  Ich gebe zu, dass ich keine wirklich gute Überschrift für zwei Dinge fand, die darunter fallen. Als Beans bezeichnet man Datenobjekte und wir werden sehen, wie die gegenseitig Ereignisse melden. Threads sind parallel laufende Unterprogramme.

- Teil VII: Netzwerke und Internet

  Wir werden die Kommunikation zwischen Computern erproben, die Romeo und Julia spielen. Und dann wird Java Mails versenden, Webseiten lesen und mit Google Maps spielen.

  Vor allem werden wir uns die Profitechniken von Java Enterprise Edition ansehen. Da fallen dann die Schlagwörter aus den Stellenanzeigen wie Application Server, Java Server Pages, JPA, REST und SOAP. Wir drehen also am ganz großen Rad der Client-Server-Architektur.

- Teil VIII: Werkzeuge

  Mit dem Debugger können Sie Programme Schritt auf Schritt verfolgen und so auch verborgene Fehler finden. Wir bauen Programme, die sich automatisch selbst testen, und verwenden eine Versionskontrolle, um in alten Versionen zu wühlen, aber auch, damit ein Programmierteam auf dem gleichen Stand ist.

- Teil IX: Top-Ten-Teil

  Hier finden Sie eine Hitparade der beliebtesten Programmierfehler und ganz persönliche Dinge.

Das ist ja doch ganz schön viel geworden. Erstaunlich, dass es so exakt zwischen die Buchdeckel passt.

Symbole, die in diesem Buch verwendet werden

Als ich dieses Buch schrieb, durfte ich mir ein paar Symbole aussuchen, die Sie nun immer wieder sehen werden. Sie müssen sie nicht auswendig lernen. Sie werden

vermutlich schon am Inhalt merken, was sie bedeuten. Aber damit alles seine Ordnung hat, stelle ich Ihnen die Symbole kurz vor.


Neben diesem Symbol finden Sie das Konzentrat eines Abschnitts. Etwas, was Sie sich auf Ihren Spicker schreiben würden, wenn jemand Sie prüfen würde.


Ein Tipp lässt manche Arbeit etwas einfacher werden. Damit solche Hilfestellungen nicht verloren gehen, stehen sie neben diesem Symbol.


Wenn es da noch etwas zu erwähnen gibt, was zum Verständnis hilft, finden Sie dieses Symbol.


Bevor Sie mit einem Fehlgriff Daten zerstören, den PC schädigen oder die Welt pulverisieren, sollten Sie den Text hinter diesem Symbol lesen. Die Welt oder Ihr Computer werden Ihnen dankbar sein.


Manchmal gibt es ungeheuer spannende Hintergründe zu einem Thema, das den Anfänger aufgrund der Komplexität etwas verschrecken könnte. Bevor der Informatiker in mir durchgeht, habe ich versucht, dieses Symbol zu verwenden. Sie können dem Ausflug folgen oder den Inhalt einfach ignorieren.

Wie es weitergeht

Die Formalitäten sind erledigt. Sie können sich mit Ihrem Computer einschließen und die wunderbare Welt der Java-Programmierung entdecken. Dabei wird Ihr PC Dinge kennenlernen, die er nie zuvor gesehen hat.

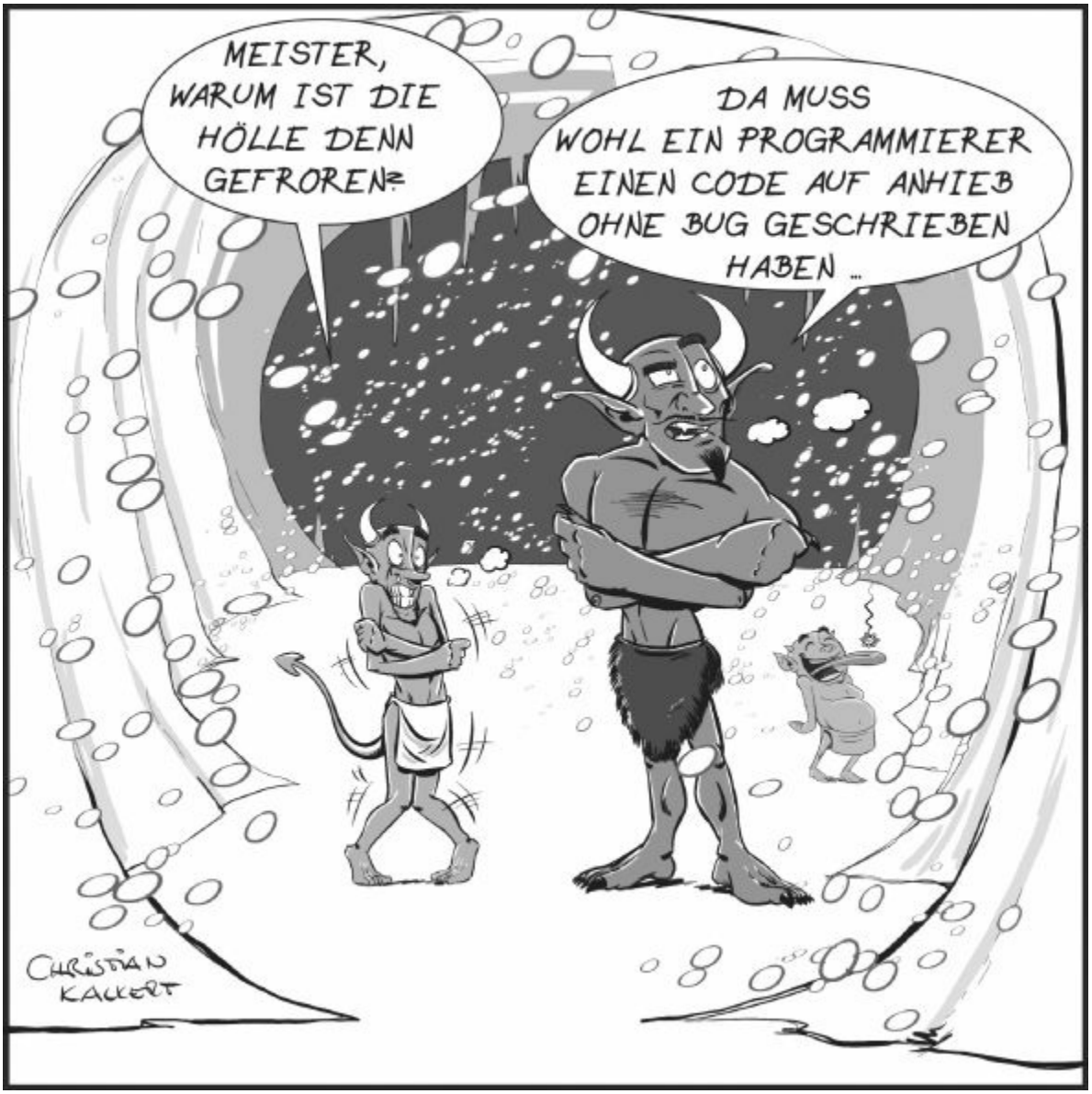

Zu diesem Buch gibt es eine Supportseite im Internet: `http://www.willemer.de/java4d`. Die Lösungen zu den Aufgaben im Buch und die Quelltexte aus diesem Buch finden Sie auch unter: `http://www.wiley-vch.de/ISBN9783527714506`.

Falls Sie dieses Buch richtig gut finden, senden Sie eine Mail an den Verlag, schreiben Sie eine Rezension für Amazon und informieren Sie das Fernsehen. Falls nicht, schicken Sie eine Mail an `javafuerdummies@willemer.de`. Ich werde sie lesen, und sobald ich mich davon erholt habe, dass ein Leser mich nicht verstanden hat, werde ich Ihnen antworten.

Arnold Willemer
Norgaardholz

Teil I

Programmieren lernen



Kapitel 1

Ein Loblied auf Java

IN DIESEM KAPITEL

Bequem und abgesichert, aber vor allem objektorientiert

Übersetzen oder übersetzen?

Vorteile der Portabilität

Von dem Autor eines Fachbuches für eine Programmiersprache erwartet man, dass er im ersten Kapitel ein Loblied auf diese Sprache singt. Wenn Sie dieses Buch kaufen wollen, haben Sie vermutlich gute Gründe, Java zu lernen. Und da werde ich Ihnen natürlich nicht abraten und so einen potenziellen Leser verlieren! Wie sollte ich dann Bestseller-Autor werden?

Das Profil von Java

Falls Sie die Programmiersprache, die Sie lernen wollen, nach dem Namen aussuchen, sind Sie hier richtig. Java! Kaum eine Programmiersprache hat solch einen hübschen Namen! Klingt Java nicht nach Südsee, Urlaub und weißen Stränden? Benannt wurde Java eigentlich nach Kaffee, der aus Java kommt. Auch das ist schön. Denn wer von uns mag denn keinen Kaffee?

Wer tatächlich weder Kaffee noch Südseeinseln mag, wird wohl Python programmieren müssen. Aber ob Schlangen wirklich schöner sind? Nebenbei bemerkt, ist die Sprache Python tatsächlich sehr nett. Ich könnte Ihnen da ein Buch empfehlen. Aber vielleicht sollte ich doch lieber erzählen, welche Besonderheiten Java so hat.

Objektorientiert

Java ist *objektorientiert*. Und weil objektorientiert beim Programmieren heutzutage ganz wichtig ist, drängt Java den Programmierer dazu, indem es massiv Hilfen zur Verfügung stellt. Nun ist die objektorientierte Programmierung, von ihren Fans auch kurz OOP genannt, wirklich hilfreich bei der Bewältigung großer Projekte. Darum sollten Sie Java diesen Zwang nicht übel nehmen.

Damit ich nicht allzu sehr ins Schwärmen gerate, werde ich hier auf Details zur OOP verzichten und dies an geeigneter Stelle nachholen.

Abgesichert und gut gepolstert

Ein typisches Merkmal von Java ist der Versuch, so viele Fehlerquellen auszuräumen wie irgend möglich und so den Programmierer vor seinem eigenen Fehlverhalten zu beschützen. Ein gutes Beispiel ist die Müllabfuhr, die *Garbage Collection*. Programmierer vergessen schon mal die Beseitigung ihrer Speicherleichen. Und wir wissen ja alle, was mit alterndem Müll geschieht: Er beginnt langsam intensiv zu stinken.

Aber trotz aller Vorsichtsmaßnahmen durch Java ist eine sorgfältige Programmierung nicht überflüssig geworden. Ja, es gibt das Gerücht, dass auch schon einmal ein Java-Programm abgestürzt sei.

Portabel

Wirklich brillieren kann Java allerdings auf dem Gebiet der Portabilität. Der Programmierer schreibt ein Programm, übersetzt es und es läuft auf Linux, Mac und Windows. Das ist schon sehr beeindruckend. Bei vielen anderen Programmiersprachen müssten Sie das Programm für jede Plattform neu schreiben. Ein Java-Programm sollte hingegen eigentlich auch auf jedem Telefon, der programmierbaren Fernbedienung, dem Toaster und auf allen Internetseiten laufen. Aber da ist sie wieder, die lästige Realität. Bei den Telefonen sind die Displays sehr klein, und bei Webanwendungen gibt es aus Sicherheitsgründen keinen Zugriff auf die lokalen Dateien. So müssen vielleicht nicht immer gleich Extra-Süppchen gekocht werden, aber manchmal muss die Beilage an das jeweilige Ziel angepasst werden. Weniger kulinarisch ausgedrückt: Sie können zwar auch auf exotischen Umgebungen Java programmieren, müssen aber ein paar Einschränkungen und Besonderheiten in Kauf nehmen.

Und dann gibt es noch marktpolitische Gründe, warum Java-Programme nicht überall da laufen dürfen, wo sie könnten. So hat Microsoft schon früh seine Unterstützung für Java zurückgezogen, als klar wurde, dass der Benutzer bei portablen Programmen frei wählen kann, ob er statt Windows vielleicht ein ganz anderes Betriebssystem verwenden möchte. Microsoft konnte zwar nicht verhindern, dass Java-Programme auch auf Windows laufen, sie müssen sich die Umgebung, in der sie laufen, aber selbst mitbringen. Apple unterbindet die Programmierung von iPhone-Apps in Java, angeblich um Sicherheitslücken zu vermeiden. Ein Schelm, der »Android« dabei denkt.

Alle Standards eingebaut

Wenn eine Anwendung überall laufen soll, ergibt sich zwangsläufig, dass Java alles an Bord haben muss, was das Programm je benötigen könnte. So ist insbesondere die grafische Oberfläche der Programme aus der Sicht des Java-Programmierers überall gleich. Andere Programmierer müssen für Windows entweder Win32, MFC oder .NET lernen. Für Linux sollten Sie GTK oder Qt beherrschen. Und natürlich hat der Mac auch eine ganz eigene Programmierschnittstelle. Dasselbe gilt auch für

Datenbanken oder die Prozessverwaltung. Der Java-Programmierer programmiert alle Datenbanken gleich. Willkommen in der wunderbar langweiligen Welt der standardisierten Umgebung!

Performance

Ja, zugegeben: Java-Programme haben nicht den Ruf, mit quietschenden Reifen ihre Aufgaben zu erledigen. Das beginnt schon beim Start einer Java-Anwendung. Da startet nämlich zunächst die virtuelle Maschine durch, bevor das Programm das erste Mal husten kann. Nun weiß jeder Power-User, dass Java-Programme interpretiert werden (was das ist, steht im nächsten Abschnitt), und Interpreter haben seit den Zeiten von BASIC den Ruf elender Schnarchnasen.

Zum Glück sind die Zeiten der BASIC-Interpreter vorbei. Java übersetzt das Programm in einen vorkompilierten Code, der auf schnelle Ausführung optimiert ist, und so muss sich das Java-Laufzeitsystem nicht mit irgendwelche Textzeilen quälen. Außerdem ist es auch schon über zwanzig Jahre her, dass die meisten C-64 ihren Weg in den Elektronikschrott gefunden haben. Und seither sind unsere Computer so schnell geworden, dass Sie einen aktuellen Computer vom Discounter nur an seine Grenzen fahren können, indem Sie ihm mit der Konvertierung von Spielfilmen drohen oder einen Ego-Shooter spielen wollen, bei dem Sie das Blut in 3D spritzen sehen wollen. Auf einem Durchschnittsrechner werden Sie also keine Verzögerung bei der Ausführung der Java-Programme bemerken. Überraschenderweise benutze ich gerade für den Videoschnitt seit Jahren ein Java-Programm, das ich unter anderem wegen seiner Geschwindigkeit schätze.

Programmiersprache, Interpreter und Compiler

Java ist eine Programmiersprache. Programmiersprachen sind für Menschen gemacht, damit sie darin Anweisungen für Computer codieren können. Das Problem ist, dass Computer eigentlich solche Sprachen nicht verstehen. Sie schwärmen eher für Zahlenkombinationen, die sie dann für Befehle halten und ausführen. Diese Diskrepanz muss irgendwie überbrückt werden. Es gibt zwei Arten von Programmen, die genau dies tun:

- Ein *Compiler* arbeitet wie ein Übersetzungsbüro. Man schickt den Quellcode hin und erhält den Maschinencode zurück. Mit dem Ergebnis füttert man den Computer.
- Ein *Interpreter* entspricht eher einem Simultan-Dolmetscher. Jeder Programmbefehl wird während der Laufzeit des Programms abgearbeitet.

Und was macht nun Java? Beides! Ich habe ja schon erwähnt, dass der Java-Code

interpretiert wird. Aber das ist nur die halbe Wahrheit. Zuvor übersetzt der Java-Compiler, der von seinen Freunden `javac` genannt wird, den Quellcode in einen Maschinencode für eine Maschine, die es gar nicht gibt. Und dieser Maschinencode wird dann auf jeder Zielmaschine interpretiert. Der Interpreter nennt sich geschwollen *Java Runtime System* und wird mit seinem Spitznamen `java` aufgerufen.

Die virtuelle Maschine und das ultimative Wohnmobil

Sowohl Compiler als auch Interpreter einzusetzen, erscheint zunächst völlig unsinnig. Das dahinter stehende Prinzip lässt sich aber an meinem Traum eines ultimativen Wohnmobils recht anschaulich erläutern. Es ist zusammengebaut aus einem Kleinlaster und einem alten Wohnwagen. Da sind richtig gute Ideen eingebaut. Die Betten sind bequem und mit einem Griff zur Sitzecke umgebaut. Die energieeffiziente Küche nutzt den kleinen Platz ideal. Die Dusche verbraucht kaum Wasser und hat doch einen vollen Strahl. Jetzt stellen Sie sich noch vor, das Wohnmobil habe einen speziellen Antrieb, der auf Solar-energie und Wasserstoff basiert. Wenn es nicht fährt, nutzt es die Sonne, um das bei Regen aufgefangene Wasser in Wasserstoff und Sauerstoff aufzuteilen.

Das Super-Wohnmobil hat lediglich einen Nachteil. Es fährt nur auf der Straße. Das ist für ein Wohnmobil nicht ungewöhnlich, ich fahre aber nun mal genauso gerne mit einem Boot in Urlaub. Es wäre also toll, wenn ich als Unterbau auch einen Bootsrumpf daruntersetzen könnte. Darum hat die Wohnzelle eine klare Verbindung, an denen die Anschlüsse und das Zahnrad für den Antrieb sitzen, und schon kann die Wohnzelle einfach durch einen Kran vom Boot auf den Laster und umgekehrt gesetzt werden. Wenn man nun noch jemanden findet, der einen Luftkissenunterbau herstellt, könnte man sogar in den Everglades Urlaub machen und Alligatoren füttern. Letzteres ist aber vermutlich verboten. Vor allem wenn man als Futter die Schwiegermutter verwendet.

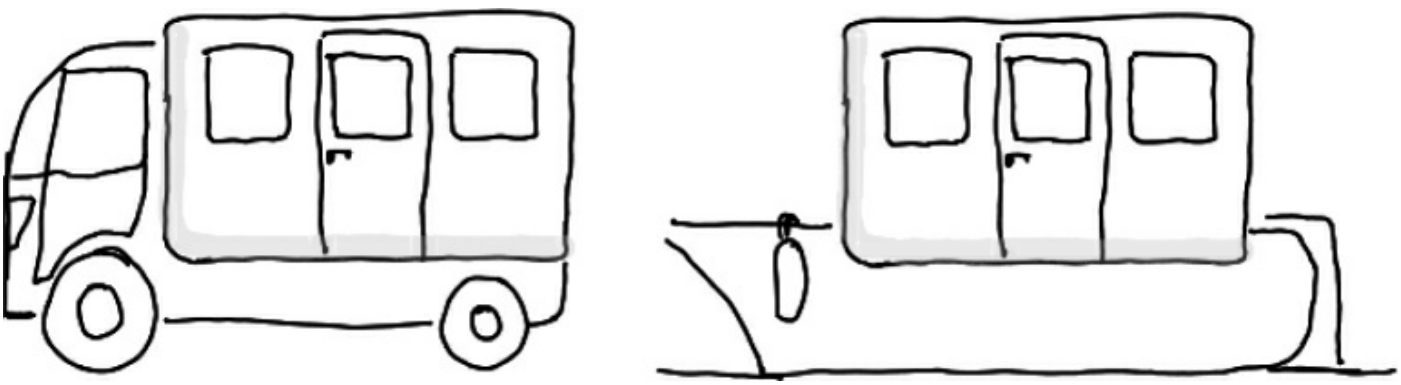


Abbildung 1.1: Vom Wohnmobil zum Wohnboot

Und was hat das jetzt mit der virtuellen Maschine zu tun? Ganz einfach! Das übersetzte Java-Programm ist die Kabine. Es verändert sich nach der Erstellung nicht mehr. Es setzt auf einem Unterbau auf, der es von der harten Wirklichkeit der verschiedenen Computersysteme trennt. Ob das Vehikel zu Wasser oder zu Land unterwegs ist, muss die Kabine nicht wissen. Der Unterbau ist also die virtuelle Maschine und die Kabine die Java-Anwendung.

Die virtuelle Maschine ist das *Java Runtime Environment (JRE)*. Bei einigen Systemen ist es bereits installiert oder eingebaut. Aber bei fast allen anderen Systemen kann man es nachinstallieren.

Wo wir gerade bei dreibuchstabigen Kürzeln sind: Neben dem JRE gibt es noch das *JDK*. Das JDK ist das *Java Development Kit*. Das ist der Werkzeugsatz, den der Programmierer benötigt, um Java-Programme zu entwickeln. Es enthält beispielsweise den Compiler, der aus dem Source-Code den Zwischencode erzeugt, der dann später von der JRE ausgeführt werden kann.


Das JDK (Java Development Kit) enthält den Compiler für den Code der virtuellen Maschine. Das JRE (Java Runtime Environment) interpretiert diesen Code für die Zielmaschine.

Vergleich mit anderen Programmiersprachen

Wer mit Computern zu tun hat, stellt fest, dass die Branche gelegentlich etwas an Sachlichkeit zu wünschen übrig lässt. Die Emotionswelle entspricht manchmal der von Fußballfans. Wer eine Programmiersprache kennt, wird sie bis aufs Messer verteidigen und Beweise über die Unfähigkeit aller anderen Programmiersprachen auf den Tisch legen. Dabei sind Erfahrungen mit der gegnerischen Sprache scheinbar eher hinderlich.

Der Hintergrund dieser Emotionen ist vermutlich die Angst, das eigene Wissen könnte obsolet werden, wenn sich die andere Programmiersprache oder das andere Betriebssystem durchsetzt.

Portabilität und Konsequenzen

Java hat seine Kompetenz da, wo Portabilität benötigt wird. Bei sorgfältiger Programmierung kann das Programm ohne Neuübersetzung auf verschiedenen Computern laufen.

Das hat zur direkten Folge, dass es klare Vorgehensweisen geben muss, wie man Datenbanken und grafische Oberflächen programmiert, ohne auf die Spezialitäten einzelner Produkte eingehen zu müssen. Es ergibt sich also eine hohe

Standardisierung.

Akademische Reinlichkeit

Das Design der Sprache Java ist durch eine gewisse Ideologie geprägt. Man möchte den Programmierer dazu zwingen, saubere objektorientierte Programme zu schreiben. Das hat etwas für sich, wenn man sieht, dass Cobol-Programmierer in jeder Programmiersprache Cobol programmieren können. Und Cobol ist wahrhaft keine übersichtliche oder wartungsfreundliche Programmiersprache.

Client-Server-Systeme

Seinen wirklichen Vorteil spielt Java im Bereich der verteilten Systeme aus. Mit Java schreiben Sie nicht nur einfache Anwendungen, sondern auch Client-Anwendungen, Servlets, die auf Server-Anfragen warten, aufgedonnerte HTML-Seiten oder REST-Server, die Ressourcen verteilen.

Kapitel 2

Entwicklung mit IDEen und ohne

IN DIESEM KAPITEL

Werkzeugkiste für den Programmierer

Die Befehle und die Entwicklungsumgebung

Grüßen im August

Ich denke, dass nun der Tatendrang aus Ihnen hervorbricht, denn es sollen Programme entstehen. Dazu benötigen Sie als Programmierer Werkzeug und davon gibt es reichlich und erstaunlicherweise alles kostenlos.

Die meisten Programmierer bevorzugen eine *IDE*. IDE hat leider nicht zwangsläufig etwas mit Idee zu tun, sondern ist die Abkürzung für *Integrated Development Environment*, also *integrierte Entwicklungsumgebung*. Diese Programme enthalten einen Editor, den Compiler, einen Debugger, eine Projektverwaltung und einige sehr praktische Tools.

Aber wo wir gerade bei Idee sind: Es ist nicht falsch, wenn ein Java-Programmierer eine Idee davon hat, wie es unter der Motorhaube seiner IDE aussieht. Dazu besuchen wir das Java Development Kit.

Ein kleines Programmbeispiel als Opfer

Um mit den Entwicklungsumgebungen wenigstens kurz spielen zu können, brauchen wir erst einmal ein kleines Java-Programm. Seit dem Erscheinen der Programmiersprache C hat es sich eingebürgert, ein Minimalprogramm zu erstellen, das »Hallo Welt!« auf den Bildschirm schreibt. In Java sieht das so aus:

```
public class ErsteKlasse {
    public static void main(String[] args) {
        System.out.println("Hallo Welt!");
    }
}
```

Listing 2.1: Erster Versuch

Viele Details dieses kleinen Programms werden erst im Laufe des Buches genauer erläutert. Aber damit nicht der Eindruck entsteht, die Zeilen seien vom Himmel gefallen, darf ich Sie vielleicht kurz mit ihnen bekannt manchen.

- `public`: Das Schlüsselwort `public` ist ein Modifizierer. Das bedeutet, dass er den folgenden Bereich etwas verändert. In diesem besonderen Fall bedeutet `public`, dass die folgende Klasse von außen frei zugänglich ist. Es gibt keine Geheimnisse, Paparazzi werden arbeitslos. Das Wort `public` taucht noch einmal in der nächsten Zeile auf. Dort veröffentlicht es eine Methode. Eine Methode ist etwas, das etwas tut.
- `class`: Das Schlüsselwort `class` ist keine Auszeichnung, sondern ist auch für weniger tolle Programme zwingend. Jedes Java-Programm besteht aus mindestens einer Klasse. Dem Schlüsselwort folgt der Name der Klasse, den der Programmierer aussucht. Hier ist es `ErsteKlasse`.
- Geschweifte Klammern: Mit einem geschweiften Klammernpaar wird ein Block von Anweisungen zusammengefasst. Sie können ineinander verschachtelt werden, wie das hier zu sehen ist.
- `static`: Das Schlüsselwort `static` ist ein Modifizierer für eine Methode oder ein Attribut, die es nur einmal pro Klasse gibt. Es muss genau da hin und da es ohne Vorwissen schwer zu begründen ist, müssen Sie mir vorläufig einfach vertrauen, auch wenn es schwerfällt.
- `void main(String[] args)`: Mit `main` beginnt die Hauptfunktion des Programms. Funktionen heißen in Java im Allgemeinen Methoden. Zwischen den runden Klammern stehen Parameter, die von außen an das Programm herangetragen werden. Auch dies lassen wir erst einmal ruhen, bis wir auf Parameter näher eingehen. In diesem Programm wird eh noch nicht auf sie zugegriffen. Dennoch müssen die Parameter benannt werden.

  Und `void`? Das heißt nichts. Wirklich! Gar nichts. Es sagt hier, dass die Hauptfunktion – ich meine natürlich Hauptmethode – gar nichts an das System zurückgibt. Verrückterweise steht das, was hinten rauskommt, bei der Methodendeklaration vorn. Sie werden sich daran gewöhnen müssen.
- `System.out.println`: Alles, was in den Klammern steht, wird über den Systemausgabekanal angezeigt, oder weniger hochtrabend ausgedrückt: Es wird auf dem Bildschirm ausgegeben. Text wird dazu in Anführungszeichen gesetzt.
- Die geschweiften Klammern werden jeweils wieder geschlossen. Der Compiler wird etwas pinselig, wenn das nicht passiert.

Der Programmtext steht in einer Datei, die den Namen der Klasse tragen muss und auf die Extension *java* endet, hier also *ErsteKlasse.java*. Die Groß- und Kleinschreibung ist signifikant. Das birgt eine Überraschung, weil es Windows egal

ist, ob seine Dateien in Groß- oder Kleinbuchstaben geschrieben werden. Dort würden *ErsteKlasse.java* und *ersteklasse.java* dieselbe Datei bezeichnen.

Der Quelltext darf übrigens nicht mit einer Textverarbeitung wie Word oder OpenOffice.Writer geschrieben werden. Ein Quelltext wird mit einem Editor geschrieben, der keinerlei zusätzliche Informationen wie Schriftarten oder Ausrichtungen hinterlegt. All das würde einen Compiler nur irritieren. Stattdessen werden nur die nackten Texte geschrieben. Man spricht auch von *rohem Text (raw text)*.

Dafür sind viele dieser Editoren auf Programmierung spezialisiert und färben beispielsweise die Quelltexte schön bunt, damit der Programmierer schnell erkennt, was Schlüsselworte sind, oder ob alle geöffneten Klammern auch wieder geschlossen werden. Da können Textteile zusammengeklappt werden oder auch Referenzhilfen angezeigt werden. Sie werden Ihre Textverarbeitung deshalb langfristig bestimmt nicht vermissen!


Verwenden Sie zum Programmieren einen Editor statt einer Textverarbeitung! Der Compiler verzehrt Quelltexte am liebsten schön roh.

Java-Programmierung mit dem JDK

Die Mindestausstattung, die Sie für die Java-Programmierung brauchen, finden Sie im *Java Development Kit (JDK)*. Allerdings enthält es keine grafischen Werkzeuge, sondern Programme, die in klassischer Weise über die Kommandozeile aufgerufen werden. Programmierer, die so arbeiten, haben den Ruf von Leuten, die sich lieber im Wald ein Zelt aus Zweigen bauen, als eine Ferienwohnung auf Mallorca zu mieten.

Zunächst müssen Sie das JDK aber erst mal haben. Linux-Benutzer installieren das JDK über das distributionseigene Installationstool. Windows- und Mac-Benutzer laden das JDK von der Website von Oracle:

`http://www.oracle.com/technetwork/java/javase/downloads/index.html`

Hier suchen Sie nach dem Link zum Download des JDK der Java Standard Edition (Java SE). Nun können Sie Ihre Betriebssystemversion auswählen. Bei Windows gibt es für 32-Bit- und 64-Bit-Versionen je eine EXE-Datei. Natürlich müssen Sie noch die Lizenzbedingungen akzeptieren. Dann können Sie die Datei herunterladen. Diese klicken Sie doppelt an und folgen den Installationsanweisungen. Auch für Mac finden Sie eine Standardinstallationsdatei, die Sie ebenso herunterladen und doppelklicken.

Windows

Das JDK kommt in Form einer ausführbaren Datei, die eine ganz normale

Installation durchführt, wie Sie sie auch von anderen Programmen her kennen. Sie können alle Schritte einfach bestätigen. Wenn Sie die Installation fragt, wo alle Dateien hingeschafft werden sollen, sollten Sie aufmerken. Typischerweise installiert die Routine in ein Verzeichnis, das mit *C:\Program Files\Java\jdk* beginnt. Das Verzeichnis enthält auch die Versionsnummer. Merken Sie sich diesen Pfad!

Wenn Sie den Java-Compiler und andere Tools von der Kommandozeile aufrufen wollen, dann sollten Sie nun noch die Umgebungsvariable `PATH` anpassen. Diese können Sie bei Windows erreichen, indem Sie die ⊞-Taste mit der Pause-Taste zusammen drücken. Es erscheint ein Fenster, in dem Sie Erweiterte Systemeinstellungen anklicken. Es erscheint ein weiteres Fenster. Unter dem Reiter Erweitert finden Sie den Button Umgebungsvariablen. Es startet ein weiteres Dialogfenster, in dem Sie `PATH` als neue Variable anlegen können. Als Wert hängen Sie den Pfad des JDK an, dem Sie noch ein Verzeichnis *bin* anhängen, weil sich darin die ausführbaren Programme befinden.

```
PATH=%PATH%;C:\Programme\Java\jdk\bin
```

Sollten Sie noch Windows XP verwenden, öffnen Sie das Start-Menü und klicken mit der rechten Maustaste auf Arbeitsplatz. Im Kontextmenü wählen Sie Eigenschaften aus und kommen dann über Erweitert zu Umgebungsvariablen.

MacOS

Liebhaber des Macintoshs finden auf ihrem Gerät bereits ein komplettes JDK, sofern ihr Gerät mit einem Betriebssystem bis 10.6 ausgestattet ist. Allerdings ist die dort lauffähige Java-Version 6 schon historisch wertvoll.

Tatsächlich wird diese Version von Apple auch für neuere Versionen von MacOS X angeboten, allerdings nur als JRE. Nach der Installation dieser Version gibt es häufig erhebliche Probleme mit diesem Gerät.

Statt also bei Apple zu suchen, sollten Sie sich lieber der Website von Oracle zuwenden und dürfen dort das aktuelle JDK herunterladen. Die Datei ist eine typische Installationsdatei für den Mac und wird durch einen einfachen Doppelklick installiert wie jedes andere Programm auch. Dabei sind Administrationsrechte erforderlich, weil das JDK grundsätzlich für alle Benutzer des Rechners eingerichtet wird.

Linux

Wenn Sie das JDK unter Linux installieren wollen, tun Sie sich den Gefallen und halten Sie sich an die normalen Paket-Installationen Ihrer Linux-Distribution. Erstens ist es der leichteste Weg. Zweitens sind Sie sicher, dass Sie keine Programme zweifelhafter Herkunft installieren. Und drittens sorgt die Installationsroutine dafür, dass alle Umgebungsvariablen korrekt gesetzt sind und Sie einfach anfangen können.

Rufen Sie das zu Ihrer Distribution gehörige Installationstool auf. Das könnte Synaptec, Software-Center oder Muon heißen, je nachdem, welche Distribution und welchen Desktop Sie verwenden. Darin suchen Sie nach dem Stichwort »JDK« und finden das Paket *default-jdk*. Dieses Pseudopaket verweist auf die aktuellste Version des Standard-JDKs. Dieses installieren Sie. Alle weiter benötigten Pakete werden automatisch mitinstalliert. Alle Umgebungsvariablen werden automatisch gesetzt.

Compiler aufrufen

Damit haben Sie das Werkzeug an Bord, um Ihre Java-Programme kompilieren und dann auch ausführen zu können.

Der Java-Compiler heißt `javac` und kann direkt mit dem Namen der Quelltextdatei von der Konsole aus gestartet werden. Unter Windows heißt diese `cmd.exe`. Sie können sie direkt über das Menü Start|Ausführen aufrufen. Sie finden sie auch unter dem Namen Eingabeaufforderung im Zubehör. Auf dem Mac suchen Sie beispielsweise über ⌘ + Leertaste nach dem Programm Terminal. Unter Linux gibt es an jeder Ecke eine Terminal-Emulation, meist unter dem Namen Terminal oder Konsole in den Untermenüs Zubehör oder System.

Sie wechseln mit dem Befehl `cd` in das Verzeichnis, in dem Ihr Java-Quelltext *ErsteKlasse.java* steht und rufen dann den Compiler auf:

```
javac ErsteKlasse.java
```

Falls Sie Fehlermeldungen bekommen, die besagen, dass `javac` nicht gefunden wurde, prüfen Sie Ihre Installation. Sollte der Compiler den Quelltext beanstanden, schieben Sie bitte zuerst mir die Schuld in die Schuhe und prüfen Sie dann, ob Sie beim Abtippen einen Fehler gemacht haben. Sollten hingegen keine Fehler gemeldet werden, erzeugt der Compiler aus der Datei *ErsteKlasse.java* eine Datei namens *ErsteKlasse.class*.

Diese Datei kann nicht einfach gestartet werden, weil sie weder unter Windows noch Mac oder Linux selbstständig läuft, deswegen kommt jetzt das Java Runtime Environment ins Spiel.

Das Runtime Environment starten Sie von der Konsole aus mit `java`. Als Parameter erhält es den Dateinamen der vorkompilierten Java-Klasse, aber bitte ohne die Extension *.class*, denn sonst gibt es böse Fehlermeldungen.

```
java ErsteKlasse
```

Wenn alles gutgegangen ist, erscheint auf der Konsole die Zeile:

```
Hallo Welt!
```

Nun bestehen große Programme aus ganz vielen Klassen und so auch aus ganz vielen Dateien. Braucht das Programm weitere Klassen, so wird es im aktuellen Verzeichnis nach den passenden Dateien mit der Erweiterung *class* suchen. Das ist

auch der Grund, warum der Dateiname dem Klassennamen entsprechen muss.

Aber was macht man jetzt unter Windows, wenn eine Klasse `MyClass` und die andere `myclass` heißt? Java unterscheidet zwar Groß- und Kleinschreibung, aber Windows tut dies bei Dateinamen nicht. Beide Klassen könnten also gar nicht nebeneinander in einem Verzeichnis stehen. An dieser Stelle helfen aber die Konventionen. Bei Klassennamen pflegt man den ersten Buchstaben großzuschreiben. Wortanfänge werden gern durch Großbuchstaben angezeigt, wie bei `ErsteKlasse`. Der Klassenname `ersteklasse` widerspräche also der Konvention.

Auch wenn Sie das bei den ersten Programmen noch nicht betrifft, stellt sich natürlich die Frage, ob das nicht sehr lästig ist, wenn man bei einem großen Projekt vielleicht mehrere Hundert Dateien installieren muss. So viele Klassen kommen da schon locker zusammen. Irgendwie sollte man das Ganze zusammenpacken können.

Ob Sie es glauben oder nicht: Das ist möglich! Sie hatten es nach dieser Einleitung wohl auch schon vermutet. Aber nun kommen wir zum komplizierteren Teil: Wie macht man das?

Das Packprogramm heißt *jar*. Übersetzt heißt es zwar *Krug*, was irgendwie passt, aber eigentlich ist der Name eine Abwandlung von *tar*, dem Standard-Packprogramm von Linux, dessen Syntax das Programm auch folgt. Sie müssen aber nicht alle Möglichkeiten kennenlernen. Es reicht zu wissen, wie Sie eine gepackte Anwendung zusammensetzen können.

Das Runtime Environment würde beim Start eines solchen Pakets natürlich schon gern wissen, in welcher der vielen Klassendateien eigentlich die Hauptfunktion `main` steckt. Dazu schreiben Sie ein Manifest. Das mag hochgestochen klingen, aber ausnahmsweise habe ich mir das nicht ausgedacht. Es heißt wirklich so.

Das Manifest könnte in der einfachsten Form den folgenden Aufbau haben:

```
Manifest-Version: 1.0
Main-Class: ErsteKlasse
```

Im Abschluss muss noch eine Leerzeile hinzugefügt werden, sonst bekommt das Laufzeitsystem wieder seine Migräne.

Welchen Dateinamen Sie für die Manifest-Datei verwenden, ist egal. Das Manifest wird vom Befehl `jar` später immer unter dem Namen *MANIFEST.MF* im Archiv abgelegt. Im folgenden Aufruf gehen wir davon aus, dass Ihr Manifest unter dem Dateinamen *manifest.txt* erstellt wurde.

Der folgende Befehl erzeugt ein Programmpaket namens *MeinProgramm.jar*, entnimmt das Manifest der Datei *manifest.txt* und bindet die beiden Klassendateien *ErsteKlasse.class* und *NochEine.class* ein:

```
jar -cfm MeinProgramm.jar manifest.txt ErsteKlasse.class NochEine.class
```

Die entstandene Datei *MeinProgramm.jar* kann fast wie die Klassendateien über das Runtime Environment System aufgerufen werden. Allerdings muss die Option `-jar` noch angegeben werden:

```
java -jar MeinProgramm.jar
```

Wenn Sie die *jar*-Dateien als ausführbar auf Ihrem Computer anmelden, können Sie sie sogar direkt von der grafischen Oberfläche doppelklicken.

Sie haben jetzt gesehen, wie Sie Java-Programme über die Konsole ausführen können. Zugegebenermaßen ist das alles etwas umständlich, sodass Sie sich vielleicht freuen werden, wenn ich Ihnen mitteile, dass Ihnen Entwicklungsumgebungen die Arbeit deutlich erleichtern können.

Eclipse

Die Suche nach einem freundlichen Partner für die Java-Programmierung führt häufig zu Eclipse. Eclipse enthält einen Editor, der Gedanken lesen kann. Jedenfalls schlägt er immer wieder vor, wie ein Befehl weitergehen könnte. Er übersetzt bei jeder Sicherung schnell das Programm und streicht nicht nur rot an, was falsch ist, sondern macht auch ein paar Vorschläge zur Korrektur. Mit dem Debugger können Sie Ihr fertiges Programm Schritt für Schritt durchlaufen und feststellen, warum der Computer vielleicht ganz andere Vorstellungen von Ihrer Logik hat als Sie.



Im Buch wird entweder mit Eclipse oder im Einzelfall mit dem JDK gearbeitet. Allerdings dürften Sie keine Probleme haben, wenn Sie eine andere IDE verwenden.

Installation

Von der Seite `www.eclipse.org` können Sie sich die aktuelle Version von Eclipse herunterladen. Sie geben an, auf welchem Betriebssystem Sie entwickeln dürfen oder müssen.

Auf der Website finden Sie auch Unterstützung, wenn Sie Probleme haben. Allerdings ist dort alles sehr englisch, vor allem die Sprache.

Unter Windows

Nach dem Download der passenden Windows-Version von `www.eclipse.org` erhalten Sie eine ZIP-Datei. Darin finden Sie einen Ordner namens *eclipse*. Diesen legen Sie dann von Hand an eine freie Stelle im Computer. Da es ein Programm ist, passt es am besten in das Verzeichnis *C:\Programme*. Natürlich dürfen Sie das nur mit Administratorrechten.

Leider gibt es kein Symbol auf dem Desktop oder im Hauptmenü. Es ist aber leicht

möglich, im Ordner *C:\Programme\eclipse* die Datei namens *eclipse.exe* mit der rechten Maustaste anzuklicken und im Kontextmenü auszuwählen, dass ein Eintrag ins Hauptmenü gemacht werden soll.

Das Starten funktioniert dann immer noch nicht? Eclipse nörgelt, das Development Kit sei nicht vorhanden? Dann haben Sie das JDK vermutlich noch nicht installiert. Blättern Sie also noch einmal zurück und schauen Sie nach, wie es installiert wird.

Installation unter Linux

Am einfachsten haben es wieder einmal die Benutzer von Linux, denn die können ihren Lieblings-Software-Installationsdienst wie das Software-Center unter Ubuntu oder Yast unter SUSE nach dem Stichwort »Eclipse« befragen und zur Installation auffordern. Eclipse ist anschließend Bestandteil des Systems und kann direkt über die Menüs zur Software-Entwicklung gestartet werden.

Auch wenn es möglich ist, sollten Sie eine Installation von der Website von Eclipse nur dann durchführen, wenn Sie unbedingt die aktuellste Version benötigen und Sie mit derartigen Installationen vertraut sind.

Auf dem Mac

Zunächst laden Sie sich das passende Eclipse für den Macintosh von `www.eclipse.org` herunter. Die Datei ist eine gepackte *tar*-Datei. Nach einem Doppelklick entpackt sich die Datei von selbst. Sie finden ein Programm namens Eclipse-Installer (siehe Abbildung 2.1). Der Mac warnt Sie vor den Gefahren des Internets und anschließend können Sie die Installation über den Button ÖFfnen beginnen.

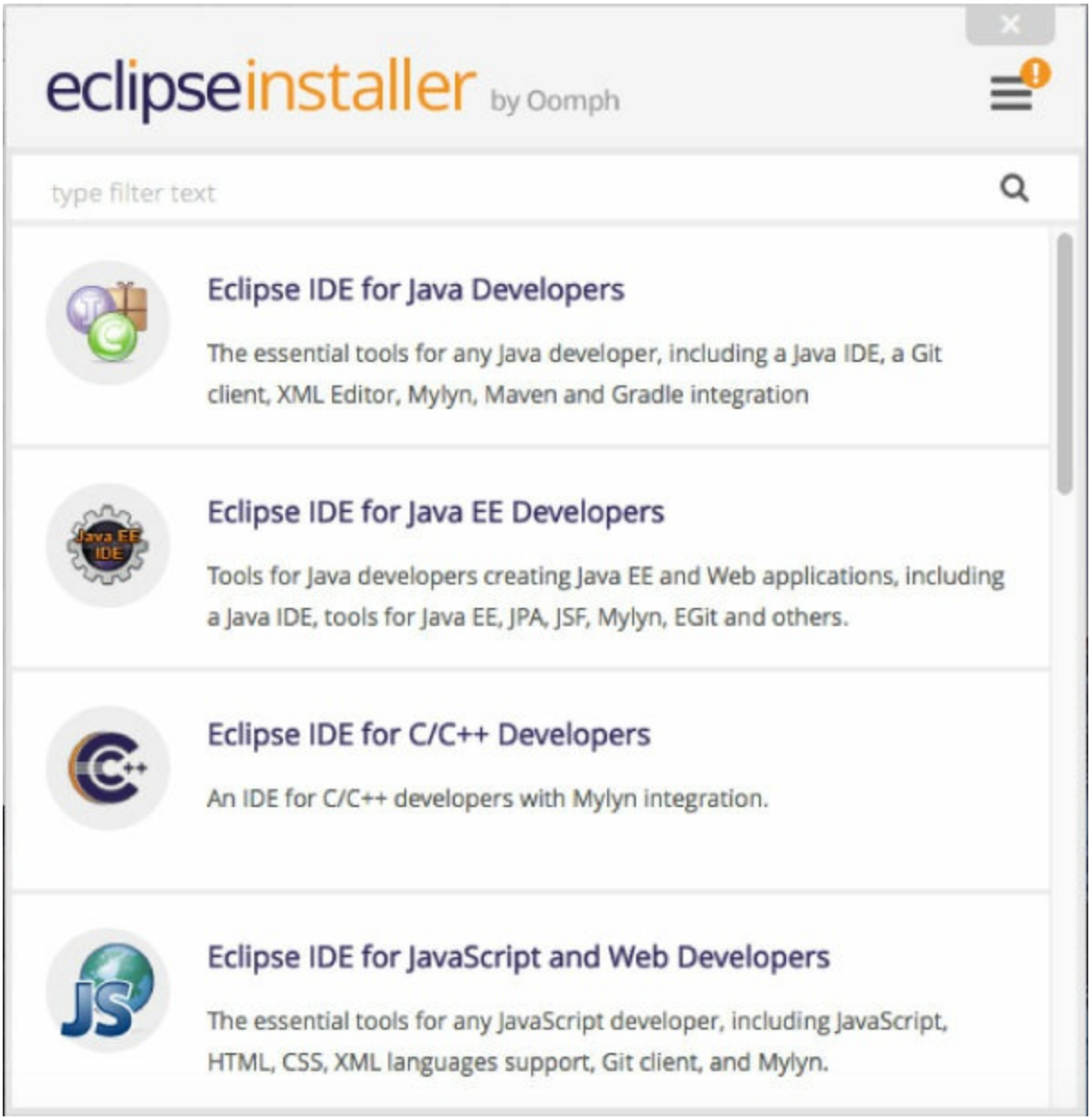


Abbildung 2.1: Eclipse-Installer

Sie können nun auswählen, ob Sie Eclipse als IDE für Java-Entwickler verwenden oder eine der anderen Optionen wählen möchten. Da Sie Java lernen wollen, wäre die erste Entscheidung erst einmal ausreichend. Im weiteren Verlauf des Buches werden wir allerdings auch JEE beschnüffeln. Da die Variante für Java EE Developers alle Elemente der Standardversion enthält, sollten Sie diese wählen. Sobald Sie die Lizenzbedingungen akzeptiert haben, dürfen Sie installieren.

Anlegen eines neuen Projekts

Noch bevor Eclipse mit dem Hauptfenster erscheint, fragt es, wo denn bitte der *Workspace* liegen soll. Das ist das Verzeichnis, in dem später alle Projekte und deren Dateien abgelegt werden sollen. Sie wissen es auch nicht? Nicht so schlimm, denn Eclipse macht selbst einen Vorschlag, den man in der Regel akzeptieren kann.

Beim Start erscheint zunächst ein Willkommensbildschirm, der Ihnen anbietet, ein paar Informationen über Eclipse zu erfahren und andere Dinge mehr. Wenn Sie

davon ermüdet sind, können Sie diesen durch das kleine Kreuz rechts neben dem Wort »Welcome« entfernen. Dieser Bildschirm wird beim Start so lange immer wieder auftauchen, bis Sie rechts unten das verantwortliche Häkchen gefunden und entfernt haben.

Ein Projekt für ein Java-Programm

Wenn Sie mit Eclipse ein Programm schreiben wollen, macht Eclipse etwas hochgestochen sofort ein Projekt daraus.

1. Ein neues Projekt legen Sie über die Menüpunkte File|New|Java Project an.
2. Es baut sich ein Dialog auf, in dem Eclipse Genaueres wissen will. Das Wichtigste ist zunächst der Projektname. Dieser Name wird später auch als Name für das Programm verwendet. Wir verwenden hier mal den Namen `ErsterTest`. Sie mögen meinen Einfallsreichtum bezüglich der Namenswahl nicht schätzen, aber es steht Ihnen völlig frei, einen besseren Namen zu wählen. Ich verspreche, nicht beleidigt zu sein.
3. Unten im Dialog wird nun der Button Finish sichtbar. Den klicken Sie an und Ihr erstes Projekt ist angelegt.

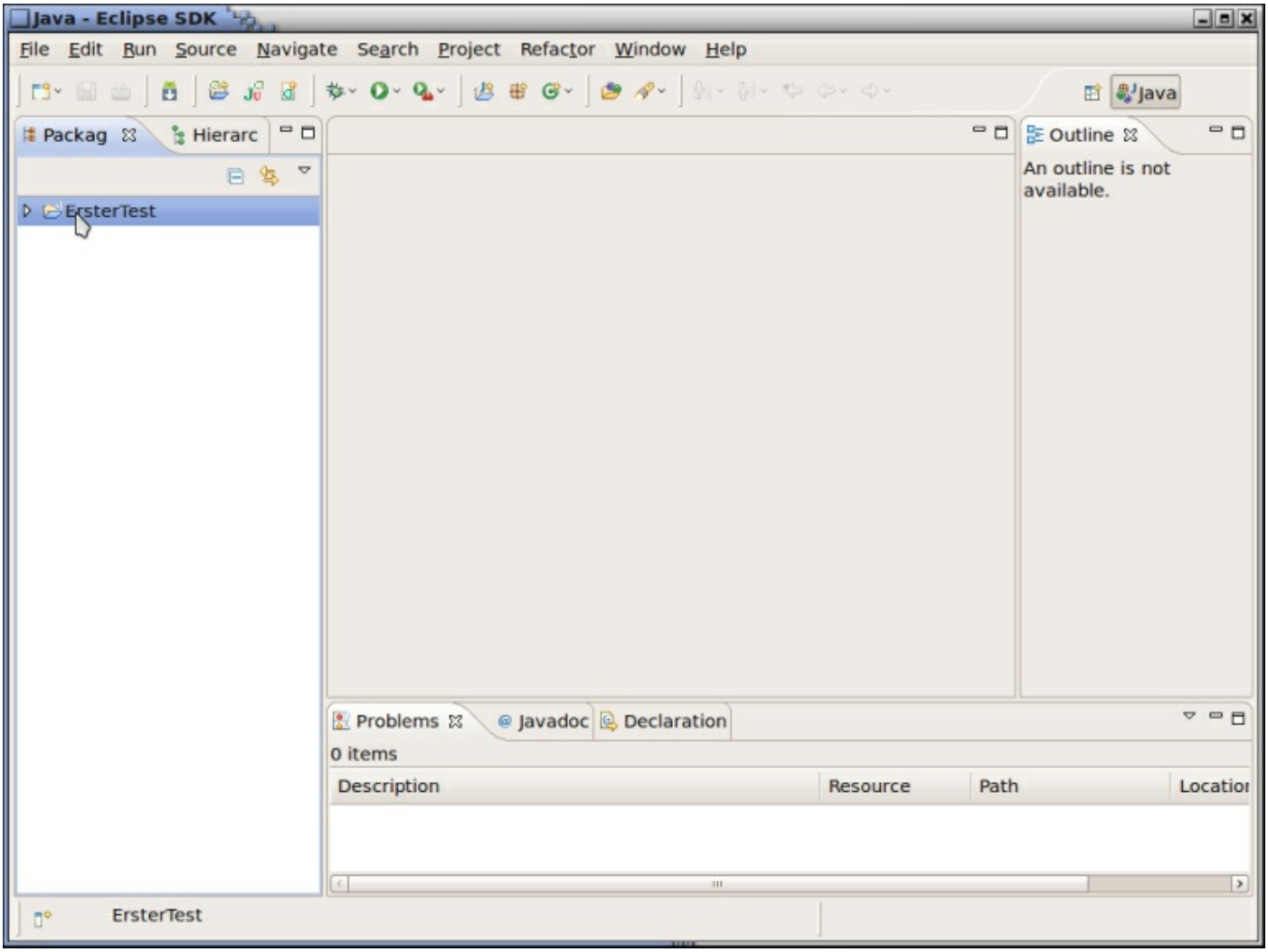


Abbildung 2.2: Eclipse nach Anlegen des Projekts

In der linken Spalte sehen Sie nun das Projekt `ErsterTest`. Um mit einem Projekt etwas zu tun, können Sie es mit der rechten Maustaste anklicken. Für ein Programm benötigen wir eine Klasse, weil in Java ohne Klasse gar nichts geht. Also wählen wir aus dem Kontextmenü New und dann Class. Und wieder erfreut uns ein Dialog mit Eingabemöglichkeiten.

- Den Eintrag für Source Folder können Sie lassen. Er bezeichnet den Ort, wo die Dateien liegen sollen.
- Im Feld Name blinkt der Cursor. Hier erwartet Eclipse den Namen der Klasse. Und mit meinem bereits legendären Einfallsreichtum schlage ich `ErsteKlasse` vor. Leerzeichen dürfen dort nicht erscheinen, sonst wird der Dialog im Kopfbereich rot.
- Das Feld Package lassen Sie erst einmal leer. Hier jammert der Dialog mit gelbem Kopfbereich, dass man doch bitte etwas anderes als `default` nehmen möchte. Aber heute sind wir Sadist.

  Sollten Sie doch Erbarmen haben, können Sie auch gern `erstesPaket` verwenden. Der Kleinbuchstabe am Anfang ist Absicht. Ansonsten würde sich Eclipse nämlich schon wieder beklagen, weil Package-Namen üblicherweise mit Kleinbuchstaben beginnen.
- Sie ersparen sich Tipparbeit, wenn Sie einen Haken an dem Feld Public Static Void Main(String[] Args) setzen. Es steht bei Which Method Stubs Would You Like TO Create? und bewirkt, dass Eclipse den Rahmen für ein kleines Programm erstellt.
- Nun klicken Sie auf Finish. Und wie von Zauberhand entstehen im mittleren Editorfeld ein paar Programmzeilen, die ich auch nicht besser hätte machen können.



Eclipse markiert mit roter Farbe einen Fehler. Die Farbe Gelb steht für eine Warnung. Hier könnten Sie also etwas besser machen. Warnungen weisen manchmal auf verborgene Fehler hin.

```
package erstesPaket;
public class ErsteKlasse {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}
```

Listing 2.2: Von Eclipse generiertes Programm

In der Zeile unter »TODO« können Sie nun die folgende Zeile einfügen, die Sie

schon von unserem Kleinprogramm her kennen.

```
System.out.println("Hallo Welt!");
```

In der Programmanweisung wird über das Objekt `out` der Klasse `System` die Methode `println` aufgerufen. Sollten Sie den letzten Satz nicht ganz verstehen, ist das ein guter Grund, das Buch noch ein wenig weiter zu lesen. Vielleicht haben Sie gesehen, wie während des Eintippens immer wieder Menüs aufgeklappt sind. Eclipse ahnt schon, dass Sie auch nicht genau wissen, welche Bestandteile welche Klassen haben, und schlägt Ihnen darum immer wieder etwas vor. Die Methode `println` jedenfalls druckt alles, was in der Klammer steht, auf dem Bildschirm aus. Diesen Text werden Sie bei der Ausführung des Programms also irgendwo finden.

Übersetzung und Testlauf

Bevor das Programm gestartet werden kann, muss es der Compiler in die Finger bekommen. Tatsächlich müssen Sie sich darum nicht kümmern, weil unter dem Menü Project ein Haken bei Build Automatically steht. Das ist der Normalzustand. Und so übersetzt Eclipse von sich aus das Programm immer dann, wenn es meint, dass es mal wieder angebracht wäre.

Fehlerbehandlung

Sollte es dabei Fehler finden, setzt es eine rote Markierung neben die Zeile. Der fehlerhafte Ausdruck ist rot unterkringelt. Auch rechts neben dem Scrollbar erscheint eine kleine rote Markierung, die Sie in einer längeren Quelltextdatei durch Anklicken direkt zum Fehler führt.

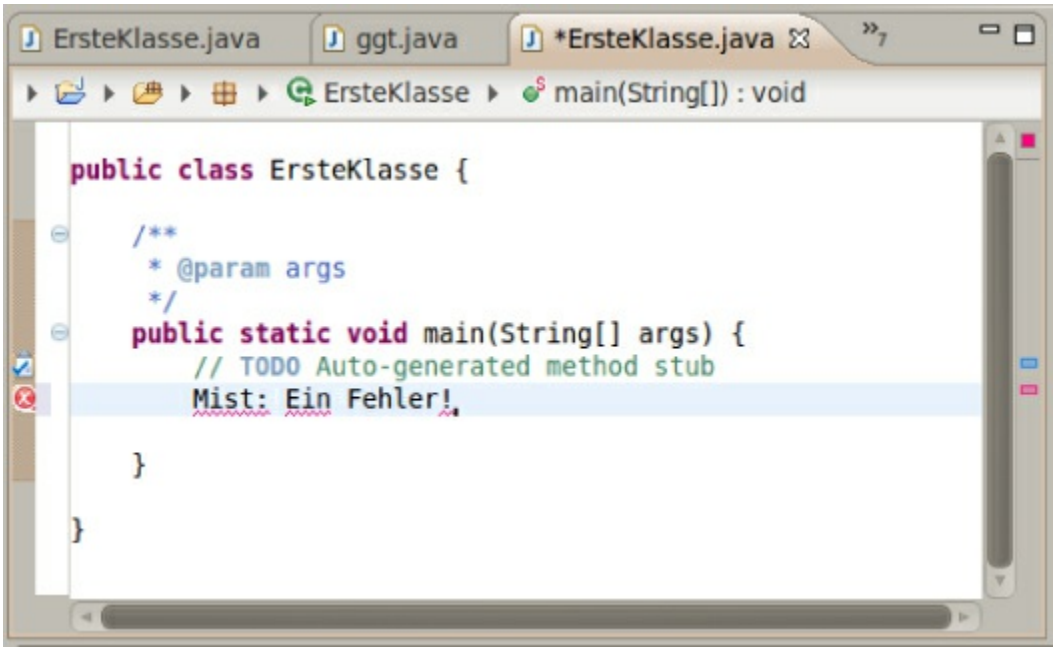


Abbildung 2.3: Ein Fehler im Eclipse-Editor

Wenn Sie die Maus über die Fehlermarkierung schieben, erzählt Ihnen Eclipse, was es an dem Programm nicht so toll findet. Klicken Sie die Fehlermeldung links neben der Zeile an, unterbreitet Eclipse sogar Vorschläge, wie Sie diese missliche

Situation beheben können. Durch Anklicken des Vorschlags erscheinen noch weitere Hinweise in einem Nachbarfenster. Doppelklicken Sie den Vorschlag und Eclipse setzt ihn sofort um und erzeugt dazu selbstständig Quellcode, wenn es erforderlich ist.



Mit ein wenig Übung können Sie durch Fehleranklicken Eclipse dazu bringen, einen Großteil Ihres Programms für Sie zu schreiben. So macht Programmieren richtig Spaß.

Eine Übersetzung wird auf jeden Fall erzwungen, wenn Sie über das File-Menü sichern.

Programmstart

Nun wollen wir das Programm starten. Sie finden in der Menüleiste den Punkt Run, der vielversprechend aussieht. Und tatsächlich. Dort gibt es noch einmal den Punkt Run. Eclipse will nun wissen, ob Ihr Programm ein »Java-Applet« oder eine »Java-Application« ist. Ich kann Ihnen versichern, dass es eine Applikation ist.

Nun fragt Eclipse eventuell, ob es nicht sicherheitshalber mal Ihr Programm speichern sollte, bevor es gestartet wird. Das ist eigentlich eine ganz tolle Idee. Nachdem Sie dies mit OK bestätigt haben, passiert nichts.

Oder? Wenn Sie genau hinsehen, können Sie unterhalb Ihres Programmtextfensters ein schmales Fenster sehen, das mit Console überschrieben ist. Dort steht nun plötzlich »Hallo Welt!«.

Schlagen Sie sich auf die Schulter: Sie haben Ihr erstes Programm unter Eclipse zum Laufen gebracht.

Erzeugen einer JAR-Datei

Sie wollen Ihr Programm nun aber auf einen USB-Stick schieben und an alle Ihre Freunde schicken, damit die sehen, dass Sie programmieren können? Bei der Gelegenheit können Sie gleich prüfen, ob ich Ihnen die Wahrheit erzählt habe und Java-Programme wirklich auf Windows, Linux und Mac laufen.

1. Klicken Sie das Projekt mit der rechten Maustaste an. Im Menü wählen Sie Export....
2. Es erscheint ein Dialog, in dem der Typ der Zieldatei ausgewählt werden kann. Unter der Gruppe Java finden Sie den Punkt Runnable JAR File.
3. Ein Klick auf Next führt zu dem Dialog, in dem der Name der Datei und ihr Ort bestimmt wird.
4. Mit dem Button Browse... können Sie einen Ordner auswählen, in den die Datei geschrieben werden soll. Als Dateiname hinter JAR File geben Sie beispielsweise *gruss.jar* an.

5. Nun klicken Sie Finish an und finden an der Stelle, die Sie angegeben haben, die Datei *gruss.jar*.

NetBeans

NetBeans ist wie Eclipse eine integrierte Entwicklungsumgebung, die Ihnen kostenlos und für alle Plattformen zur Verfügung steht. Ich verwende in diesem Buch zwar Eclipse, NetBeans ist aber auch ein feines System, das zu Recht viele Freunde hat. Die Stärken von NetBeans liegen vor allem darin, dass Sie sich Ihre Oberflächen zusammenklicken können.

Installation

NetBeans wird kostenlos im Internet unter der Heimatadresse `www.netbeans.org` angeboten. Auf der Download-Seite können Sie sich für Windows oder Mac eine Version herunterladen. Für Linux gibt es dort natürlich auch eine Version. Aber bei Linux ist es einfacher und sinnvoller, Software über die Linux-Paketverwaltung zu installieren.

`http://netbeans.org/downloads`

NetBeans wird für alle möglichen Plattformen wie Windows, Linux, Mac oder Solaris angeboten. Sogar ein Paket mit einem OS-unabhängigen Zip findet sich in der Liste. Da NetBeans in Java geschrieben ist, sollte es schließlich im Idealfall auf jeder Plattform laufen, für die eine JRE verfügbar ist.

Sie werden für den Download nach der gewünschten Technologie gefragt. Für die Aufgaben in den ersten Kapiteln dieses Buches benötigen Sie mindestens Java SE. Es ist aber durchaus klug, gleich die Komplettversion zu verwenden, die auch mit JEE umgehen kann. Sie können an dieser Stelle die Sprache auch auf Deutsch umstellen. Wenn Sie den Download starten, erhalten Sie ein Installationspaket, das auf Ihr System zugeschnitten ist.

- Windows: Sie erhalten eine ausführbare Installationsdatei mit der Dateiendung *.exe*, die Sie als Administrator starten.
- Linux: Sie erhalten eine Installationsdatei mit der Dateiendung *.sh*, die als Shell-Skript vom Administrator gestartet werden muss, beispielsweise über den folgenden Befehl.
  ```
  sudo bash netbeans-8.2-linux.sh
  ```
- MacOS: Sie erhalten eine ausführbare Installationsdatei mit der Dateiendung *.dmg*, die Sie doppelklicken.



Abbildung 2.4: NetBeans wird installiert.

Es startet das NetBeans-Installationsprogramm, das Sie einmal um das Akzeptieren der Lizenzbedingungen bittet und anschließend zeigt, wo das Paket installiert wird, und Änderungen erlaubt.

Nach dem ersten Start bietet Ihnen NetBeans eine nette Kennenlernrunde an. Sobald Sie damit fertig sind, können Sie neben der Lasche Startseite das Kreuz anklicken, um mit NetBeans arbeiten zu können.

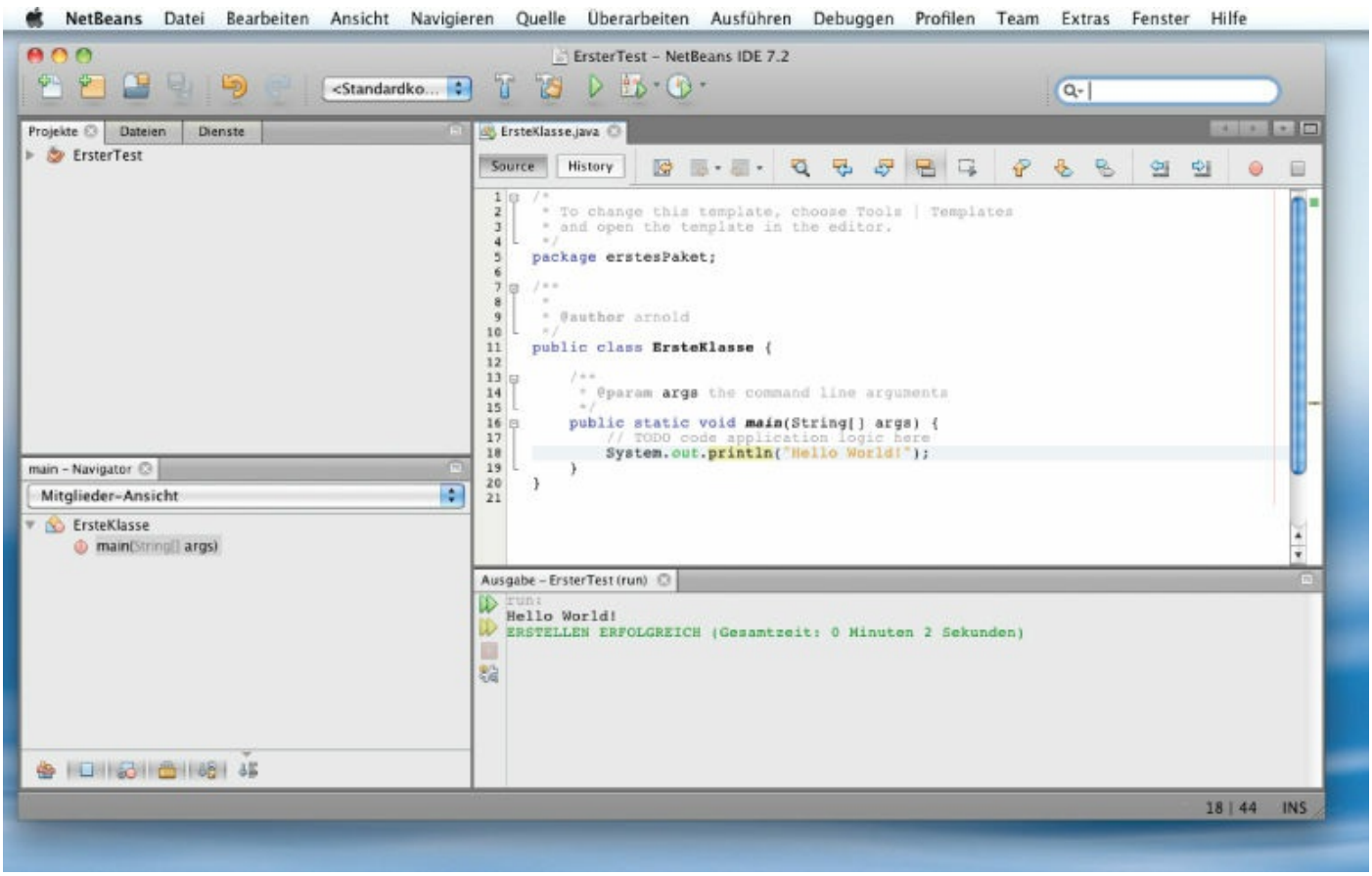


Abbildung 2.5: Die IDE NetBeans

Anlegen eines neuen Projekts

Wenn Sie ein neues Programm schreiben wollen, müssen Sie zunächst ein neues Projekt eröffnen, das Sie über das Menü File|New Project ... (Datei|Neues Projekt...) erstellen können. Daraufhin öffnet sich ein Dialog, der zwei Listen anzeigt. Aus der Liste der Kategorien wählen Sie Java und aus der Liste der Projekte Java Application. Über den Button Next (Weiter) gelangen Sie erwartungsgemäß in den nächsten Dialog, dem Sie mit weiteren Informationen über das Projekt eine Freude machen können.

Analog zum Eclipse-Projekt soll der Projektname auch hier ErsterTest heißen. In dem Eingabefeld, das mit Create Main Class (Hauptklasse Erzeugen) beschriftet ist, steht erstertest.Main. Der Anteil vor dem Punkt bezeichnet den Packagenamen und der dahinter ist der Name der Hauptklasse des Projekts. Um analog zum Eclipse-Projekt zu bleiben, ändern wir dies in erstesPaket.ErsteKlasse.

Der daraus entstehende Programmrahmen ist sehr ähnlich zu dem von Eclipse, wenn Sie die Kommentare nicht berücksichtigen. Ein Kommentar ist alles zwischen /* und */, sowie alles, was in der Zeile hinter zwei Schrägstrichen // steht.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package erstesPaket;

/**
 *
 * @author arnold
 */
public class ErsteKlasse {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }

}
```

Listing 2.3: Generiertes Programm

Unter der Zeile, in der »TODO«««« steht, schreiben wir nun noch den Gruß an die Welt. Ansonsten hätte das Programm ja gar nichts zu tun.

```
System.out.println("Hallo Welt!");
```

Übersetzung und Testlauf

Das frisch erstellte Programm soll nun zeigen, was es kann. Im Menü wird Run|Run Main Project (Ausführen|Projekt Ausführen) angeklickt oder die Taste F6 verwendet. Und im Fenster rechts unten erscheint die Ausgabe:

```
run:
Hallo Welt!
BUILD SUCCESSFUL (total time: 0 seconds)
```

Erzeugen einer JAR-Datei

Um eine JAR-Datei zu erzeugen, klicken Sie in der linken Spalte in der Liste Projects das Projekt *ErsterTest* mit der rechten Maustaste an. Es fällt Ihnen ein Kontextmenü entgegen, aus dem Sie Clean And Build (Bereinigen Und Erstellen) auswählen. Im Ausgabefenster rechts unten erscheinen die Meldungen, die die Generierung der JAR-Datei begleiten. Die letzten Zeilen sind dabei interessant.

```
Building jar: /home/...NetBeansProjects/ErsterTest/dist/ErsterTest.jar
To run this application from the command line without Ant, try:
java -jar “/home/.../NetBeansProjects/ErsterTest/dist/ErsterTest.jar”
jar:
BUILD SUCCESSFUL (total time: 1 second)
```

Die Meldung besagt, dass NetBeans erfolgreich eine JAR-Datei erzeugt hat und wo Sie sie auf Ihrem Computer finden können. Der Ort könnte bei Ihnen etwas anders aussehen.

Aufgabe: Grüßaugust selbst erstellen

Lassen Sie sich nichts von mir erzählen, was alles angeblich klappt und was nicht. Programmieren lernt man in erster Linie dadurch, dass man es tut.

Erstellen Sie deswegen ein neues Programm mit dem JDK oder mit Eclipse oder mit was auch immer Sie möchten. Grüßen Sie doch mal jemand anderen als die Welt: den Mond, den Nachbarn, Ihren Lieblingsautor! Und grüßen Sie anschließend in zwei oder mehr Zeilen zum Beispiel alle Verwandten auf einmal. Vielleicht grüßt ja mal einer zurück.

Experimentieren Sie mit der IDE und machen Sie ruhig mal etwas kaputt. So teuer sind Computer heutzutage nicht mehr.

Für diese Aufgabe gibt es aus naheliegenden Gründen noch keine Musterlösung, für die späteren Aufgaben wird es aber welche geben. Und die können Sie sich von `www.willemer.de/java4d` herunterladen.

Kapitel 3

Aufstieg zum Programmierer

IN DIESEM KAPITEL

Klammern umschweifen Blöcke.

Wir taufen Bezeichner und kommentieren Programme.

Variablen werden als primitive Typen identifiziert.

Auf Java kann man rechnen.

Das erste Kleinprogramm und die Arbeitsumgebung habe ich Ihnen nun vorgestellt. Nichts hält uns davon ab, mit dem Programmieren zu beginnen.

Java-Programm-Skelett

Ich werde nicht müde, Ihnen das hinreißende Programm aus dem letzten Kapitel zu präsentieren. Mit zehn Wiederholungen habe ich nämlich bereits eine Seite herausgeschunden und nach wenigen Tausend Wiederholungen könnte ich das Buch allein auf diese Weise füllen. Aber der eigentlich Grund ist natürlich, dass Sie das Grundgerüst von Java-Programmen memorieren sollen.

```
public class ErsteKlasse {
    public static void main(String[] args) {
        System.out.println("Hallo Welt!");
    }
}
```

Listing 3.1: Hallo Welt zum Dritten

Das Programm wird später auf dem Bildschirm die Wörter »Hallo Welt!« ausgeben. Ich gehe mit Ihnen zeilenweise durch das Programm. Sie werden vielleicht nicht alles sofort verstehen, aber keine Sorge: Es wird alles noch einmal später im Buch ausführlicher erläutert.

```
public class ErsteKlasse {
```

Diese Befehlszeile eröffnet die Klasse `ErsteKlasse`. Sezieren wir die Bestandteile:

- `public`: Wie beim Public Viewing bedeutet das Schlüsselwort `public` auch hier, dass alles, was folgt, ruhig jeder sehen darf. Es dürfen auch alle gern daran

herumfummeln. Hier bezieht sich `public` auf die nachfolgende Klasse `ErsteKlasse`. Später lernen Sie, wie Sie aus Teilen Ihres Programms eine Privatsache machen. Und dann werden wir auch klären, wie und warum man so etwas tut.

- `class`: Java will damit nicht seine besondere Klasse als Programmiersprache herausstellen, sondern verwendet Klassen, um Programmteile gegeneinander abzugrenzen. Die Einteilung eines Programms in Klassen stellt die Basis der objektorientierten Programmierung dar. Sie können darauf wetten, dass der Begriff der Objektorientierung in diesem Buch noch einmal auftaucht.
- `ErsteKlasse`: Jede Klasse erhält einen Namen. Nicht immer ist er so originell gewählt wie in diesem Beispiel. Aber Sie können ihn frei wählen. Üblicherweise lassen Programmierer Klassennamen mit einem Großbuchstaben beginnen.
- `{ }`: Geschweifte Klammern umfassen einen Block. Es ist möglich, diese Klammern zu verschachteln. Sie verhalten sich also ein wenig wie die Schalen einer Zwiebel, riechen aber glücklicherweise nicht so. Die öffnende, geschweifte Klammer an dieser Stelle markiert den Beginn der Klasse. Irgendwann muss eine schließende, geschweifte Klammer folgen, sonst wird der Computer sauer.

```
public static void main(String[] args) {
```

In der inneren Schale der geschweiften Klammer wird die Methode `main` definiert. Auch diese Zeile zerlegen wir in Einzelteile:

- `public`: Auch der Inhalt der inneren Schale ist öffentlich.
- `static`: Alles, was in Java als `static` bezeichnet wird, ist einmalig, vielleicht nicht einmalig gut, aber es kommt nur einmal vor.
- `void`: Dieses Wort steht für nichts und bedeutet in diesem Fall, dass die Methode `main` an den Aufrufer nichts zurückgibt. Methoden können nämlich Werte an ihre Aufrufer zurückgeben. Welchen Typ diese Rückgabe hat, steht immer vor der Methode. Im Falle von `main` ist das eben `void`, also nichts.
- `main`: Der Name dieser Funktion lautet `main`. Das steht für das englische Wort für »Haupt«, hat also nichts mit dem Fluß bei Frankfurt zu tun. Für den Programmstart müssen Sie den Namen `main` verwenden. Wenn Sie später eigene Methoden verwenden, dürfen Sie den Namen frei wählen.
- `(String[] args)`: Jede Methodendefinition führt nach dem Methodennamen ein Paar runde Klammern an. Innerhalb der Klammern steht, was der Aufrufer der Methode an Daten übergibt. Tatsächlich erhält die Methode `main` unter dem Namen `args` eine Reihe von Zeichenketten. Zunächst können Sie diese

Parameter ignorieren.

- `{ }`: Wieder folgt eine öffnende, geschweifte Klammer. Sie umfasst den Inhalt der Methode und damit alle Befehle, die darin ausgeführt werden. Wenn Sie also eigene Programme schreiben, sollte das hier passieren.

Die statische, öffentliche Methode `main` hat eine besondere Funktion. Das Java Runtime Environment sucht nach dieser Methode, um das Programm zu starten. Fehlt diese Methode, kann das Programm nicht laufen. Egal, ob Ihnen diese Zeile gefällt oder nicht. Sie werden sie einmal in dieser Form in Ihrem Programm einbauen müssen, da ein Programm, das nicht startet, recht sinnlos ist.

```
System.out.println("Hallo Welt!");
```

Diese Zeile ist tatsächlich die einzige, die zu Aktivitäten führt.

- `System.out.println`: Das Programm ruft eine von Java vorgefertigte Methode namens `println` auf. Der Befehl bedeutet so viel wie »drucke den Inhalt der Klammern mit einem abschließenden Zeilenumbruch«. Die Methode ist Bestandteil des Pakets `out`, das wieder im Paket `System` enthalten ist. Sie werden durch Punkte getrennt.
- `("Hallo Welt!")`: Innerhalb der Klammern steht ein in Anführungszeichen gesetzter Text. Die Anführungszeichen sind nicht nur in wissenschaftlichen Arbeiten ein Hinweis auf ein wörtliches Zitat, sondern auch in Java ein Hinweis darauf, dass der Text dazwischen unverändert ausgegeben werden soll.
- `;`: Jede Anweisung wird mit einem Semikolon am Ende verziert. Wenn Sie es weglassen, ist es allerdings nicht nur ein Schönheitsfehler. Der Compiler wird sich sofort beschweren.

Es folgt eine schließende, geschweifte Klammer, die die innere Schale und damit die Methode `main()` schließt. Die nachfolgende schließende Schweifklammer schließt dann die äußere Schale der Klasse `ErsteKlasse`. Und damit gehört die Methode `main()` zur Klasse `ErsteKlasse`.

Programmerweiterungen

Das Programm aus Listing 3.1 enthält nur eine Zeile, die wirklich etwas tut, nämlich die Ausgabe auf dem Bildschirm mittels `System.out.println`. Statt die Leute zu grüßen, soll der Computer das tun, wofür er erfunden wurde, nämlich rechnen. Dazu ersetzen wir den Inhalt der Klammern durch eine einfache Rechenaufgabe:

```
public class ErsteKlasse {
    public static void main(String[] args) {
        System.out.println(17+4);
    }
}
```

Listing 3.2: Rechenergebnisse statt Grüße

Natürlich ist das Schreiben eines Java-Programms für diese Aufgabe ein Overkill. Selbst diejenigen, die in der Grundschule geschlafen haben, werden im späteren Leben mit einem einfachen Taschenrechner schnellere Ergebnisse liefern.

Starten des Programms unter Eclipse

Das Programm ist eingetippt. Nun ist es mit der Faulheit vorbei, der Computer soll etwas tun, und zwar das Programm starten. Das können Sie auf verschiedene Arten herbeiführen:

- ✔ Klicken Sie mit der rechten Maustaste Ihr Projekt an und wählen Sie im Kontextmenü Run AS|Java Application aus.
- ✔ Rufen Sie im Hauptmenü Run|Run auf.
- ✔ Drücken Sie die Tastenkombination Strg + F11.
- ✔ Klicken Sie auf das grüne Dreieck in der Symbolleiste, wie in Abbildung 3.1 zu sehen. Wenn Sie auf den Pfeil rechts neben dem Dreieck klicken, sehen Sie, welche Programme in letzter Zeit gestartet wurden. Daraus können Sie sich gern eines aussuchen und noch einmal starten.

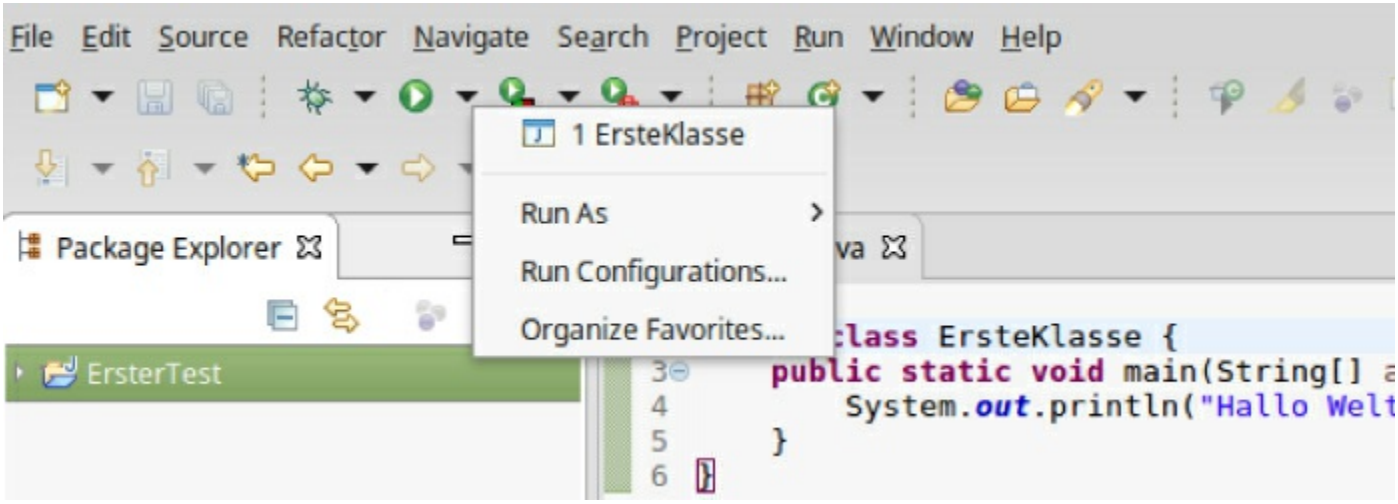


Abbildung 3.1: Start über das grüne Dreieck oder dessen Menü

Wenn Sie dieses Programm starten, erscheint in dem Eclipse-Fenster ganz unten, das mit Console beschriftet ist, statt des Grußes die Zahl 21.

Sie sehen, dass in Listing 3.2 die Anführungszeichen verschwunden sind. Ansonsten würde »17+4« erscheinen und Sie müssten es selbst ausrechnen. Das

wollen wir nicht. Ansonsten hätten Sie sich keinen teuren Computer kaufen müssen.

Eingabe des Benutzers

Sie können nun die Zahlen durch andere ersetzen und werden begeistert feststellen, wie toll Java addieren kann. Es ist natürlich doof, dass Sie ständig die Werte im Programm ändern müssen. Es wäre nett, wenn ein Anwender nach dem Start wenigstens eine der beiden Zahlen ändern könnte. Die folgenden zwei Zeilen sorgen dafür, dass eine Benutzereingabe gespeichert wird:

```
java.util.Scanner scan = new java.util.Scanner(System.in);
int zahl;
zahl = scan.nextInt();
```

Die erste Zeile erzeugt einen Scanner, zu Deutsch Abtaster. Er bezieht sich auf den Standardeingabekanal `System.in`, den man weniger schwülstig als Tastatur bezeichnen könnte. Wie diese Zeile, insbesondere der Befehl `new` funktioniert, werden Sie in späteren Kapiteln herausfinden können. Derzeit müssen Sie es vielleicht hinnehmen wie die Zaubersprüche von Harry Potter, deren genaue Übersetzung weniger interessant ist als ihre Wirkung.

Die zweite Zeile legt einen Datenspeicher für Zahlen an. Das Programm muss die später eingegebene Zahl irgendwo lassen. In solchen Fällen werden Variablen Verwendet. Das Schlüsselwort `int` gibt an, dass die Variable ganze Zahlen aufnimmt. Es folgt der Name der Variablen, die etwas fantasielos `zahl` genannt wurde.

Die dritte Zeile verwendet das in der ersten Zeile angelegte Objekt `scan` und fordert über dessen Methode `nextInt` vom Programmanwender die nächste Eingabe ein und interpretiert diese als Zahl. Damit die Zahl in die Variable gelangt, steht vor dem Aufruf von `scan.nextInt` die Variable namens `zahl`, gefolgt von einem Gleichheitszeichen.



Die Kombination aus Variable und Gleichheitszeichen leitet eine *Zuweisung* ein. Alles, was dem Gleichheitszeichen folgt, wird dabei in die Variable gestopft.

Wenn Sie anschließend eine weitere Zahl vom Benutzer haben wollen, müssen Sie die zweite Zeile wiederholen. Vermutlich ist es aber sinnvoll, dass Sie sich dann einen neuen Namen statt `zahl` einfallen lassen. Ansonsten wird der bisherige Wert überschrieben.

```
public class ErsteKlasse {
    public static void main(String[] args) {
        java.util.Scanner scan = new java.util.Scanner(System.in);
        int zahl;
        zahl = scan.nextInt();
        System.out.println(zahl);
    }
}
```

Listing 3.3: Eingabe einer Zahl

Das Programm können Sie in Eclipse in den folgenden Schritten ablaufen lassen:

1. Sie starten das Programm wie gehabt, beispielsweise über die Tastenkombination Strg + F11.
2. Um die Eingabe in das Programm laufen zu lassen, müssen Sie das Teilfenster von Eclipse mit dem Reiter Console aktivieren. Klicken Sie zunächst auf den Reiter, dann in das darunterliegende Fenster.
3. Nun können Sie eine Zahl eingeben. In Abbildung 3.2 ist dies eine 12.
4. Nachdem Sie die Zahl eingegeben haben, erscheint diese direkt darunter.

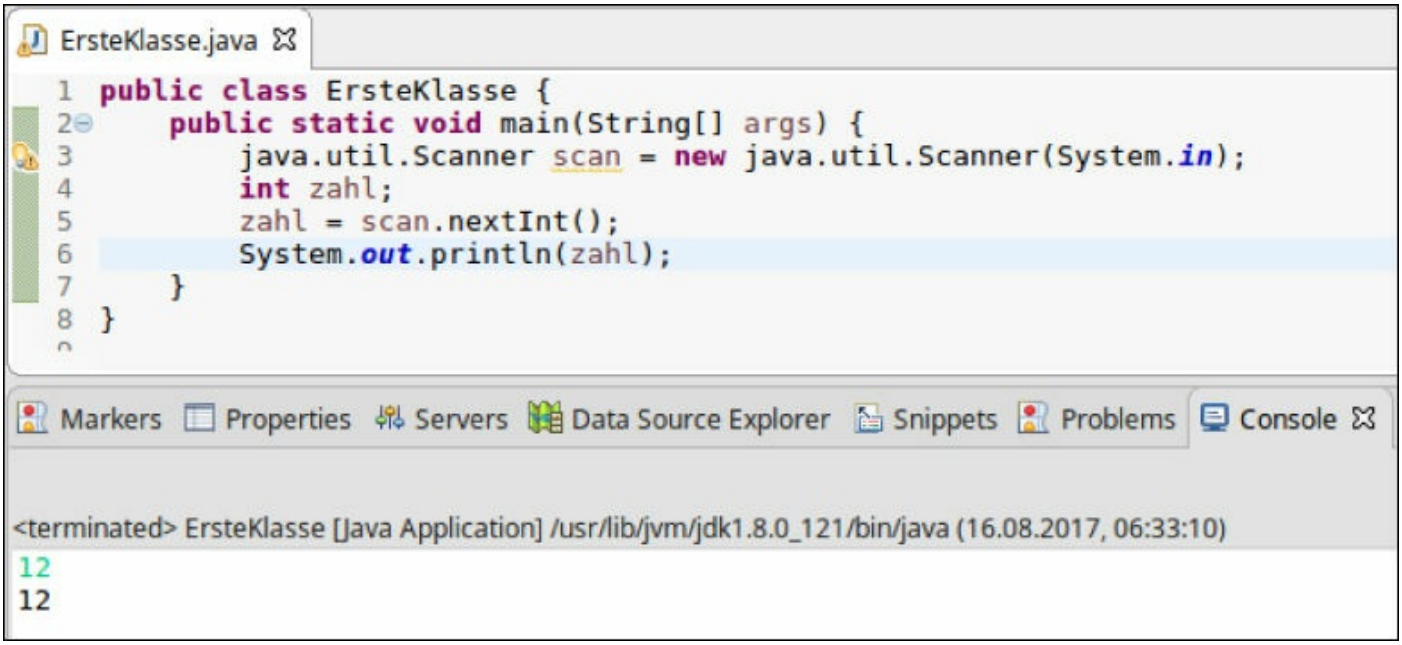


Abbildung 3.2: Eingabe einer Zahl

Die Variable `zahl` kann in Ihrem Programm verwendet werden wie eine normale Zahl. Versuchen Sie doch einmal, die Ausgabezeile durch die folgende Zeile zu ersetzen:

```
System.out.println(17+zahl);
```

Nun können Sie immer wieder neue Zahlen eingeben, die Sie schon immer einmal zu 17 addieren wollten.

Variabel rechnen

Da die eingegebene Zahl in der Speicherstelle `zahl` gespeichert ist, kann sie auch mehrfach verwendet werden. Die folgende Zeile bewirkt, dass der Inhalt der Variablen `zahl` quadriert und das Ergebnis einer anderen Variablen `quadrat` zugewiesen wird.

```
int quadrat = zahl * zahl;
```

Sie sehen, dass Java für die Multiplikation einen Stern verwendet. Das wirkt etwas weihnachtlich, hat aber seinen Grund darin, dass das mathematische Multiplikationszeichen auf der Tastatur nicht vorhanden ist. Im Programm wird die Zeile folgendermaßen eingebettet:

```
public class ErsteRechnung {
    public static void main(String[] args) {
        java.util.Scanner scan = new java.util.Scanner(System.in);
        int zahl = scan.nextInt();
        int quadrat = zahl * zahl;
        System.out.println(quadrat);
    }
}
```

Listing 3.4: Berechnung einer Quadratzahl

Nach dem Start klicken Sie in das Teilfenster Console und geben eine Zahl ein. Das Quadrat der Zahl erscheint direkt unter der Eingabe.

Variable Attribute

Fressen wir uns doch beim Thema Variablen ruhig ein wenig fest. Immerhin ist das Speichern von Informationen in der Informatik von besonderer Bedeutung. Computer haben ein fantastisches Gedächtnis. Zumindest ist es besser als meins.

Die Speicher der Variablen liegen im Hauptspeicher des Computers, der spätestens beim Beenden des Programms verloren geht. Trösten Sie sich! Wir werden Wege finden, Daten auch längerfristig zu speichern.



Im Umfeld von Java finden Sie auch die Bezeichnung *Attribute*. Variablen, die zu einer Klasse gehören, werden nämlich Attribute genannt. Da in Java aber jedes Programm zu einer Klasse gehört, sind eigentlich alle Variablen streng genommen Attribute.

Variablen sind Aufnahmebehälter für Daten, vergleichbar mit einem Koffer. Relevant ist der Inhalt. Aber nach außen sehen Sie nur den Koffer. Der Koffer sollte folgende Eigenschaften haben:

- ✔ Der Inhalt sollte hineinpassen, wenn möglich sogar exakt. Ein Schrankkoffer

mit einer Unterhose darin ist Verschwendung. Wenn Sie den Koffer nicht mehr schließen können, schützt er den Inhalt nicht mehr und ist sinnlos.

Variablen besitzen einen Typ, der kennzeichnet, welcher Inhalt am besten in die Variable passt.

- Sie sollten von außen erkennen können, was sich im Koffer befindet. Wenn Sie einen Geigenkoffer sehen, erkennen Sie an der äußeren Form, was darin ist. Ein Maschinengewehr ist größer, passt also gar nicht hinein. Koffer, die äußerlich ihren Inhalt nicht erkennen lassen, werden Sie vermutlich beschriften.

  Ähnlich macht man es bei Variablen. Damit klar ist, was sie enthält, gibt man ihr einen passenden Namen.


Variablen tragen einen Namen und enthalten Daten, die im Hauptspeicher liegen.

Abgefüllt und ausgefüllt

Wie kommen nun die Inhalte in die Koffer? Ganz einfach: Koffer auf, Gitarre hinein, Koffer zu. Bei Java ist es noch einfacher! Man stellt den Koffer auf die linke Seite, stellt ein Gleichheitszeichen zwischen Koffer und Gitarre und schon ist die Gitarre im Koffer. Man nennt diesen Vorgang dann eine Zuweisung.

```
mehrwertsteuersatz = 19;
```

Alles, was links vom Zuweisungszeichen steht, wird *LValue* genannt. Sie könnten es auch als »Etwas, dem etwas zuweisbar ist» bezeichnen. Sehen Sie? *LValue* klingt doch besser.


Alles was rechts vom Gleichheitszeichen steht, wird in der links vom Gleichheitszeichen stehenden Variablen abgelegt.

Auch Variablen wollen getauft sein

Falls Sie jemals Eltern darum beneidet haben, dass diese ihren Kindern Namen geben durften, haben Sie als Programmierer den richtigen Weg eingeschlagen. Unter Java dürfen Sie Programme, Klassen, Attribute, Methoden, Pakete und viele weitere Dinge benennen, müssen diese aber weder erziehen noch ernähren.

Falls Sie selbst Kinder haben, wissen Sie, wie lange es dauert, sich einen einzigen Namen einfallen zu lassen. Wenn es genauso lange dauert, bis alle Variablen eines Programms benannt sind, können Jahre vergehen. Aber zum Glück müssen Programmierer die Namen nicht mit der Verwandtschaft abgleichen, sondern nur mit dem Compiler. Und der ist da sehr anspruchslos.

- Der Name muss mit einem Buchstaben beginnen.

- Es dürfen Ziffern und Buchstaben in beliebiger Reihenfolge und Anzahl folgen.
- Der Name darf nicht mit einem reservierten Wort der Sprache Java kollidieren.

Nun hat Java etwas originelle Ansichten darüber, was eigentlich ein Buchstabe ist, der für Bezeichner gewählt werden darf. Im Gegensatz zu vielen anderen Programmiersprachen sind neben a bis z auch die nationalen Sonderzeichen erlaubt. Sie können also deutsche oder dänische Umlaute oder griechische Buchstaben verwenden. Auch Währungszeichen und Sonderzeichen wie der Unterstrich sind möglich. Aber wenn Sie einmal einen Franzosen fluchen gehört haben, als er versuchte, auf seiner französischen Tastatur ein deutsches ß einzutippen, dann wissen Sie, warum sich die meisten Programmierer auf die internationalen Buchstaben beschränken.

Grundsätzlich gilt, dass Java genau zwischen Groß- und Kleinschreibung unterscheidet. Ein `Haus` ist etwas anderes als ein `haus`.


Vergeben Sie Namen, die möglichst klar Auskunft über den Inhalt geben. Sparen Sie nicht mit Buchstaben, aber übertreiben Sie es nicht.

Eine Variable hat immer einen Namen, den Sie sich frei aussuchen können. Sie dürfen Alfredo oder Eulalia wählen. Die fantasievolle Auswahl mag ungeheuer fröhlich sein. Welch ein Spaß, wenn der Kollege über einem Programm brütet, weil er nicht weiß, welcher Inhalt in der Variablen `maerchenwald` steckt! Ist es der Preis oder die Mehrwertsteuer? Besser ist es aber, wenn der Name dokumentiert, was Sie in der Variablen ablegen. Sinnvollere Namen wären also in diesem Fall `preis` oder `mehrwertsteuer`.

Sie merken schon: Variablennamen beginnen typischerweise mit einem Kleinbuchstaben. Ein Name kann dann aus beliebig vielen Buchstaben und Ziffern zusammengesetzt sein, wobei Java wie immer Groß- und Kleinbuchstaben voneinander unterscheidet.

Variablen und Typen

Variablen haben verschiedene Typen. Wenn Variablen die Behälter für Inhalte sind, dann sind sie im Falle von Java Formkoffer. In einen Gitarrenkoffer passen nur Gitarren. In einen Laptopkoffer passen Laptops, aber keine Gitarre. Und auch für Geigen gibt es spezielle Koffer, in die weder ein Cello noch eine Gitarre passen.

Bei anderen Programmiersprachen wie beispielsweise Perl oder PHP gleicht eine Variable eher eine Tüte, in die man alles Mögliche stopfen kann. So eine Tüte ist natürlich sehr viel einfacher im Gebrauch. Ja, aber wie blöd, wenn man zur LAN-Party mit seinem Laptop will und dann aus der Tüte eine Geige zieht. Darum achtet Java sehr streng darauf, dass Sie die Typen beachten.

Es gibt Variablen für Zahlen, Zeichen und Texte. Wenn Sie versuchen sollten, einer

Zahlenvariablen einen Text zuzuweisen, meckert der Compiler. Dessen penible Nörgelei ist zwar lästig, hat aber auf lange Sicht ihre Vorteile. Denn die falsche Zuordnung kann später zu Irritationen führen, wenn durch diese Variable dividiert werden soll und Java nicht weiß, wie dort der Text der Nationalhymne hineingekommen ist.

Der Typ einer Variablen bestimmt also auch die möglichen Operationen: Eine Division durch einen Text ist schwierig.

Primitive Typen

Primitive Typen kennen wir als Leute, die am Stammtisch fremdenfeindliche Sprüche klopfen. Im Zusammenhang mit Java sind *primitive Typen* einfache Datentypen wie Zahlen oder Zeichen. Nichtprimitive Typen gibt es auch. Die haben Referenzen und haben vor allem auch Klasse. Darum werden wir sie noch kennenlernen, wenn es um objektorientierte Programmierung geht. Auch wenn primitive Typen zu komplexeren Typen zusammengefasst werden, handelt es sich nicht mehr um primitive Typen. Das passt allerdings nicht ganz zur realen Welt. Dort werden primitive Typen durch Rudelbildung im Allgemeinen nur noch primitiver.

Ganze Zahlen

Ganze Zahlen kennen Sie noch aus der Schule, damals als Sie das Zählen lernten. Ganze Zahlen können auch negativ sein. Dann steht ihnen ein Minuszeichen voran. Java nennt den Typ für ganze Zahlen *int* für *Integer*.

Java kennt für ganze Zahlen noch zwei weitere Typen: *long* und *short*. Der Unterschied liegt im Wertebereich. Eine Variable vom Typ `short` kann maximal den Wert 32.767 enthalten. Bei `int` sind es schon über 2 Milliarden und der Typ `long` schafft über 9 Trillionen. Das hat seine Ursache im Platzbedarf. Eine Variable vom Typ `short` benötigt den halben und eine Variable vom Typ `long` den doppelten Speicherplatz einer Variablen vom Typ `int`.

Kommen wir noch einmal auf die Gitarrenkoffer zurück. Bei den Gitarrenkoffern gibt es nämlich Unterschiede. Eine 12-saitige Gitarre passt nicht in den Koffer einer 6-saitigen Gitarre, einfach weil sie länger ist. Umgekehrt ist es kein Problem. Auch eine Kindergitarre passt in den Koffer einer Standardgitarre. Im Zweifelsfall muss man noch ein Handtuch hineintun. Umgekehrt geht es nicht. Die Standardgitarre passt nicht in den Kindergitarrenkoffer.

So ist es auch mit den verschiedenen Integer-Typen. Der Inhalt einer Variablen vom Typ `short` lässt sich bequem in einer Variablen vom Typ `long` unterbringen. Der Java-Compiler sagt dann auch gar nichts dazu. Beim Versuch, es umgekehrt zu machen, wird er sich deutlich räuspern.

Ganzzahlige Literale

Sie können einer ganzzahligen Variablen natürlich ganzzahlige Zahlenwerte zuweisen. Einen festen Zahlenwert nennt man auch *Literal*. Das könnte beispielsweise eine 123 sein. Ein normales ganzzahliges Literal ist vom Typ `int`. Wollen Sie einem Literal den Typ `long` zuordnen, hängt man ihm ein kleines oder großes L an, also 123L.

Ganze Zahlen sind natürlich ganz einfach. Lauter Ziffern halt. Vielleicht noch ein Minuszeichen davor, wenn man eher negativ eingestellt ist. Eine Kleinigkeit sollten Sie aber beherzigen: Lassen Sie eine Zahl nicht mit 0 beginnen. Sie könnte sich glatt für oktal halten und dann kommt etwas ganz anderes heraus, als Sie gedacht haben. Obwohl ich vermute, dass Sie furchtbar gern Ihre Freunde mit Ihrer Kenntnis von oktalen Zahlen beeindrucken würden, spare ich mir die nächsten fünf Seiten mit den ausführlichen Erklärungen. Meine Lektorin hat mir nämlich gesagt, ich würde Sie damit nur anöden und ich solle Ihnen endlich das Programmieren beibringen, anstatt mit meinen Kenntnissen der Zahlensysteme zu protzen. Und da hat sie wohl recht.

Casting

Nun kann es aber natürlich sein, dass eine Variable vom Typ `long` eine so kleine Zahl enthält, dass sie auch prima in eine Variable vom Typ `short` passt. Das ist dem Compiler aber egal. Der wird sich stur stellen. Aber als Programmierer sind Sie in der priviligierten Position, den Compiler zwingen zu können. Dazu erklären Sie, dass der Inhalt einer `long`-Variablen doch in eine `short`-Variable passt.

```
public class Casting {
    public static void main(String[] args) {
        short kurz = 2;
        long lang = 3;
        kurz = (short) lang;
        System.out.println(kurz);
    }
}
```

Listing 3.5: Casting: Große Zahl in kleinem Koffer

Sie sagen mit `(short)`, dass Sie die Verantwortung übernehmen und der Compiler gefälligst tun soll, was Sie sagen. Was passiert aber, wenn in `lang` beispielsweise tatsächlich mit 65000 ein Wert steht, der viel zu groß für eine schmächtige `short`-Variable ist? Probieren Sie es aus! Und Sie sehen selbst: Aus 65000 wird -536! Beim Casting hackt der Compiler gnadenlos den Teil der Variablen ab, der nicht in eine `short`-Variable passt. Und wenn in dem Abgehackten etwas anderes als 0 drinstand, gibt es eben Murks. Und aufgrund der Art wie Java seine Zahlen intern ablegt, kann es da auch plötzlich zu negativen Werten kommen.


Casting ist nicht nur im Fernsehen brutal.

Kaputte Zahlen

Nachdem Sie schon so viel Freude an den ganzen Zahlen hatten, werden Sie sich mit Wonne daran erinnern, dass in der Schule anschließend die Brüche kamen. Nun gibt es in Java zwar keine Brüche, aber immerhin Fließkommazahlen, also die Dezimalbrüche.

Und Sie erinnern sich, dass ich bei unserem Beispielprogramm vorab verraten hatte, dass diese vom Typ `float` sind. Aber auch von den Fließkommatypen kennt Java nicht nur den einen Typ, sondern auch noch einen weiteren namens `double`. Und der Name lässt schon erahnen, dass dieser doppelt so viel Platz im Speicher belegt. Das wiederum hat zwei Konsequenzen: Die Zahlen können größer werden, aber vor allem können sie mehr Nachkommastellen speichern und sind dadurch genauer.


Mit den Jahren hat sich die Völlerei durchgesetzt. Java verwendet heutzutage als Standard für Fließkommazahlen `double`.

Das Programm in Listing 3.6 berechnet den Preis für einen Gitarrenkoffer. Preise haben in der Regel Nachkommastellen. Wir verwenden `double`, auch wenn Gitarrenkoffer üblicherweise nicht so teuer sind, als dass dies notwendig wäre. Ansonsten bleibt zu bemerken, dass Java auch außerhalb der Weihnachtszeit einen Stern zum Multiplizieren verwendet und dass der Schrägstrich das Divisionszeichen ist.

```
public class GitarrenKoffer {
  public static void main(String[] args) {
    double koffer = 49;
    double mehrwertsteuersatz = 19;
    double mwst = koffer*mehrwertsteuersatz/100;
    System.out.println(mwst);
  }
}
```

Listing 3.6: Ein Gitarrenkoffer und seine Mehrwertsteuer

Es gibt drei Variablen: `koffer`, `mehrwertsteuersatz` und `mwst`.

Den ersten beiden wird direkt ein Wert zugewiesen, nämlich einmal der Nettopreis des Gitarrenkoffers in Euro und einmal der Mehrwertsteuersatz. Der Wert der Variablen `mwst` wird dann aus diesen Variableninhalten errechnet.

Projekt anlegen und ausführen

Wie bereits bei den Grüßprogrammen legen Sie ein passendes Projekt in Eclipse an.

Darin lassen Sie sich eine Klasse `GitarrenKoffer` erstellen und geben im Editor die Zeilen oben ein. In Eclipse starten Sie über das Menü Run|Run und sehen dann im Konsolenfenster das Ergebnis 9.31. Es gehen also 9,31 Euro an unsere Regierung.

Sie können natürlich auch eine andere IDE verwenden oder eine Datei *GitarrenKoffer.java* erstellen, diese mit dem Befehl `javac` übersetzen und dann mit dem Befehl `java` ausführen.

```
javac GitarrenKoffer.java
java GitarrenKoffer
9.31
```

Aber nun wollen wir noch wissen, was uns der Gitarrenkoffer insgesamt kostet. Dazu müssten wir den Nettopreis für den Gitarrenkoffer und die Mehrwertsteuer zusammenzählen. Aber warum soll das Programm das nicht auch noch tun? Dazu versuchen Sie, die folgenden Zeilen an der richtigen Stelle in das Programm einzufügen:

```
double preis;
preis = koffer+mwst;
System.out.println(preis);
```

Listing 3.7: Ein Gitarrenkoffer inklusive Mehrwertsteuer

Wenn alles geklappt hat, sehen Sie die folgenden Zeilen auf dem Bildschirm:

```
9.31
58.31
```

Damit läuft Ihr erstes Programm, das selbst etwas tut. Da sind natürlich jetzt ein paar Fragen offen. Ich hatte es etwas eilig, um nicht allzu lange in der theoretischen Phase zu bleiben. Vermutlich haben Sie schon erraten, wie das Ganze funktioniert. Was Ihnen nun noch rätselhaft erscheint, werden Sie in den nächsten Abschnitten näher kennenlernen.

Initialisierung

Das erstmalige Zuweisen eines Wertes einer Variablen nennt man *Initialisierung*. Das haben Sie im Beispiel bereits gesehen.

```
float koffer = 49;
```



Es ist klug, jede Variable bei der Deklaration zu initialisieren. Java beschafft zwar den Platz für die Variable, besetzt sie aber nicht immer mit einem Wert. Wird die Initialisierung übersehen, ist das Risiko groß, dass Sie mit einem völlig zufälligen Wert in der Variablen weiterrechnen und später wunderliche Ergebnisse sehen.

Sie können auch gleich mehrere Variablen in einer Zuweisung belegen.

```
int a, b, c;
a = b = c = 3;
```

Der Variablen `b` wird hier das Ergebnis der Zuweisung `c = 3` zugewiesen. Eine Zuweisung kann also eben wieder ausgewertet werden. Java wertet die Zuweisungen von rechts nach links aus. Zunächst wird der Variablen `c` der Wert 3 zugewiesen. Der Inhalt von `c` wird dann an `b` durchgereicht und schließlich erhält `a` den Wert von `b`.

Kaputte Literale

Fließkommazahlen enthalten verrückterweise gar kein Komma, sondern einen Punkt. So lautet der Mehrwertsteuersatz beispielsweise 0.19 statt 0,19. Das liegt daran, dass Java aus dem angelsächsischen Umfeld stammt.

Fließkommaliterale sind nicht etwa vom Typ `float`, sondern vom Typ `double`. Wenn Sie einem Literal unbedingt klarmachen wollen, dass es vom Typ `float` und nicht `double` sein soll, hängen Sie ihm ein großes oder kleines F an, hier also 0.19f.

Um Fließkommaliterale von ganzzahligen Literalen abzusetzen, können Sie einfach .0 anhängen, wie beispielsweie bei 123.0.

Fließkommazahlen können recht groß werden. Zwei Millionen ist eine 2 mit sechs Nullen. Als Java-Literal sähe dies so aus: 2000000. Das ist etwas unübersichtlicher als die Schreibweise 2E6. Inhaltlich entspricht das eben einer 2 mit 6 Nullen oder genauer $2 \cdot 10^6$. Zweieinhalb Millionen können Sie als 2.5E6 darstellen. Übrigens darf anstelle des großen E auch ein kleines e verwendet werden.

Hinter dem E können auch negative Zahlen stehen. Das ist dann für sehr kleine Werte. 8 Promille sind beispielsweise 8 Tausendstel. In Java können Sie 8 Promille als 8E-3 darstellen. Das E steht für die Zahl 10. Die nachfolgende Zahl ist deren Exponent. 10^{-1} ist genauso ein Zehntel wie 1E-1. Also könnten wir den Mehrwertsteuersatz auch als 19E-2 schreiben. Überlegen Sie, welche Zahl 235e-2 ist. Oder was ist 0.001e4? Das ist aber eigentlich alles ein alter Hut. Ihr Taschenrechner macht es genauso.



Eine Fließkommazahl beginnt optional mit einem Vorzeichen. Auf jeden Fall gibt es die Mantisse. Es kann ein Exponent folgen, der durch ein E angekündigt wird. Der Exponent kann wiederum ein Vorzeichen haben.

Die *Mantisse* könnte man als Basiszahl bezeichnen. Sie enthält mindestens eine Ziffer. Optional kann sie maximal einen Dezimalpunkt enthalten. Den gleichen Regeln folgt der Exponent.

Noch ein paar Primitive

Neben den vorgestellten Zahlen kennt Java noch ein paar primitive Typen.

- `boolean`: Da gibt es auch noch den Typ `boolean`. Er speichert die Wahrheitswerte und kennt nur zwei Inhalte: `true` und `false`. Wir werden damit noch unseren Spaß bei Abfragen und Schleifen haben.
- `byte`: Da Computer total auf Bytes stehen, hat ihm Java zu Ehren den primitiven Typ `byte` gestiftet. Er umfasst 8 Bits. Damit kann eine Variable dieses Typs Werte zwischen -128 und +127 speichern. Diesen Typ können Sie einsetzen, um Datenstrukturen fremder Programme nachzuahmen oder wenn Sie Daten von technischen Geräten auslesen.
- `char`: Der Typ `char` kann ein Zeichen, also einen Buchstaben, eine Ziffer oder ein Satzzeichen aufnehmen. Auch mit diesem Typ werden wir uns schon sehr bald näher auseinandersetzen.

Tabelle 3.1 zeigt eine Übersicht darüber, wie viel Platz die verschiedenen Primitive benötigen und was sie aufnehmen können.

Typ	Speicher	Beschreibung
boolean	8 Bit	Wahrheitswert: true oder false
byte	8 Bit	Ein Byte: -128 bis 127
char	16 Bit	Buchstaben
short	16 Bit	-32.768 bis 32.767
int	32 Bit	-2.147.483.648 bis 2.147.483.647
long	64 Bit	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
float	32 Bit	Zahl mit Nachkommastellen
double	64 Bit	Mehr Platz für Nachkommawerte

Tabelle 3.1: Primitive Typen und ihre Dimensionen

Ein Bit ist die kleinste Informationseinheit: Sie kann eine 0 oder eine 1 enthalten. Technisch gesehen ist ein Bit durch eine Leitung zu realisieren, auf der entweder Spannung anliegt oder eben nicht.



Sie werden sich fragen, wieso Java dann 8 Bits für einen Wahrheitswert benötigt. Da dieser nur wahr oder falsch sein kann, müsste er doch mit einem Bit abzuhandeln sein. Es mag sein, dass es auf den ersten Blick Speicherplatz spart, Wahrheitswerte in einem Bit abzulegen. Die kleinste direkt adressierbare Speichereinheit ist aber ein Byte mit eben 8 Bits. Der Aufwand, dort das eine Bit herauszufummeln, wäre größer, als einfach 8 Bits zu nehmen.

Konstanten

Manche Dinge ändern sich nie. Beispielsweise wird ein Skatblatt immer 32 Karten enthalten. Da solche Zahlen nicht variabel sind, könnte sich der Programmierer

verleiten lassen, das Literal 32 im Programm direkt zu verwenden. Aber Literale sind immer recht abstrakt und unflexibel. Wenn Sie beispielsweise im Programm die Zahl 6 verwenden, könnte das die Anzahl der Flaschen in einem Sixpack sein. Oder die Anzahl der Spieler einer Volleyballmannschaft. Oder die Anzahl der Kugeln, die beim Lotto gezogen werden. Oder die Anzahl der Beine einer Spinne.

Damit die eine 6 von der anderen 6 unterscheidbar ist, sollten Sie sie in einem Java-Programm mit einem Namen belegen. In Java wird dazu eine Variablendefinition angelegt, die gleich mit dem Literal initialisiert wird. Mit dem Modifizierer `final` wird angezeigt, dass der Wert später nicht mehr geändert werden kann. Solch eine besondere Variable, die nicht variabel ist, bezeichnet man als Konstante.

```
final int SIXPACKFLASCHEN = 6;
final int VOLLEYBALLSPIELER = 6;
final int LOTTOKUGELN = 6;
final int SPINNENBEINE = 6;
```

Und sollte ich mich bei den Spinnenbeinen verzählt haben und irgendein Biologe besteht darauf, dass es vielleicht doch 8 sind, so kann ich das an einer Stelle im Programm ändern und muss nicht das Literal 6 im Programm suchen und prüfen, ob das wirklich Spinnenbeine, Volleyballspieler, Lottozahlen oder Flaschen sind.

```
final int SPINNENBEINE = 8;
```

Die Namen von Konstanten bestanden schon in der Sprache C aus Großbuchstaben. Und tatsächlich pflegt man es auch in Java so zu tun. Prinzipiell können die gleichen Bezeichner verwendet werden wie bei Variablen, aber die Schreibweise in Großbuchstaben ist üblich.



Als Faustregel sollten Sie außer bei 0 und 1 Konstanten statt Zahlenwerten verwenden. Der Name der Konstanten dokumentiert nämlich, was die Zahl bedeutet.

Kleines Rechenwerk

Es wird Sie nicht überraschen: Computer können rechnen und Java kann es auch. Die Rechenanweisungen ähneln der Schreibweise in der Mathematik. Addiert wird mit dem Pluszeichen und mit dem Minuszeichen wird subtrahiert. Zum Multiplizieren nehmen Sie nicht nur in der Adventszeit einen Stern. Ein mittelhoher Punkt ist nämlich auf der Tastatur nicht verfügbar. Für die Division steht der Schrägstrich zur Verfügung.

Als weiteren Operator kennt Java noch das Prozentzeichen. Es wird aber nicht etwa für die Prozentrechnung verwendet, sondern bedeutet *Modulo*. Damit bezeichnet man den Rest bei einer Division. 25 geteilt durch 7 ergibt 3 Rest 4. Also ist 25%7

eben 4.

Das Plus- und das Minuszeichen können auch als Vorzeichen verwendet werden. Der Wert -6 ist also genauso negativ wie im wirklichen Leben. Damit ergibt sich die folgende Tabelle der Rechensymbole:

Zeichen	Bedeutung
+	positives Vorzeichen
-	negatives Vorzeichen
+	Summe
-	Differenz
*	Produkt
/	Quotient
%	Modulo (Rest)

Tabelle 3.2: Operatoren in Java

Plus und Minus gibt es zweimal: einmal als Vorzeichen und einmal als Operator, genauso wie Sie und ich das auch machen, wenn wir auf einem Zettel rechnen.

Ausdruck höchster Rechenkünste

Das Ergebnis rechnerischer Bemühungen wird, wie Sie bereits gesehen haben, gern in Variablen geparkt. Dazu stellt man die Rechenanweisung auf die rechte Seite eines Gleichheitszeichens.

```
ergebnis = 12 * 3 + 4;
brutto   = netto + tara;
mwstAnteil = preis * 0.19;
```

Da wir die linke Seite einer Zuweisung bereits als LValue kennen, wäre es konsequent, eine solche Rechenanweisung RValue zu nennen. Stattdessen bevorzugen die meisten Programmierer die Bezeichnung *Ausdruck*. Im Gegensatz zum LValue, der meist tatsächlich links neben dem Gleichheitszeichen steht, findet man einen Ausdruck durchaus auch an anderer Stelle, beispielsweise als Bestandteil eines anderen Ausdrucks oder einfach zwischen den Klammern von `System.out.println`.


Das Schema von Variablenzuweisungen lautet: `LValue = Ausdruck;`

Java zählt

Bei einer Zuweisung wird zuerst der Ausdruck fertig berechnet und dann im LValue sicher verstaut. Das ist beim Zählen außerordentlich praktisch, weil Sie die gleiche Variable auf der linken wie auf der rechten Seite des Gleichheitszeichens verwenden können.

```
zaehler = zaehler + 1;
```

Das sieht ulkig aus, nicht wahr? Zeigen Sie das bloß keinem Mathematiker. Der bekommt sofort einen Allergieschub. In Java funktioniert das aber völlig problemlos, weil der Compiler den Teil rechts vom Gleichheitszeichen zuerst auswertet und dann erst der Variablen auf der linken Seite zuweist.

Da das Erhöhen oder Erniedrigen einer Variablen um einen bestimmten Betrag immer wieder vorkommt, gibt es eine Abkürzung. Dabei wird der Operator vor das Gleichheitszeichen gestellt.

```
zaehler = zaehler + 1; // ist das Gleiche wie ...
zaehler += 1;          // ... dies!
```

Das funktioniert auch mit dem Minus, dem Stern oder dem Schrägstrich und auch mit anderen Beträgen als 1. Sie verdoppeln also einen Wert mit der folgenden Zeile:

```
wert *= 2;
```

Inkrementieren und dekrementieren

Das Hochzählen einer Variablen wird in der Praxis sehr oft benötigt. Darum gibt es sogar noch eine Verkürzung der verkürzten Schreibweise: Vor oder hinter die Variable wird ein doppeltes Pluszeichen geschrieben. Tatsächlich gibt es einen Unterschied, wo das doppelte Plus steht. Wird das ++ vor eine Variable geschrieben, wird die Variable vor dem Erhöhen ausgewertet. Das folgende Beispiel zeigt die Konsequenz:

```
int zaehler = index = 5;
ergebnis = zaehler++; // ergebnis ist 5
ergebnis = ++index;   // ergebnis ist 6
```

Sie können auch eine Variable um 1 vermindern. Das nennt man dann *dekrementieren*. Dazu wird dann analog ein doppeltes Minuszeichen verwendet.

Punkt vor Strich

Wie in der Mathematik üblich, gibt es eine Rangfolge bei der Verwendung der Operatoren. Sie kennen wahrscheinlich noch den Spruch: »Punkt- geht vor Strichrechnung«, was bedeutet, dass zuerst multipliziert oder dividiert wird und dann erst addiert oder subtrahiert wird. Wenn Sie eine andere Reihenfolge wollen, müssen Sie eine Klammer setzen. Und das Schöne ist: Sie können diese Regel für Java einfach weiterverwenden.

Ein einfaches Beispiel: Wir kaufen einen Kasten Cola. Die Cola kostet 50 Cent. Das Flaschenpfand beträgt 20 Cent und für den Kasten muss ein Pfand von 2 Euro bezahlt werden. Wenn in dem Kasten 12 Flaschen sind, betragen die Kosten:

```
12 * (0.50 + 0.20) + 2
```

Die Klammern müssen so stehen, dass vor dem Multiplizieren die Kosten je Flasche summiert sind. Und wenn wir schon einmal dabei sind, verwenden wir auch gleich Konstanten. Dann wird die Formel gleich viel lesbarer.

```
class Flaschenpfand {
    public static void main(String[] args) {
        final int flaschenAnzahl = 12;
        final double colaPreis = 0.50;
        final double flaschenPfand = 0.20;
        final double kastenPfand = 2.0;
        double preis = 0.0;
        preis = flaschenAnzahl * (colaPreis + flaschenPfand)
                               + kastenPfand;
        System.out.println(preis);
    }
}
```

Listing 3.8: Flaschenpfandberechnung

Ganzzahlig kontra Fließkomma

Java verwendet ganzzahlige Operationen, so lange kein Fließkommawert in die Berechnung einfließt. Beispielsweise errechnet die folgende Zeile 0.0, obwohl Sie vielleicht eine 4 erwarten würden.

```
double ergebnis = 12/24*8;
```

Der Grund liegt darin, dass Java zuerst 12 durch 24 teilt. Aus Sicht von Java ist sowohl 12 als auch 24 eine ganze Zahl. Ganzzahlig gerechnet, ergibt 12 durch 24 aber 0 mit Rest 12. Diese 0 wird mit 8 multipliziert und bleibt darum eine 0. Wird dagegen die 12 oder die 24 mit einem F oder einem D als Fließkommazahl markiert, wird die Division nicht ganzzahlig durchgeführt. Das ergibt dann 0.5, was wiederum mit 8 multipliziert die erhoffte 4 ergibt.

```
double ergebnis = 12f/24*8;
```

Alternativ kann auch eine Nachkommastelle angegeben werden, selbst wenn diese 0 ist. Auch das zwingt zur Fließkommaberechnung.

```
double ergebnis = 12.0/24*8;
```

Der Fluch der zehn Finger

Wenn Sie das Programm aus Listing 3.8 abgetippt und gestartet haben, werden Sie festgestellt haben, dass als Ergebnis 10,3999999 herauskommt! Das ist gemein! Im Grunde kennen Sie dieses Verhalten wahrscheinlich von einem normalen Taschenrechner. Wenn Sie 1 durch 3 teilen, kommt 0,3333 heraus. Multiplizieren Sie nun das Ergebnis wieder mit 3, kommt nicht etwa die 1, sondern 0,99999 heraus.

Vermutlich ist Ihnen das sogar einsichtig, weil ein Drittel eben nicht in das dezimale Zahlensystem passt. So entsteht eine Periode, die ein Taschenrechner nun mal nicht verarbeiten kann. Aber in dem Beispiel teilen wir nirgends durch 3. Wir teilen überhaupt nicht!

Doch, wir teilen in diesem Beispiel sehr wohl, zwar nicht durch 3, aber durch 100. Denn 0,50 und 0,20 sind 50 geteilt durch 100 und 20 geteilt durch 100. Leider ist aber ein Zehntel für ein binäres Zahlensystem, wie es der Computer verwendet, genauso eine Periode wie ein Drittel für das dezimale Zahlensystem.

Das bedeutet, dass es beim Rechnen mit dezimalen Nachkommabeträgen immer wieder zu solchen Ungenauigkeiten führen kann. Dieses Problem hat nicht nur Java, sondern das haben alle Programmiersprachen, die mit binär codierten Fließkommazahlen rechnen.

 Das Aufsummieren von Zehnteln führt immer wieder zu Ungenauigkeiten.

Wenn Sie also mit Zehnteln rechnen, sollten Sie auch damit rechnen, dass der Computer eine Periode daraus macht. Was tun Sie dann? Sie müssen auf- oder abrunden. Durch geschicktes Casting können Sie ausnutzen, dass eine ganze Zahl alle Nachkommastellen abhackt. Sie wollen aber nicht alle Nachkommastellen erschießen, sondern nur die Bruchteile der Cents. Dazu multiplizieren Sie die Zahl vor dem Casting mit 100. Das klappt beinahe gut. Aus 10,3999999 wird nun 1039. Allerdings wollten Sie eigentlich erreichen, dass 1040 daraus wird. Dazu addieren Sie vor dem Multiplizieren noch 0,005. Damit kippt der Wert nach oben, falls der Bruchteil größer als ein halber Cent ist. Anschließend teilen Sie den Wert einfach wieder durch 100.0.

```
double rundpreis = (long)(preis*100+0.005)/100.0;
```

Zum Runden gibt es in der Math-Bibliothek den Aufruf `round`. Die Bibliotheken werden in Kapitel 14 beschrieben. Damit sieht die Rundung so aus:

```
double rundpreis = Math.round(preis*100)/100.0;
```

Zeichen

Neben den Zahlen müssen natürlich manchmal auch Buchstaben in Variablen gespeichert werden. Dafür gibt es den Datentyp `char` (von englisch *character*, also »Buchstabe«), den ich eben schon kurz erwähnt hatte. Damit der Java-Compiler die Buchstaben, die Sie als Literale ins Programm setzen, nicht für Variablennamen hält, werden `char`-Literale in Hochkommata gesetzt (').

Zeichen sind auch nur Zahlen

```
char zeichen = 'A';
char nochEinZeichen = '7';
```

Listing 3.9: Zeichenzuweisungen



Sie finden auf der Tastatur einige Zeichen, die wie ein Hochkomma aussehen, aber eigentlich Akzente sind und prompt vom Compiler nicht verstanden werden. Wenn Sie genau hinschauen, sind diese Zeichen leicht nach links oder rechts abgeknickt. Das richtige Hochkomma befindet sich auf einer deutschen Tastatur über dem Lattenzaun (#).

Falls Sie bei der 7 in Listing 3.9 die Hochkommata vergessen haben sollten, werden Sie feststellen, dass der Compiler gar nicht gemeckert hat. Das hat damit zu tun, dass Zeichen eigentlich als Zahlen codiert werden. Verwenden Sie doch einmal statt der 7 eine 65 und geben Sie das Zeichen auf dem Bildschirm aus.

```
public class CharTest {
    public static void main(String[] args) {
        char zeichen = 65;
        System.out.println(zeichen);
    }
}
```

Listing 3.10: Zeichen aus Zahlen

Da erscheint ein großes A, weil das Zeichen ‘A’ als 65 codiert ist. Übrigens ist die Codenummer des Zeichens ‘7’ die Zahl 55. Probieren Sie es einmal aus!

Die 55 ergibt eine 7 auf dem Bildschirm, während eine 7 ein merkwürdiges Zeichen ergibt. Wer denkt sich so etwas aus? Ich war es bestimmt nicht. Aber diese Codierung geht auf den American Standard Code for Information Interchange zurück, kurz als ASCII bezeichnet. Er beschreibt, wie Zeichen codiert werden. Die ersten 32 Zeichen sind Steuerzeichen wie Wagenrücklauf, Tabulator oder Beep. Die nächsten 16 sind Satzzeichen, dann kommen die Zahlen, die Großbuchstaben und die Kleinbuchstaben.

Warum, so fragt sich der geneigte Leser, gibt `System.out.println` diese Zahlen nicht einfach aus? Das hängt damit zusammen, dass `System.out.println` eine `char`-Variable in den Klammern vorfand. Daraus schließt die Ausgabe, dass der Programmierer nicht etwa eine Zahl, sondern ein Zeichen sehen wollte, und wird die Umsetzung nach ASCII durchführen.

Der Datentyp `char` ist zwar eigentlich für Zeichen gedacht, kann aber eben auch die Codierungen der Zeichen aufnehmen und arbeitet dann wie ein kleiner `int`-Typ.

Internationale Abwege

Der ASCII-Code hat sein A von American und so kann man sich leicht vorstellen,

dass man sich über die europäischen Sonderzeichen keine großen Gedanken gemacht hat. Ein `char` kann 256 Zeichen aufnehmen. Für die englischen Zeichen, Ziffern, Satz- und Sonderzeichen reichten die ersten 128 Plätze. Der Rest war den Amerikanern egal.

Das sahen die Europäer etwas anders und so stritten sie sich um die restlichen 128 Plätze. Sie stellten fest, dass der Platz für die westeuropäischen Zeichen reichte. Spätestens bei der Verwendung des griechischen und des russischen Alphabets quoll alles über. So schufen sie für jede Region einen eigenen ISO-Standard.

Wenn jede Region ihren eigenen Standard hat, ist das auch nicht schön. Also murkste man weiter, indem man statt den bisherigen 8 Bits eines Bytes nun 16 Bits in zwei Bytes verwendete. Das war natürlich eine üble Platzverschwendung, wenn man keine internationalen Sonderzeichen benötigte. Immerhin durften die Amerikaner die ersten 127 Zeichen ihres ASCII behalten.

Erst dann bemerkte man, dass die Europäer den gleichen Fehler gemacht hatten wie die Amerikaner. Sie glaubten, es gäbe sonst niemanden auf der Welt. Beim Blick auf die asiatischen Zeichen wurde klar, dass das auch mit 16 Bits nicht mehr zu schaffen ist.

Dann kam die Idee, dass man erst einmal alle denkbaren Zeichen sammelt und durchnummeriert. Diese einheitliche Nummerierung ist der Unicode eines Buchstabens. Für die in der Informatik meistbenötigten Zeichen verwendet man nach wie vor die ersten 128 Positionen des ASCII. Da ein Byte 256 Werte darstellen kann, ist das höchste Bit des Bytes frei. Damit kann man signalisieren, dass das nächste Byte für die Codierung herangezogen werden muss. Dieser Kniff kann im nächsten Byte wiederholt werden, sodass beliebig lange Codierungen erzeugt werden können. Dieses Verfahren wird UTF genannt. UTF-8 verwendet 8 Bits, also 1 Byte für jedes Zeichen. Wenn ein deutscher Umlaut auftaucht, sind es dann vielleicht 2 Bytes. Und asiatische Sonderzeichen könnten dann mehrere Bytes in Anspruch nehmen, je nachdem, welche Unicode-Nummer das Zeichen hat.

Als Java eingeführt wurde, verwendete man dort UTF-16 und konnte damit die meisten europäischen und amerikanischen Zeichen in 2 Bytes abbilden. Erst bei asiatischen Zeichen werden es mehr.

Jeder Wechsel der Codierung führte dazu, dass ein deutsches Ä eine neue Nummer bekam, da die internationalen Zeichen immer wieder anders belegt wurden. Manchmal sehen Sie die Ergebnisse noch in Mails oder Texten, deren Umlaute plötzlich kryptisch aussehen. Wollen wir also hoffen, dass man bei UTF erst einmal ein paar Jahrzehnte bleibt.

Auch für die internationalen Unicode-Zeichen können Sie Literale schreiben, selbst wenn Sie keine chinesische Tastatur zur Hand haben. Voraussetzung ist nur, dass Sie den Unicode des Zeichens kennen. Dann stellen Sie vor diese Nummer die

Kombination \u. Darauf folgt eine vierstellige hexadezimale Codierung. Falls Sie nicht wissen, was hexadezimal ist, müssen Sie nicht in Trauer ausbrechen. Sie müssten sowieso in einer Tabelle nachschlagen und abschreiben. Ich überlegte noch, ob ich Ihnen die Tabelle aller weltweit verfügbaren Sonderzeichen in einem zweiten Band zu diesem Buch aufschreiben soll. Aber irgendwie fand der Verlag die Idee gar nicht toll. So habe ich es dabei belassen, Ihnen ein Beispiel zu geben. Hier enthält die Variable `zeichenA` den Buchstaben `'A'`.

```
char zeichenA = '\u0041';
```

Steuerzeichen

ASCII codiert nicht nur sichtbare Zeichen, sondern auch Steuerzeichen, wie etwa das Zeichen für eine neue Zeile oder das Backspace-Zeichen. Beispielsweise können Sie mit *Carriage Return* einen Wagenrücklauf auslösen. Dieser wird als Zeichen mit `'\r'` dargestellt. Codiert ist er als dezimal 13. Geben Sie dieses Zeichen auf einen Drucker, wird er die Zeile noch einmal von vorn beginnen. Erst wenn Sie ein *Line Feed*, also einen Zeilenvorschub mit `'\n'` oder dem ASCII-Code 10 auslösen, geht der Drucker eine Zeile weiter. Lassen Sie Letzteres bei einem alten Nadeldrucker weg, so können Sie mit einer Zeile von Minuszeichen, die Sie immer wieder mit *Carriage Return* abschließen, eine wirkungsvolle Perforation in das Endlospapier hineindrücken. Wenn Sie anschließend ein paar Mal *Form Feed*, also Seitenvorschübe per `'\f'` oder dem ASCII-Code 12 hinterherschicken, dürfte das Papier reißen, aus dem Transporttraktor fallen und damit einen tollen Papierstau erzeugen. Sollten Sie nur Laserdrucker haben, müssen Sie Ihren Administrator auf eine andere Art und Weise ärgern.

Sie sehen also, wie ungeheuer nützlich diese Steuerzeichen sind. Darum gibt es für diese auch feste Literale, so dass Sie nicht die ASCII-Tabelle bemühen müssen.

Zeichen	Bedeutung
\b	Backspace
\t	Tabulatorzeichen
\n	Zeilenvorschub
\r	Wagenrücklauf
\f	Seitenvorschub
\\	Backslash
\'	Hochkomma
\”	Anführungszeichen

Tabelle 3.3: Backslash-Sequenzen für Steuerzeichen

Zeichenkettenliterale

Wenn mehrere Buchstaben aneinanderhängen, spricht man von einer Zeichenkette oder von einem String. Da Strings für Bildschirmausgaben und andere Zwecke besonders wichtig sind, haben sie ein eigenes Literal. Zeichenketten werden in Anführungszeichen gesetzt, also auf der Tastatur das Zeichen über der 2.


Nicht verwechseln: Ein einzelnes Zeichen wird in Hochkommata eingeschlossen, ein String in Anführungzeichen.

```
System.out.println("Hallo, Ihr da!");
```

Es gibt auch einen passenden Typ String. Dieser gehört aber nicht zu den primitiven Typen und darum werden wir ihn uns erst später in Kapitel 6 noch einmal ausgiebiger anschauen.

Mit Zeichen spielen und rechnen

Mit dem ersten Programm wollen wir ein wenig die Codierung der Zeichen erkunden. Dazu weisen wir einer `char`-Variablen zuerst den Buchstaben `'A'` zu. Anschließend erhöhen wir die Variable um 1. Was kommt dabei für ein Buchstabe heraus? Sie ahnen es schon. Aber glauben Sie mir nichts, sondern probieren Sie es aus. Wenn Sie vom `'A'` die Zahl 16 abziehen, erhalten Sie eine `'1'`. Und wenn Sie zu einem `'A'` 32 hinzuaddieren, erhalten Sie ein kleines `'a'`. Das klingt zugegeben etwas abgedreht, bietet Ihnen aber einen schönen Rahmen, um ein bisschen mit den Zeichen zu spielen.

```
public class ZeichenZeigen {
    public static void main(String[] args) {
        char zeichen = 'A';
        zeichen++;
        System.out.println(zeichen);
        zeichen -= 16;
        System.out.println(zeichen);
        zeichen += 16+32;
        System.out.println(zeichen);
    }
}
```

Listing 3.11: Zeichencodierung ausprobieren

Probieren Sie weiter! Folgen alle Zeichen aufeinander? Wo ist die `'0'`? Jetzt wäre es natürlich schön, wenn man einfach eine Wiederholung in das Programm einbauen könnte, damit man das ganze Alphabet darstellen kann. Geduld! Das kommt früher, als Sie denken.

Ordnende Elemente

In diesem Abschnitt geht es darum, wie Programme strukturiert werden.

Blöcke in geschweiften Klammern

Ineinander verschachtelte geschweifte Klammern werden Ihnen bei Java immer wieder begegnen. Eine geschweifte Klammer umfasst immer einen *Block*. Ein Block kann eine beliebige Menge von Anweisungen aufnehmen, die er zu einer Anweisung zusammenfasst.

Beispielsweise wirkt der Befehl `if` nur auf die nächste Anweisung. Sollen es gern mehrere sein, kann man sie einfach mit geschweiften Klammern zusammenfassen. Daneben gibt es aber Bereiche, die immer zwingend geschweifte Klammern erfordern. Sie haben das schon bei Methoden und Klassen gesehen.



Verwechseln Sie nicht die geschweiften mit den runden Klammern. Beide haben sehr unterschiedliche Aufgaben. Und nur keine Sorge: Die runden Klammern werden zu einem passenden Zeitpunkt ausführlich erläutert.



Geschweifte Klammern fassen einen Block von Anweisungen zu einer Anweisung zusammen.

Einrückung

An den vorgestellten Programmbeispielen können Sie sehen, dass nach dem Öffnen der Klammer die nächste Zeile ein paar Leerzeichen nach rechts rückt und mit dem Schließen einer Klammer wieder zurückkehrt. Dieses Ein- und Ausrücken schreibt der Compiler zwar nicht vor, sorgt aber für eine bessere Übersicht. Darum tun das die meisten Java-Programmierer. Am besten tun Sie es auch. Falls Sie Eclipse verwenden, werden Sie feststellen, dass Eclipse dies für Sie erledigt.



Wenn Sie irgendwann einmal in Eclipse die Einrückung durcheinandergebracht haben, können Sie den Bereich mit der Maus markieren und dann mit der Tastenkombination Strg + I (großes i) die Einrückung nachrichten.



Für die Gestaltung von Java-Programmen gibt es Konventionen, die im *Java Style Guide* vorgeschlagen werden. Sie finden diesen unter dem Titel *Java Code Conventions* auf der Website von Oracle zum Download als PDF oder online zum Stöbern.

`http://www.oracle.com/technetwork/java/codeconvtoc-136057.html`

Leider liegen die Originale in englischer Sprache vor. Sie finden aber über eine Suchmaschine auch genügend deutsche Übersetzungen. Sie können daran später kontrollieren, ob ich in diesem Buch alles richtig gemacht habe.

Verbindlich werden solche Vorschriften allerdings erst dadurch, dass Ihr Chef sagt, wie die Java-Programme aussehen sollen. Und da er bezahlt, ist er wichtiger als der Java-Style-Guide.

Semikolon

Java ist es eigentlich recht egal, wie Sie den Quelltext über die Datei verteilen. Das Ende einer Anweisung erkennt Java am Semikolon. Fehlt es, kann der Compiler recht maulig werden, weil er nicht weiß, wann der alte Befehl aufhört und wo der neue Befehl anfängt. Und natürlich schiebt er Ihnen die Schuld in die Schuhe.

Die geschweiften Klammern sind bereits ein Abschluss. Darum benötigen sie kein Semikolon, wenn es hinter einer Anweisung bereits geschweifte Klammern gibt.

Kommentare sind nett zu lesen

Java ermöglicht es dem Programmierer, Texte in den Quellcode zu schreiben, die vom Compiler übersprungen werden. Diese Texte nennt man Kommentare und sie richten sich an Programmierer, die später einmal die Freude haben, das Machwerk zu bewundern. Diese Texte sollten erzählen, was der Autor sich dabei gedacht hat, als er etwas auf diese Weise programmiert hat. Ein Programm wird nämlich öfter gelesen als geschrieben. Und bedenken Sie, dass derjenige, der das Programm fluchend überarbeiten muss, Sie selbst sein könnten.

Ein Kommentar beginnt mit der Kombination aus /* und wird so lange vom Übersetzungsvorgang ignoriert, bis die Kombination */ erscheint. Ein solcher Kommentar kann über mehrere Zeilen gehen.

```
/* Diesen Kommentar kann man über mehrere Zeilen ziehen.
   Er hört auf, wenn ein Stern und ein Schrägstrich
   aufeinanderfolgen, etwa so: */
```

Etwas tückisch ist, dass Kommentare im Gegensatz zu Blöcken nicht verschachtelt werden können. Sollten Sie also einen Bereich auskommentieren wollen, der bereits einen solchen Kommentar enthält, wird das erste */ den Compiler dazu motivieren, wieder zu arbeiten. Spätestens das zweite */ wird dann als Fehler interpretiert.

```
/* Dieser Kommentar enthält einen anderen.
   Der nächste Kommentaranfang wird ignoriert.
   Er ist ja auskommentiert.
   /* Dieser Komentar war schon früher da! */
   Der Compiler hat ein Kommentarende gefunden, also denkt
   er hier bereits, dass es sich um Java handelt.
   Dumm gelaufen! */
```

Es gibt noch ein weiteres Kommentarzeichen. Es besteht aus zwei Schrägstrichen. Diese Variante sorgt dafür, dass der Rest der Zeile vom Übersetzer nicht mehr beachtet wird.

```
class Datum { // Restkommentar
```



Kommentieren Sie nicht, was Sie machen, sondern, warum Sie es so machen.

Das »Was« kann ein erfahrener Java-Programmierer schon herausfinden. Aber warum Sie diesen Weg beschritten haben, wissen nur Sie und diejenigen, denen Sie es erzählen.

Kommentare für Javadoc

Eine besondere Form der Kommentierung fügt der Schrägstrich-Stern-Kombination einen weiteren Stern hinzu, also /**. Natürlich wird der Compiler auch dies ignorieren. Aber der Inhalt dieser Kommentare kann von Javadoc verwendet werden, um eine Dokumentation über Ihr Programm zu erstellen.

Das folgende Beispiel greift ein wenig vor, weil Sie ja noch gar nicht genau wissen, was eine Klasse oder eine Methode ist. Auch Parameter und Rückgabewerte sind Ihnen noch fremd. Das wird sich aber schon bald ändern. Im Moment geht es erst einmal nur darum, zu verstehen, wie Javadoc-Kommentare funktionieren. Wenn Sie dann später Programme dokumentieren wollen, kommen Sie hierher zurück.

```java
/**
 * Dieser Kommentar gilt für die Klasse JavaDocMich
 * <p>
 * Sie sehen, dass HTML-Tags eingebettet werden können.
 * Mit dem Zeichen @ können Sie verschiedene Informationen
 * definieren.
 *
 * @author Arnold Willemer
 * @version 1.0
 */
public class JavaDocMich {

    /**
     * Dieser Kommentar gilt für die nächstkommende Methode
     * @param erster  Dieser Wert kommt in die Methode rein.
     * @param zweiter Da gibt es noch einen Wert.
     * @return gibt die Summe aus all dem zurück.
     */
    public int tuwas(int erster, int zweiter) {
        return erster+zweiter;
    }

    /**
     * Dieser Kommentar gilt für die nächstkommende Methode.
     * Das ist main und die hat keinen return.
     * @param args  Das kommt in die Methode rein.
     */
    public static void main(String[] args) {
        System.out.println("Kuckuck!");
    }
}
```

Listing 3.12: Kommentare für Javadoc

Sie sehen, dass bestimmte Einträge mit dem @-Zeichen beginnen. Sie bezeichnen Einträge für das Dokumentationssystem, wie beispielsweise @author. Dahinter können Sie als Parameter Ihren Namen schreiben.

Mit dem Befehl `javadoc` können Sie aus dem Quelltext eine Dokumentation als Website generieren.

```
javadoc JavaDocMich.java
```

Um sich die Dokumentation anzusehen, rufen Sie Ihren Browser mit der Datei *index.html* auf.

```
firefox index.html
```

Nun können Sie sich durch die Dokumentation Ihres Programms klicken.



Aber selbst dann, wenn Sie nicht die Absicht haben, eine beeindruckende Dokumentation Ihres Programms zu generieren, sollten Sie Javadoc-Kommentare verwenden. Eclipse wird die passenden Javadoc-Kommentare in einem Tooltipp einblenden, wenn Sie die Maus über einen Aufruf oder eine sonstige Verwendung der kommentierten Elemente setzen.

Aufgaben

Welche Bezeichner sind korrekt?

```
h12aus
hAUS
12haus
Haus
HAUS
hæus
$haus
1$job
```

Die Lösung ist in der Fußnote versteckt.[1] Bevor Sie nachschauen, versuchen Sie doch einfach, die Variablen in einem Programm zu verwenden.

Welche Kommentierung ist korrekt?

Im Beispiel sollen alle Befehle außerhalb der Kommentare für den Compiler sichtbar sein und alle Kommentare für den Compiler unsichtbar sein.

```
befehl1; /* kommentar */ befehl2;
```

In der Fußnote finden Sie die Lösung.[2]

```
befehl1; // kommentar // befehl2;
```

In der Fußnote finden Sie die Lösung.[3]

```
/*
// kommentar1
// kommentar2
befehl1;
*/
befehl2;
```

In der Fußnote finden Sie die Lösung.[4]

Zahlensalat

Korrigieren Sie die folgende Zeile! Warum ist sie falsch?[5]

```
3.14 = pi;
```

Versuchen Sie herauszufinden, ob die folgenden Zahlenliterale korrekt sind und wenn ja, welchem Wert sie entsprechen. Sie bekommen ein paar Vorschläge.

	Korrekt?	Wert?
12,1	j/n	12,1 / 12 / 0 / 1 / 0,1
12,1L	j/n	12,1 / 12 / 0 / 1 / 0,1
12.1F	j/n	12,1 / 12 / 0 / 1 / 0,1
12.1f	j/n	12,1 / 12 / 0 / 1 / 0,1
12,1D	j/n	12,1 / 12 / 0 / 1 / 0,1

Welche der folgenden Fließkommazahlen sind korrekt?

```
.12e-12
1.2e-1.2
-1.2e-12
3.14159
314.159
314159.
314159.0
```

Um wie viel unterscheiden sich die folgenden Zahlen?[6]

```
3.14e2  3140e-1
```

Bringen Sie die folgenden beiden Definitionen in die richtige Reihenfolge. Wenn Sie sicher sind, wie die Zeilen lauten sollen, bauen Sie sie einfach in das Grüßprogramm ein. Dann können Sie Ihren Compiler entscheiden lassen, ob das Ergebnis richtig ist.

```
KONSTANTE int 5 =  static ; final
0.12 eineVariable double ; =
```

Typenberechnungen

Zum Abschluss wollen wir noch einmal Zeichen mit ganzen Zahlen und Fließkommazahlen verrechnen. Als Ausgangssituation verwenden wir das folgende, kleine Programm:

```
class TypBerechnung {
    public static void main(String[] args) {
        char zeichen = '0';
        int zahl = 12;
        float fliess = 12;
        double ergebnis = zeichen + zahl + fliess;
        System.out.println(ergebnis);
    }
}
```

Listing 3.13: Mit Zeichen rechnen

Ersetzen Sie nun die Zeile

```
double ergebnis = zeichen + zahl + fliess;
```

nacheinander durch folgende Zeilen. Überlegen Sie zuerst, was dabei herauskommt.

Dann prüfen Sie es im Compiler nach.

```
double ergebnis = zeichen = zahl + fliess;
double ergebnis = zahl / zeichen;
double ergebnis = fliess / zeichen;
```

Anmerkungen

[1]Nur die Bezeichner, die mit einer Ziffer beginnen, sind falsch.

[2]Alles richtig.

[3]`befehl2` ist auskommentiert.

[4]`befehl1` steckt zwischen den Kommentarklammern.

[5]Bei einer Zuweisung steht die Variable links.

[6]Gar nicht.

Kapitel 4

Entscheiden und Wiederholen

IN DIESEM KAPITEL

Unter Umständen werden if-Befehle ausgeführt.

Wir geben uns Logeleien hin.

Schleifen rotieren kopf- und fußgesteuert.

Sie können jetzt schon Schritt für Schritt nacheinander verschiedene Operationen zu einem Programm zusammenstellen. Das ist prima. In diesem Kapitel soll es darum gehen, dass Ihr Programm auf bestimmte Situationen reagiert und dass Sie wiederkehrende Aufgaben in einer Schleife rotieren lassen.

Nur unter einer Bedingung

Im Leben eines jeden Programms ist einmal der Moment gekommen, endlich Entscheidungen zu treffen. Er oder ich! Sein oder Nichtsein? Pest oder Cholera.

Java kennt Abfragen, die es einem Programm ermöglichen, auf Bedingungen zu reagieren. Aus der Mathematik kennen Sie vielleicht solche Bedingungen. So darf man nicht durch 0 dividieren oder aus negativen Zahlen die Wurzel ermitteln. Das Computerspiel wird ein Explosionsgeräusch erzeugen, wenn der Abstand zur Rakete 0 ist. Das Fahrzeug der Zukunft wird vielleicht die Handbremse nicht entriegeln, wenn der Alkoholwert im Blut des Fahrers einen gewissen Pegel überschreitet.

if-Anweisung

Die Sprache Java verwendet für Abfragen den Befehl `if`. Er funktioniert ähnlich wie die Weiche bei der Eisenbahn. Durch eine Bedingung wird der Hebel gesteuert, der die Weiche umlegt, sodass der Verlauf des Programms einen anderen Weg nimmt.

In Abbildung 4.1 kommt der Zug von links. Nur wenn der Hebel umgestellt ist, kommt der Zug an den Stationen vorbei und arbeitet diese ab. Der Hebel wird durch die Bedingung gesteuert. In Java würde eine solche Weiche im Quellcode etwa so aussehen:

```
if (hebel==rechts)
    bahnhof.anfahren();
```

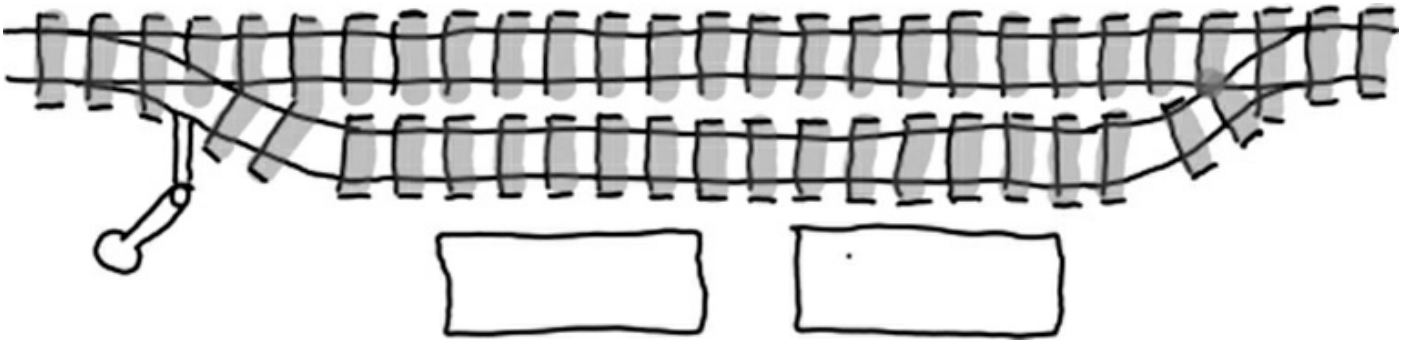


Abbildung 4.1: Eine Weiche bringt eine Verzweigung des Ablaufs.

Das sieht vielleicht auf den ersten Blick zu abstrakt aus. Aber wenn wir bei dem einfachen Beispiel der Mathematik bleiben, dass man besser nicht durch 0 teilen sollte, ist das folgende Beispiel schon einfacher zu verstehen.

```
if (nenner!=0)
    quotient = zaehler / nenner;
```

Listing 4.1: `if` schützt vor einer Division durch 0.

- Die Bedingung wird durch den Befehl `if` eingeleitet.
- In der Klammer hinter dem Befehl `if` steht die Bedingung.
  - Die Zeichenkombination == bedeutet *gleich*.
  - Die Kombination `!=` bedeutet *ungleich*.
  - Das Ausrufezeichen allein bedeutet *nicht*.
- Die dem `if`-Befehl folgende Anweisung wird nur ausgeführt, wenn die Bedingung in der Klammer zutrifft.



Um dieses Verhalten zu verdeutlichen, pflegt man die bedingte Anweisung einzurücken. Das müssen Sie aber nicht tun, denn Sie wissen ja, dass der Compiler Sie lieb hat, egal wie Sie den Quelltext optisch gestalten.

Nur eine Anweisung unter eine Bedingung stellen zu können, ist natürlich etwas spartanisch. Sollen mehrere Anweisungen nur dann ausgeführt werden, wenn die Bedingung erfüllt ist, benötigen Sie einen Block, den Sie wie gewohnt in geschweifte Klammern setzen.

```
if (munition>0) {
    // ballern
    // auf Treffer prüfen
}
```

Listing 4.2: Zu viel Munition hat mehrere Konsequenzen.

Um festzustellen, ob der Schuss in diesem Beispiel ein Treffer war, brauchn Sie wieder eine Abfrage. Dann werden die Bedingungen verschachtelt. Und obwohl hier nur eine Anweisung in dem Block steht, ist es übersichtlicher, bei jeder Abfrage einen Block zu bilden.

```
if (munition>0) {
    // ballern
    if (treffer>0) {
        score++;
    }
}
```

Listing 4.3: Verschachtelte Bedingungen

 Syntaxaufbau `if`:

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

```
if (Bedingung) {
    Anweisung1;
    Anweisung2;
}
```

Operatoren

Die Bedingung steht immer in der Klammer hinter dem `if`. Das Ergebnis einer Bedingung kann nur wahr oder falsch sein. Eine andere Möglichkeit gibt es nicht. Falls Ihr alter Mathelehrer Ihnen vielleicht etwas vom Gesetz des ausgeschlossenen Dritten erzählt hatte, wissen Sie spätestens jetzt, dass er damit nicht die Vorschriften für das Schließen einer Ehe gemeint hat. In Java ergibt die Prüfung der Bedingung daher immer eines der beiden Literale für Wahrheitswerte, die Sie bereits kurz kennengelernt haben: `true` oder `false`.

Wo wir gerade bei zwei sind: Eine Bedingung besteht nicht immer, aber oft aus zwei Operanden, die verglichen werden. Dazwischen steht dann ein Vergleichsoperator, der größer, kleiner, gleich oder ungleich darstellen kann. Die Tabelle zeigt eine Übersicht der numerischen Vergleichsoperatoren.

Operator	Bedeutung
a == b	ist a gleich b?
a != b	ist a ungleich b?
a > b	ist a größer als b?
a >= b	ist a größer oder gleich b?
a < b	ist a kleiner als b?
a <= b	ist a kleiner oder gleich b?

Tabelle 4.1: Vergleichsoperatoren



Für die Prüfung auf Gleichheit werden, anders als in der Mathematik, zwei Gleichheitszeichen verwendet – wenn Sie versehentlich nur eines einfügen würden, würden Sie nicht auf Gleichheit prüfen, sondern stattdessen einen Wert zuweisen! Freundlicherweise wird Java Sie allerdings in diesem Fall warnen.

Vielleicht ist es eine Eselsbrücke, dass eine Abfrage auf Gleichheit immer zwei mögliche Zustände kennt. Egal, wie Sie sich das am besten einprägen können: Ein Gleichheitszeichen ist die Zuweisung, zwei Gleichheitszeichen stellen den Vergleichsoperator dar.

Alternativen heißen Else

Was passiert eigentlich, wenn die Bedingung nicht zutrifft? Dann überspringt das Programm alle Befehle, die zum `if` gehören, und macht dahinter weiter. Manchmal soll aber auch in dem Fall, dass die Bedingung nicht zutrifft, etwas passieren, zum Beispiel: Wenn der Alkoholpegel unter einem Grenzwert liegt, darf ich Auto fahren. Andernfalls muss ich mir ein Taxi rufen. Dieser andere Fall wird in Java durch den Befehl `else` eingeleitet.

```
if (alkoholPegel<GRENZWERT) {
     // Sie duerfen mit dem Auto fahren
} else {
    // Taxi rufen!
}
```

Listing 4.4: Entweder saufen oder fahren

Oder schauen wir uns das Ballerspiel an! Da wird man nachladen, wenn kein Schuss mehr da ist.

```
if (munition>0) {
    // ballern
} else {
    // nachladen
}
```

Listing 4.5: Zu wenig Munition hat auch Konsequenzen.

Um einen Plan B auszuführen, bietet die `if`-Anweisung einen `else`-Zweig an. Er wird ausgeführt, wenn die Bedingung nicht zutrifft.



Syntaxaufbau if-else:

```
if (Bedingung)
     Anweisung;
else
     Gegenanweisung;
```

```
if (Bedingung) {
    Anweisung1;
    Anweisung2;
} else {
```

```
            Gegenanweisung1;
            Gegenanweisung2;
        }
```

Als Beispiel für die Anwendung einer `if-else`-Konstruktion können Staffelpreise aller Art dienen. So könnte der Übernachtungspreis einer Ferienwohnung niedriger sein, wenn man sie länger als eine Woche mietet. Immerhin spart der Vermieter die Reinigung und den Aufwand beim Mieterwechsel.

```
class Ferien {
    public static void main(String[] args) {
        final int tage = 11;
        double preis = 0.0;
        if (tage>7) {
            preis = tage * 45.0;
        } else {
            preis = tage * 60.0;
        }
        System.out.println(preis);
    }
}
```

Listing 4.6: Ferienwohnungspreise

Logeleien

Die Welt ist manchmal kompliziert. So kann es sein, dass eine einfache Bedingung nicht reicht. Manchmal sind mehrere Einflussgrößen zu berücksichtigen. Was passiert, wenn es mehr als eine Bedingung gibt?

Wenn alles gelten muss: UND

Sie dürfen nur Auto fahren, wenn Sie einen Führerschein besitzen und wenn Sie nüchtern sind. Zwei Bedingungen, beide müssen zutreffen. Man spricht hier von einer logischen UND-Verknüpfung.

Unter Java wird diese Situation durch zwei kaufmännische Und-Zeichen `&&` dargestellt. Man könnte den Eindruck gewinnen, dass in der booleschen Ecke von Java alles doppelt ist.

Wenn also für eine Berechnung der Nenner ungleich 0 und das Quadrat größer oder gleich 0 sein müssten, würde man das in Java so formulieren:

```
if ((nenner!=0) && (quadrat>=0))
```

An sich sind die zusätzlichen Klammern um die Teilbedingungen gar nicht erforderlich. Sie erhöhen aber die Lesbarkeit und wirken auch irgendwie hübsch,

finden Sie nicht?

Da die Wahrheit bei Programmierern nur die Werte `true` und `false` kennt, lässt sich schnell eine Tabelle aufstellen, welche Ergebnisse die Verknüpfung unter allen denkbaren Wahrheitswerten liefert. Solche Tabellen nennt man Wahrheitswertetabelle (siehe Tabelle 4.2).

A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

Tabelle 4.2: Wahrheitswertetabelle für UND

Es ist sehr einfach, eine solche Wahrheitswertetabelle durch ein Programm zu erstellen oder zu prüfen.

```
class WahrheitUnd {
    public static void main(String[] args) {
        System.out.println( true  && true );
        System.out.println( true  && false );
        System.out.println( false && true );
        System.out.println( false && false );
    }
}
```

Listing 4.7: Wahrheitstabellenprogramm für UND

Die Ausgabe besteht aus einem `true` in der ersten Zeile, gefolgt von dreimal `false` und entspricht damit denen der Wahrheitstabelle 4.2 von UND.



Alle Teilbedingungen müssen erfüllt sein, damit die gesamte Bedingung erfüllt ist:

Wer den Führerschein hat UND nüchtern ist, darf Auto fahren.

Wohltemperierte Fische im Wellness-Intervall

In einer Anleitung für das Halten bestimmter Fische im Aquarium stand zu lesen, dass die Wassertemperatur zwischen 22 und 28 Grad liegen müsste. Wie würde eine entsprechende Abfrage in Java formuliert sein? Spontan würden wir Folgendes schreiben:

```
if (22 <= temperatur <= 28) { // das klappt nicht!
```

Das wird Ihnen der Compiler nicht für Geld und gute Worte abnehmen. Wieso versteht Java das nicht? Gehen wir das mal Schritt für Schritt durch. Zunächst liest

der Compiler den folgenden Ausdruck:

```
22 <= temperatur
```

Er vergleicht die Zahl 22 mit einer Variablen, die auch eine Zahl enthält. Das Ergebnis dieses Ausdruck ist wahr oder falsch, also vom Typ `boolean`. Nennen wir das Ergebnis einfach mal `istZuKalt`. Dann wäre der nächste Schritt:

```
if (istZuKalt <= 28) { // das klappt nicht!
```

Hier wird verglichen, ob ein Wahrheitswert kleiner als eine Zahl ist, und das geht richtig schief. So geht es nicht. Also versuchen wir uns erst einmal in den Teilausdrücken. Der Vergleich zwischen 22 und der Variablen `temperatur` hatte ja schon geklappt.

```
if (22 <= temperatur)
```

Aber das reicht nicht. Wir brauchen noch eine Abfrage, ob die Temperatur kleiner oder gleich 28 ist. Das ist aber nicht so schwer.

```
if (temperatur <= 28) {
```

Nun müssen wir beides kombinieren. Eine Verknüpfung muss her. Beides muss gelten, also benötigen Sie ein UND!

```
public class TemperaturBereich {
    public static void main(String[] args) {
        int temperatur = 26;
        if ((22 <= temperatur) && (temperatur <= 28)) {
            System.out.println("Prima Aquarium!");
         } else {
            System.out.println("Fische tot!");
        }
    }
}
```

Listing 4.8: Aquarientemperaturüberwachung

Ändern Sie ein paar Mal die Temperatur und prüfen Sie, ob die Abfrage sauber arbeitet!

Wenn mindestens eines gilt: OR

Neben der UND-Verknüpfung, bei der alle Teilbedingungen zutreffen müssen, gibt es die ODER-Verknüpfung, bei der es reicht, wenn eine der Teilbedingungen zutrifft. Beispiel: Sie dürfen einen Kinofilm anschauen, wenn Sie den Eintritt zahlen oder wenn Sie eine Freikarte besitzen. Nur eine der Bedingungen muss zutreffen.

Übrigens werden Sie auch ins Kino kommen, wenn Sie sowohl den Eintritt bezahlen als auch eine Freikarte haben. Das ist vielleicht nicht besonders schlau,

aber zulässig.



Das Zeichen für eine logische Oder-Verknüpfung ist der doppelte senkrechte Strich (||). Auf einer deutschen PC-Tastatur finden Sie den senkrechten Strich ganz links unten beim Kleinerzeichen. Damit er erscheint, müssen Sie zusätzlich AltGr drücken. Auf einem Mac finden Sie ihn unter Alt + 7.

```
if ((nenner>0) || (nenner<0)) {
```

Die Bedingung wird zutreffen, wenn `nenner` größer als 0 oder kleiner als 0 ist. Nur wenn beide Aussagen falsch sind, ist die Gesamtaussage falsch. Das Beispiel ist also die poetische Ausdrucksweise dafür, dass `nenner` ungleich 0 sein soll.



Es muss mindestens eine Teilbedingung erfüllt sein, damit die gesamte Bedingung erfüllt ist:

Wer Eintritt zahlt ODER eine Freikarte hat ODER eingeladen wird, darf ins Kino.

Da mir gerade die Tabellen ausgegangen sind, helfen Sie sich doch mit dem folgenden kleinen Programm:

```
class WahrheitOder {
    public static void main(String[] args) {
        System.out.println( true  || true );
        System.out.println( true  || false );
        System.out.println( false || true );
        System.out.println( false || false );
    }
}
```

Listing 4.9: Wahrheitstabellenprogramm für ODER

Das Ergebnis kennen Sie vermutlich schon vor der Ausführung. Genau, die ersten drei Zeilen liefern `true`, die letzte `false`. Und was für ein Glück, ich habe doch noch eine Tabelle gefunden:

A	B	A && B
true	true	true
true	false	true
false	true	true
false	false	false

Tabelle 4.3: Wahrheitswertetabelle für ODER

Exklusiv ODER

Die ODER-Verknüpfung ergibt ja auch dann `true`, wenn mehr als eine Teilbedingung zutrifft. Im normalen Sprachgebrauch ist mit »oder« allerdings meist ein »entweder ... oder« gemeint. Wenn jemand sagt, dass er mit der Bahn oder mit dem Auto fahren will, wird er nicht beides gleichzeitig fahren können. Insofern ist es also keine ODER-Verknüpfung. Wenn wie hier die gesamte Bedingung `false` sein soll, wenn mehr als eine Teilbedingung zutrifft, dann brauchen Sie das exklusive ODER, das auch kurz XOR genannt wird.

Java verwendet als Zeichen für XOR das Dach ^. Die Taste funktioniert bei Ihnen vielleicht nicht auf Anhieb. Drücken Sie zweimal drauf! Auf dem Mac müssen Sie die Kombination Alt + ⇧ + K drücken. Das Erstellen der Wahrheitswertetabelle ist schnell programmiert:

```
class WahrheitXOR {
    public static void main(String[] args) {
        System.out.println( true  ^ true );
        System.out.println( true  ^ false );
        System.out.println( false ^ true );
        System.out.println( false ^ false );
    }
}
```

Listing 4.10: Wahrheitstabellenprogramm für XOR

Als Ergebnis erhalten Sie die folgenden Werte:

A	B	A ^ B
true	true	false
true	false	true
false	true	true
false	false	false

Tabelle 4.4: Wahrheitswertetabelle für exklusives ODER



Es muss genau eine Teilbedingung erfüllt sein, damit die gesamte Bedingung erfüllt ist:

ENTWEDER Sie fahren mit der Bahn ODER Sie fliegen mit dem Flugzeug.

Genau das Gegenteil: NOT

Und dann gibt es noch eine unäre logische Verknüpfung. Unär heißt, dass es nur auf einen Operanden wirkt. Diese Verknüpfung ist das NOT und wird durch ein Ausrufezeichen dargestellt. Wollen Sie also einen logischen Ausdruck umkehren,

können Sie einfach eine Klammer darum machen und ein Ausrufezeichen davorstellen.

Die Wahrheitswertetabelle ist bei NOT noch kleiner. Es gibt ja nur einen Operanden, der maximal zwei Zustände annehmen kann.

```
class WahrheitNOT {
    public static void main(String[] args) {
        System.out.println( ! true  );
        System.out.println( ! false );
    }
}
```

Listing 4.11: Wahrheitstabellenprogramm für NOT

Auch die Wahrheitstabelle wird ausgesprochen übersichtlich:

A	!A
true	false
false	true

Tabelle 4.5: Wahrheitswertetabelle für NOT



Es muss der Operand falsch sein, damit die gesamte Bedingung erfüllt ist:

Sie sind anwesend, wenn Sie NICHT abwesend sind.

Negieren, aber richtig: De Morgan

Die Negierung verknüpfter Bedingungen ist etwas tückisch. Der englische Mathematiker De Morgan hat das in eine schöne Formel gepackt, die besagt, dass bei einer Negierung eines booleschen Ausdrucks die Teilbedingungen negiert werden müssen und dass ein UND zum ODER wird und ein ODER zum UND. Wenn Sie nun englische Mathematiker spontan nicht mehr so lieb haben, sollten Sie ein Beispiel nehmen. Dann wird es verständlich.

Umkehrung einer UND-Verknüpfung

Betrachten wir folgendes Beispiel: Man darf nur Auto fahren, wenn man einen Führerschein hat und nüchtern ist. Beide Bedingungen müssen zutreffen, der gesamte Ausdruck ist also eine UND-Verknüpfung.

Führerschein	nüchtern	Führerschein && nüchtern	negiert
true	true	true	false
true	false	false	true
false	true	false	true
false	false	false	true

Tabelle 4.6: Wahrheitswertetabelle zulässigen Fahrens und der Negation

Wir drehen das Beispiel um. Wann dürfen Sie nun nicht fahren? Naheliegend wäre: Wenn Sie keinen Führerschein haben UND wenn Sie nicht nüchtern, also betrunken sind. Das stimmt zwar erst mal, weil Betrunkene ohne Führerschein definitiv nicht Auto fahren sollten, es reicht aber nicht aus. Die Wahrheitstabelle beweist es:

kein Führerschein	nicht nüchtern	kein Führerschein && nicht nüchtern
false	false	false
false	true	false
true	false	false
true	true	true

Tabelle 4.7: Wahrheitswertetabelle der Umkehrung mit UND

Sie dürfen schließlich auch dann nicht Auto fahren, wenn Sie nüchtern sind, aber keinen Führerschein haben. Wenn Sie nicht nüchtern sind, ist es völlig egal, ob Sie einen Führerschein haben. Sie dürfen nicht fahren. Es reicht also bereits eine Teilbedingung, um nicht fahren zu dürfen! Also muss die richtige Verknüpfung der beiden Verneinungen ein ODER sein. Schauen Sie sich die Wahrheitstabelle mit der ODER-Verknüpfung an und es passt zur Negierung der Tabelle 4.6.

kein Führerschein	nicht nüchtern	kein Führerschein || nicht nüchtern
false	false	false
false	true	true
true	false	true
true	true	true

Tabelle 4.8: Wahrheitswertetabelle der Umkehrung mit ODER



Die Negierung einer UND-Verknüpfung ist eine ODER-Verknüpfung der negierten Teilbedingungen.

Wohlfühltemperatur für Fische

Kommen wir noch einmal auf die Betriebstemperatur des Aquariums zurück. Die Abfrage, ob die Wassertemperatur optimal ist, lautet:

```
if ((22 <= temperatur) && (temperatur <= 28)) {
```

Nun versuchen Sie die Umkehrung. Sie wollen feststellen, ob die Temperatur außerhalb des Bereichs liegt.

```
if (!(temperatur >= 22) || !(temperatur <= 28)) {
```

Listing 4.12: Aquarientemperaturüberwachung zum Zweiten

Die Bedingung ist ja nun wirklich reichlich unleserlich. Aber `!(temperatur>=22)` ist ja wohl das Gleiche wie `temperatur<22`. Also:

```
if ((temperatur < 22) || (temperatur > 28)) {
```

So funktioniert es! Wenn die Temperatur kleiner als 22 ODER höher als 28 wird, liegt sie außerhalb der gesunden Temperatur für Fische. UND wäre hier Quatsch, weil die Temperatur ja nicht gleichzeitig höher als 28 und niedriger als 22 sein kann. Sobald mindestens eine der beiden Grenzen durchbrochen wird, ist bereits der Karren im Dreck. Damit ist dies das Gegenteil der Aussage, dass die Temperatur größer oder gleich 22 UND kleiner oder gleich 28 sein muss, damit die Fische gesund bleiben.

Umkehrung einer ODER-Verknüpfung

Wir spielen nun eine Oder-Verknüpfung durch. Dabei setzen wir einmal voraus, dass auch bei der ODER-Verknüpfung die Negierung in ein UND kippt. Sie dürfen nicht ins Kino, wenn Sie keinen Eintritt bezahlt haben UND keine Freikarte haben UND nicht eingeladen sind. Trifft nur eine der Bedingungen nicht zu, stimmt die Aussage nicht, dass Sie nicht ins Kino kommen.



Die Negierung einer ODER-Verknüpfung ist die UND-Verknüpfung der negierten Teilbedingungen.

Und weil ich für heute wirklich genug gearbeitet habe, beweise ich Ihnen das jetzt nicht mehr. Prüfen Sie es per Wahrheitstabelle oder noch besser: Schreiben Sie ein Programm, das diese Aussage überprüft.

Boolesche Variablen

Variablen vom Typ `boolean` haben Sie ja bereits kennengelernt.

```
boolean istErledigt = false;
```

Diese Variable könnte den Status einer Aufgabe speichern. Sie kann dann direkt in einer Abfrage verwendet werden.

```
if (istErledigt) {
```

Natürlich kann auch ein Ausrufezeichen davor gesetzt werden, um zu prüfen, ob die Aufgabe eben noch nicht erledigt ist.

```
if (!istErledigt) {
```

Einer booleschen Variablen kann ein beliebiger logischer Ausdruck zugewiesen werden.

```
istErledigt = zaehler >= MAXANZAHL;
```

Und das alles, obwohl `boolean` nur zwei Werte annehmen kann. Das nenne ich mal einen primitiven Typ.

Das Fragezeichen und der bedingte Ausdruck

Eine etwas ungewöhnliche Form der Abfrage ist die bedingte Zuweisung. Sie können sie einsetzen, wenn sich der Wert einer Zuweisung in Abhängigkeit einer Bedingung ändert. Beispielsweise kann es sein, dass der Stundenlohn am Sonntag höher ist als sonst. Wenn die boolesche Variable `sonntag` also wahr ist, erhält die Variable `stundenlohn` einen höheren Wert. Bisher würden Sie das so schreiben:

```
if (sonntag) {
    stundenlohn = 25;
} else {
    stundenlohn = 20;
}
```

Listing 4.13: Sonntagsarbeit

Weil Programmierer meist ziemlich schreibfaul sind, können Sie das auch kürzer ausdrücken, und zwar mit dem Fragezeichenoperator:

```
stundenlohn = sonntag ? 25 : 20;
```

Der Ausdruck auf der rechten Seite des Gleichheitszeichen liest sich so, dass der Doppelpunkt zwei Ausdrücke trennt. Ist die Bedingung vor dem Fragezeichen wahr, wird der Ausdruck links vom Doppelpunkt verwendet, andernfalls der rechte.



Bei komplexen Ausdrücken führt der Fragezeichenoperator schnell zu unleserlichen Programmen. Dann benötigt ein `if` zwar ein paar Zeilen mehr, entspannt aber den Leser.



Syntax des Fragezeichenoperation:

```
Bedingung ? WahrAusdruck : FalschAusdruck;
```

Die Fallunterscheidung – switch case

Es gibt Situationen, in denen Sie eine Variable auf verschiedene Inhalte abfragen wollen. Ein gutes Beispiel dafür sind die Stockwerke eines Kaufhausfahrstuhls. Je nachdem, wo sich der Fahrstuhl befindet, könnte er den passenden Text ausgeben, etwa »Etage 4: Herrenunterbekleidung und andere Wegwerfartikel«.

Eine solche Abfrage können Sie durch gestaffelte `if`-Abfragen umsetzen.

```
public class KaufhausIf {
    public static void main(String[] args) {
        int stockwerk = 3; // hier ändern zum Testen!
        if (stockwerk == 1) {
            System.out.println("Damenbekleidung");
        } else if (stockwerk == 2 ) {
            System.out.println("Spielwaren");
        } else if (stockwerk == 3) {
            System.out.println("Haushaltswaren");
        } else { // wenn kein Fall passt
            System.out.println("Sie verlassen das Kaufhaus.");
        }
    }
}
```

Listing 4.14: Fahrstuhltexte per gestaffeltem `if`

Wenn Sie das Beispiel im Geiste durchlaufen, werden Sie erkennen, dass jede Abfrage bis zur dritten fehlschlägt und so der `else`-Zweig durchlaufen wird, der jeweils wiederum aus einer `if`-Anweisung besteht. Erst dann wird die Ausgabe angestoßen. Der nachfolgende `else`-Zweig kann dann ignoriert werden, sodass der Ablauf direkt an das Ende des Programms geleitet wird.

Wenn Sie wie hier immer dieselbe Variable nach einzelnen Werten abfragen, gibt es eine Sonderform der Abfrage, die Fallunterscheidung mit `switch` und `case`. Sie macht das Programm zwar nicht kürzer, aber ermöglicht eine bessere Übersicht darüber, welche Werte wo abgefragt werden.

```
public class KaufhausSwitch {
    public static void main(String[] args) {
        int stockwerk = 3; // hier ändern zum Testen!
        switch (stockwerk) { // Unterscheidungsmerkmal stockwerk
        case 1: // springt hierhin, wenn stockwerk==1 ist
            System.out.println("Damenbekleidung");
            break;
        case 2: // springt hierhin, wenn stockwerk==2 ist
            System.out.println("Spielwaren");
            break;
        case 3: // springt hierhin, wenn stockwerk==3 ist
            System.out.println("Haushaltswaren");
            break;
        default: // nur wenn kein anderer case passt
            System.out.println("Sie verlassen das Kaufhaus.");
        }
    }
}
```

Listing 4.15: Der Fahrstuhl

Beachten Sie, dass am Ende jedes Falls ein `break` steht. Das führt dazu, dass das Programm aus dem `switch`-Block hinausspringt. Das wird normalerweise genau das sein, was Sie haben möchten. Wenn Sie kein `break` setzen, wird das Programm zwar nach wie vor vom `switch` zum passenden `case` springen, danach aber auch die Anweisungen in dem oder den noch folgenden Fällen. Wenn also `stockwerk` eine 2 enthält, würde zwar der Fall für 1 übersprungen und der für 2 ausgeführt, darüber hinaus würden aber auch die Anweisungen für Fall 3 und den `default`-Fall ausgeführt, was ja keinen Sinn ergibt. Damit das nicht passiert, schließt man diesen Fall mit einem `break` ab und überspringt damit die Anweisungen, die für die anderen Stockwerke vorgesehen sind.

Es gibt aber auch Situationen, in denen es sehr praktisch ist, dass man das `break` weglassen kann.

Im folgenden Beispiel soll die Besteuerung nach Artikelgruppe unterschieden werden. Diese wird in der Variablen `artikelgruppe` vom Typ `char` gespeichert. Die Waren der Gruppen B, N und P werden mit 7 %, die Waren der Gruppen S und A werden mit 19 % besteuert.

```
switch (artikelgruppe) {
case 'B':
case 'N':
case 'P':
   mwstSatz = 7;
   break;
case 'S':
case 'A:
   mwstSatz = 19;
   break;
}
```

Listing 4.16: Wenn man `break` weglässt



Das Fehlen eines `break`-Befehls am Ende eines Falles kann leicht übersehen werden. Der Compiler warnt nicht davor, weil es zulässig ist. Aber oft hat sich der Programmierer etwas ganz anderes dabei gedacht.

Also: Wenn Sie einmal absichtlich ein `break` weglassen, schreiben Sie besser einen Kommentar, damit bei einer späteren Wartung nicht plötzlich ein `break` eingefügt wird, wo es nicht hingehört.

Diese Fallunterscheidung hat übrigens keinen `default`-Zweig. Das ist nicht zwingend erforderlich. Alle nicht explizit genannten Fälle werden dann einfach nicht behandelt.



Syntax einer Fallunterscheidung:

```
switch (variable) {
case wert:
     anweisung1;
     break;
case nochnwert:
     anweisung2;
default:
     anweisung3;
}
```

Endlich lernen wir Schleifen zu binden!

Die Schuhindustrie hat der Schleife den Kampf angesagt. Durch Kletten und Slippen wird das Betreten des Schuhs vereinfacht, sodass viele Zeitgenossen gar keine Schleife mehr binden können.

Beim Programmieren gehören die Schleifen längst nicht zum alten Eisen. Aber sie wird nicht gebunden, sondern bildet wiederholende Abläufe nach. Auch unterscheiden sich die Java-Schleifen von der Dauerschleife in den Fernseh-Shopping-Kanälen in der Hinsicht, dass es bei Java eine Ende-Bedingung gibt.

Ein typisches Beispiel für eine solche Wiederholung ist das Zählen. Wenn Sie ohne eine Schleife die Zahlen von 1 bis 20 ausgeben wollten, müssten Sie 20 Zeilen untereinander schreiben. Eine blöde Herangehensweise! Überlegen Sie, was Sie tun, wenn Sie zählen. Und dann versuchen wir, dies in ein Java-Programm umzusetzen:

1. Sie legen eine 1 in eine Variable und geben diese auf dem Bildschirm aus. Ein guter Anfang!
2. Nun müssen Sie zur 2 gelangen und von da aus zur 3 und so fort. Dazu wird auf den vorigen Wert eine 1 aufaddiert und auch ausgegeben. Dann addieren Sie wieder eine 1 und geben Sie aus und so weiter und so fort.
3. Und wenn es keinen Stromausfall gibt, werden die Ausgaben niemals enden. Außer Sie haben daran gedacht, zu prüfen, ob die Zahl 20 überschritten wurde.

Damit haben wir die drei Aspekte der Schleife zusammen:

1. Startwert: Die Variable setzen Sie auf 1!
2. Schleifenanweisung: Sie geben die Variable aus und erhöhen sie.
3. Schleifenbedingung: Die Variable darf nicht größer als 20 werden.

Wenn Sie die Schleife als Eisenbahnstrecke darstellen würden, sähe sie etwa so aus:

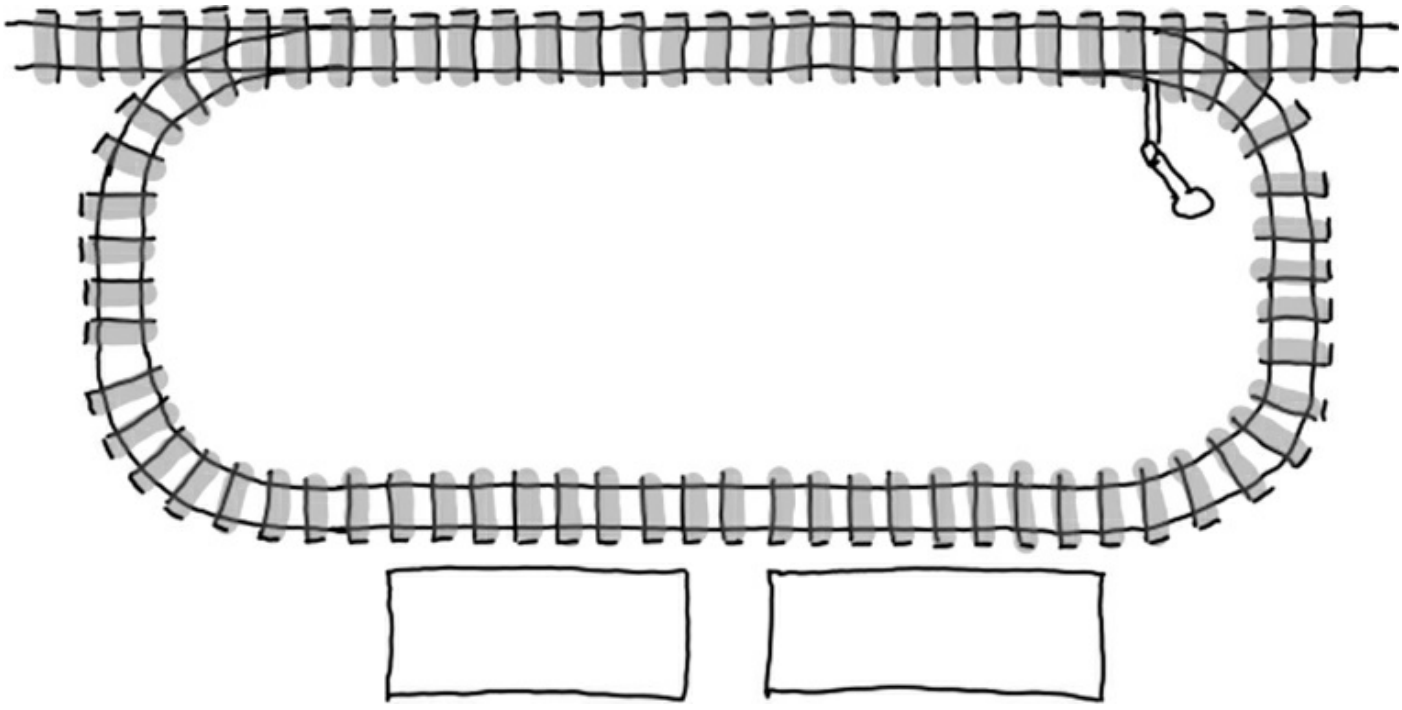


Abbildung 4.2: Eine Weiche führt zur Wiederholung.

Der Startwert wird vor der ersten Weiche gesetzt. An der Weiche wird dann gefragt, ob der Wert schon über 20 ist. Wenn nicht, fährt der Zug in die Schleife. Dort kommt er an einer oder mehreren Stationen vorbei. An der ersten Station wird der Wert ausgegeben und bei der nächsten Station erhöht. Irgendwann kommt der Zug wieder an der Weiche vorbei, wo der Wert wieder geprüft wird, und das Spiel beginnt von vorn.

Die kopfgesteuerte while-Schleife in Java

Es gibt drei verschiedene Arten von Schleifen in Java. Die wohl verbreitetste ist die `while`-Schleife.

1. Der Befehl `while` leitet die Schleife ein.
2. In Klammern folgt die Bedingung, unter der die Schleife ausgeführt wird.
3. Die folgende Anweisung bildet den Schleifenkörper. Es können auch mehrere Anweisungen sein, wenn sie mit geschweiften Klammern zu einem Block zusammengefasst werden.

Die auf den `while`-Befehl folgende Anweisung wird so lange wiederholt, bis die Bedingung nicht mehr erfüllt ist.

Nun können Sie ein Programm schreiben, das von 1 bis 20 zählt. Der Rahmen ist so wie immer. Sie legen ein Projekt an, definieren eine Klasse, die Sie beispielsweise `Zaehlen` nennen. In der `main`-Methode bauen Sie eine Schleife ein. Als Zählvariable verwenden Sie eine Variable namens `zahl` vom Typ `int`.

```
public class Zaehlen {
    public static void main(String[] args) {
        int zahl = 1; // mit dem Startwert initialisieren
        while (zahl <= 20) { // Die Schleifenbedingung
            System.out.println(zahl); // Der Schleifenkörper
            zahl++;
        }
    }
}
```

Listing 4.17: Zählen für Anfänger

1. Da das Zählen mit 1 starten soll, wird die Variable `zahl` mit 1 initialisiert.
2. In der Schleifenbedingung wird geprüft, ob `zahl` kleiner oder gleich 20 ist. Das ist beim ersten Durchlauf natürlich der Fall.
3. Im Schleifenkörper wird `zahl`, im ersten Schleifendurchlauf also 1, in einer

neuen Zeile ausgegeben. In der nächsten Zeile wird `zahl` um 1 erhöht.

4. Nach Ablauf des Schleifenkörpers springt das Programm wieder zum Schleifenkopf. Dort wird wieder geprüft, ob `zahl` immer noch kleiner oder gleich 20 ist, und weil das auch im zweiten Durchlauf so ist, wird wieder der Schleifenkörper ausgeführt.
5. Das Ganze wird so lange wiederholt, bis `zahl` gleich 21 ist. Dann wird die Bedingung nicht mehr erfüllt und die Schleife wird beendet.

Lassen Sie uns ein paar Gedankenexperimente machen.

- ✔ Was passiert wohl, wenn wir die beiden Anweisungen innerhalb des Schleifenkörpers austauschen? Denken Sie nach, ich gehe derweil und hole mir einen Kaffee.

  Fertig? Okay, Sie dürften zu dem Ergebnis gekommen sein, dass die erste dargestellte Zahl eine 2 sein wird, weil `zahl` vor der Ausgabe erhöht wird. Entsprechend wird die letzte Zahl 21 sein.
- ✔ Das war schon nett. Nun überlegen Sie, was passiert, wenn Sie statt `<=` als Operator `<` verwenden. Denken Sie in Ruhe nach, während ich den Kaffee wieder wegbringe.

  Na klar: Die Schleife wird natürlich eine Runde weniger drehen und damit die Zahlen von 2 bis 20 ausgeben.
- ✔ Was würde passieren, wenn Sie die Variable `zahl` nicht mit 1, sondern mit 0 initialisieren?

  Richtig: Das Programm würde mit der Ausgabe von 1 beginnen, nun 20 Mal rotieren und bei 20 enden.

Abgesehen davon, dass es immer eine gute Übung ist, zu durchdenken, was ein Programm tut, sehen Sie an den Überlegungen, dass ein Programm, das von 1 bis 20 zählt, ganz anders aussehen könnte. Und tatsächlich werden Informatiker aus gutem Grund meist bei 0 anfangen zu zählen und als Abbruchkriterium die Kleiner-Relation verwenden.

```
public class ZaehlenBis20 {
    public static void main(String[] args) {
        int zahl = 0; // mit dem Startwert initialisieren
        while (zahl < 20) { // Die Schleifenbedingung
            zahl++;
            System.out.println(zahl); // Der Schleifenkörper
        }
    }
}
```

Listing 4.18: Auch von 1 bis 20 zählen

Quadratzahlenliste

Nachdem Sie jetzt die Grundlage verstanden haben, bauen wir das Programm etwas aus und lassen es zusätzlich zu den Zahlen auch noch die Quadratzahlen ausgeben. Dafür brauchen Sie drei statt wie bisher eine Zeile für die Ausgabe. Die erste Zeile gibt die Zählzahl aus, dann kommt ein Trenner und dann die Quadratzahl.

```
public class Quadrat1 {
    public static void main(String[] args) {
        int zahl = 0; // mit dem Startwert initialisieren
        while (zahl < 20) { // Die Schleifenbedingung
            zahl++;
            System.out.print(zahl);
            System.out.print(": ");
            System.out.println(zahl*zahl);
        }
    }
}
```

Listing 4.19: Quadratzahlen

Was soll das mit dem `print`? Das hieß doch vorher `println`! `println` steht für »print line« und beginnt nach der Ausgabe eine neue Zeile. Fehlt das »ln«, dann werden die weiteren Ausgaben in die gleiche Zeile geschrieben, wodurch Sie wundervolle Tabellen komponieren können.



Syntaxaufbau `while`-Schleife:

```
while (Bedingung)
    Schleifenanweisung;
```

```
while (Bedingung) {
    Schleifenanweisung1;
    Schleifenanweisung2;
}
```

Zählschleife mit for

Sie werden sich vielleicht schon gefragt haben, warum ich Ihnen mit dem Zählen so auf die Nerven gehe. Sie wollen vermutlich nicht programmieren lernen, um einen Computer zählen zu lassen. Tatsächlich ist das Zählen aber wichtig, weil Listen typischerweise durchnummeriert sind. Und da ist die Fähigkeit des Zählens gefragt. Es ist sogar so wichtig, dass fast alle Programmiersprachen eine spezialisierte Schleife für das Zählen kennen.

Die `for`-Schleife ähnelt funktional der `while`-Schleife, fasst aber die immer wiederkehrenden Elemente einer Schleife in ihrem Kopf zusammen. Drei Bestandteile stehen in der Klammer hinter dem Befehl `for`, sodass man nichts vergessen kann.

1. Initialisierung
2. Schleifenbedingung
3. Schlussanweisung

Die Initialisierung setzt beispielsweise den Zähler auf 0. Die Bedingung prüft, ob die Schleife durchlaufen werden soll. Die Schlussanweisung wird immer am Ende der Schleife ausgeführt. Hier erhöht sie beispielsweise den Zähler. Formal werden diese drei Elemente in die Klammer hinter dem Befehl `for` eingetragen, jeweils durch ein Semikolon getrennt.

Das Quadratzahlenprogramm aus dem vorigen Abschnitt verkürzt die `for`-Schleife in eine etwas übersichtlichere Form.

```
public class Quadrat2 {
    public static void main(String[] args) {
        for (int zahl=1; zahl <= 20; zahl++) {
            System.out.print(zahl);
            System.out.print(": ");
            System.out.println(zahl*zahl);
        }
    }
}
```

Listing 4.20: Quadrieren mit for

Zinswachstum

Lassen Sie uns noch ein weiteres Beispiel programmieren. Wenn Sie beispielsweise Ihr Geld auf ein Sparkonto mit guten Zinsen einzahlen und warten, bis es immer mehr wird, können Sie dies in einer Schleife simulieren. Sie sehen, wie Ihr Geld langsam über die Jahre immer mehr wird. Natürlich können Sie auch die folgende Formel nachschlagen und in den Taschenrechner eingeben.

$$Kapital = Startkapital \cdot \left(\frac{1 + Zinssatz}{100}\right)^{Jahre}$$

Aber es ist doch schöner, den stetigen Anstieg zu sehen. Nehmen wir an, Sie zahlen immer am Ende eines Jahres 2.400 Euro auf ein Konto ein, das 6 % Zinsen abwirft. Machen Sie bitte nicht mich verantwortlich, wenn Ihre Bank Ihnen nicht so hohe Zinsen zahlt. Meine tut es nämlich auch nicht. Aber man darf ja mal ein wenig träumen.

Wir schlagen den Zins auf die Summe des Vorjahres und addieren die 2.400 Euro. Und das wiederholen wir so lange, bis die Jahre um sind. Nehmen wir als Beispiel 10 Jahre:

```
public class Kapital {
    final static double zinsSatz = 0.06;
    final static double zahlung = 2400;
    final static int maxJahr = 10;
    public static void main(String[] args) {
        double kapital = 0.00;
        for (int jahr=0; jahr<maxJahr; jahr++) {
            kapital = kapital * (1 + zinsSatz) + zahlung;
            System.out.print(jahr);
            System.out.print(" - ");
            System.out.println(kapital);
        }
    }
}
```

Listing 4.21: Wundersame Kapitalvermehrung

Haben Sie es gemerkt? Die Klammer der `for`-Schleife sieht etwas anders aus. Die erste Schleife initialisierte `jahr` mit 1, wie es Menschen zu tun pflegen. Die Schleife läuft dann, so lange `jahr` kleiner oder gleich 20 ist. Computer beginnen gern bei 0 zu zählen. Damit es immer noch zwanzig Runden sind, darf dann die Schleife nur noch laufen, solange zahl kleiner als 20 ist.

In manchen Fällen möchten Sie vielleicht mehrere Anweisungen in die Initialisierung oder in die Schlussanweisung der `for`-Klammer stellen. Das geht, wenn Sie diese Anweisungen durch ein Komma trennen. Sie sollten aber sparsam damit umgehen, weil es die Übersichtlichkeit nicht unbedingt erhöht.

```
for (int jahr=0; jahr<maxJahr;
     kapital=kapital*(1+zinsSatz)+zahlung, jahr++) {
```



Die `for`-Schleife ist eine kompakte Schreibweise einer `while`-Schleife, in der die typischen Bestandteile wie Initialisierung, Bedingung und

Schlussanweisung im Schleifenkopf friedlich koexistieren.

```
for (Initialisierung ; Schleifenbedingung ; Schlussanweisung)
    Anweisung;
```

Auch für die `for`-Schleife gilt, dass die Anweisung in den meisten Fällen durch einen Block von Anweisungen ersetzt wird.

Die fußgesteuerte do-while-Schleife

Die `while`- und `for`-Schleifen haben gleich vorn einen Türsteher, der kontrolliert, ob dem Programm Zutritt zu gewähren ist. Man nennt sie auch kopfgesteuerte Schleifen, und das, obwohl Türsteher nicht als typische Kopfarbeiter gelten. Bei diesen Schleifen kann es passieren, dass der Schleifenkörper überhaupt nicht durchlaufen wird.

In manchen Fällen soll der Schleifenkörper auf jeden Fall durchlaufen werden. Erst danach kann entschieden werden, ob ein weiterer Durchlauf erforderlich ist oder nicht. Für solche Fälle bietet Java einen dritten Schleifentyp an: die `do-while`-Schleife. Die `do-while`-Schleife ist deutlich gastfreundlicher und bittet das Programm erst einmal herein. Am Ende des Durchlaufs wird dann geprüft, ob es nicht eine Runde mehr sein darf. Diese Art von Schleifen nennt man fußgesteuerte Schleifen.

Bei dem Beispiel mit der Eisenbahn würde der Gleisplan nun so aussehen:

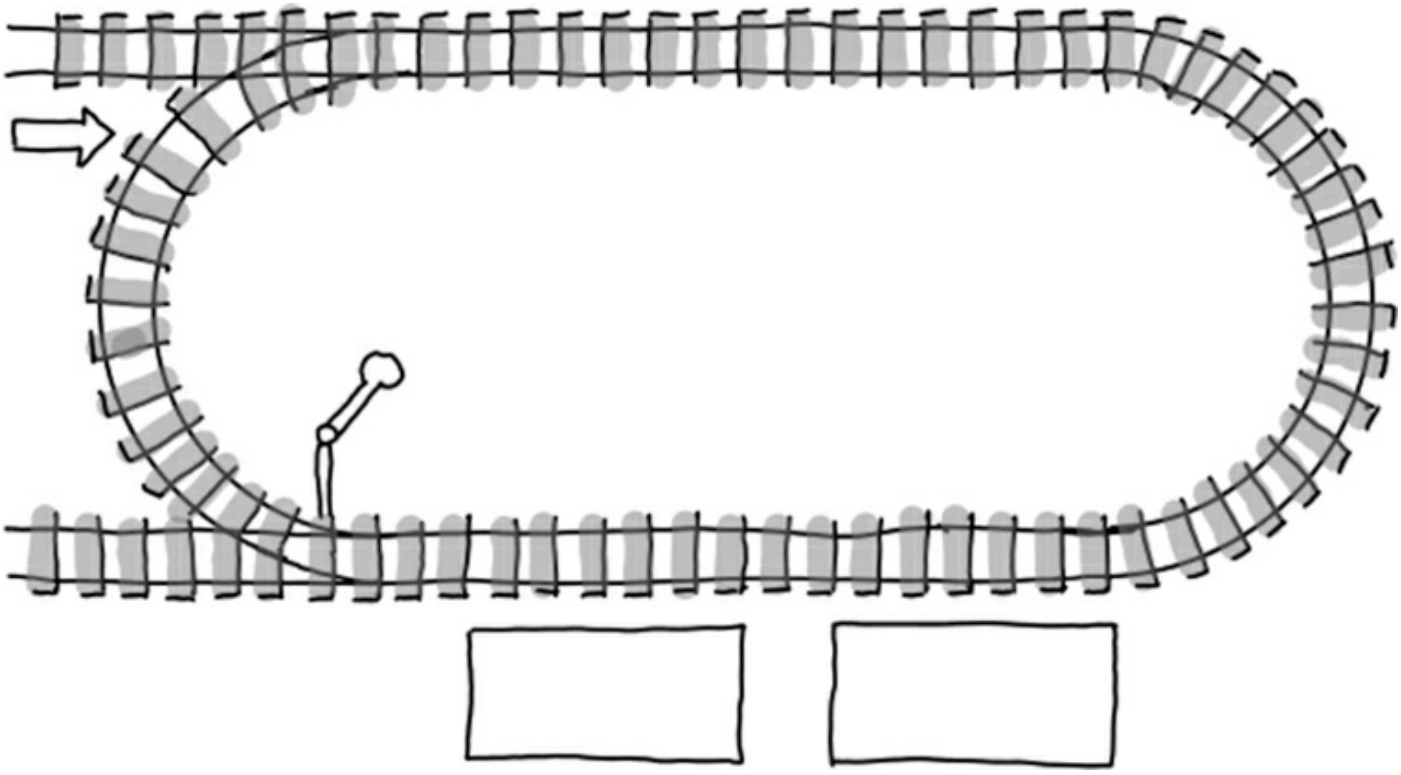


Abbildung 4.3: Eine Weiche am Ende der Schleife

Jetzt liegen die Stationen an der Strecke vor der Weiche. Steht die Weiche so, dass

der Kreis verlassen wird, sind die Stationen mindestens einmal abgefahren worden.

Syntaktisch wird der Anfang der Schleife mit dem Befehl `do` markiert. Es folgt die Anweisung, die wiederholt werden soll. Die Schleife schließt mit dem `while`-Befehl, dessen Klammer die Bedingung enthält, unter der die Schleife wiederholt wird. Natürlich können Sie das Quadratzahlenbeispiel auch mit einer `do-while`-Schleife realisieren.

```
public class Quadrat3 {
    public static void main(String[] args) {
        int zahl = 1; // mit dem Startwert initialisieren
        do { // Schleifenbeginn
            System.out.print(zahl);
            System.out.print(": ");
            System.out.println(zahl*zahl);
            zahl++;
        } while (zahl <= 20); // Die Schleifenbedingung
    }
}
```

Listing 4.22: Fußgesteuerte Quadratur

Viel hat sich nicht verändert, wozu also der Aufwand für noch eine Schleife? Die Prüfung am Ende passt zur Kontrolle von Benutzereingaben. Erst einmal muss die Schleife durchlaufen werden, damit der Benutzer auch eine Chance bekommt, etwas einzugeben. Dann wird geprüft, ob die Eingabe fehlerhaft war. Das würde zu einer Wiederholung der Schleife und damit zu einer erneuten Benutzereingabe führen.

 Syntaxaufbau `do-while`-Schleife:

```
do
    Anweisung;
while (Bedingung);
```

```
do {
    Anweisung1;
    Anweisung2;
} while (Bedingung};
```

Eine `do`-Schleife prüft erst am Ende, ob die Bedingung erfüllt wurde.

Wenn Schleifen brechen müssen

Wenn so eine Schleife vor sich hindreht, kommt sie manchmal zu Erkenntnissen, die einen sofortigen Ausstieg nahelegen. Damit Schleifen spontan in ihrem Ablauf unterbrochen werden können, gibt es in Java den Befehl `break`.

```
for (int i=0; i<100; i++) {
    anweisung1;
    if (fehler) {
        break;
    }
    anweisung2;
}
```

Listing 4.23: Kurz und gemein

Nun ist es in fast allen Lebensbereichen gut, spontan zu sein. Nur in der Programmierung schätzt man eher ein planvolles Vorgehen. Und das besagt, dass wir den Ausstieg durch eine boolesche Variable sauberer simulieren können als durch einen `break`-Befehl.

```
boolean rotiere = true;
for (int i=0; rotiere && i<100; i++) {
    anweisung1;
    if (fehler) {
        rotiere = false;
    } else {
        anweisung2;
    }
}
```

Listing 4.24: Sauberer Ausbruch

Der Vorteil dieser Lösung ist, dass alle Bedingungen, die den Ausstieg aus der Schleife bewirken, im Kopf deutlich genannt sind. Auch dass `anweisung2` nur ausgeführt wird, wenn kein Fehler auftritt, ist hier deutlich zu erkennen. Darum lässt der Befehl `break` nicht nur die Schleife brechen, sondern auch den ein oder anderen Programmierer, der das Programm später warten soll. Der Abbruch des Programms ist nämlich an der Schleifenbedingung nicht erkennbar.

Hinzu kommt, dass die scheinbar überflüssige boolesche Variable später noch von Wert sein kann, weil sie die Information enthält, dass die Schleife vorzeitig durch einen Fehler beendet wurde.

Ein ähnlicher Befehl ist `continue`. Er springt ans Schleifenende, verlässt die Schleife allerdings nicht, sondern führt direkt zur Überprüfung der Schleifenbedingung. Er ist meist durch eine gut plazierte `if`-Anweisung zu ersetzen.

Ich will das Thema aber nicht weiter vertiefen, weil die Befehle nicht gerade zur Übersicht im Programm beitragen und prinzipiell auch nicht erforderlich sind. Und deswegen ist es letztlich so: Ein Anfänger sollte diese Befehle nicht benutzen, und ein erfahrener Programmierer wird sie nicht benutzen.

Überblick behalten mit Struktogrammen

Programme wuchern. Während Sie bisher Schleifen und Abfragen einzeln betrachtet haben, werden erst Programme daraus, indem man sie aneinanderhängt oder gar ineinander verschachtelt. Das bedeutet, dass innerhalb von Schleifen Abfragen gestellt werden oder dass Schleifen innerhalb von Schleifen rotieren. Dieses Verweben von Schleifen und Abfragen wirkt vor allem auf den Anfänger verwirrend. Um einen Überblick zu gewinnen, haben sich Programmierer immer schon Darstellungen geschaffen, mit denen die Programmabläufe grafisch geordnet werden können. Eine dieser Darstellungen sind die Nassi-Schneidermann-Diagramme, die auch als *Struktogramme* bezeichnet werden.

Die Elemente eines Struktogramms bestehen vor allem aus Schleifen und Abfragen. Sie sind rechteckig und können so baukastenartig untereinander gestellt werden, wenn sie aufeinander folgen. Die Elemente enthalten aber selbst wieder Rechtecke, in denen wiederum Elemente eingesetzt werden können, um eine Verschachtelung darzustellen. Um einen Algorithmus darzustellen, reichen bereits drei Elemente: Abfrage, kopfgesteuerte Schleife und fußgesteuerte Schleife.

Abfrage

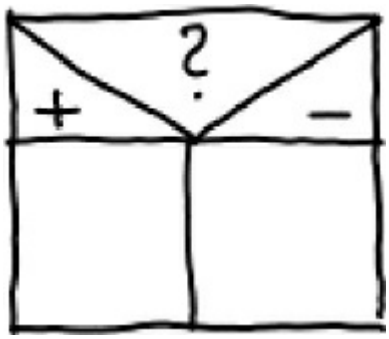


Abbildung 4.4: Struktogramm für `if`

In das obere Dreieck setzen Sie die Bedingung der Abfrage. Der linke Block wird ausgeführt, wenn die Bedingung zutrifft, der rechte, wenn er nicht zutrifft. Sie können in jeden dieser Blöcke Anweisungen hineinsetzen. Das können wiederum Abfragen sein. So stellen Sie eine verschachtelte Abfrage dar.

Kopfgesteuerte Schleife

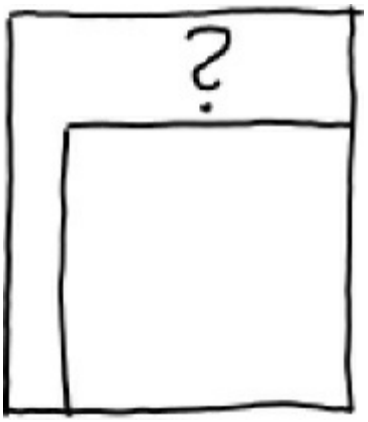


Abbildung 4.5: Struktogramm für `while`

Bei einer kopfgesteuerten Schleife wird oben die Schleifenbedingung eingetragen. Der Block rechts unten umfasst den Schleifenkörper. Da er rechteckig ist, können Sie dort prima weitere Schleifen und Abfragen einbauen.

Fußgesteuerte Schleife

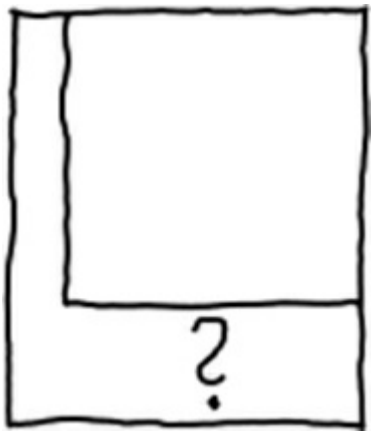


Abbildung 4.6: Struktogramm für `do`

Bei der fußgesteuerten Schleife befindet sich die Abfrage am Ende der Schleife.

Ermittlung des größten gemeinsamen Teilers

Wir werden nun Struktogramme einsetzen, um einen Algorithmus zu erfassen. Aus dem Struktogramm werden wir dann schnell ein Programm erstellen können. Als Beispiel verwenden wir den größten gemeinsamten Teiler (ggT) zweier Zahlen. Dabei handelt es sich um den größten Wert, durch den sich die beiden Zahlen teilen lassen. Für die Zahlen 36 und 24 wäre der ggT eben 12, für 28 und 16 wäre er 4.

Sie können den ggT ermitteln, indem Sie die beiden Zahlen in ihre Primfaktoren zerlegen. Das Produkt der Primzahlen, die beiden Zahlen gemeinsam sind, ergibt den ggT. Das funktioniert auf jeden Fall. Es gibt da aber noch eine bessere Methode, die man dem griechischen Mathematiker Euklid zuschreibt.

Euklid erkannte, dass der ggT zweier Zahlen – nennen wir sie a und b – zusätzlich auch der ggT der Differenz von a und b ist. Im Beispiel haben 28 und 16 also denselben ggT wie ihre Differenz, also 12. Da es einfacher ist, den ggT kleinerer

Zahlen zu bestimmen, kann man für die weitere Untersuchung 28 weglassen und den ggT von 16 und 12 betrachten. So kann man fortfahren, bis irgendwann eine Differenz von 0 entsteht. Der letzte Wert muss dann der ggT sein. Probieren wir das gleich mal aus.

```
28 - 16 = 12
16 - 12 = 4
12 - 4 = 8
8 - 4 = 4
4 - 4 = 0
```

Also muss 4 der ggT von 28 und 16 sein. Hurra, es klappt!

Wenn Sie dafür ein Programm schreiben möchten, benötigen Sie eine Schleife, die so lange läuft, bis die Differenz einmal 0 wird. Innerhalb der Schleife wird die Differenz gebildet. Zuvor müssen Sie aber sicherstellen, dass die kleinere Zahl von der größeren abgezogen wird. Das erreichen Sie mit einer Abfrage. Diese muss in den Körper der Schleife eingebaut werden. Sie erhalten so das Diagramm aus Abbildung 4.7.

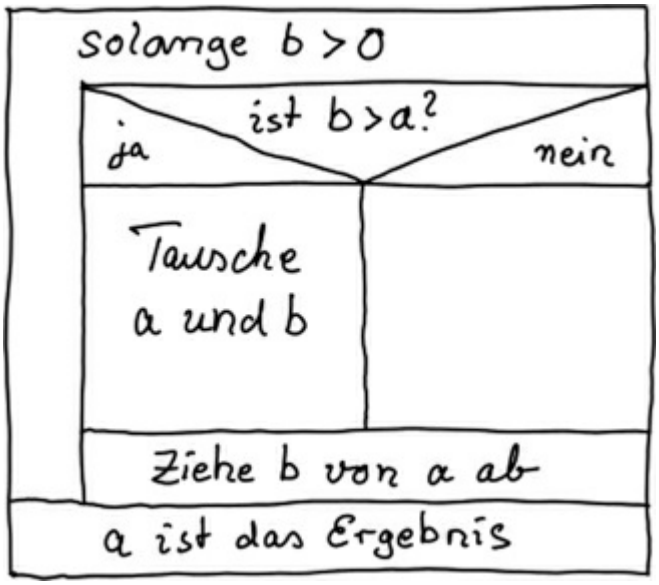


Abbildung 4.7: Struktogramm für die ggT-Berechnung

Damit haben wir eine schicke Grafik, die wir dekorativ im Flur vor dem EDV-Raum aufhängen können. Der Personalchef wird beeindruckt sein. Das ist wichtig. Aber wie kommen wir damit zu einem Programm? Zunächst füllen Sie das Struktogramm mit den entsprechenden Java-Befehlen.

In Abbildung 4.8 sieht alles schon sehr nach Java aus und es ist nicht so schwer, daraus ein fertiges Programm zu machen. Sie müssen nur noch den altbekannten Rahmen eines Java-Programms darum herumsetzen und schon können Sie es dem Compiler vor die Füße werfen.

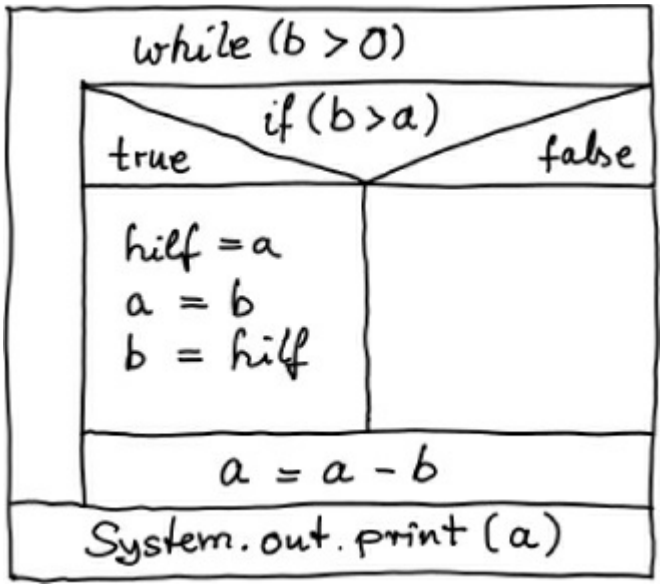


Abbildung 4.8: Struktogramm für ggT im Java-Code

```
public class GGT {
    public static void main(String[] srgs) {
        int a=24, b=36;
        while (b>0) {
            if (b>a) {
                // Tausche a und b
                int hilf = a;
                a = b;
                b = hilf;
            }
            a = a - b;
        }
        System.out.println(a);
    }
}
```

Listing 4.25: Das fertige ggT-Programm

Keine Sorge: Natürlich erstellen Programmierer nicht für jedes Programm jedes Mal ein Struktogramm. Mit ein bisschen Übung gelingt die Umsetzung direkt. Aber wenn Algorithmen verzwickter werden, ist ein Struktogramm eine schöne Methode, seine Gedanken zu ordnen.

Aufgaben

Programmieren ist Übungssache. Starten Sie Ihren Computer und legen Sie los. Sollten Sie die Lösung nicht herausfinden, könnten Sie die Musterlösung herunterladen.

http://www.willemer.de/java4d

Schauen Sie nicht zu früh in die Musterlösung. Es ist nicht entscheidend, welche Lösung ich gefunden habe, sondern dass Sie eine Lösung finden. Und die darf sich gern von meiner unterscheiden. Vielleicht ist Ihre Lösung sogar besser.

Notengebung

An einer Akademie wird die Notengebung anhand einer Prozenttabelle vorgegeben.

Prozent	Note
unter 30%	6
unter 50%	5
unter 67%	4
unter 81%	3
unter 92%	2
ab 92%	1

Tabelle 4.9: Notentabelle

Schreiben Sie ein Programm `Notengebung`, das die `double`-Variable `prozent` auswertet und daraus die Note ermittelt.

Minimum per Fragezeichenoperator

Schreiben Sie ein Programm `Minimum`, das von den beiden Variablen a und b den kleineren Wert auf dem Bildschirm ausgibt.

Notendarstellung per Fallunterscheidung

Eine Akademie möchte die Noten in den Zeugnissen gern ausschreiben, also »ausreichend« statt »4«. Schreiben Sie ein Programm `Notenausgabe`, das die Note in der `int`-Variablen `note` als Text ausgibt.

Das Depot

Kapitalvermehrung ist so schön! Leider finden Banken das auch und wollen mitverdienen. Schreiben Sie doch ein Programm `Depot`, das die Kapitalentwicklung Ihres Vermögens von 2.000 Euro errechnet, wenn die Bank Ihnen 4,5% Zinsen gewährt, aber 100 Euro jährlich an Depotgebühren haben will.

1x1

In der Grundschule gibt es Tabellen für das 1x1. Erstellen Sie ein Programm `EinMalEins`, das eine solche Tabelle erstellt. Damit die Spalten hübsch untereinander stehen, sollten Sie zwischen den Zahlen ein Tab-Zeichen ausgeben. Sie erinnern sich? ‘\t’

Tiere zählen

Ein Bauer hat doppelt so viele Katzen wie Gänse und wiederum doppelt so viele Kühe wie Katzen. Er hat dann noch doppelt so viele Hühner wie Kühe. Wenn er die

Beine durchzählt, sind es 252. Wie viele Gänse hat er? Schreiben Sie ein Programm `TiereZaehlen`!

Das Schachbrett

Die Legende besagt, dass ein König dem Erfinder des Schachspiels einen Wunsch gewährte. Daraufhin wünschte sich dieser Reiskörner. Auf das erste Feld des Schachbretts sollte ein Korn kommen. Auf jedes weitere Feld sollte immer die doppelte Menge an Körnern fallen, also auf das zweite zwei und auf das dritte vier und so weiter. Bekanntlich hat das Schachbrett acht Spalten und acht Reihen. Schreiben Sie ein Programm `Schachbrett`, das die Menge an Körnern berechnet. Übrigens werden Sie für die Variable, die die Körner zählt, weder mit `int` noch mit `long` hinkommen. Verwenden Sie lieber `float` oder `double`.

Schleifende Hunde und Füchse

Eine Aufgabe aus dem Mathematikwettbewerb vierte Klasse aus dem Jahr 2002 hat in meinem Bekanntenkreis die Runde gemacht. Interessanterweise haben vor allem die Erwachsenen Probleme damit gehabt, die mathematisch vorgebildet waren. Dagegen konnten Kinder die Aufgabe relativ leicht lösen, weil sie sie iterativ angingen. Es wäre wohl die ideale Hausaufgabe, um zu erreichen, dass die Eltern endlich nicht mehr die Hausaufgaben ihrer Kinder lösen:

Ein Hund hetzt einen Fuchs. In der Zeit, in der der Fuchs neun Sprünge macht, springt der Hund sechs Mal. Aber die Sprünge des Hundes sind länger. Mit nur drei Sprüngen legt der Hund den gleichen Weg zurück wie der Fuchs mit sieben Sprüngen. Der Fuchs hat einen Vorsprung von 60 Fuchssprüngen. Nach wie vielen seiner Sprünge holt ihn der Hund ein?

Lösung

Versuchen Sie es erst selbst. Das macht mehr Spaß! Wenn Sie hängen, lesen Sie Absatz für Absatz weiter.

Wenden Sie etwas Lehrbuch-Psychologie an. Wenn der Willemer als Thema gerade Schleifen behandelt hat, wird dieses Problem ja wohl mit Schleifen zu lösen sein.

Wir bauen eine Schleife, in der der Fuchs und der Hund jeweils eine Runde weiterlaufen. Wie oft läuft die Schleife? Natürlich so lange der Fuchs nicht eingeholt wird.

Was passiert nun genau innerhalb der Schleife, also bei jedem Schritt? Der Fuchs und der Hund bewegen sich jeweils ein Stück weit nach vorn. Betrachten wir den Fuchs: Er macht jede Runde neun Sprünge. So weit so einfach. Wo der Fuchs gerade ist, merken wir uns in der Variablen posFuchs.

Nun der Hund. In der gleichen Zeit macht der Hund nur sechs Sprünge. Allerdings sind die Sprünge länger. Der Hund braucht für die gleiche Distanz drei Sprünge, auf die der Fuchs sieben Sprünge braucht. Also ist der Hundesprung 7/3 Mal so

lang wie ein Fuchsprung. Die Position des Hundes ist also 6 * 7/3. Da kann man die 6 durch die 3 im Nenner kürzen. Na prima, also bewegt sich der Hund 2*7, also 14 Sprünge pro Runde. Die Position des Hundes speichern wir in der Variablen `posHund`.

Vor der Schleife müssen wir dem Fuchs noch den Vorsprung geben. In der Variablen `posFuchs` steht also 60, in `posHund` eine 0. Also ergibt sich das folgende Programm:

```
public class FuchsHund {
    public static void main(String[] args) {
        int posFuchs = 60;
        int posHund = 0;
        while (posHund < posFuchs) {
            posFuchs = posFuchs + 9;
            posHund = posHund + 14;
        }
        System.out.println(posHund*3/7);
    }
}
```

Listing 4.26: Fuchs und Hund

Die Ausgabe soll die Zahl der Hundesprünge liefern. Das Programm rechnet aber in Fuchssprüngen. Sie müssen die Fuchssprünge noch in Hundesprünge umrechnen, also mit 3 multiplizieren und durch 7 teilen.

Falls Sie mich beim WWF anschwärzen wollen, was für blutrünstige Aufgaben ich verwende, verweise ich hiermit auf das Kultusministerium, das diese Aufgabe erfunden hat.

Kapitel 5

Funktionierende Methoden

IN DIESEM KAPITEL

Methoden machen aus mehreren Anweisungen einen Befehlsaufruf.

Daten fließen per Parameter hinein und per `return` heraus.

Nicht nur Autos, sondern auch Methoden können überladen sein.

Manche Methoden rufen sich selbst auf und nennen sich dann Rekursion.

Dieses Kapitel soll sich mit Methoden befassen. Wenn man es böse formulieren würde, tun Methoden selbst eigentlich gar nichts. Sie sammeln lediglich Anweisungen. Das Kapitel zu überschlagen wäre aber dennoch keine gute Idee. Methoden bilden für die in ihnen enthaltenen Anweisungen eine Schnittstelle und bilden so etwas wie selbst gebastelte Befehle. Übrigens haben Sie ja schon die ganze Zeit eine Methode verwendet. Die Methode heißt `main`. Fangen wir doch damit an.

Vorstellung der Methode main

Anhand der Methode `main` werden Sie nun die wichtigsten Bestandteile einer Methode kennenlernen. In Listing 5.1 werden wir also noch einmal ein vertrautes Bild sehen. Die Methode beginnt in der zweiten Zeile und endet in der vorletzten. Darum herum befindet sich die in Java obligatorische Klassendefinition.

```
public class Klasse {
    public static void main(String[] args) {
        System.out.println("Hallo!");
    }
}
```

Listing 5.1: Die Methode `main`

Eine Methode weist folgende Bestandteile auf:

- **Namen:** Unsere Methode heißt `main`.
- **Rumpf:** Zwischen den geschweiften Klammern stehen alle Befehle, die in der Methode ausgeführt werden. In diesem Fall ist das nur eine Zeile, nämlich die, die mit `System.out.println` beginnt.

- **Rückgabetyp:** Eine Methode kann ein Ergebnis liefern, beispielsweise eine ganze Zahl. Dann wäre ihr Typ `int`. Der Rückgabetyp steht vor dem Namen und ist in diesem Fall `void`. Das ist ein Sondertyp. Er sagt nämlich, dass diese Methode geizig ist und nichts zurückgibt.
- **Parameter:** Hinter dem Namen gibt es eine runde Klammer. Die enthält all das, was von außen an die Methode übergeben wurde. In unserem Fall ist das `String[] args`. Die eckigen Klammern stehen für Arrays. Aber das heben wir uns genauso auf wie `String`. Das ist ein Datentyp für Zeichenfolgen, aber auch damit amüsieren wir uns wie erwähnt später.
- **Modifizierer:** Zwei Begriffe fehlen noch. Der eine ist `public`. Die Methode ist öffentlich zugänglich. Über die Öffentlichkeitsarbeit der Methoden wird es im Zusammenhang der Klassen noch ausgiebiger gehen.

  Der andere Modifizierer heißt `static`. Statische Methoden gibt es nur ein Mal auf dieser Welt oder zumindest in einem Programm. Auch wenn ein Mal vielleicht wenig ist: Wer ein Mal da ist, ist auf jeden Fall verlässlich da. Für die Hauptmethode eines Programms ist das ungeheuer wichtig. Es muss ja einen verlässlichen Einsprungpunkt geben, den der Interpreter aufrufen kann.



Es gibt bei `static` einen Schneeballeffekt. Alle Methoden, die eine statische Methode benutzt, müssen ebenfalls statisch sein. Und da `main` wegen des Programmstarts statisch ist, wird der Modifizierer `static` ein häufiger Begleiter sein.

Methoden gliedern und sparen Arbeit

Die Methode `main` kann mehrere Zeilen umfassen, eigentlich sogar beliebig viele. Aus Sicht des Aufrufers gibt es aber nur einen Namen für die Methode, nämlich `main`.

Wir können eine neue Methode schreiben und diese von `main` aufrufen lassen. Da Methoden meist etwas tun, erhalten sie sinnvollerweise Verben als Namen und nach Konvention werden diese Verben kleingeschrieben. Schreiben wir also eine Methode `tudochwas` und die soll dann auf dem Bildschirm ausgeben: »Ich tu doch was!«

```
public class Klasse {
    public static void tudochwas() {
        // (2) (5)
        System.out.println("Ich tu doch was!");
        // (3) (6)
    }
    public static void main(String[] args) {
        // (1)
        tudochwas();
        // (4)
        tudochwas();
        // (7)
    }
}
```

Listing 5.2: Die Methode `tudochwas`

In den Kommentaren sehen Sie Zahlen in Klammern. Sie sollen andeuten, wie das Programm schrittweise ausgeführt wird.

- Das Programm startet also an Stelle (1). Von dort wird `tudochwas` aufgerufen.
- Also arbeitet sich das Programm an (2) vorbei bis zum Ende des Unterprogramms (3).
- Von dort geht es zurück hinter den Aufruf nach (4).
- Dort befindet sich der erneute Aufruf von `tudochwas`.
- Das führt dazu, dass das Programm an Stelle (5) landet, was natürlich die gleiche Position wie (3) ist. Allerdings hat sich das Programm für den Rücksprung eine andere Position gemerkt.
- Das Programm erreicht Position (6) und kehrt daraufhin zu Position (7) zurück.
- Dort endet `main` und damit das Programm.

In der Methode `main` sehen Sie, wie die Methode `tudochwas` zwei Mal aufgerufen wird. Es ist also möglich, Methoden beliebig oft aufzurufen. Die Methode `tudochwas` könnte natürlich beliebig viele Anweisungen enthalten, den Kölner Dom fertig bauen und Loch Ness auf Würmer durchsuchen. Dennoch benötigt man nur einen kurzen Aufruf, um die Methode zu starten.



Eine Methode fasst Anweisungen zu einem Block zusammen, der an beliebiger Stelle beliebig oft aufgerufen werden kann. Der Aufrufer kehrt nach der Ausführung immer direkt hinter den Aufruf zurück.

Wenn Sie also bei der Programmierung einige Anweisungen haben, die immer

wieder in dieser Reihenfolge benötigt werden, dann ist es Ihre heilige Pflicht, diese zusammen in eine Methode zu stecken und diese an den benötigten Stellen aufzurufen.


Ein wichtiges Prinzip der Programmierung lautet, dass man niemals ein und denselben Code zweimal schreiben soll. Nicht unbedingt nur, weil Programmierer faule Säcke sind, sondern vor allem deshalb, weil ein Fehler dann nur an einer Stelle korrigiert werden muss.

Andere Programmiersprachen verwenden für Methoden die Bezeichnung Funktionen. Methoden sind Funktionen, die in einer Klasse definiert sind. Und da bei Java eigentlich alles Klasse ist, sind die Funktionen auch Methoden.

Salzkartoffeln kochen für Anfänger

Ich hoffe, Sie halten nicht gerade Diät. Im nächsten Beispiel geht es nämlich um Essen. In jedem Kochbuch für Anfänger dürfte direkt hinter dem Abschnitt »Wasser kochen ohne Anbrennen« stehen, wie man Salzkartoffeln zubereitet. Ich sollte vielleicht ein Kochbuch schreiben. Das muss dann auch nicht alle paar Jahre aktualisiert werden.

Man könnte also sagen, dass das Kochbuch die Methode `kocheSalzkartoffeln` definiert:

- Füllen Sie einen Topf mit Wasser und geben Sie 8 Kartoffeln und 20 kg Salz hinzu.
- Bringen Sie das Wasser zum Kochen.
- Garen Sie die Kartoffeln so lange, bis sie sich bei der Gabelprobe weich anfühlen.

Da man zu vielen Gerichten Salzkartoffeln servieren kann, kann das Kochbuch einfach immer wieder die Methode `kocheSalzkartoffeln` aufrufen, indem man schreibt: »Dazu bereiten Sie Salzkartoffeln.«

Beim Korrekturlesen fällt mir gerade etwas auf. Wie unangenehm! Mit 20 kg Salz im Wasser entstehen wohl eher gepökelte Kartoffeln als gekochte. Das reicht bequem für eine ordentliche Salzvergiftung. Es sollte natürlich 20 g heißen.

Hätte ich in meinem Kochbuch das Salzkartoffelnkochen zu jedem Rezept kopiert, müsste ich nun diesen Fehler bei jedem Gericht korrigieren, zu dem es Salzkartoffeln gibt. So allerdings muss nur einmal kg durch g ersetzt werden.


Wer in einem Programm Quelltext kopiert, kopiert damit auch die Fehler.

Ein wenig unflexibel ist das Ganze trotzdem. Acht Kartoffeln sind für ein Candle-Light-Dinner zu viel und für eine Party zu wenig. Darum würde ich in meinem Kochbuch vielleicht schreiben, dass diese Menge für vier Personen gedacht ist. Der Leser müsste dazu immer selbst ausrechnen, wie viel Salz benötigt wird.

Der Programmierer kann da mehr! Er kann der Funktion als Parameter übergeben, wie viele Personen versorgt werden sollen. Dann kann die Funktion selbst die Mengen von Kartoffeln und Salz berechnen. Im Rezeptbuch würde das dann einem »pro Person« entsprechen, aber der Leser müsste es selbst ausrechnen. Lassen Sie uns doch eine Methode `kocheSalzkartoffeln` in Java schreiben:

```
public class SalzkartoffelTest {
    static void kocheSalzkartoffeln(int personen) {
        int salz = personen * 5;
        int kartoffeln = personen * 2;
        System.out.print("Ein Topf voller Wasser, ");
        System.out.print(salz);
        System.out.print(" g Salz und ");
        System.out.print(kartoffeln);
        System.out.println(" Kartoffeln dazu.");
    }
    static void brateFrikadellen(int personen) {
        // füllen Sie diese Methode doch selbst!
    }
    public static void main(String[] args) {
        System.out.println("Kartoffeln mit Frikadellen:");
        int personen=4;
        kocheSalzkartoffeln(personen);
        brateFrikadellen(personen);
    }
}
```

Listing 5.3: Rezeptprogramm mit Salzkartoffelmethode

Sie sehen im Programm `SalzkartoffelTest` nun drei Methoden: `kocheSalzkartoffeln`, `brateFrikadellen` und `main`. Die Reihenfolge der Methoden ist dem Compiler völlig egal. In unserem Beispiel ruft die Methode `main` zuerst `kocheSalzkartoffeln` auf und übermittelt ihr, dass vier Personen zum Essen kommen. Es werden die Anweisungen in `kocheSalzkartoffeln` ausgeführt. Sobald das Programm damit fertig ist, kehrt es zum Aufrufort zurück und bearbeitet die nächste Anweisung, die hier ein Aufruf von `brateFrikadellen` ist. Auch hier werden nun die Anweisungen der Methode ausgeführt und das Programm kehrt zurück. Methoden können auch ihrerseits Methoden aufrufen. `System.out.println` ist nämlich ebenfalls eine Methode und Sie sehen ja, dass sie auch von

`kocheSalzkartoffeln` aufgerufen werden kann.

Methodenkopf

Eine Methode wird immer durch einen Methodenkopf eingeleitet. Die Modifizierer `public` und `static` kennen Sie schon. Beide ignorieren wir auch weiterhin.

Es folgt noch so ein Wort, das Sie zwar von `main` kennen, aber noch nie richtig erklärt bekamen: `void`! Dazu müssen Sie wissen, dass Methoden in Java oft einen Wert zurückgeben. Der Datentyp dieses Werts wird hier im Methodenkopf festgelegt. Nun geben aber einige Methoden tatsächlich überhaupt nichts zurück und dann gibt man als Rückgabetyp eben `void` an, denn *void* heißt *nix gibt's*. Weder die Methode `main` noch die Methode `kocheSalzkartoffeln` liefern irgendetwas an die wartende Menschheit zurück. Sie tun ihren Dienst, aber liefern kein abschließendes Ergebnis. In den nächsten Abschnitten zeige ich Ihnen mal ein paar ordentliche Methoden mit richtigen Rückgabewerten.

Bei der Namensgebung gilt, dass eine Methode typischerweise eine Tätigkeit ist. Also verwendet man möglichst ein Verb, und auch Methodennamen sollten mit einem Kleinbuchstaben beginnen.

Parameterklammern

Über den Namen `kocheSalzkartoffeln` kann die Methode später aufgerufen werden. Was dahinter zwischen den runden Klammern steckt, sind Parameter. Sie schleusen Daten von außen in die Methode. Es kann auch Methoden ohne Parameter geben. Bei denen müssen die Klammern dann leer sein. Weglassen dürfen Sie sie nicht.

Methodenrumpf

Die geschweifte Klammer geht auf und damit beginnt der Methodenrumpf, in dem beliebig viele Programmzeilen folgen können, die alle zur Methode gehören. Das geht so lange, bis die geschweifte Klammer schließt.

 Eine Methode wird nach dem folgenden Muster definiert.

```
Modifizierer Rückgabetyp Methodenname(Parameter) {
    Anweisung1;
    Anweisung2;
}
```

Methoden mit Datenfluss

Eine Methode ähnelt in ihrem Datenfluss ein wenig einem Fleischwolf. Oben gibt es den Einfüllstutzen, in den Fleischstücke und Gewürze hineinkommen – dieser entspricht den Parametern. Und unten an der Seite gibt es den Auslass, aus dem das Hackfleisch oder die Wurst herauskommt. Das entspricht dem Rückgabewert, der

mit dem Befehl `return` zurückgegeben wird.

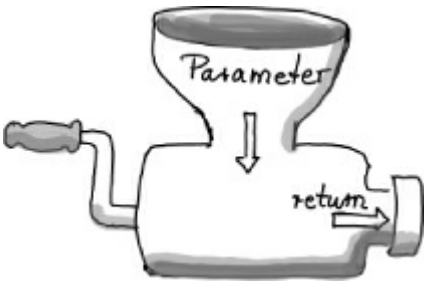


Abbildung 5.1: Aus Parametern wird der Rückgabewert.

Das folgende Programm soll die Kosten einer rauschenden Party ermitteln. Das Rauschen kostet nichts, aber die Getränke schon. Da es davon reichlich geben soll, gibt es Getränke nur im Kasten.

Damit der Kastenpreis nicht für jedes Getränk einzeln berechnet werden muss, gibt es eine Methode `kastenPreis`, die über ihre Parameter die Preise auch verschiedenster Getränke berechnet.

```
public class ColaKasten {
    static double kastenPreis(int flaschenAnzahl,
                              double flaschenPreis,
                              double flaschenPfand,
                              double kastenPfand) {
        double preis = 0.0;
        preis = flaschenAnzahl * (flaschenPreis + flaschenPfand)
                               + kastenPfand;
        return preis;
    }
    public static void main(String[] args) {
        double colaPreis = 0.0, bierPreis=0.0;
        colaPreis = kastenPreis(12, .5, .2, 2.0);
        bierPreis = kastenPreis(20, .4, .2, 2.0);
        System.out.println(colaPreis);
        System.out.println(bierPreis);
    }
}
```

Listing 5.4: Getränkekasten

Schauen wir uns die Methode `kastenPreis` etwas genauer an.

- Wie gehabt beginnt sie mit dem Modifizierer `static`. Der ist notwendig, weil sie von der statischen Methode `main` aufgerufen wird.
- Der Rückgabewert ist dieses Mal `double` anstatt `void`. Die Methode soll ja schließlich die Kosten zurückgeben.
- Die Parameter sind der Übersicht halber über mehrere Zeilen verteilt.

Nebeneinander hätten sie eh nicht auf die Buchseite gepasst. Die Parameternamen können innerhalb der Methode wie lokale Variablen behandelt werden, die von außen vorbesetzt wurden.

- Die Variable `preis` ist lokal für diese Methode deklariert. Von außen ist sie unsichtbar. Sobald der Block der Methode endet, wird auch die Variable verschwinden.
- Erst durch den Befehl `return` wird sie über den Rückgabewert nach außen kopiert. Darum muss sie vom Typ `double` sein, damit sie dem Rückgabetyp entspricht.

Da wir bei diesem Beispiel zum ersten Mal mehrere Parameter haben, hier noch der Hinweis, dass die Parameter beim Aufruf der Methode in der Reihenfolge übergeben werden müssen, wie sie im Methodenkopf definiert sind.

Die Parameter zwischen den Klammern sehen genauso aus wie Variablendefinitionen. Und das ist nicht aus Versehen so. Die Parameter sind Variablen, die zur Methode gehören, die aber vom Aufrufer initialisiert werden.

Was passiert nun, wenn Sie die Variablen innerhalb der Methode verändern? Werden dann die Werte beim Aufrufer auch verändert? Keine Sorge! Die übergebenen Daten werden in diese Variablen kopiert.


Beim Aufruf werden die übergebenen Werte in die Parameter kopiert. Die Deklaration zeigt dem Aufrufer, welchen Typ die Methode erwartet.

Die Parameterdeklaration ist aber nicht nur ein zarter Hinweis an den Aufrufer. Der Compiler prüft, ob der Parameter des Aufrufs zu den Parametern der Methode passt. Und falls die Parameter einen falschen Typ verwenden oder die Anzahl nicht stimmt, dann weigert sich der Compiler, das Programm zu übersetzen.

Rückgabe eines Wertes

Der Befehl `return` gibt nicht nur einen Wert an den Aufrufer zurück, sondern beendet sofort den Ablauf der Methode. Darum steht der Befehl auch typischerweise am Ende einer Methode.

Anweisungen, die hinter dem Befehl `return` stehen, sind unerreichbar. Wenn der Compiler das merkt, meckert er, weil er davon ausgeht, dass der Programmierer da etwas übersehen hat.

Hinter Abfragen können Sie die Methode per `return` verlassen, ohne dass der Compiler einschreitet. Es deutet aber darauf hin, dass in Ihrem Ablauf etwas unschön ist, ähnlich wie beim Einsatz von `break` in einer Schleife.

Ist der Methodentyp `void`, kann die Methode keinen Wert zurückgeben. Dennoch kann der Befehl `return` auftreten, allerdings dann ohne Wertangabe. Er beendet die

Methode und kehrt zum Aufrufer zurück.

Hat die Methode einen anderer Typ als `void`, verlangt der Compiler auch eine `return`-Anweisung mit der Angabe eines Wertes, der bitteschön auch gern mit dem Methodentyp kompatibel sein soll.

Aufruf von Methoden

Der Aufruf einer Methode mit Rückgabewert kann genauso als Ausdruck in eine Berechnung eingebaut werden wie eine Konstante oder eine Variable. So berechnet die folgende Zeile die Kosten Ihrer Feier durch Aufruf der eben gezeigten Methode `kastenPreis` und schlägt noch einmal die Miete für einen Saal oben drauf.

```
kostenFeier = kastenPreis(12, .5, .2, 2.0) + SAALMIETE;
```

Der Rückgabewert einer Methode muss vom Aufrufer nicht zwingend ausgewertet werden. Eine Methode mit einem Rückgabetyp kann aufgerufen werden wie eine void-Methode.

Überladen von Methoden

Für die Bildschirmanzeige von Variablen und Texten haben wir bisher immer wieder gern die Methode `System.out.println` verwendet. Diese Methode scheint ja offenbar alles an Parametern zu schlucken, was die Sprache Java so hergibt. Das ist ungewöhnlich, denn gerade erst hat der Willemer behauptet, der Parameter einer Methode müsse von einem festgelegten Typ sein! Der Trick ist, dass es nicht nur eine Methode `println` gibt, sondern mehrere. Jede von ihnen bedient einen anderen Parametertyp. Das können Sie in Ihren Programmen auch tun.

Java lässt es nämlich zu, dass mehrere Methoden den gleichen Namen tragen, sofern sie unterschiedliche Parameter haben. Man spricht davon, dass diese Methoden gleichen Namens sich *überladen*. Wenn so eine überladene Methode im Programm aufgerufen wird, sucht der Compiler die passende Version für den jeweiligen Datentyp selbst heraus.



Weil erst die Kombination aus Methodenname und Parametern eine Methode eindeutig machen, spricht man auch von der Signatur. Der Rückgabewert ist nicht signifikant. Er wird also bei der Prüfung, welche der überladenen Methoden verwendet werden soll, nicht herangezogen.

Überladene Experimente

Das Programm `Ueberladung` wird mehrere Methoden mit dem Namen `bestimmeParameter` anbieten, die jeweils einen anderen Parametertyp haben. Dann rufen wir die Methode von `main` aus mehrfach mit unterschiedlichen Parametern auf und schauen, ob der Compiler gute Arbeit leistet.

```
public class Ueberladung {
    static void bestimmeParameter() {
        System.out.println("keine Parameter");
    }
    static void bestimmeParameter(int a) {
        System.out.println("Parameter int");
    }
    static void bestimmeParameter(char c) {
        System.out.println("Parameter char");
    }
    static void bestimmeParameter(double d) {
        System.out.println("Parameter double");
    }
    public static void main(String[] args) {
        bestimmeParameter(12);    // Aufruf mit int-Literal
        bestimmeParameter('c');   // Aufruf mit char-Literal
        bestimmeParameter(12.4);  // Aufruf mit double-Literal
        bestimmeParameter();      // Aufruf ohne Parameter
        bestimmeParameter(12.4f); // Aufruf mit float-Literal
        bestimmeParameter(12L);   // Aufruf mit long-Literal
    }
}
```

Listing 5.5: überladen

Das Programm wird die folgenden Zeilen auf dem Bildschirm ausgeben:

```
Parameter int
Parameter char
Parameter double
keine Parameter
Parameter double
Parameter double
```

Die ersten vier Zeilen hätten Sie vermutlich auch ausgegeben, wenn Sie das Programm wären. Sie sind also keine Überraschung.

Bei der fünften Zeile wird ein `float`-Parameter angegeben. Da es keine Methode für `float` gibt, verwendet Java die Methode mit dem `double`-Parameter. Mit einem anderen Parameter wäre die Nachkommastelle nicht zu retten gewesen. Sie wissen ja, dass der Datentyp `float` einen kleineren Wertebereich als der Datentyp `double` hat, weswegen Werte vom Typ `float` problemlos von Variablen des Typs `double` aufgenommen werden können.

Der letzte Aufruf übergibt eine Zahl vom Typ `long`. Spontan hätten Sie vielleicht erwartet, dass die `int`-Methode diesen Fall übernehmen würde, das geht aber nicht, weil `int` einen kleineren Wertebereich als `long` hat, was zu Problemen führen würde. Der einzige Datentyp, der in diesem Fall einen größeren Wertebereich hat,

ist `double`, weswegen die `double`-Methode auch diese Anforderung übernimmt.

Was passiert, wenn Sie die Methode mit dem Parameter `char` entfernen? Dann müsste ja eigentlich der Aufruf `bestimmeParameter('c')` nicht funktionieren und zu schweren Compiler-Protesten führen. Probieren Sie es aus!

Übersetzen funktioniert trotzdem. Aufrufen funktioniert auch, aber es meldet sich die Methode mit dem Parameter `int`, da ein Typ `char` in einen `int` passt, dort aber dann als ganze Zahl behandelt wird.

Und jetzt klauen Sie doch bitte mal dem Programm die Methode mit dem Parameter `double`! Wie Sie schon vermuten, wird nun der Compiler furchtbar husten, weil er weder den Aufruf mit `float`- noch mit `double`-Parametern einer Methode zuordnen kann.



Wenn der Aufrufer keinen exakt passenden Parameter sucht, schaut Java, ob es eine Methode mit einem kompatiblen Parameter gibt, und verwendet dann diese.

Rekursionen

Sie können Methoden von unterschiedlichen Methoden aus aufrufen. Aber was passiert, wenn eine Methode sich selbst aufruft? Das würde eine Art Endlosschleife ergeben, sofern man nicht darauf achtet, dass der Selbstaufruf auch irgendwann stoppt. Tatsächlich gibt es so etwas und man spricht in diesem Fall von einer *Rekursion*.

Eine Methode, die sich selbst aufruft, leidet keineswegs an Narzissmus. Es gibt für solche Selbstaufrufe tatsächlich sinnvolle Anwendungsfälle. Tatsächlich ist eine Rekursion eine Art Schleife. Im Gegensatz zu einer Schleife kann sich die rekursive Methode immer wieder neue Parameter übergeben. Besonders leistungsfähig werden Rekursionen, wenn es im Methodenkörper mehrere Selbstaufrufe gibt. Der Nachteil: An diese Art des Denkens muss man sich erst gewöhnen.

Zunächst zeige ich Ihnen an einem sehr einfachen Beispiel, wie ein Selbstaufruf funktioniert. Wir programmieren einen Countdown:

```
public class RekurCountDown {
    static void countDown(int n) {
        if (n>0) { // Dies verhindert eine Endlos-Rekursion.
            System.out.println(n);
            countDown(--n); // Hier ist der Selbstaufruf.
        }
    }
    public static void main(String[] args) {
        countDown(12);
    }
}
```

Listing 5.6: Wir zählen auf die Rekursion.

Die Methode `countDown` erhält als Parameter den Wert, den sie herunterzählen soll.

1. Die Methode `countDown` sichert zunächst ab, dass die Rekursion irgendwann einmal endet. Das tut sie, indem sie nur tätig wird, wenn der Parameter größer als 0 ist. Ist der Parameter 0, wird die Methode sofort enden und zum letzten Aufruf zurückspringen.
2. Der Parameterwert wird ausgegeben.
3. Der um 1 verminderte Parameterwert wird in einem Selbstaufruf an `countDown` übergeben. Dort wird er wiederum auf 0 überprüft ausgegeben und das Spiel geht weiter.
4. Der Rücksprung erfolgt immer zu der Position, wo der Parameter noch um 1 höher war. Erst beim letzten Rücksprung landet das Programm in `main`. In all den Rücksprüngen wird aber kein Code mehr durchlaufen.

Manchmal habe ich die Gabe des Gedankenlesens. Und ich bin sicher, dass Sie gerade denken: »Das programmiere ich aber schneller in einer Schleife.« Und soll ich Ihnen etwas sagen: Sie haben recht! Und nicht nur das! Die Schleife wäre nicht nur schneller programmiert, sie würde den Countdown auch schneller ausführen als die Rekursion.

Ich will gar nicht behaupten, dass alle Probleme, die per Rekusion lösbar sind, auch so gelöst werden sollten. Aber bestimmte Probleme sind eben rekursiv einfacher zu lösen. Ein typisches Beispiel sind die Türme von Hanoi. Ich bin sicher, Sie haben erst letztlich Ihrem Nachbarn erzählt, dass Sie zu gern wüssten, wie man die Türme von Hanoi mit einem Java-Programm lösen kann, richtig?

Die Türme von Hanoi

Ach, Sie kennen die Türme von Hanoi gar nicht? Na, vielleicht nicht unter diesem Namen. Das Spiel besteht aus drei Stäben. Auf einem Stab liegen mehrere, nach oben immer kleiner werdende Scheiben, meist aus Holz. Weil es so kompliziert ist,

habe ich eine Zeichnung angefertigt.

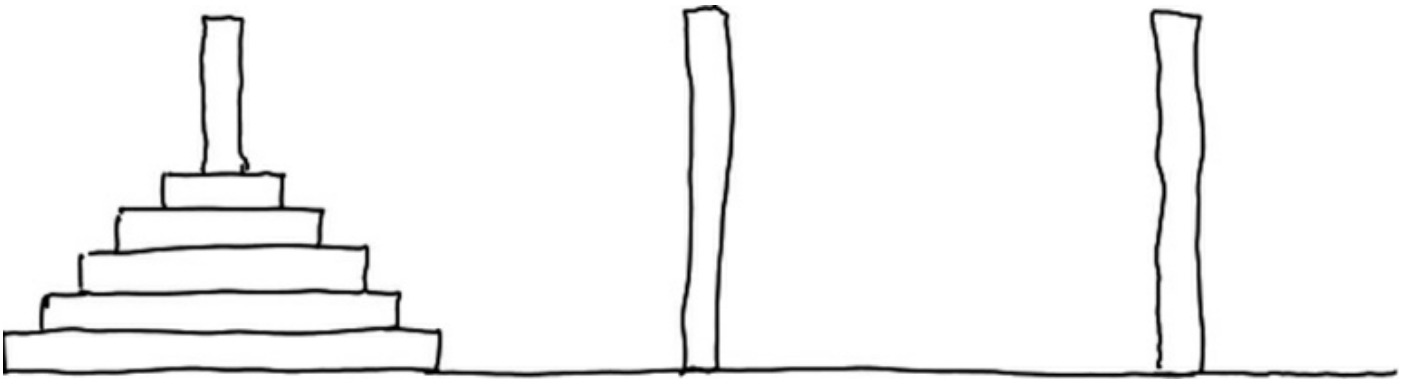


Abbildung 5.2: Türme von Hanoi

Das Ziel ist es, die Scheiben von einem auf einen der anderen Stäbe zu bekommen. Dabei darf immer nur eine Scheibe bewegt werden und es darf nie eine größere Scheibe auf einer kleineren liegen.

Der Grundgedanke der Lösung basiert darauf, dass die Rekursion die Arbeit vor sich herschiebt, bis sie plötzlich nicht mehr da ist. Damit werden Rekursionen doch gleich sehr sympathisch, nicht wahr?

- ✔ Wir schreiben eine Methode `ziehe`, die n Scheiben von einem Stab mit der Nummer `von` auf einen anderen Stab mit der Nummer `nach` verschiebt. Soll `ziehe` gefälligst zusehen, wie sie die Scheiben umsetzen kann.
- ✔ Die Methode `ziehe` ruft nun sich selbst auf und gibt in den Parametern an, dass n-1 Scheiben auf den freien Stab zu schieben sind.
- ✔ Sind die n-1 Scheiben verschoben, kann sie die unterste Scheibe direkt auf den Zielstab legen. Das geht einfach. Es ist nur eine Scheibe und der Zielstab ist frei.
- ✔ Danach kann die Methode `ziehe` wieder sich selbst aufrufen und per Parameter anweisen, dass die n-1 in der Mitte geparkten Scheiben ganz nach rechts zu verschieben sind.

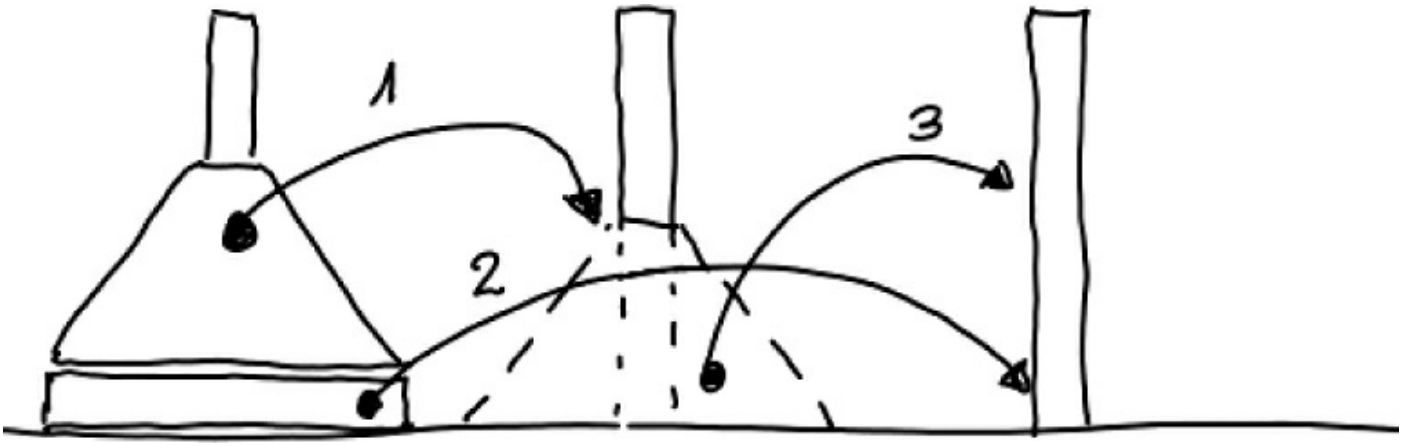


Abbildung 5.3: Lösung für die Türme von Hanoi

Halt, halt! Wie bekommen wir denn den Stapel mit den n-1 Scheiben zunächst von links zur Mitte und dann von dort nach rechts? Na, genauso! Betrachten wir das Verschieben vom linken Stab zur Mitte. Wir schieben nun n-2 Scheiben auf den rechten Stab. Wieder notieren wir, dass wir die Scheibe n-1 nun auf den mittleren Stab legen. Dann wird der Stapel mit den n-2 Scheiben vom rechten Stab wieder in die Mitte geholt. Die rekursive Methode hat also etwa folgenden Ablauf:

```
static void ziehe(int von, int nach, int scheibenZahl) {
    if (scheibenZahl>0) {
        int frei = berechneFreienStab(von, nach);
        ziehe(von, frei, scheibenZahl-1);
        zeigeZug(von, nach);
        ziehe(frei, nach, scheibenZahl-1);
    }
}
```

Listing 5.7: Prinzip der rekursiven Methode

Wie wir den freien Stab ermitteln, heben wir uns für später auf. Wir werden dafür eine Methode schreiben. Die Anzeige des Zuges erfolgt durch die Stabangaben. Wenn Sie einen Roboter irgendwo herumstehen haben, könnten Sie den natürlich auch ansteuern. Erst einmal gibt die Methode einfach nur den Stab aus, von dem die Scheibe genommen wird, und den Stab, auf dem sie landen soll.

Spielen wir erst einmal die rekursive Methode für drei Scheiben durch, die von links nach rechts wandern sollen. Die `main`-Methode hat `ziehe` aufgerufen mit `links`, `rechts` und 3 als Parameter. Nun schauen wir an, welche Anweisungen ausgeführt werden. Der Übersicht halber sind diese bei jedem rekursiven Aufruf etwas eingerückt.

```
ziehe(links, mitte, 2)
  ziehe(links, rechts, 1)
    ziehe(links, mitte, 0) // Ende der Rekursion: Rücksprung!
    print(links - rechts);
    ziehe(mitte, rechts, 1)
    // Ende der Methode: Rücksprung!
  print(links - mitte)
  ziehe(rechts, mitte, 1)
    ziehe(rechts, links, 0) // Ende der Rekursion: Rücksprung!
    print(rechts - mitte);
    ziehe(mitte, rechts, 1)
    // Ende der Methode: Rücksprung!
  // Ende der Methode: Rücksprung!
print(links - rechts)
ziehe(mitte, rechts, 2)
  ziehe(mitte, links, 1)
    ziehe(mitte, rechts, 0) // Ende der Rekursion: Rücksprung!
    print(mitte - links)
    ziehe(rechts, links, 0) // Ende der Rekursion: Rücksprung!
    // Ende der Methode: Rücksprung!
  print(mitte - rechts);
  ziehe(links, rechts, 1)
    ziehe(links, mitte, 0) // Ende der Rekursion: Rücksprung!
    print(links - rechts)
```

```
    ziehe(mitte, rechts, 0) // Ende der Rekursion: Rücksprung!
    // Ende der Methode: Rücksprung!
  // Ende der Methode: Rücksprung!
// Ende der Methode: Rücksprung!
```

Sie sehen auch, wie schnell eine Rekursion immer weiter um sich greift. Mit zwei Scheiben hätten Sie es vielleicht noch im Kopf hinbekommen. Nun filtern wir die `print`-Anweisungen in ihrer Reihenfolge aus:

```
    print(links - rechts);
  print(links - mitte)
    print(rechts - mitte);
print(links - rechts)
    print(mitte - links)
  print(mitte - rechts);
    print(links - rechts)
```

Nun suchen Sie drei unterschiedlich große Bremsscheiben, Teller oder Legosteine und folgen Sie den Anweisungen von oben nach unten. Sie sehen, dass es funktioniert.

Wenn Ihnen noch nicht klar geworden ist, wie diese Technik, die Arbeit vor sich herzuschieben, funktioniert, spielen Sie es noch einmal mit zwei Scheiben durch oder lesen Sie den Abschnitt vielleicht noch einmal von vorn. Die Rekursion ist sehr mächtig, aber nicht auf Anhieb verständlich.

Der freie Stab

Ein Problem haben wir allerdings noch nicht gelöst. Wie markieren wir die Stäbe und wie findet die Methode heraus, welches der freie Stab ist? Die Markierung der Stäbe ist trivial. Wir nummerieren sie einfach durch: 1, 2, 3.

Nun muss die Methode aus den beiden Stab-Parametern berechnen, welches der freie Stab ist. Wir könnten ja ein paar `if`-Befehle setzen. Es gibt sechs denkbare Kombinationen.

```
if (von==1 && nach==2)
    frei = 3;
if (von==1 && nach==3)
    frei = 2;
if (von==2 && nach==3)
    frei = 1;
if (von==3 && nach==1)
    frei = 2;
if (von==3 && nach==2)
    frei = 1;
if (von==2 && nach==1)
    frei = 3;
```

Listing 5.8: Ausführliches Stäbchensuchen ...

Schneller geht es, wenn man einfach die beiden Stabnummern `von` und `nach`

addiert. Addiert man alle drei Stäbe (1, 2 und 3), ergibt sich immer 6. Also ergibt sich der dritte Stab, wenn ich die Summe der beiden anderen Stäbe von 6 abziehe.

```
int frei = 6 - (von + nach);
```

Listing 5.9: ... und die Kurzversion

Das fertige Programm

Nun müssen wir diese Zeile noch in unsere Methode einbauen und damit ist das Programm fertig. Lediglich die Ausgabe sollten wir noch etwas schön gestalten.

```
public class TuermeVonHanoi {
    public static void main(String[] args) {
        ziehe(1, 3, 3);
    }

    static void ziehe(int von, int nach, int scheibenZahl) {
        if (scheibenZahl>0) {
            int frei = 6 - (von + nach);
            ziehe(von, frei, scheibenZahl-1);
            zeigeZug(von, nach);
            ziehe(frei, nach, scheibenZahl-1);
        }
    }

    private static void zeigeZug(int von, int nach) {
        zeigeStab(von);
        System.out.print(" - ");
        zeigeStab(nach);
        System.out.println("");
    }

    static void zeigeStab(int stabnr) {
        switch(stabnr) {
            case 1: System.out.print("links "); break;
            case 2: System.out.print("mitte "); break;
            case 3: System.out.print("rechts"); break;
        }
    }
}
```

Listing 5.10: Türme von Hanoi rekursiv gelöst

Sie können die Scheibenzahl beliebig erhöhen. Allerdings steigt dann auch die Anzahl der Ausgaben erheblich. Nun holen Sie sich die Türme von Hanoi aus dem

Kinderzimmer und spielen Sie die Anweisungen nach. Ich wünsche einen angenehmen Abend!

Aufgaben

Nun sind Sie dran! Hier finden Sie zwei kleine Übungen, bei denen Sie überprüfen können, ob Sie die Methoden verstanden haben.

Der Sektkelch

Gibt es etwas zu feiern? Gut! Der Computer kann ja mitfeiern, indem er einen Sektkelch darstellt. Zum Zeichnen nehmen wir Sterne für die Farbe Schwarz und das Leerzeichen für Weiß. Das Programm soll `Sektkelch` heißen.

```
*********
 *******
  *****
   ***
    *
    *
    *
    *
*********
```

Die Öffnung des Glases und der Fuß haben einen Radius, der exakt der Länge des Stiels entspricht. Damit allerdings der Stiel schön in der Mitte ist, nehmen wir noch eine 1 hinzu. Im Beispiel oben beträgt die Länge des Stiels 4, die Breite des Fußes und der Öffnung ist 9.

Die Methode `main` ruft die Methode `zeichneSektkelch` mit dem Parameter für die Stiellänge auf. Diese ruft die drei Methoden `zeichneKelch`, `zeichneStiel` und `zeichneFuss`. Diese drei Methoden verwenden die Methoden `zeichneLeer` und `zeichneStern` um eine bestimmte Zahl von Leerzeichen und Sternen zu zeichnen. Alternativ können Sie diese beiden Methoden in einer Methode `zeichneChar` zusammenfassen, die dann als weiteren Parameter den Buchstaben erhält, den sie zeichnen kann.

Noch einmal mit ggT

Nachdem wir den größten gemeinsamen Teiler im vorigen Kapitel einmal kennengelernt haben, wollen wir gar nicht mehr ohne ihn leben. Nun wollen wir ihn nicht nur berechnen. Nein, wir wollen ihm eine eigene Methode widmen. Und wenn ich hier »wir« schreibe, meine ich natürlich Sie!

Schauen Sie zuerst auf den Datenfluss, also Parameter und Rückgabewert. Die Methode soll aus zwei Eingangswerten den größten gemeinsamen Teiler errechnen. Es liegt in der Natur der Sache, dass die verwendeten Werte ganzzahlige Werte sind. Damit sieht der Rahmen der Methode... Ach was! Das können Sie selbst. Und den Algorithmus kennen Sie ja noch aus dem letzten Kapitel.

Kapitel 6

Die Strings: Viele, viele Buchstaben

IN DIESEM KAPITEL

String: Wenn es viel Text sein soll.

Zahlen werden Strings. Strings werden Zahlen.

Veränderbare Strings im `StringBuilder`.

Endlich Klarheit über das String-Array `args` in `main`.

Nichts prägt das menschliche Miteinander so sehr wie die menschliche Sprache. Man kann so schön ausdrücken, was man voneinander hält. Das führt allerdings zugegebenermaßen nicht immer zu Frieden auf Erden. Sprache besteht aus Wörtern, die ihrerseits eine Aneinanderreihung von Buchstaben sind. Und natürlich will Java auch mitmischen und Sprache speichern.

Variablen und Texte

Java stellt für Zeichenfolgen einen eigenen Datentyp namens `String` zur Verfügung. Ein String ist kein primitiver Typ. Das leuchtet ein, wenn er so etwas Wertvolles wie die menschliche Sprache aufnimmt. Aber der Hauptgrund dafür ist, dass man vorher nicht weiß, ob eine String-Variable nur einen kurzen Fluch oder die Verordnung für die Hygienevorschriften landwirtschaftlicher Fahrzeuge aufnehmen soll.

Für den Typ `String` gibt es Literale, also schlicht Wörter und Sätze, wie zum Beispiel »Hallo Welt!«. Das überrascht Sie jetzt nicht besonders, weil Sie solche Zeichenkettenliterale ja schon seit Ihren ersten Programmen verwenden. Sie stehen immer in Anführungszeichen. Dabei müssen Sie immer die Anfühgungszeichen über der Taste 2 verwenden. Alle anderen Arten von Anführungszeichen betrachtet Java nur als merkwürdige Gebilde und stolpert darüber.

Sie können ein solches Literal einer String-Variablen einfach zuweisen.

```
String ausgabe = "Mein Name ist Bond.";
```

Referenzen statt Kopien

Strings können sehr unterschiedlich groß sein, aber fast immer sind sie größer als die primitiven Typen, die Sie bislang kennen. Damit nicht unnötig Daten durch den eh schon verschwitzten Speicher gescheucht werden müssen, hat sich Java etwas ausgedacht: Der String wird beim Zuweisen nicht kopiert, sondern nur eine Referenz gebildet. So können mehrere String-Variablen auf denselben String zeigen. Die Hintergrundüberlegung geht dahin, dass die meisten Texte, wenn sie einmal erstellt wurden, danach keine große Änderung mehr erfahren. Die Verordnung für die Hygienevorschriften landwirtschaftlicher Fahrzeuge wird so selten geändert, dass man damit rechnen muss, dass der Computer zwischendurch mehrfach neu gestartet werden wird.

Wenn Sie eine String-Variable einer anderen String-Variablen zuweisen, entsteht keine Kopie, sondern die Zielvariable erhält eine Referenz auf das Original. Das spart natürlich Speicherplatz, wenn sich darin besagte Verordnung befindet. Denn warum sollte man einen solchen Unfug zweimal speichern?

Leer oder null?

Eine String-Variable kann auf verschiedene Weisen nichts enthalten.

- Wenn eine String-Variable eine leere Zeichenkette enthält, wird dies durch zwei direkt aufeinanderfolgende Anführungszeichen dargestellt, die sich quasi ganz eng aneinanderkuscheln. Die Variable enthält dann eine Referenz auf einen String, der selbst aber leer ist.
- Eine String-Variable kann aber auch gar keine Referenz besitzen. Dann ist sie mit `null` besetzt. Eine solche Variable ist eine derartige Null, dass sie nicht einmal feststellen kann, ob sie einen leeren String enthält, ohne dass das Programm ins Trudeln gerät.



Bevor Sie irgendetwas mit einer String-Variablen tun, sollten Sie erst einmal prüfen, ob sie vielleicht `null` ist. Ansonsten könnte Ihr Programm schneller beendet sein als geplant.

```
String leererString = ""; // Leer, aber ein String!
String nullString = null; // Nicht zur Weiterarbeit empfohlen
```

Texteingabe

Wenn der Anwender Texte eingeben soll, können Sie wieder den Scanner verwenden, den Sie schon bei der Eingabe von Zahlen verwendet haben. Der Scanner besitzt eine Methode `nextLine`, die – wie schon der Name vermuten lässt – eine Zeile von der Tastatur einliest und diese als String zurückgibt.

```
import java.util.Scanner;

public class StringEingabe {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String string = scanner.nextLine();
        System.out.println(string);
    }
}
```

Listing 6.1: Einlesen eines Strings von der Tastatur

Textmanipulationen

Sie können mit Strings nicht rechnen. Sie können sie aber zusammenfügen, umwandeln, durchsuchen und andere Dinge damit anstellen. Allerdings können Sie sie nicht verändern. Das hört sich widersinnig an? Keineswegs! Denn durch die genannten Operationen verändern Sie zwar nicht die Ausgangs-Strings. Stattdessen werden neue Strings gebildet.



Aber was passiert mit den unveränderlichen Original-Strings, die nicht mehr gebraucht werden? Java-Programmierer sind da ein klein wenig Umweltschweine. Sie lassen die Strings einfach herumliegen. Irgendwann kommt die Java-Müllabfuhr und schafft all die Strings weg, die niemand mehr braucht. Die Java-Müllabfuhr gibt es tatsächlich. Damit es nicht so komisch klingt, verwendet man allerdings den englischen Namen: *Garbage Collection*.

Texte verbinden

Der Operator +, der Zahlenwerte so hübsch summiert, kann bei Strings verwendet werden, um sie miteinander zu verbinden. Ich zeige Ihnen das in einem kleinen Beispiel:

```
class DarfIchMichVorstellen {
    public static void main(String[] args) {
        String ausgabe = null;
        // Das Pluszeichen fügt Strings zusammen.
        ausgabe = "Mein Name ist Bond." + " James Bond.";
        System.out.println(ausgabe);
    }
}
```

Listing 6.2: Bond, die Doppel-Null

Als Ergebnis landet in der Variablen `ausgabe` der berühmte Spruch: »Mein Name

ist Bond. James Bond.«

Sie können mit dem Pluszeichen nicht nur Strings, sondern auch einzelne Zeichen an einen String anhängen.

```
String string = "Man nennt mich " + "00" + '7';
```

Selbst Zahlen können Sie anhängen. Das ist sehr nützlich, wenn Sie Zahlen in Ausgaben einbetten wollen.

```
int zahl = 17 + 4;
String string = "Drei Wochen haben" + zahl + " Tage.";
```

Das funktioniert. Auch wenn die Art der Berechnung etwas fragwürdig ist.

Strings haben so ihre Methoden

Im Gegensatz zu primitiven Typen haben Strings Methoden. Diese Methoden können Sie auf eine String-Variable anwenden. Um das deutlich zu machen, wird an die String-Variable ein Punkt angehängt und dann wird der Methodenaufruf mit seinen Parametern angehängt.

```
String ausgabe = "Mein Name ist Bond. James Bond.";
char zeichen = ausgabe.charAt(1);
```

Hier wird die String-Methode `charAt` auf den String `ausgabe` angewandt. Die Methode fischt Ihnen aus einem String ein einzelnes Zeichen heraus. Der Parameter von `charAt` ist ein ganzzahliger Wert, der die Position des gesuchten Buchstabens übergibt. Hier müssen Sie beachten, dass die String-Position immer mit 0 anfängt. Die Methode liefert also das Zeichen ‘e’.


Die Positionen in einem String beginnen bei 0, nicht bei 1.

Neben `charAt` gibt es noch ein ganzes Rudel Methoden, die die Verwendung der Strings erst richtig lecker machen. Damit wir die alle ausprobieren können, werden wir hier erst einmal einen String definieren, mit dem wir ein wenig Schabernack treiben können.



Abbildung 6.1: James Bond in seiner Rolle als String

Wir legen diesen in die Variable `ausgabe`, mit der wir dann spielen können:

```
String ausgabe = "Mein Name ist Bond. James Bond.";
```


Eine Methode kann aber nur auf eine Variable angewandt werden, die nicht `null` enthält. Sollte das passieren, bricht das Programm an dieser Stelle zusammen. Das wollen Sie aber nicht.

Sobald die Möglichkeit besteht, dass eine String-Variable den Wert `null` enthalten könnte, ist es kein dummer Gedanke, vor dem Aufruf einer Methode dies zu prüfen:

```
if (ausgabe != null) {
    zeichen = ausgabe.charAt(1);
} else {
    zeichen = ' ';
}
```

Listing 6.3: Prüfung, ob die String-Variable auf `null` steht

Zusammengesetzt mit Plus und concat

Statt mit dem Pluszeichen können Sie auch durch den Aufruf der Methode `concat` zwei Strings verbinden. Mit dem Pluszeichen kennen Sie es ja schon:

```
ausgabe = "Mein Name ist Bond." + " James Bond.";
```

Sie können aber auch die Methode `concat` aufrufen:

```
ausgabe = "Mein Name ist Bond.".concat(" James Bond.");
```

Ja, das sieht komisch aus. Die Methode hängt einfach direkt am String-Literal. Die Methoden können eben nicht nur an String-Variablen gehängt werden.

```
ausgabe = "Mein Name ist Bond.";
ausgabe = ausgabe.concat(" James Bond.");
```

Das sieht aber fast so aus, als ob doch der String in der Variablen `ausgabe` verändert worden wäre. Lassen Sie sich nicht täuschen! Die Methode `concat` hat aus dem alten Inhalt von `ausgabe` und dem als Parameter übergebenen String einen neuen String gebildet. Das Ergebnis wird der Variablen `ausgabe` zugewiesen. Der vorige Inhalt »Mein Name ist Bond.« verliert seine Referenz und wird so ein Fall für die Garbage Collection.

Strings und ihresgleichen

Wenn Sie zwei Zahlen auf Gleichheit prüfen wollen, dann verwenden Sie die doppelten Gleichheitszeichen. Bei Strings funktioniert das in den meisten Fällen auch. Aber die Aussage »meistens« ist nicht schön, wenn man Programme schreiben will, die immer gleich funktionieren sollen.

Die richtige Art, Strings zu vergleichen, liegt darin, die Methode `equals` zu verwenden. Die Methode wird an den einen String angehängt und erhält den zweiten String als Parameter. Also etwa so:

```
boolean gleich = ausgabe.equals("Mein Name ist Bond. James Bond.");
```

Nach dieser Anweisung enthält die Variable `gleich` den Wert `true`. Wenn Sie dagegen die beiden Gleichheitszeichen verwenden, prüfen Sie lediglich, ob die beiden Strings die gleiche Referenz verwenden. Das folgende Programm müsste

also »nicht gleich« ausgeben, weil die Variablen `jb007` und `bond` ja auf unterschiedliche Zeichenketten referenzieren.

```
public class Gleichheit {
    public static void main(String[] args) {
        String jb007 = "Mein Name ist Bond. James Bond.";
        String bond  = "Mein Name ist Bond." + " James Bond.";
        if (jb007==bond) {
            System.out.println("gleich");
        } else {
            System.out.println("nicht gleich");
        }
    }
}
```

Aber tatsächlich erhalten Sie hier die Meldung »gleich«. Das liegt daran, dass Java ein wenig optimiert. Bei der Erstellung des Strings für die Variable `bond` merkt Java, dass es schon einen solchen String gibt, und weist `bond` eine Referenz auf `jb007` zu. Damit sind beide Referenzen gleich und der Vergleich mit den zwei Gleichheitszeichen funktioniert. Falls aber diese Optimierung mal nicht klappt, dann stimmt der Vergleich nicht. Darum nehmen Sie bitte die Methode `equals`. Die ist verlässlich.

Beim Vergleich mit `equals` werden Groß- und Kleinbuchstaben streng unterschieden. Ist das nicht gewünscht, sollten Sie sich die Methode `equalsIgnoreCase` näher anschauen. Diese Methode achtet nicht auf die Groß- und Kleinschreibung.

```
boolean gleich = ausgabe.equalsIgnoreCase(
        "MEIN name ist bOND. James bond.");
```

Auch hier enthält die Variable `gleich` nach der Ausführung den Wert `true`.



Seit Java 7 dürfen Strings sogar für die Fallunterscheidung `switch` verwendet werden. Welch ein Aufstieg!

Dieser Text hat Längen

Die Methode `length` ermittelt, wie viele Zeichen ein String enthält.

```
int laenge = ausgabe.length();
```



Wenn Sie die Arrays kennenlernen, werden Sie feststellen, dass auch Arrays eine Länge haben und diese über `length` zu ermitteln ist. Allerdings ist die Länge `length` bei Arrays ein Attribut, also eine Integer-Variable. Bei Strings ist `length` eine Methode und benötigt ein Klammernpaar am Ende. Fragen Sie

mich nicht, wer sich das ausgedacht hat.

Inhaltsangabe eines Strings

Wenn die Länge eines Strings 0 ist, dann ist er wohl leer. Damit Sie sich auch anders ausdrücken können, gibt es die Methode `isEmpty`. Sie stellt fest, ob ein String leer ist

```
boolean leer = ausgabe.isEmpty();
```

Hier wird das Ergebnis wohl eher falsch sein. Kurz im Programm prüfen ... Und richtig! Das ist falsch! Weil der String nicht leer ist, liefert die Methode `isEmpty` den Rückgabewert `false`.

Sehr viel spannender ist die Methode `contains`. Mit ihr können Sie feststellen, ob in einem String ein anderer String enthalten ist.

```
boolean istDrin = ausgabe.contains("Bond");
```

Diese Anweisung liefert den Wert `true`, weil »Bond« ja in dem Satz enthalten ist. Was aber tun Sie, wenn Sie bei der Überprüfung keinen Wert auf Groß- oder Kleinschreibung legen? Dann kombinieren Sie die Methode `toLowerCase` mit der Methode `contains` und übergeben als Parameter einen kleinen »bond«.

```
boolean istDrin = ausgabe.toLowerCase().contains("bond");
```

Der String wird durch die Methode `toLowerCase` in Kleinbuchstaben verwandelt und das Ergebnis mit `contains` verglichen.

Geschnittene Strings

Die Methode `trim` entfernt Leerzeichen, die vielleicht am Anfang oder am Ende eines Strings stehen.




```
String neu = ausgabe.trim();
```

Diese Methode ist vor allem hilfreich, wenn Eingabefelder ausgelesen werden. Manchmal tippt der Benutzer versehentlich ein paar Leerzeichen vor oder nach der Eingabe ein. Diese können bei der Auswertung zu unerwarteten Ergebnissen führen.

Wo bist du?

Mit der Methode `indexOf` kann bestimmt werden, an welcher Stelle ein Suchbegriff in einem String enthalten ist. Sollte der Suchstring nicht enthalten sein, liefert die Methode -1 zurück. Im folgenden Beispiel ist es allerdings die Zahl 14.

```
int position = ausgabe.indexOf("Bond");
```

Wie gut, dass ich zufällig ein Beispiel gewählt habe, in dem das Wort »Bond« zweimal vorkommt. Denn manchmal reicht es nicht, das erste Auftreten zu erwischen, sondern man möchte alle Treffer sammeln. Für diesen Fall gibt es eine überladene Methode `indexOf`, die außer dem Vergleichstring auch noch die Angabe der Position erlaubt, ab der gesucht werden soll. Das erste Auftreten war bei Position 14, also können Sie ab Position 15 weitersuchen.

```
int position = ausgabe.indexOf("Bond", 15);
```

Wollen Sie alle Positionen ermitteln, bauen Sie darüber eine Schleife, bis die Variable `position` einen Wert enthält, der kleiner als 0 ist.

```
int position = ausgabe.indexOf("Bond");
while (position>=0) {
    System.out.println("Position: " + position);
    position = ausgabe.indexOf("Bond", position+1);
}
```

Die Ausgabe sieht dann so aus:

```
Position: 14
Position: 26
```

String-Zerlegung

Die Methode `substring` liefert einen Teilstring zurück, der ab der übergebenen Position beginnt. Beispielsweise wissen Sie ja jetzt, dass der zweite Bond ab der 26. Position erscheint. Wenn Sie diese 26 der Methode `substring` als Parameter übergeben, gibt sie den Teil des Strings ab der Position 26 zurück.

```
String bond = ausgabe.substring(26);
```

Das Ergebnis ist nun »Bond.«. Der Punkt am Ende stört etwas. Zum Glück kann bei `substring` auch ein zweiter Parameter angegeben werden, der die Endposition angibt.

```
String bond = ausgabe.substring(26, 30);
```



Abbildung 6.2: Die Wirkung von `substring(26, 30)`

Damit wäre Bond isoliert. Aber keine Sorge: 007 bleibt ja nie lange einsam.

Ersatzbeschaffung

Sie können auch mit der Methode `replace` einen Teil des Strings durch einen anderen ersetzen. Falls Sie der Meinung sind, dass Bond längst durch einen echten Helden abgelöst werden müsste, fragen Sie nicht Blofeld, sondern `replace`.

```
String ichBins = ausgabe.replace("Bond", "Willemer");
```

Das Ergebnis lautet dann:

```
Mein Name ist Willemer. James Willemer.
```

Wenn der Computer das ausspuckt, muss es wohl stimmen. Der Computer hat schließlich immer recht.

Die Methode ersetzt offenbar alle Vorkommnisse des Wortes »Bond«. Darüber hinaus gibt es eine Abart, die `replaceAll` lautet. Sie kann mehr. Sie versteht es, mit regulären Ausdrücken umzugehen.

```
String replaceAll(String such, String ersatz)
```

Regulärer Ausdruck

Reguläre Ausdrücke ermöglichen eine Reihe von Platzhaltern und Regeln, die den Suchstring spezifizieren. Wenn Sie oft komplexe Ersetzungsprobleme haben, sollten Sie sich unbedingt damit näher befassen. Die Syntax sieht auf den ersten Blick etwas abschreckend aus. Aber was ein echter Held ist, der lässt sich dadurch nicht abhalten.



Falls Sie sich doch eher nicht zu den Helden zählen und keine direkte Anwendung für reguläre Ausdrücke sehen, können Sie den Abschnitt einfach überspringen. Merken Sie sich, wo die regulären Ausdrücke beschrieben sind, und kommen Sie bei Bedarf wieder hier her.

Das folgende Beispiel ersetzt alle Zeichenfolgen, die mit einem Großbuchstaben anfangen und dann ein kleines o enthalten. Es folgt ein beliebiges Zeichen und dann ein d.

```
String ich007 = ausgabe.replaceAll("[A-Z]o.d", "Willemer");
```

- Der Punkt steht für irgendein Zeichen.
- In eckigen Klammern wird beschrieben, welcher Buchstabe an dieser Stelle erwartet wird. Hier kann eine Aufzählung oder durch das Minuszeichen ein Bereich angegeben werden.

Die Suchmaske würde also nicht nur auf Bond, sondern auch auf Mond oder Mord passen, aber nicht auf blond. Dagegen passen Fond und Ford, Fort wieder nicht. Selbst Ao4d würde also passen. Aber eigentlich heißt der R2D2! Und der kommt nicht bei 007 vor, sondern in Star Wars und passt sowieso nicht auf den regulären

Ausdruck.

Mit dem Stern kann man noch verrücktere Dinge tun. Er erlaubt eine beliebige Anzahl von Wiederholungen des voranstehenden Zeichens. `e*` steht also für kein e, für e, ee oder eeeeeeee. Die Kombination `.*` steht darum für beliebig viele beliebige Zeichen.

```
String nocheiner = ausgabe.replaceAll("[A-Z].*e", "Willemer");
```

Das Ergebnis sieht übrigens so aus:

```
Willemers Bond.
```

Die Methode findet zuerst das M und das passt zu `[A-Z]`. Dann sucht sie bis zum letzten e im String. Das s hinter Willemer stammt vom Ende des Wortes James, dessen e das letzte e in diesem String ist.

Zeichen	Bedeutung
.	Ein Punkt passt auf ein beliebiges Zeichen
[bdg]	passt auf den Buchstaben b, d oder g
[b-g]	passt auf jeden Buchstaben zwischen b und g
[^b-g]	passt auf jeden Buchstaben, der nicht zwischen b und g liegt
q*	passt auf 0 bis unendlich viele Buchstaben q
r+	passt auf ein bis unendlich viele Buchstaben r
w?	passt auf 0 oder einmal w

Tabelle 6.1: Reguläre Ausdrücke

Typwandler

7 und `"007"` unterscheiden sich darin, dass das eine eine Zahl und das andere ein String ist. Das könnte nett nebeneinander stehen, aber manchmal muss das Eine in das Andere überführt werden. Beispielsweise verwenden die grafischen Oberflächen für ihre Ein- und Ausgabefelder ausschließlich Strings. Um mit diesen Eingaben rechnen zu können, müssen aus den Strings Zahlenwerte gemacht werden. Dasselbe gilt umgekehrt: Sollen Zahlenwerte angezeigt werden, müssen diese für die Ausgabe oft in Strings umgewandelt werden.

Aus einer Zahl einen String machen

Wie man eine Zahl an einen String anhängen kann, haben Sie ja schon gesehen. Wollen Sie eine Zahl in einen String überführen, reicht es, die Zahl an einen leeren String anzuhängen:

```
int zahlenVariable = 7;
String zahlString = "" + zahlenVariable;
```

Das funktioniert nicht nur mit Zahlenvariablen, sondern auch mit den meisten Objekten. Dabei greift Java in solch einer Situation einfach auf die Methode `toString` zu.



Wenn Sie eines Tages eine eigene Klasse von Objekten erstellen, implementieren Sie einfach die Methode `toString` und schon kann Java auch Ihr Objekt durch Anhängen an einen String konvertieren. Daraufhin können Sie auch `System.out.println` verwenden, um Ihre Klasse auszugeben, da der Parameter `String` von `System.out.println` überschrieben wurde.

Aus einem String eine Zahl gewinnen

Wenn man aus einer Zahl so einfach einen String machen kann, geht das doch bestimmt auch genauso in die andere Richtung? Etwa so ...

```
int zahl = 0 + "12"; // das geht in die Hose!
```

Nein, das funktioniert so nicht! Wirklich schade! Glücklicherweise hat Java eine Methode zur Verfügung gestellt, die diese Aufgabe übernimmt. Sie finden solche Methoden in den Wrapperklassen. Diese werden an anderer Stelle noch ausführlicher beschrieben.

```
int zahl    = Integer.parseInt("122");
double wert = Double.parseDouble("12.2");
```

Sie können aus einem String eine Zahl, in diesem Beispiel `“122”` oder `“12.2”` auslesen, indem Sie die Klasse `Integer` oder `Double` verwenden, je nachdem, ob es eine ganze Zahl oder eine Fließkommazahl werden soll.



Der Begriff *parse* stammt aus dem Compilerbau. Ein Parser ist ein Programm, das einen Text liest und dessen Befehle erkennt und auswertet.

Aber die Parse-Methoden sind etwas weinerlich. Kaum ist vorn oder hinten ein Leerzeichen im String, reißen sie das ganze Programm herunter. Aber mit der String-Methode `trim` können Sie das Problem der Leerzeichen bereits im Vorfeld schnell beseitigen.

Schwieriger wird es schon, wenn ein fieser Anwender völlig wirre Zeichen statt Zahlen eingibt. Was kann man tun? Das verrate ich Ihnen später in Kapitel 13. Alternativ könnten Sie ja auch selbst parsen. Im Rahmen dieses Kapitels werden Sie noch sehen, wie es geht.

StringBuffer und StringBuilder

Manche Dinge ändern sich nie. Strings zum Beispiel. Bei jeder Änderung wandert der alte String in den Müll und ein neuer wird verwendet. Das scheint in Zeiten des

Müllrecyclings eigentlich nicht mehr ganz zeitgemäß. Es muss doch auch andere Möglichkeiten geben.

Und wenn der Autor schon einmal den Abschnitt mit solch einem Gesülze beginnt, hat er natürlich irgendetwas in der Tasche, mit dem er den Leser beeindrucken will.

Also gut, zugegeben: Wenn Sie also doch einmal einzelne Zeichen innerhalb eines Strings verändern wollen, können Sie die Klassen `StringBuffer` und `StringBuilder` einsetzen. Sie ermöglichen Änderungen an ihren Zeichenketten, ohne jedes Mal ein neues Objekt anzulegen.



Der Hauptunterschied zwischen `StringBuffer` und `StringBuilder` zum String liegt darin, dass der String keine Methode `setCharAt` zur Verfügung stellt.

Ihren Vorteil spielen diese Klassen nur aus, wenn man einen String wirklich immer wieder ändert. Dabei ist der `StringBuilder` noch einmal etwas schneller als der ältere `StringBuffer`. `StringBuffer` ist eher sozial eingestellt und achtet darauf, dass alles gut geht, auch wenn das Programm in mehreren Threads nebenläufig auf den Daten arbeitet. `StringBuilder` interessiert sich nicht für Threads. Dadurch ist er schneller. Das braucht er auch nicht, weil Sie mehrere Threads leicht selbst synchronisieren können. Sie glauben mir nicht? Warten Sie nur ab! In Kapitel 28 zeige ich es Ihnen.

Die Methoden

Beide Klassen bringen die gleichen Methoden mit. Hier eine kleine Vorstellungsrunde.

- `void setCharAt(int pos, char c)`

  Im Gegensatz zum normalen String können im `StringBuffer` und im `StringBuilder` einzelne Zeichen effizient ersetzt werden. Und genau dies tut die Methode `setCharAt`. Der erste Parameter ist die Position und der zweite Parameter das neue Zeichen.
- `StringBuffer reverse()`

  Diese Methode dreht den Inhalt des Strings komplett um.
- `StringBuffer append(StringBuffer str)`

  Mit der Methode `append` wird ein String an den anderen angehängt. Erst wenn der intern vorgehaltene Reservespeicher überfüllt ist, wird neuer Speicher angefordert.
- `StringBuffer insert(int pos, String str)`

  Durch den Aufruf der Methode `insert` kann ein String eingefügt werden. Der erste Parameter gibt die Position an und der zweite den String, der eingefügt wird.

- ✔ `StringBuffer delete(int start, int ende)`

  Die Methode `delete` entfernt den Teil des Strings, der durch die Parameter `start` und `ende` bestimmt wird.

Ein `StringBuilder` oder `StringBuffer` kann natürlich auch mittels der bekannten Methode `toString` in einen normalen String überführt werden. Beim Vergleich zwischen einem String auf der einen Seite und `StringBuffer` oder `StringBuilder` auf der anderen Seite meldet die Methode `equals` immer den Wert `false`.

Beispiel Pfadumbau

Lassen Sie mich den Umgang mit den Stringmanipulationen an einem kleinen Beispiel lebendiger machen. Linux verwendet nach seinem Vorbild UNIX den Schrägstrich / als Pfadtrenner. Den hatte MS-DOS aber bereits als Optionsschalter missbraucht, bevor es lernte, was Verzeichnisse sind. Als dann MS-DOS endlich Verzeichnisse beherrschte, musste ein anderes Zeichen als der Schrägstrich her und man wählte den Backslash \. Etwas blöd zu tippen und auszusprechen. Leider wurde das Ding zusammen mit den Laufwerksbuchstaben von MS-DOS an Windows vererbt und wurde so zur Geißel aller Programmierer. Wenn Sie sowohl unter Linux als auch unter Windows arbeiten, kann die unterschiedliche Pfadschreibweise zu Problemen führen. Für Sie natürlich nicht, denn Sie können die Pfade in die jeweils richtige Schreibweise umwandeln.

substring und indexOf per String

Versuchen wir es zuerst mit den Methoden `substring` und `indexOf`: Wir suchen alle Schrägstriche und hängen die Teile dazwischen an den neuen Windows-Pfad. Dabei schummeln wir einfach noch je einen Backslash ein.

```
class StringPfadUmbau {
    public static void main(String[] args) {
        String linuxPfad = "/media/usbstick/daten/unfug";
        String windowsPfad = "";
        int pos = 0;  // akt. Position im String
        int oldPos = 0; // wo wir zuletzt waren
        // wiederholen, wenn es noch weitere Schrägstriche gibt
        while (pos>=0) {
            pos = linuxPfad.indexOf('/', oldPos);
            if (pos>=0) {
                // Der Windows-Pfad ist der alte
                // plus dem Backslash
                // und dem Pfad zwischen den Schrägstrichen.
                if (pos>oldPos) {
                    windowsPfad = windowsPfad + '\\'
                            + linuxPfad.substring(oldPos, pos);
                }
                oldPos = pos+1;
            }
        }
        // Sollte nach dem letzten Schrägstrich noch ein Pfad
        // kommen, muss der natürlich auch übernommen werden.
        if (oldPos<linuxPfad.length()) {
           windowsPfad = windowsPfad + '\\'
               + linuxPfad.substring(oldPos,
                       linuxPfad.length());
        }
        System.out.println(windowsPfad);
    }
}
```

Listing 6.4: Der Pfadumbau mit Strings



Sie wundern sich vielleicht, dass die Backslash-Zeichen immer paarweise auftreten. Das liegt daran, dass der Backslash in einem String-Literal immer Sonderzeichen einleitet. Will man den Backslash selbst, muss man ihn doppelt verwenden.

replace

Kann man für den Austausch der Pfadtrenner nicht viel einfacher die Methode `replace` nehmen? Probieren Sie es einmal:

```
class ReplacePfadUmbau {
    public static void main(String[] args) {
        String linuxPfad = "/media/usbstick/daten/unfug";
        String windowsPfad = linuxPfad.replace("/", "\\");
        System.out.println(windowsPfad);
    }
}
```

Listing 6.5: Replace baut den Pfad noch kürzer um.

Und siehe da: Es klappt wunderbar!

Der Pfadumbau mit dem StringBuilder

Der Austausch der Slashes zu Backslashes dürfte mit dem `StringBuilder` schneller und ressourcenschonender sein. Schließlich müssen nicht ständig neue Strings gebildet werden. Die Garbage Collection kann etwas entspannen.

```
class BuilderPfadUmbau {
    public static void main(String[] args) {
        StringBuilder pfad = new
            StringBuilder("/media/usbstick/daten/unfug");
        int pos = 0;
        int oldPos = 0;
        while (pos>=0) {
            // Schrägstrichposition merken
            pos = pfad.indexOf("/", oldPos);
            if (pos>=0) {
                // einfach durch Backslash ersetzen
                pfad.setCharAt(pos, '\\');
            }
            if (pos>=0) oldPos = pos+1;
        }
        System.out.println(pfad);
    }
}
```

Listing 6.6: Pfadumbau mit `StringBuilder`

Sie müssen also abschätzen, ob es sich lohnt, mehr Aufwand zu betreiben, um im Gegenzug den Speicher und das Umschieben der Texte im Speicher zu sparen. Bei kurzen Texten dürfte die Methode `replace` gewinnen, bei sehr großen Texten die Arbeit mit dem `StringBuilder`.



Und jetzt, nachdem Sie so fleißig an Ihrem Pfadumbau gearbeitet haben, verrate ich Ihnen, dass Sie auch unter Windows den Schrägstrich als

Pfadtrenner einsetzen können. Microsoft hatte wohl doch ein Einsehen.

Aufgaben

Programmieren lernt man nur durch Programmieren. Außerdem merken Sie erst dann, ob ich Ihnen wirklich alles gut erklärt habe.

Bond, seine Längen und Ausgaben

Überlegen Sie, was das Programmbeispiel in Listing 6.7 tut! Sie dürfen Ihren Compiler zuhilfe nehmen. Ich würde es eh nicht merken.

```
String ausgabe = "Mein Name ist Bond. James Bond.";
int laenge = ausgabe.length();
for (int i=0; i<laenge; i++) {
    System.out.print(ausgabe.charAt(i));
}
```

Listing 6.7: Was tut dieses Programm?

Anzahl der Suchmuster

Schreiben Sie ein Programm, das zählt, wie oft ein String in einem anderen String vorkommt.

String-Tanga

Das Wort »String« gibt es im Deutschen eigentlich ja nicht und es erfreut sich nur im Zusammenhang mit dem String-Tanga einer gewissen Verbreitung. Auf der anderen Seite verspricht das Wort einen Hauch von Erotik. Ich habe das ausprobiert und mich im String-Tanga vor den Spiegel gestellt. Ich kann Ihnen versichern, dass dieses Versprechen nicht eingehalten wird. Es ist gar nicht erotisch, sondern sieht eher aus wie eine abgebundene Presswurst. Also sollten Sie Ihre Assoziationen überdenken.

Wie Sie sicher wissen, ist das Wort Tanga aus dem Wort Tang entstanden, an das man die Selbstlaute, die im Wort vorkommen, hinten angehängt hat. Schreiben Sie ein Programm namens Tanga, das ein Wort nach Selbstlauten durchsucht und diese in dieser Form an das Wort anhängt. Aus Schlot würde dann Schloto, aus Kaffee Kaffeeaee, aus Maus Mausau und so fort.

Palindrom

Ein Palindrom ist ein String, der sich von hinten nach vorn gleich liest (zum Beispiel »Reittier« oder »Rentner«). Das Programm soll den Inhalt einer String-Variablen darauf testen, ob er ein Palindrom darstellt und damit rückwärts wie vorwärts gelesen gleich ist. Als Ergänzung könnten Sie das Programm erweitern, und Leerzeichen überspringen, um auch Palindromsätze zu erkennen.

Noch einmal Linux- und Windows-Pfad

Verfeinern wir noch einmal unser Beispielprogramm, in dem ein Linux-Pfad in einen Windows-Pfad umgesetzt wird. Der Linux-Pfad */home/willemer/* entspricht unter Windows *C:\Dokumente und Einstellungen\willemer*. Können Sie das Programm so ändern, dass es das berücksichtigt? Die Musterlösung finden Sie unter dem Namen `HeimatPfadUmbau.java` zum Download.

Wahr oder falsch?

Ein kleiner Multiple-Choice-Test:

- ✔ String ist ein primitiver Typ.
- ✔ Das Pluszeichen kann mit Strings verwendet werden.
- ✔ Ein String-Literal kann direkt einer String-Variablen zugewiesen werden.
- ✔ Die Buchstaben werden über rechteckige Klammern angesprochen.
- ✔ String-Methoden können auch über ein Literal aufgerufen werden.
- ✔ Einer String-Variablen kann null zugewiesen werden.

Sollten Sie bei einer Frage unsicher sein, taugt dieses Buch nichts.

Reguläre Namensgleichheit

Wer nicht gerade einen sehr einfachen Namen hat, kennt das Problem. Die Anschriften auf der Post muten manchmal abenteuerlich an. Briefe, die an mich gerichtet sind, beginnen zwar fast immer noch mit Wil. Aber dann wird es unübersichtlich. Einig sind sich die Namensgeber aber immer darin, dass ich auf mer ende. Oder mar?

Schreiben Sie ein Programm `FindWillemer`, das den Namensversuch als Parameter übergibt. Das Programm soll `true` ausgeben, wenn die oben genannte Ähnlichkeit zum Namen Willemer erreicht wurde.

Vielleicht finden Sie aber auch ein wiederkehrendes Muster für Ihren Namen. Oder den Namen Ihrer Stadt, Ihres Lieblingsmusikers oder Ihres Nachbarn. Programmieren Sie auch das.

Einfach mal selbst parsen

Wenn Sie aus einem String eine Zahl gewinnen wollen, bietet Java Ihnen die Methoden `parseInt` oder `parseDouble` an. Bei deren Aufruf müssen Sie aber immer damit rechnen, dass es knallt, wenn der String nicht so ganz den Vorstellungen von Java entspricht. Darum schreiben wir einfach selbst eine solche Funktion. Sie sucht das erste Zeichen, das zwischen ‘0’ und ‘9’ liegt. Aus der Ziffer wird eine Zahl und diese in der Ergebnisvariablen gespeichert. Für jede weitere gefundene Ziffer wird die Ergebnisvariable erst mit 10 multipliziert und dann die gefundene Ziffer aufaddiert. Im Prinzip genauso, wie Sie selbst intuitiv eine mehrstellige Zahl lesen. Kommt keine Ziffer mehr, steht in der Ergebnisvariablen

die Zahl. Am Schluss geben Sie das Ergebnis aus. Damit ist es völlig egal, ob vor oder hinter der Zahl noch irgendwelche Texte stehen. Zu schwer? Ach was!

```
class Parse {
   public static void main(String[] args) {
       String input = "haha 1245 huhu";
       int pos=0;
       long ergebnis = 0L;
       char zeichen = input.charAt(pos);
```

So ein Mist! Ich glaube, die nächsten Zeilen sind etwas durcheinander geraten. Wären Sie vielleicht so freundlich, die Zeilen wieder in die richtige Reihenfolge zu bringen? Danke!

```
    }
    }
       zeichen = input.charAt(pos);
       zeichen = input.charAt(pos);
       ergebnis += zeichen-'0';
    while (zeichen<'0' || zeichen >'9') {
    while (zeichen>='0' && zeichen <='9') {
       pos++;
       pos++;
       ergebnis *= 10;
```

Ab hier stimmt es wieder!

```
       System.out.println(ergebnis);
   }
}
```

Wenn Sie mögen, könnten Sie auch noch selbst etwas dazuprogrammieren. Ergänzen Sie doch den kleinen Parser um die Fähigkeit, Fließkommazahlen zu ermitteln. Und wenn Sie schon mal dabei sind, sollte er auch noch sowohl den Punkt als auch das Komma als Dezimaltrenner erkennen. Viel Spaß!

Kapitel 7

Arrays

IN DIESEM KAPITEL

Arrays als Variablen in Reih und Glied

`for` und die Liebe zu Arrays

Keine Science-Fiction: Arrays im mehrdimensionalen Raum

Häufig trifft man Werte im Rudel an. So bestehen die Lottozahlen aus sechs ganzen Zahlen, die erst gemeinsam wirklich Freude machen. Um solche Wohngemeinschaften in Java zu betreuen, hat der Programmierer das *Array* zur Verfügung (statt Array wird manchmal auch der Begriff *Felder* verwendet). Stellen Sie sich Arrays als nebeneinander angeordnete, durchnummerierte Kästchen vor, die je ein Element aufnehmen können. Alle Elemente müssen vom gleichen Typ sein.



Abbildung 7.1: Gleichartige Objekte dicht an dicht und nummeriert

Ähnlich einer Hausnummer kann dann jedes in einem Array abgelegte Element über eine Nummer angesprochen werden, die man als *Index* bezeichnet. Um es nicht zu einfach zu machen, fängt die Nummerierung nicht bei 1, sondern bei 0 an. Die Kästchennummer wird in rechteckige Klammern hinter die Array-Variable gesetzt.

```
haus[12]
```

Hier wäre es zum Beispiel das Haus mit der Nummer 12 und damit das 13. Haus, weil die Zählung ja immer bei 0 beginnt. Der Ausdruck `haus[12]` kann bei einer

Zuweisung sowohl links als auch rechts vom Gleichheitszeichen stehen, er verhält sich also wie jede gewöhnliche Variable.

Wir bauen ein Array

Aber bevor Sie Ihre Sachen in ein Haus hineintragen, müssen Sie das Array für die Häuser erst einmal bauen. Und da es sich um einen Neubau handelt, verwendet man den Befehl `new`. Da Häuser nicht zu den Standardtypen von Java gehören und der Eigenbau von Typen erst später gezeigt wird, verwenden wir hier die Lottozahlen, die bekanntermaßen ganze Zahlen sind.

Lottozahlen sind sechs Zahlen, die erst gemeinsam sinnvoll sind, und damit ein Paradebeispiel für Arrays darstellen. Vermutlich wurden sie nur erfunden, um Programmierbuchautoren das Leben zu erleichtern.

```
int lotto[] = new int[6];
(1)   (2)  (3)(4)   (5)
```

1. Zuerst wird der Datentyp angegeben, der in jedem Feld des Arrays abgelegt werden soll. Für die Lottozahlen ist der Typ `int` passend.
2. Es folgt der Name des Arrays. Das Array für die Lottozahlen nennen wir `lotto`. Hinter oder vor den Array-Namen muss noch ein Paar leere rechteckige Klammern gesetzt werden, um klarzustellen, dass das Array ein Array ist.
3. Nun kommt das Zuweisungszeichen, also ein Gleichheitszeichen.
4. Anschließend folgt der Befehl `new`, der den Platz für das Array im Speicher schafft und damit das Array erstellt.
5. Hinter dem Befehl `new` gibt man an, was es werden soll. Dazu wird wieder der Typ aufgegriffen, den man am Anfang der Zeile verwendet hat, und in rechteckigen Klammern angegeben, wie viele Elemente das Array haben soll.



Die Größe eines Arrays kann man nachträglich nicht ändern, nachdem sie einmal festgelegt ist. Für »dynamische Arrays«, deren Größe sich später noch ändern lässt, bietet das Java Collection Framework eine `ArrayList` an. Die zeige ich Ihnen später – halten Sie durch!

Mit dieser Anweisung wurde das Array angelegt und jedes Element des Arrays kann nun mit einem Wert befüllt werden.

```
class Lotto {
    public static void main(String[] args) {
        final int LOTTO = 6;
        int lotto[] = new int[LOTTO];
        lotto[0] = 1;
        lotto[1] = 4;
        lotto[2] = 9;
        lotto[3] = 16;
        lotto[4] = 25;
        lotto[5] = 36;
    }
}
```

Listing 7.1: Anlegen und Füllen eines Arrays



Ein Array mit sechs Elementen wird durch den Aufruf von new int[6] erzeugt. Der Index reicht dann von 0 bis 5.

Das Füllen jedes einzelnen Elements ist sehr mühselig, funktioniert aber. Sie können ein Array auch erstellen, indem Sie es mit einer Liste von Werten initialisieren. Dadurch verkürzt sich das Beispiel um die vielen Zuweisungen.

```
class Lotto {
    public static void main(String[] args) {
        int lotto[] = { 1, 4, 9, 16, 25, 36 };
    }
}
```

Listing 7.2: Anlegen und Initialisieren eines Arrays

Wenn Sie der Array-Variablen eine bereits vorbesetzte Liste zuweisen, ist ein `new` nicht mehr erforderlich. Ein Array wird durch in geschweifte Klammern gesetzte und durch Kommata voneinander getrennte Literale initialisiert.

Mit for durchschleifen

Sie könnten die Initialisierung auch durch eine `for`-Schleife ersetzen. Und natürlich können Sie auf die gleiche Weise den Inhalt des Arrays Wert für Wert auf dem Bildschirm anzeigen.

```
class Lotto {
    public static void main(String[] args) {
        final int LOTTO = 6;
        int lotto[] = new int[LOTTO];
        for (int i=0; i<LOTTO; i++) {
            lotto[i] = (i+1)*(i+1);
        }
        for (int i=0; i<LOTTO; i++) {
            System.out.println(lotto[i]);
        }
    }
}
```

Listing 7.3: Füllen und Anzeigen des Arrays per Schleife

Die Schleife mit `for` und Arrays passen ganz wunderbar zusammen. Sie sind wie füreinander geschaffen! Ist das nicht romantisch? Und glauben Sie mir: Hier ist die Spitze des Spannungsbogens noch nicht erreicht. Wenn Sie nämlich nur das Array auslesen wollen, dann gibt es noch eine spezielle Version.

Bei dieser Spezialschleife müssen Sie in der `for`-Klammer nur eine Variable einsetzen, die dem Typ eines Array-Elements entspricht und dann, durch einen Doppelpunkt abgetrennt, das Array, das durchlaufen wird.

```
for (int zahl : lotto) {
   System.out.println(zahl);
}
```

Listing 7.4: Ein speziell geschliffenes Array

Innerhalb der Schleife wird auf die im Schleifenkopf definierte Variable zugegriffen, die bei jeder Umdrehung das nächste Element des Arrays enthält. Java weiß, wie groß das Array dimensioniert ist, und so gibt es keine Unfälle mit der Array-Grenze. Die Schleife ist zudem übersichtlicher.



Auch wenn Sie nun glauben zu wissen, wie die Lottozahlen zustande kommen, sollten Sie sich nicht blind darauf verlassen. Es kann sein, dass die Lottozahlen ausgerechnet das nächste Mal anders lauten als im Programmbeispiel.

Übung: Ein Array für die Monatstage

Falls Sie einmal das Vergnügen haben, mit Kalenderdaten zu arbeiten, werden Sie feststellen, dass dort großes Chaos herrscht. Dass es zwölf Monate gibt, stammt wohl aus der Zeit, als Menschen noch zwölf Finger hatten. Aber noch viel lästiger

ist es, dass die Monate eine unterschiedliche Anzahl von Tagen haben. Wollen Sie also wissen, wie viele Tage Ihr Silvestervorsatz gehalten hat, wenn Sie am 23. September wieder anfangen zu rauchen, raucht spätestens dann auch Ihr Kopf. Je nach Monat müssen Sie zwischen 28 und 31 Tage aufaddieren. Das gibt hinreißende Fallunterscheidungen.

Sie können sich das Leben deutlich vereinfachen, wenn Sie sich ein Array zulegen, in dem die Monatstage abgelegt sind.

```
int tageImMonat[] =
        { 31, 28, 31, 30, 31, 30,
          31, 31, 30, 31, 30, 31 };
```

Listing 7.5: Array mit den Tagen pro Monat

Sie sehen, dass im Array die jeweilige Anzahl der Monatstage so angelegt ist, dass Sie mit der Monatsnummer als Index die Anzahl des jeweiligen Monats erhalten. Nun muss natürlich der Einwurf kommen, dass der Februar im Schaltjahr ja 29 Tage hat. Bei einem bekannten Navi-Hersteller fehlte dieser Einwurf, woraufhin einige Modelle ab März 2012 überhaupt nicht mehr navigierten. So wirkungsvoll können Programmierfehler sein. Also sollten Sie sich darum kümmern. Abfragen beherrschen Sie ja längst.

Kommen wir zurück zum Array. Um die vergangenen Tage im September zu ermitteln, müssen Sie nun die Zahlen aller Monate vor dem Zielmonat aufsummieren.

```
public class DatumsArray {
    public static void main(String[] args) {
        int tageImMonat[] =
            { 31, 28, 31, 30, 31, 30,
              31, 31, 30, 31, 30, 31 };
        int dattag = 23; int datmonat = 9;
        int tage = 0;
        for (int i=0; i<datmonat-1; i++) {
            tage += tageImMonat[i];
        }
        tage += dattag;
        System.out.println(tage);
    }
}
```

Listing 7.6: Anzahl der Tage seit Jahresbeginn

Das Programm in Listing 7.6 tut genau dies. Das Zieldatum wird in den Variablen `dattag` und `datmonat` festgelegt. Das Programm ermittelt beim 23.9. 266 Tage. Falls Sie das im Kopf nicht so schnell nachrechnen können, verwenden Sie als

Beispiel den 13. Februar. Das sollte 44 ergeben.

Auf dieser Basis ist es auch nicht schwer, den Schalttag einzubauen oder eine Methode zu schreiben, die die Differenz in Tagen zwischen zwei Kalendertagen bestimmt.

Übung: Ein Kartenspiel

Spielkarten kann man auch in einem Array speichern. Für ein Skatblatt benötigen Sie 32 Karten und der Einfachheit halber nummerieren wir sie durch.

```
public class Kartenspiel {
    static final int KARTENZAHL = 32;
    public static void main(String[] args) {
        int[] blatt = new int[KARTENZAHL];
        for (int i=0; i<KARTENZAHL; i++) {
            blatt[i] = i;
        }
    }
}
```

Listing 7.7: Ein Kartenspiel als Integer-Array

Um die Karten zu mischen, schreiben Sie eine Methode `mische`, die einige Male zufällig gewählte Karten miteinander tauscht.

```
static void mische(int[] karten) {
    java.util.Random zufall = new java.util.Random();
    final int MISCH = 10000;
    for (int misch=0; misch<MISCH; misch++) {
        int i = zufall.nextInt(KARTENZAHL);
        int j = zufall.nextInt(KARTENZAHL);
        int tausch = karten[i];
        karten[i] = karten[j];
        karten[j] = tausch;
    }
}
```

Listing 7.8: Kartenspiel mischen

Das Geheimnis, wie man aus Zahlen von 0 bis 31 Spielkartenwerte und -farben macht, liegt in der Division durch 4. Wird der Wert durch 4 geteilt, ergeben sich die Kartenwerte von 0 bis 7, die Sie den Spielkartenwerten 7, 8, 9, 10, B, D, K und A zuordnen können. Der Restwert der Division ergibt 0 bis 3, die Sie den Farben Kreuz, Pik, Herz und Karo zuweisen können.

Auch bei der Darstellung der Karten helfen Arrays. Für Farbe und Wert wird je ein Array angelegt, in dem abgelegt wird, was der Benutzer später zu sehen bekommt. Für die Kartenwerte habe ich ein Array von `char` angelegt. Leider benötigt eine 10 zwei Buchstaben. Hier habe ich mir mit einem Z geholfen. Für die Farben habe ich ein Array von `String` gewählt. Hier könnte man auch `char` verwenden, wenn man Kreuz als Treff bezeichnet, so dass Karo und Kreuz unterscheidbar sind.

```
static void zeige(int karte) {
    String[] farbe = { "Kreuz", "Pik  ", "Herz ", "Karo " };
    char[] wert = {'A', 'K', 'D', 'B', 'Z', '9', '8', '7'};
    System.out.print(wert[karte/4]+" "+farbe[karte%4]);
}
```

Listing 7.9: Anzeige einer Karte

Wenn Sie Skat spielen wollen, werden Sie drei Spieler benötigen, die jeweils ein Drittel der Karten bekommen. Dazu teilen wir die Karten ganzzahlig durch die Anzahl der Spieler. Für jeden Spieler gibt es ein Array und in einer Schleife verteilen wir die Karten. Das alles kann in der Methode `main` passieren:

```
mische(blatt);
int[] a = new int[KARTENZAHL/SPIELER];
int[] b = new int[KARTENZAHL/SPIELER];
int[] c = new int[KARTENZAHL/SPIELER];
for (int i=0; i<KARTENZAHL/SPIELER; i++) {
    a[i] = blatt[i*SPIELER];
    b[i] = blatt[i*SPIELER+1];
    c[i] = blatt[i*SPIELER+2];
}
```

Listing 7.10: Karten auf die Spieler a, b und c verteilen

Die Karten von jedem Spieler könnten Sie nun anzeigen. Allerdings sind die ja noch gemischt. Aber mit der Methode `java.util.Arrays.sort`, die Ihnen Java freundlicherweise zur Verfügung stellt, können Sie das Spieler-Array zuvor wieder sortieren. Ich habe es Ihnen am Spieler a vorgemacht.

```
java.util.Arrays.sort(a);
for (int karte: a) {
    zeige(karte);
    System.out.print('|');
}
System.out.println();
```

Listing 7.11: Karten von Spieler a anzeigen

Ich überlasse es Ihnen, die Fragmente zusammenzusetzen. Vielleicht programmieren Sie ja eines Tages daraus ein komplettes Skatspiel. Das könnte für

den Anfang noch etwas schwer sein. Aber wenn Sie etwas tun wollen, könnten Sie ja versuchen, die übrigen zwei Karten, die man als Skat bezeichnet, noch zusätzlich anzuzeigen.

Das String-Array der Methode main

Vielleicht ist dies ein guter Zeitpunkt, zu erläutern, was Sie mit dem Parameter der Methode `main` machen können. Sie erinnern sich: Es geht um `String[] args`, was hinter `main` in den Klammern steht. Mit Ihrem jetzigen Wissen erkennen Sie die Variable `args` als ein Array von Strings. Und dieses Array enthält die Argumente, mit denen das Programm aufgerufen wird.

Wenn Sie an der Konsole ein Programm aufrufen, geben Sie zunächst den Programmnamen und dann weitere Parameter an, die man auch als Argumente bezeichnet. Beispielsweise würde `edit text.txt` einen Editor rufen und das Argument wäre die Datei *text.txt*, die Sie mit dem Programm bearbeiten wollen.

Auch Java-Programme können so aufgerufen werden. Allerdings wird zunächst das Java Runtime System gestartet, das den Befehlsnamen `java` trägt. Der Aufruf eines in Java geschriebenen Editors würde also beispielsweise `java edit text.txt` lauten. Gibt der Benutzer Argumente ein, wie hier den Dateinamen *text.txt*, so findet das Programm diese in dem String-Array `args`.

Die Übergabe von Argumenten von der Konsole lässt sich am einfachsten vorführen. Das folgende Programm gibt den Inhalt des String-Arrays `args` einfach auf dem Bildschirm aus.

```
public class ZeigArgs {
    public static void main(String[] args) {
        // Durchlaufe args und lege die Elemente je in str ab
        for (String str : args) {
            // Für jedes Argument eine eigene Zeile
            System.out.println(str);
        }
    }
}
```

Listing 7.12: Zeige die Aufrufparameter!

Nun starten wir das Programm von der Konsole und geben ihm ein paar Parameter mit.

```
java ZeigArgs Anton “Stan Laurel” Oliver Hardy
Anton
Stan Laurel
Oliver
Hardy
```

Der Befehl `java` ruft das Java-Runtime-System auf, das das Programm `ZeigArgs` startet. Natürlich muss es vorher kompiliert worden sein. Hinter dem Programmnamen werden nun die Argumente aufgezählt. Die Argumente werde durch Leerzeichen voneinander getrennt. Das Programm gibt jedes Argument in einer eigenen Zeile aus, da wir `println` verwenden.

Dadurch sieht man sehr schön, dass Stan Laurel durch die Anführungszeichen zu einem Argument zusammengefasst wird.


Wenn Sie also ein Argument übergeben wollen, das Leerzeichen enthält, verpacken Sie es in Anführungszeichen.

Aufbruch in neue Dimensionen

Falls Sie Fan der Linearen Algebra sind, werden Sie Matrizen heiß und innig lieben und sich vermutlich schon seit den ersten Seiten dieses Buches fragen, wie man zweidimensionale Arrays anlegt. Wenn Sie eher ein Spieler sind und die Felder auf einem Schachbrett positionieren wollen, werden Sie sich fragen, wie man in Java mit zweidimensionalen Arrays umgeht. Wenn Sie mit Aktien spekulieren, werden Sie sich fragen, ob es eine Möglichkeit gibt, Kurse vorherzusehen. Damit ist bewiesen: Zwei von drei Menschen wollen mehr über zweidimensionale Arrays wissen.

Listing 7.13 zeigt, wie ein zweidimensionales Array angelegt wird. Bei der Variablendefinition werden nun doppelt so viele rechteckige Klammern verwendet wie bei einem eindimensionalen Array. Um auf das Feld eines zweidimensionalen Arrays zuzugreifen, versehen Sie die Array-Variable mit zwei rechteckige Klammern.

```
public class ZweiDimArray {
    public static void main(String[] args) {
        final int MAXX = 9;
        final int MAXY = 7;
        int [][]brett = new int[MAXX][MAXY];
        brett[8][6] = 42;
        for (int i=0; i<MAXX; i++) {
            for (int j=0; j<MAXY; j++) {
                System.out.println(brett[i][j]);
            }
        }
    }
}
```

Listing 7.13: Ein zweidimensionales Array

Was Sie in zwei Dimensionen machen können, gelingt Ihnen auch in mehreren Dimensionen. Das geht alles gut, solange Ihnen nicht die rechteckigen Klammern ausgehen.

Sie können ein zweidimensionales Array auch als ein Array von Arrays anlegen. Um ein zweidimensionales Array `brett` der Dimension 9 mal 7 zu erstellen, fordern Sie mit `new` zunächst ein Array der Dimension 9 an. Anschließend erzeugen Sie in jedem Element je ein Array der Dimension 7.

```
final int MAXX = 9;
final int MAXY = 7;
int [][]brett = new int[MAXX][]; // x-Richtung
for (int i=0; i<MAXX; i++) {
    brett[i] = new int[MAXY]; // y-Richtung
}
```

Listing 7.14: Alternatives zweidimensionales Array

Das Ergebnis können Sie sich so vorstellen wie in Abbildung 7.2.

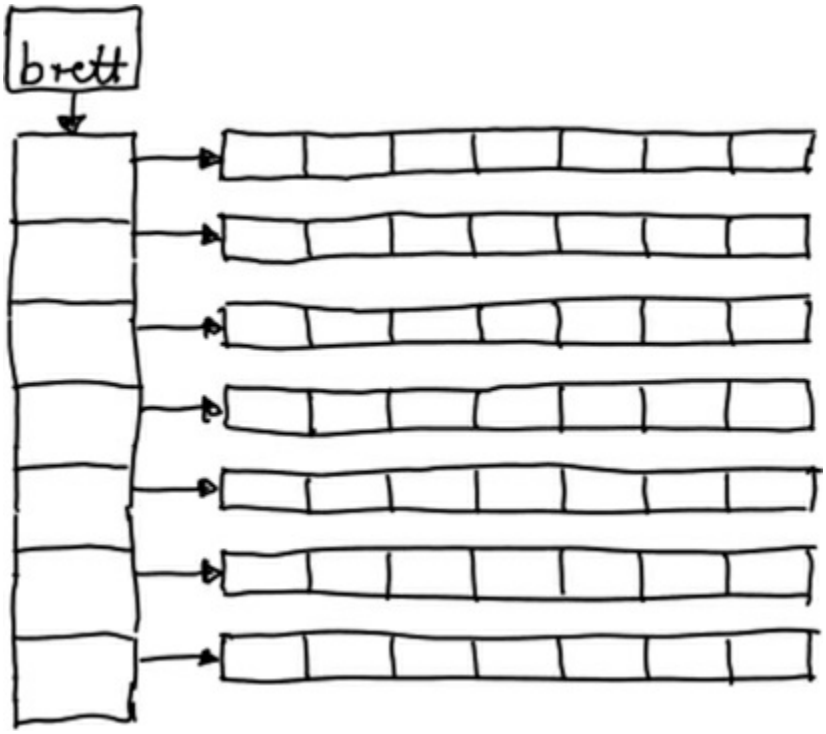


Abbildung 7.2: Ein zweidimensionales Array ist ein Array von Array-Referenzen.

Da jede Zeile einzeln angelegt wird, wäre es auch kein Problem, ein dreieckiges Array zu erzeugen. Dazu schreiben Sie in die eckige Klammer statt `MAXY` einfach `i+1`. Dann hat die erste Zeile ein Element, die zweite Zeile zwei Elemente und so weiter.

Und sicherheitshalber eine letzte Frage, nur um zu sehen, ob Sie es verstanden haben: Wie greift man auf das letzte Element unseres rechteckigen Arrays zu? Genau: mit `feld[8][6]` und nicht etwa mit `feld[9][7]`, weil ja jeder Index bei 0

beginnt. Aber das wissen Sie ja längst.

Aufgaben

Die eigenen Lottozahlen

Lottospielen ist doof. Sie verlieren eh immer – es sei denn natürlich, Sie schummeln sich Ihre Lottozahlen selbst zurecht. Das ist viel berechenbarer!

Zunächst müssen Sie sich ein Array für die Lottozahlen anlegen. Dann geht es um die Erzeugung von Lottozahlen. Sie könnten in Kapitel 14 spicken, aber das wäre hochgradig unfair. Das geht gar nicht!

Schreiben Sie eine Methode `getLotto`. Sie soll aus der Position im Array eine Lottozahl generieren. Beispielsweise könnten Sie das Quadrat der Position bestimmen und dazu die Summe aller Zahlen bis zur Position aufsummieren und anschließend den Wert quadrieren. Bei solchen Berechnungen kann die 49 leicht überschritten werden. Also verwenden Sie die Modulo-Operation.

Doppelte Lottozahlen

Klappt das? Wirklich? Keine Doppelten? Glück gehabt! Aber zur Sicherheit schreiben Sie doch eine Methode `entdopple`, die die Lottozahlen auf doppelte Zahlen überprüft und gegebenenfalls neu besetzt. Testen Sie doch einmal, ob Ihre Entdopplung funktioniert, indem Sie das Lotto-Array mit 12 12 12 13 23 23 vorbesetzen.

Geldwechsler

Für einen Geldwechselautomaten legen Sie die Münzwerte in Cent in einem Array ab, also 200 für eine 2-Euro-Münze, 100 für die 1-Euro-Münze, 50 für eine 50-Cent-Münze und so weiter. Mithilfe dieses Arrays können Sie nun in einer Schleife einen beliebigen Betrag so in die Münzwerte zerlegen, dass das Programm ausgibt, wie viele Münzen es für welchen Wert ausgibt.

Kapitel 8

Bermuda: Die Schiffe und das Meer

IN DIESEM KAPITEL

Ohne Spielregeln kein Spaß

Spielfeld und Schiffsflotte nachbilden

Spielen auf der Konsole

Lassen Sie uns spielen! Wir wollen im Laufe des Buches ein Spiel programmieren und weiterentwickeln. Dazu darf ich Ihnen mein Lieblingsspiel Bermuda vorstellen.

Spielregeln

Das Spiel heißt Bermuda, weil es darum geht, nach Schiffen zu suchen. Im Bermudadreieck soll es, wie man hört, schon ein paar Verluste gegeben haben. Nun soll es darum gehen, die verlorenen Schiffe wiederzufinden. Es steht eine Matrix von Peilbojen zur Verfügung. Vier Schiffe werden vermisst. Sie haben im Sturm jeweils an einer Peilboje festgemacht. Aber an welcher?

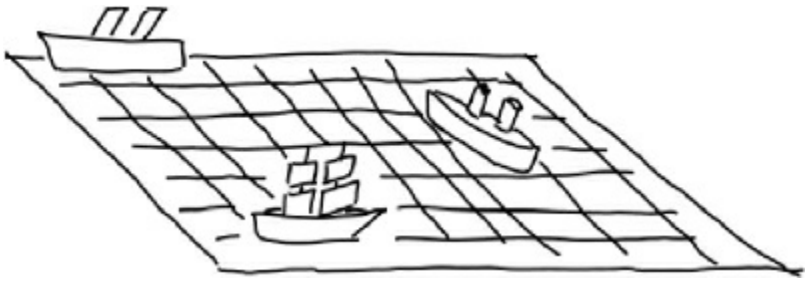


Abbildung 8.1: Das Spielfeld von Bermuda

Eine Peilboje kann melden, dass ein Schiff angelegt hat. Dann kann sie aber nicht mehr peilen. Ansonsten peilt sie in alle acht Richtungen (einschließlich der Diagonalen) und ermittelt, in wie vielen Richtungen Schiffe zu entdecken sind. Zwei Schiffe hintereinander erscheinen durch den Peilstrahl wie eines.

Das Programm muss also folgende Aufgaben erfüllen:

- ✔ Es gibt ein rechteckiges Spielfeld, in dem das Programm die Schiffe versteckt.

- Der Spieler kann nun ein Feld aufrufen und erfährt, ob dort ein Schiff angedockt liegt. So weit ähnelt es dem bekannten Spiel »Schiffe versenken«.
- Falls an der angefragten Stelle kein Schiff ist, meldet das Spiel, in wie viele Richtungen Schiffe gepeilt werden können. Als Richtung gelten neben oben, unten, links und rechts auch die Diagonalen. Es gibt also acht Richtungen, allerdings nur vier Schiffe.

  Schauen Sie sich Abbildung 8.2 an. Der Peilsender funkt in alle acht Richtungen. In drei Richtungen ist ein Schiff zu sehen. Also meldet das Spiel hier eine 3.

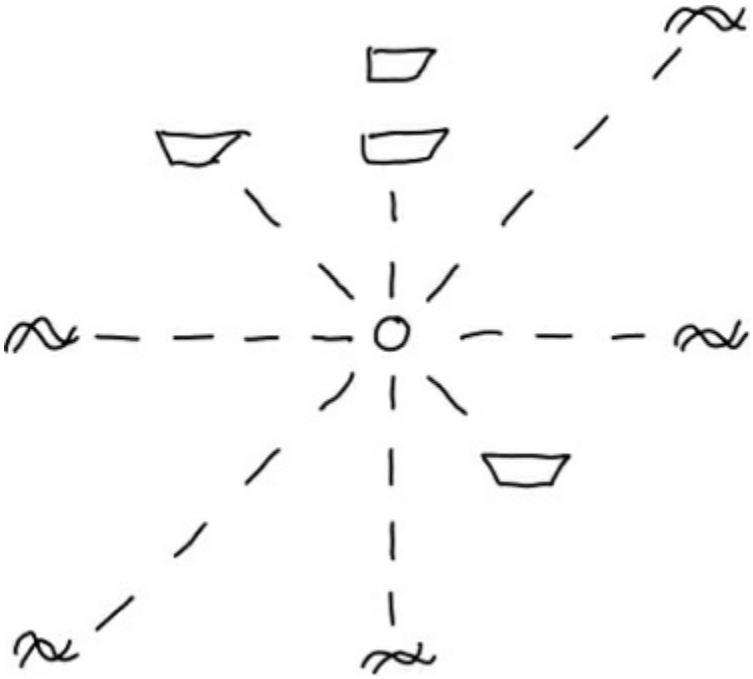


Abbildung 8.2: Richtungen abfragen in Bermuda

Aus den Ergebnissen kann der Spieler die Schnittpunkte ermitteln und so abschätzen, wo sich Schiffe befinden. Durch geschickte Fragen lässt sich das Auffinden der Schiffe beschleunigen.

Arrays für Spielfeld und Schiffe

Endlich können Sie Ihre Erkenntnisse über Arrays aus dem Kapitel 7 anwenden, einmal ein zweidimensionales Array für das Spielfeld und dann normale Arrays für die Schiffe.

Das Spielfeld

Für das Spielfeld benötigen wir ein zweidimensionales Array. Dieses wird im Spielverlauf immer wieder angezeigt. Es enthält also nicht die Schiffe, sondern die bisherigen Anfragen des Spielers. Das Spielfeld ist zu Anfang folglich leer und nimmt bei jeder Peilung ein neues Zeichen auf.

- ✔ Gefundene Schiffe können durch ein großes X repräsentiert werden.
- ✔ Die Zahl der Peilungen kann maximal einstellig sein und darum auch durch eine Ziffer repräsentiert werden.
- ✔ Jedes Feld, das noch nicht angefragt wurde, enthält Wasser. Wasser könnte natürlich sehr hübsch durch eine Welle angezeigt werden. Aber viel praktischer ist ein Pluszeichen, weil das Gesamtbild aus lauter Pluszeichen eine Art Raster bildet, das beim Suchen der Schiffe hilft.

Listing 8.1 zeigt die Definition des Spielfelds und dessen Vorbesetzung mit Pluszeichen.

```
final static char SCHIFF = 'X';
final static int MAXX = 9;
final static int MAXY = 7;
// ...
char[][] feld = new char[MAXX][MAXY];
// ...
for (int x = 0; x < MAXX; x++) {
    for (int y = 0; y < MAXY; y++) {
        feld[x][y] = '+';
    }
}
```

Listing 8.1: Das Spielfeld von Bermuda

Die Schiffe

Es gibt vier Schiffe. Das riecht nach einem Array mit vier Elementen. Aber was wollen wir eigentlich über die Schiffe speichern?

- ✔ Die Koordinaten eines Schiffes bestehen aus der X-Position und der Y-Position. Beides sind ganze Zahlen. Also brauchen wir je ein Array für X und ein Array für Y.
- ✔ Im Laufe des Spiels wird irgendwann ein Schiff gefunden. Welche Schiffe bereits gefunden wurden, müssen wir uns merken, damit wir feststellen können, ob alle Schiffe gefunden sind, und damit das Spiel beendet ist.

Die Arrays werden folgendermaßen angelegt:

```
final static int MAXSCHIFF = 4;
// ...
int schiffX[] = new int[MAXSCHIFF];
int schiffY[] = new int[MAXSCHIFF];
boolean schiffGefunden[] = new boolean[MAXSCHIFF];
```

Listing 8.2: Die Schiffe

Ein Ausflug in die Zufallsermittlung

Nun müssen die Schiffe zufällig auf dem Spielfeld verteilt werden. Dazu benötigen wir eine Zufallsfunktion. Die wird aber erst bei den Mathematik-Bibliotheken von Java vorgestellt. Was soll's: Wir schummeln einfach ein wenig und ich verrate Ihnen hier schon, wie Sie an eine Zufallszahl kommen.

- Zu Anfang des Programms müssen Sie erklären, dass Sie den Zufall, der englisch »random« heißt, importieren wollen.
  ```
  import java.util.Random;
  ```
- Später können Sie eine Variable vom Typ `Random` bilden, aus der Sie anschließend zufällige Werte herausquetschen können.
  ```
  Random zufall = new Random();
  ```
- Sie erhalten über die Methode `nextInt` einen zufälligen, ganzzahligen Wert. Für die X-Position soll der Wert aber zwischen 0 und `MAXX` liegen. Dazu verwenden Sie einfach die Modulo-Rechnung, bilden also den Rest.
  ```
  zufall.nextInt() % MAXX
  ```
- Allerdings könnte der Zufallswert auch negativ sein. Um das Minuszeichen zu entfernen, verwenden Sie die Methode `Math.abs`. Das Gesamtkunstwerk hat damit folgendes Aussehen:
  ```
  int x = Math.abs(zufall.nextInt()) % MAXX;
  ```

Nun sind wir bereit, die Schiffe zu verteilen. Die Vorbesetzung integriert die Zufallserzeugung, muss allerdings noch einmal kontrollieren, ob nicht zwei Schiffe auf der gleichen Position landen. Darum wird beim Setzen eines Schiffes eine weitere Schleife gestartet, in der geprüft wird, ob auf der neuen Position bereits ein Schiff sitzt.

```
Random zufall = new Random();
for (int i = 0; i < MAXSCHIFF; i++) {
    boolean kollidieren = false;
    do {
        int x = Math.abs(zufall.nextInt()) % MAXX;
        int y = Math.abs(zufall.nextInt()) % MAXY;
        schiffX[i] = x;
        schiffY[i] = y;
        kollidieren = false;
        for (int j = 0; j < i; j++) {
            if (schiffX[j] == x && schiffY[j] == y) {
                kollidieren = true;
            }
        }
    } while (kollidieren);
    schiffGefunden[i] = false;
}
```

Listing 8.3: Die Schiffe werden zufällig gesetzt.

Der Spielablauf

Das Feld ist vorbereitet, die Schiffe positioniert. Prinzipiell können wir spielen. Damit die Übersicht nicht verloren geht, zeige ich Ihnen zunächst den Spielablauf.

- ✔ Eine große Schleife läuft so lange, wie das Spiel nicht abgebrochen wurde oder alle Schiffe gefunden wurden.
- ✔ Das Spielfeld wird gezeigt, der Spieler gibt seinen Tipp ein.
- ✔ Der Tipp wird durch die Suche geschickt.
- ✔ Ist die Eingabe kein gültiger Wert, handelt es sich wohl um einen Spielabbruch und das Programm wird beendet.

In Java sieht das alles so aus:

```
char[][] feld = new char[MAXX][MAXY];
int schiffX[] = new int[MAXSCHIFF];
int schiffY[] = new int[MAXSCHIFF];
boolean schiffGefunden[] = new boolean[MAXSCHIFF];
init(feld, schiffX, schiffY, schiffGefunden);
int[] pos;
boolean spielAbbruch = false;
while (!spielAbbruch &&
        !sindAlleSchiffeGefunden(schiffGefunden)) {
    zeige(feld);
    pos = eingabe();
    char zeichen = suche(feld, pos[0], pos[1],
            schiffGefunden, schiffX, schiffY);
    if (pos[0]<0 || pos[1]<0) {
        spielAbbruch = true;
    } else {
        feld[pos[0]][pos[1]] = zeichen;
    }
}
if (spielAbbruch) {
    System.out.println("Hier waren die Schiffe");
    enttarneSchiffe(feld, schiffX, schiffY);
    zeige(feld);
} else {
    System.out.println("Gewonnen");
}
```

Listing 8.4: Der Rahmen für das Spiel

Damit sich das Programm nicht in Details verliert, wurden hier Methodenaufrufe eingesetzt, die durch ihre Namen verraten, was darin passieren soll. Zunächst wird Java auf die nicht aufgelösten Methoden mit Fehlermeldungen reagieren. Also müssen wir sie Schritt für Schritt implementieren.

Eine Methode für die Peilung

Um herauszufinden, in wie vielen Richtungen Schiffe zu sehen sind, verfolgt das Programm die gleiche Taktik wie der Peilstrahl. Es geht vom Ausgangspunkt in alle acht Richtungen und schaut, ob es zuerst auf ein Schiff oder den Rand des Spielfelds trifft. Jeder Schiffstreffer wird gezählt.

Wir stellen uns also auf den Ausgangspunkt und wandern nach Westen, dem Sonnenuntergang entgegen. Das heißt, wir ziehen von der x-Koordinate immer eine 1 ab, bis sie 0 ist. Das bedeutet, dass wir den Rand erreicht haben. Wenn wir vorher auf ein Schiff stoßen, ist die Suche nach Westen schon vorher beendet. Wenn wir

nach Osten wollen, wird auf die x-Koordinate eine 1 aufaddiert. Beim Marsch nach Süden wird die y-Koordinate um 1 erhöht und nach Norden wird die y-Koordinate um je 1 erniedrigt. Bei der Diagonalen nach Südosten wird x und y jeweils um 1 erhöht.


Je nach Richtung gibt es also eine Schrittdifferenz von -1, 0 oder 1 auf der x- und der y-Koordinate.

Die Richtungen werden am einfachsten durch zwei Schleifen bestimmt, die jeweils von -1 bis 1 laufen. Diese Werte werden dann als Differenz auf die Positionsvariablen `x` und `y` aufaddiert. Allerdings muss beachtet werden, dass nicht beide Differenzen gleich 0 sind. Sonst bleibt der Suchstrahl stehen und erreicht den Rand nie. Das prüfen Sie durch ein `if`.

```
static char suche(char[][] feld, int posX, int posY,
        boolean schiffGefunden[],
        int schiffX[], int schiffY[]) {
    char fund = ' ';
    int index = getSchiffIndex(posX, posY, schiffX, schiffY);
    if (index >= 0) {
        schiffGefunden[index] = true;
        fund = SCHIFF;
    }
    int funde = 0;
    for (int diffX = -1; diffX <= 1; diffX++) {
        for (int diffY = -1; diffY <= 1; diffY++) {
            if (diffX != 0 || diffY != 0) {
                int x = posX;
                int y = posY;
                int zahl = 0;
                while (zahl==0 && x>=0 && x<MAXX
                        && y>=0 && y<MAXY) {
                    x += diffX;
                    y += diffY;
                    if (getSchiffIndex(x, y,
                            schiffX, schiffY)>=0)
                        zahl = 1;
                }
                funde += zahl;
            }
        }
    }
    fund = (char) ('0' + funde);
    return fund;
}
```

Listing 8.5: Die Suche nach einem Schiff

Der Aufruf der Methode `getSchiffIndex` liefert den Index des Schiffes, das getroffen wurde, oder -1, wenn dort kein Schiff ist. Die Implementierung ist trivial. Sie sehen sie zum Schluss in der Zusammenfassung.

Benutzereingabe

Die Benutzereingabe ist etwas spartanisch, schließlich wollen wir das Spiel ja später eh mit einer grafischen Oberfläche ausstatten. Darum muss der Spieler ein wenig Disziplin zeigen. Das Programm erwartet die Koordinaten als eine Kombination aus einem Großbuchstaben, dem direkt eine Zahl folgt. Natürlich muss beides in den Spielfeldgrenzen sein. Ist irgend etwas nicht korrekt, hat das

Programm genug und interpretiert die Eingabe als Bitte um Abbruch des Programms.

Die Eingabe soll als Koordinatenpaar zurückgegeben werden. Da Java nicht in der Lage ist, zwei Werte als Rückgabe einer Funktion zu verwenden, bedienen wir uns eines schamlosen Tricks. Wir stecken die X- und Y-Koordinate einfach in ein Integer-Array von zwei Werten.

```
static int[] eingeben() {
    int[] pos = new int[2];
    pos[0] = -1;
    pos[1] = -1;
    System.out.println("\nPosition eingeben ");
    Scanner sc = new Scanner(System.in);
    String eingabe = sc.next();
    if (eingabe.length() > 1) {
        char cx = eingabe.charAt(0);
        if (cx >= 'A' && cx <= 'A' + MAXX) {
            pos[0] = (int) (cx - 'A');
        }
        char cy = eingabe.charAt(1);
        if (cy >= '1' && cy <= '1' + MAXX) {
            pos[1] = (int) (cy - '1');
        }
    }
    return pos;
}
```

Listing 8.6: Eine einfache Koordinateneingabe

Die Eingabe erfolgt über einen Scanner. Dieser wird auf `System.in` gelegt und so kann über dessen Methode `next` eine Zeile von der Tastatur eingelesen und einer String-Variablen zugeführt werden.

Anschließend wird vom ersten Zeichen der Buchstabe A abgezogen. Es ergibt sich eine Zahl zwischen 0 und 8, die die Spalte beschreibt. Das zweite Zeichen ist eine Ziffer, von der der Buchstabe ‘1’ abgezogen wird, so dass eine Zahl von 0 bis 6 die Zeile beschreibt.

Beides wird in ein Array gesteckt und an den Aufrufer zurückgegeben. Natürlich können Sie die Eingabe noch erweitern, indem Sie Leerzeichen ignorieren oder auch Kleinbuchstaben als Koordinaten erlauben.

Das komplette Listing

Nun sehen Sie alles in voller Schönheit. Das Spiel ist zwar optisch nicht der Brüller, aber man kann spielen. Ob das Spiel Spaß macht, ist natürlich Geschmackssache. Aber es ist ein nicht triviales Computerspiel, das aus den Elementen zusammengesetzt ist, die Sie bisher kennengelernt haben.

```
import java.util.Random;
import java.util.Scanner;

public class Bermuda {

    final static int MAXX = 9;
    final static int MAXY = 7;
    final static char SCHIFF = 'X';
    final static int MAXSCHIFF = 4;

    static void init(char[][] feld,
            int schiffX[], int schiffY[],
            boolean schiffGefunden[]) {
        // Spielfeld vorbereiten
        for (int x = 0; x < MAXX; x++) {
            for (int y = 0; y < MAXY; y++) {
                feld[x][y] = '+';
            }
        }
        // Schiffe vorbereiten und verteilen
        Random zufall = new Random();
        for (int i = 0; i < MAXSCHIFF; i++) {
            boolean kollidieren = false;
            do {
                int x = Math.abs(zufall.nextInt()) % MAXX;
                int y = Math.abs(zufall.nextInt()) % MAXY;
                schiffX[i] = x;
                schiffY[i] = y;
                kollidieren = false;
                for (int j = 0; j < i; j++) {
                    if (schiffX[j] == x && schiffY[j] == y) {
                        kollidieren = true;
                    }
                }
            } while (kollidieren);
            schiffGefunden[i] = false;
        }
    }

    static boolean sindAlleSchiffeGefunden(
            boolean schiffGefunden[]) {
        boolean gefunden = true;
        for (int i = 0; i < MAXSCHIFF; i++) {
            if (schiffGefunden[i] == false) {
                gefunden = false;
            }
        }
```

```
    return gefunden;
}

/**
 * Liefert den Index des Schiffes an dieser Position.
 * Ist dort nichts, gibt die Methode -1 zurück.
 */
static int getSchiffIndex(int x, int y,
        int[] schiffX, int[] schiffY) {
    int index = -1;
    int i = 0;
    while(index<0 && i<MAXSCHIFF) {
        if (schiffX[i]==x && schiffY[i]==y) {
            index = i;
        }
        i++;
    }
    return index;
}
```

```
/**
 * Ermittle die Anzahl der sichtbaren Schiffe durch
 * Ablaufen der Richtungen
 * @param feld
 * @param posX
 * @param posY
 * @return Richtungen oder Schiff als char
 */
static char suche(char[][] feld, int posX, int posY,
        boolean schiffGefunden[],
        int schiffX[], int schiffY[]) {
    char fund = ' ';
    int funde = 0;
    int index = getSchiffIndex(posX, posY, schiffX, schiffY);
    if (index >= 0) {
        schiffGefunden[index] = true;
        fund = SCHIFF;
    } else {
        for (int diffX = -1; diffX <= 1; diffX++) {
            for (int diffY = -1; diffY <= 1; diffY++) {
                if (diffX != 0 || diffY != 0) {
                    int x = posX;
                    int y = posY;
                    int zahl = 0;
                    while (zahl==0 && x>=0 && x<MAXX
                            && y>=0 && y<MAXY) {
                        x += diffX;
                        y += diffY;
                        if (getSchiffIndex(x, y,
                                schiffX, schiffY)>=0)
                            zahl = 1;
                    }
                    funde += zahl;
                }
            }
        }
        fund = (char) ('0' + funde);
    }
    return fund;
}

static void enttarneSchiffe(char[][] feld,
        int schiffX[], int schiffY[]) {
    for (int i=0; i<MAXSCHIFF; i++) {
        feld[schiffX[i]][schiffY[i]] = SCHIFF;
    }
}
```

```
// -------------- Interaktion mit dem Anwender ------------

/**
 * Zeigt das Spielfeld als Zeichenmatrix
 * @param feld - Das Spielfeld muss initialisiert sein
 */
static void zeige(char[][] feld) {
    System.out.println();
    System.out.print("  ");
    for (int x = 0; x < MAXX; x++) {
        System.out.print(" " + (char) ('A' + x));
    }
    // Zuerst zeilenweise
    for (int y = 0; y < MAXY; y++) {
        System.out.println();
        System.out.print(""+(y+1)+" ");
        for (int x = 0; x < MAXX; x++) {
            System.out.print(" " + feld[x][y]);
        }
    }
}

/**
 * Bitte den Anwender um die Position
 * @return int-Array mit x und y
 */
static int[] eingabe() {
    int[] pos = new int[2];
    pos[0] = -1;
    pos[1] = -1;
    System.out.println("\nPosition eingeben ");
    Scanner sc = new Scanner(System.in);
    String eingabe = sc.next();
    if (eingabe.length() > 1) {
        char cx = eingabe.charAt(0);
        if (cx >= 'A' && cx <= 'A' + MAXX) {
            pos[0] = (int) (cx - 'A');
        }
        char cy = eingabe.charAt(1);
        if (cy >= '1' && cy <= '1' + MAXX) {
            pos[1] = (int) (cy - '1');
        }
    }
    return pos;
}
```

```
    public static void main(String[] args) {
        char[][] feld = new char[MAXX][MAXY];
        int schiffX[] = new int[MAXSCHIFF];
        int schiffY[] = new int[MAXSCHIFF];
        boolean schiffGefunden[] = new boolean[MAXSCHIFF];
        init(feld, schiffX, schiffY, schiffGefunden);
        int[] pos;
        boolean spielAbbruch = false;
        while (!spielAbbruch &&
                !sindAlleSchiffeGefunden(schiffGefunden)) {
            zeige(feld);
            pos = eingabe();
            char zeichen = suche(feld, pos[0], pos[1],
                    schiffGefunden, schiffX, schiffY);
            if (pos[0]<0 || pos[1]<0) {
                spielAbbruch = true;
            } else {
                feld[pos[0]][pos[1]] = zeichen;
            }
        }
        if (spielAbbruch) {
            System.out.println("Hier waren die Schiffe");
            enttarneSchiffe(feld, schiffX, schiffY);
            zeige(feld);
        } else {
            System.out.println("Gewonnen");
        }
    }
}
```

Listing 8.7: Das Programm Bermuda in voller Länge

Das Spiel läuft, aber wir werden es noch ein paar Mal neu programmieren, damit Sie andere Java-Techniken kennenlernen.

Teil II

Objektorientiert und modular



Kapitel 9

Klasse Objekte

IN DIESEM KAPITEL

- Klassen haben Methoden und Attribute.
- Referenzen sind keine Objekte.
- Private und öffentliche Bereiche
- Konstruktoren
- Aufzählungen

Ohne Objektorientierung kann sich ein ordentlicher Programmierer heutzutage nirgendwo mehr blicken lassen. Es ist wichtig, Klasse zu zeigen und Referenzen zu haben. Objekte müssen erst einmal `new` sein, um später gebraucht werden zu dürfen.

Böse Zungen behaupten, die objektorientierte Programmierung, manchmal auch liebevoll OOP genannt, solle verhindern, dass noch einmal ein BASIC-Programmierer wie Bill Gates Milliarden scheffelt und damit den Software-Markt beherrscht. Das ist zwar ein angenehmer Nebeneffekt, aber tatsächlich ist die OOP pure Notwehr der Programmierer gegen die steigende Komplexität großer Projekte.

Die wenigsten Programmieraufgaben lassen sich auf einer DIN-A4-Seite zusammenfassen, wie unser Programm zur Errechnung des größten gemeinsamen Teilers. Darum wird die nächste Version von Windows oder World of Warcraft auch nicht von einem einzelnen Programmierer erstellt, sondern von einem ganzen Rudel von Software-Entwicklern. Das Problem dabei ist, dass jeder seine eigene Aufgabe zugeteilt bekommt und später alles zusammenpassen muss.

Es geht also darum, das Programm in viele kleine Teilprogramme zu zerlegen, die man separat bearbeiten kann und die hinterher auch wieder zusammenpassen. Solche Teilprogramme sind die Klassen und die Technik der Zerlegung bezeichnet man als objektorientierte Programmierung.

Ein Beispiel für objektorientierte Programmierung

Wie funktioniert die Zerlegung der größeren Programme durch die objektorientierte

Programmierung? Stellen Sie sich vor, Sie müssten die Software-Unterstützung für eine öffentliche Bibliothek schreiben.

Lange Zeit wurde das EVA-Prinzip (Eingabe, Verarbeitung, Ausgabe) bei der Erstellung von Programmen verfolgt. Wenn Sie mit den Eingaben einer Ausleihe anfangen, verlieren Sie sich sofort in Details. Welches Buch wurde ausgeliehen? Stammt das Buch vielleicht aus der Präsenzbibliothek und darf gar nicht verliehen werden? Ist der Ausleiher ein Dozent oder Student? Hat der Ausleiher vielleicht noch Bücher, die er zurückgeben muss, bevor er neue Bücher ausleiht? Alle Fragen ziehen sich fadenartig durch das Problem und vor allem finden sich auf diese Weise keine Möglichkeiten, die Aufgabe auf mehrere Programmierer zu verteilen.

Die objektorierentierte Programmierung betrachtet nicht die Funktionsabläufe, sondern isoliert zunächst die Objekte. Bei einer Bibliothek gibt es da:

- ✔ Bücher
- ✔ Regale
- ✔ Kunden, also Menschen, die Bücher ausleihen
- ✔ Mitarbeiter, also Menschen, die den Betrieb aufrecht erhalten

Die OOP schaut sich dann noch Kunden und Mitarbeiter genauer an und sagt, dass beide Arten von Objekten Spezialfälle von Personen sind.

Diese Objekte werden durch ihre Klasse bestimmt. Die *Klasse* legt zunächst fest, welche Attribute ein Objekt hat. Die *Attribute* sind Variablen, die zum Objekt gehören und den Status und die Eigenschaften des Objekts speichern. Bei einem Buch sind dies naheliegenderweise Titel, Erscheinungsjahr und Autor, aber auch, ob es zur Präsenzbibliothek gehört oder ausleihbar ist. Ein Attribut beschreibt, in welches Regal es gehört, ein anderes, bis wann es derzeit verliehen ist.

Die verschiedenen Objekte haben unterschiedliche Funktionalitäten. Ein Buch kann verliehen werden, ein Mitarbeiter hoffentlich nicht und ein Kunde schon gar nicht. Auch die Regale bleiben hier. Diese Funktionalitäten werden im objektorientierten Sprachgebrauch *Methoden* genannt.



Die Klasse stellt die Bauanleitung für Objekte dar. Attribute enthalten die Eigenschaften, den Status eines Objekts. Man könnte sie auch als Objektvariablen bezeichnen. Methoden beschreiben die Aktivitäten oder Funktionen des Objekts.

Nicht alle Details der Objekte sind für die Modellierung relevant. Für die Steuerabrechnung mag es interessant sein, ob ein Mitarbeiter verheiratet ist. Bei einem Kunden ist das unwichtig. Eine Bibliothek ist ja keine Partnervermittlung. Spinnen Sie das Beispiel weiter, und Sie werden feststellen, dass sich das Gewusel

langsam lichtet und sich in klar voneinander abgegrenzte Klassen aufteilen lässt.

Natürlich gibt es Abhängigkeiten zwischen den Objekten. So kann ein Buch an einen Kunden verliehen sein. Aber der Programmierer, der sich um das Buch kümmert, kann den Kunden als abstraktes Objekt behandeln.

Eine Klasse im Selbstbau

Schnappen Sie sich Ihren Computer, wir bauen eine Klasse. Sie können nun zu Recht einwenden, dass Sie dies ja die ganze Zeit schon getan haben. Richtig! Jedes Java-Programm besteht ja aus mindestens einer Klasse. Aber wir werden nun eine Klasse bilden, von der wir mehrere Objekte erstellen wollen.

Eine neue Klasse bedeutet, dass wir eine neue Datei erstellen. Mit einer Klasse erstellen wir oftmals einen neuen Datentyp, der aus bekannten Datentypen zusammengesetzt wird. Ein typischer Fall ist das Datum. Es besteht aus Tag, Monat und Jahr. Alle drei sind ganze Zahlen, also typische `int`-Variablen. Die Bauanleitung, also die Klasse für ein Datum, sieht in Java in seiner einfachsten Form etwa so aus:

```
public class Datum {
    int jahr;
    int monat;
    int tag;
};
```

Listing 9.1: Das Datum hat drei Attribute.

Java schreibt vor, dass jede öffentliche Klasse in einer eigenen Datei steht, deren Name der Name der Klasse ist. Wir würden darum die Datei *Datum.java* anlegen und hier die Klasse `Datum` definieren. Sie muss separat kompiliert werden.



Wenn Sie unter Eclipse eine neue Klasse zu einem Projekt anlegen wollen, klicken Sie mit der rechten Maustaste Ihr Projekt an und wählen Sie aus dem Menü den Punkt New|Class. Es erscheint ein Dialog, in dem Sie den Namen der Klasse eingeben.

Wenn Sie mit dem JDK auf der Kommandozeile arbeiten, reicht es, die Datei *Datum.java* in das gleiche Verzeichnis zu legen, in dem auch die anderen Dateien der Anwendung liegen, und dort einmal den Compiler für die Datei aufzurufen.

```
javac Datum.java
```

Es entsteht ein Kompilat unter dem Dateinamen *Datum.class*. Sobald eine andere Klasse des Projekts die Klasse `Datum` verwendet, wird es automatisch nach der Datei *Datum.class* suchen.


Der Klassenname muss als Dateiname übernommen werden. Der Quelltext bekommt *.java* angehängt, Das Kompilat erhält als Extension *.class*.

Der pflegliche Umgang mit Objekten

Die Klasse ist definiert. Im nächsten Schritt erstellen wir eine Klasse, die die neu geschaffene Klasse `Datum` verwendet und somit testet. Dazu legen Sie für das gleiche Projekt eine neue Klasse namens `TestDatum` an, die dann auch die Methode `main` enthält, die die Klasse `Datum` ausprobiert. In der Methode `main` legen Sie eine Variable vom Typ `Datum` an.

```
class TestDatum {
    static public void main(String[] args) {
        Datum meinGeburtstag;
    }
}
```

Listing 9.2: Variable des selbst gebauten Typs

Doch Vorsicht! Eine Variable vom Typ einer Klasse ist nicht das Objekt selbst, sondern eine Referenz auf das Objekt! Eine Referenz ist wie der Handgriff an einem Koffer. Ohne diesen verlieren Sie Ihren Koffer. Aber der Handgriff selbst enthält nichts. Das Objekt gibt es noch nicht. Das muss erst noch erzeugt werden. Und dazu gibt es den Befehl `new`.

```
class TestDatum {
    static public void main(String[] args) {
        Datum meinGeburtstag;
        meinGeburtstag = new Datum();
    }
}
```

Listing 9.3: Ein Objekt wird erzeugt.

Der Befehl `new` fordert einen Platz im Hochregallager für Objekte an und gibt dem Programmierer eine Referenz auf den Ort zurück, wo sein Objekt liegt. Die Variable ist die Referenz auf das Objekt, das ganz woanders liegt.

Hinter dem Befehl `new` steht noch einmal der Klassenname des erzeugten Objekts mit runden Klammern. Das sieht ganz ähnlich aus wie eine Methode. Und tatsächlich verbirgt sich dahinter ein Methodenaufruf, durch den das Objekt initialisiert wird, wie Sie bald noch sehen werden.

Die folgende Anweisung definiert eine Referenz namens `meinGeburtstag` und erzeugt durch den Befehl `new` auch gleich das dazugehörige Objekt.

```
Datum meinGeburtstag = new Datum();
 (1)       (2)       (3)(4)  (5) (6)
```

Schauen wir uns die Zeile im Detail an:

1. Zunächst wird die Klasse der Variablen genannt, die die Referenz auf das Objekt wird.
2. Es folgt der Name der Variablen, also der Referenz.
3. Die Zuweisung gibt die neu erzeugte Referenz an die Variable.
4. Der Befehl `new` fordert den Speicher an, der für die nachfolgende Typbeschreibung benötigt wird.
5. Mit `Datum` wird die Klasse bezeichnet, die das neue Objekt beschreibt.
6. Die Klammern sind wichtig. Es ist gar nicht falsch, wenn Sie diese als einen Bestandteil eines Methodenaufrufs identifizieren. Darauf kommen wir noch zurück.

Zugriff über die Referenz

Über die Referenz greifen Sie auch auf die Bestandteile des Objekts zu. Um den Umgang mit Referenz und Objekt zu demonstrieren, verwenden wir den Geburtstag von Bernhard Grzimek am 24.4.1909. Unter den Zoologen wurde Grzimek durch die Entdeckung der Steinlaus weltberühmt.

Für Grzimeks Geburtstag legen wir – wie oben gezeigt – ein Objekt an. Die Referenzvariable nennen wir `grzimeksGeburtstag`.

```
Datum grzimeksGeburtstag = new Datum();
```

Über die Referenz erreichen Sie die Attribute des Objekts. Tragen wir gerade mal Grzimeks Geburtstag ein. Sie kennen ihn sicher auch auswendig.

```
grzimeksGeburtstag.jahr  = 1909;
grzimeksGeburtstag.monat = 4;
grzimeksGeburtstag.tag   = 24;
```

Listing 9.4: Besetzung der Attribute



Die Zugriffe auf die Objektelemente darf erst erfolgen, wenn ein Objekt durch `new` angelegt wurde. Ansonsten droht ein Programmabsturz. Java kommentiert das mit einer *NullPointerException*. Wenn Ihrem Programm also mit dieser Meldung die Spielerlaubnis entzogen wird, können Sie darauf wetten, dass irgendwo ein `new` fehlt.

Die Klassen haben so ihre Methoden

Wenn Sie schon mit Software-Entwicklung zu tun hatten, kennen Sie vielleicht das Phänomen, dass der Auslieferungstermin sich eher danach richtet, wann das Software-Haus dringend Geld braucht, und nicht, wann die Software einsatzfähig

ist. Wenn so ein Termin platzt, wird ein neuer vereinbart und der ist dann in der Regel einen Monat später.

Um das abzubilden, muss der Monat um 1 erhöht werden. Sollte er schon bei 12 sein, muss er auf 1 gesetzt werden und die Jahreszahl erhöht werden. Da dieser Vorgang immer wieder gleich abläuft, schreit er förmlich nach einer Methode. Und da das immer auf den Attributen des Termins abläuft, sollte die Methode auch gleich zur Klasse hinzugefügt werden.

```
public class Datum {
    int tag, monat, jahr;
    public void verschiebe() {
        monat++;
        if (monat>12) {
            monat = 1;
            jahr++;
        }
    }
}
```

Listing 9.5: Terminverschiebung

Die Klasse `Datum` hat neben den Attributen `tag` `monat` und `jahr` nun auch noch eine Methode namens `verschiebe`. Und wie rufen Sie die Methode auf? Ganz genau so, wie Sie auch auf die Attribute zugreifen. Sie stellen den Objektnamen voran, dessen Methode aufgerufen werden soll.

```
public class TestTermin {
    public static void main(String[] args) {
        Datum abgabe = new Datum();
        abgabe.tag = 12;
        abgabe.monat = 12;
        abgabe.jahr = 2012;
        abgabe.verschiebe();
        System.out.print(abgabe.monat);
        System.out.print("/");
        System.out.println(abgabe.jahr);
    }
}
```

Listing 9.6: Terminverschiebung

Der Aufruf von `verschiebe` erfolgt über das Objekt `abgabe`, weil ja schließlich der Termin der Abgabe verschoben werden soll.

Sie könnten ja noch in die Methode `verschiebe` einbauen, dass der Termin in den Sommerferienmonaten Juli und August auf September weiterverschoben wird, oder

prüfen, ob durch das Schieben die Zeit zwischen Weihnachten und Neujahr betroffen ist, und den Termin entsprechend auf die Zeit nach dem 6. Januar weiterverschieben.



Die Methode `verschiebe` verwendet keinen Modifizierer `static`. Diese Methode wirkt auf das jeweilige Objekt und ist damit nicht nur einmal im gesamten Programm, sondern quasi für jedes Objekt anders vorhanden, weil es jeweils auf die Attribute des eigenen Objekts zugreift.

Freundliche Unterstützung von Eclipse

Eine Klasse kann eine Vielzahl von Methoden und Attributen anbieten. Als fleißiger Programmierer können Sie die zwar alle auswendig lernen. Aber wer will das schon? Die integrierten Entwicklungsumgebungen wie Eclipse oder Netbeans helfen Ihnen dabei. Sobald Sie einen Punkt hinter einen Objektnamen setzen, erscheint eine Liste aller Attribute und Methoden der Klasse, wie in Abbildung 9.1 zu sehen.

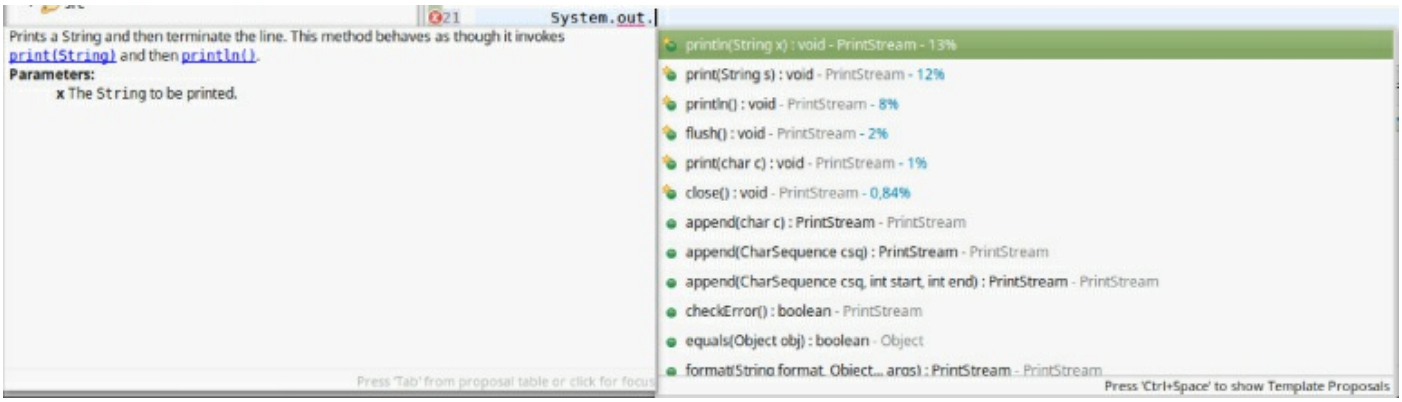


Abbildung 9.1: Eclipse zeigt die Attribute und Methoden eines Objekts.

In der Abbildung wurde der Punkt hinter `System.out` gesetzt. Nun zeigt Eclipse, welche Attribute und Methoden das Objekt `out` beherrscht. Wenn Sie in der Liste ein Element auswählen, erscheinen weitere Informationen. In Abbildung 9.1 wurde `println` ausgewählt und links neben dem Befehl erscheint eine kurze Beschreibung.

Diese Unterstützung gewährt Eclipse auch für Ihre selbst gebauten Klassen. Allerdings werden Sie die kurze Beschreibung vermissen. Das ist eigentlich logisch. Woher soll Eclipse wissen, wie Ihre selbst geschriebene Methode tickt? Das können Sie ändern, indem Sie vor Ihre Methoden einen Javadoc-Kommentar setzen.

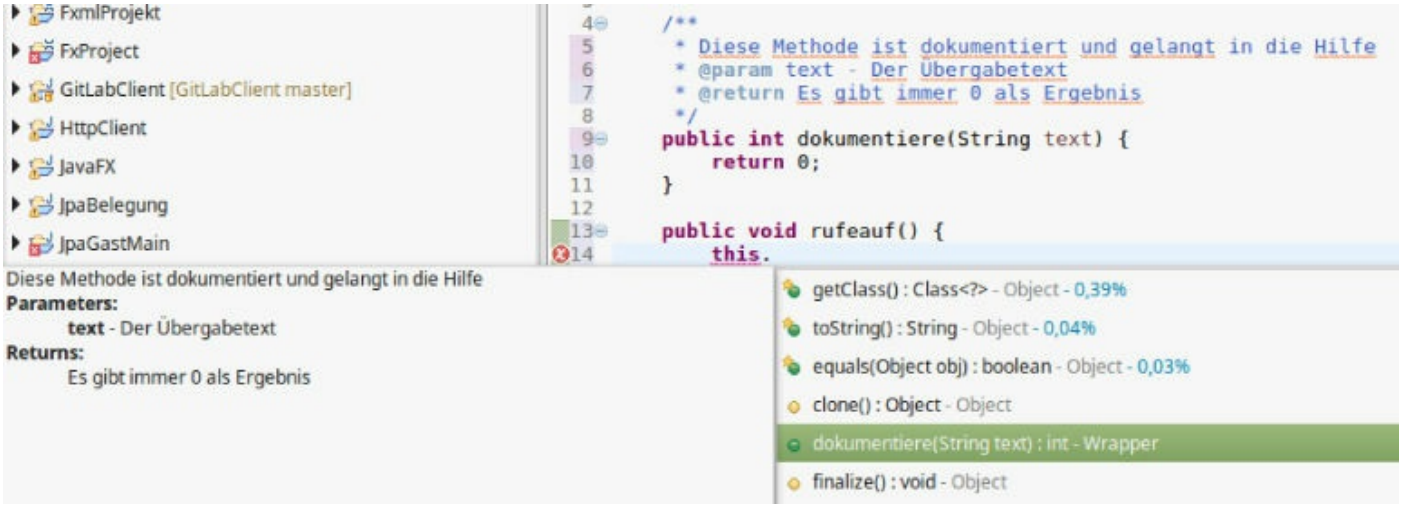


Abbildung 9.2: Eclipse-Hilfe mit Javadoc.

In Abbildung 9.2 sehen Sie, wie der Javadoc-Kommentar der Methode direkt in die Hilfe von Eclipse eingebunden wird. Die Marker `@param` und `@return` werden zur Beschreibung der Parameter und Rückgabewerte verwendet.

Stolperfalle: Kopieren von Referenzen

Dass die Variablen Referenzen sind, hat ein paar Konsequenzen. So führt die Zuweisung einer Referenz eben zu einer Kopie der Referenz und nicht zu einer Kopie eines Objekts. Das bedeutet wiederum, dass anschließend beide Variablen auf dasselbe Objekt referenzieren.

Wir spielen das mal durch. Von der Klasse `Datum` legen wir nun ein Objekt an, das Grzimeks Geburtstag speichern soll.

```
Datum grzimeksGeburtstag = new Datum();
grzimeksGeburtstag.tag   = 24;
grzimeksGeburtstag.monat = 4;
grzimeksGeburtstag.jahr  = 1909;
```

Listing 9.7: Vorsicht Falle!

Dabei fällt mir auf, dass Grzimek genau am gleichen Tag Geburstag hat wie ich! Super. Anstatt alles durchzukaspern, erzeuge ich eine Kopie von Grzimeks Geburtstag und ändere nur mein Geburtsjahr!

```
Datum meinGeburtstag = grzimeksGeburtstag;
meinGeburtstag.jahr =  1960;
```

Listing 9.8: Vorsicht Falle!

Schauen wir noch einmal, wann Grzimek geboren wurde, und wir stellen fest, dass er plötzlich blutjung ist – na ja, zumindest für seine Verhältnisse.

```
System.out.println(grzimeksGeburtstag.jahr);
```

Der Hintergrund ist, dass nicht das Objekt kopiert wurde, sondern nur eine Referenz auf das Objekt. Nach der Zuweisung von `grzimeksGeburtstag` an

`meinGeburtstag` zeigen beide Referenzen auf dasselbe Objekt. Wird im Objekt ein Attribut verändert, ändert es sich auch für die andere Referenz.



Eine Kopie der Referenz bedeutet, dass es einen weiteren Verweis auf dasselbe Objekt gibt.

Eine echte Objektkopie anfertigen

Wenn also das Zuweisen der Referenzen für eine Kopie nicht reicht, muss wohl ein neues Objekt angelegt werden und alle Elemente kopiert werden. Das können wir in einer Methode zusammenfassen, die wir in der Klasse `Datum` anlegen. Wir geben der Methode zum Kopieren einfach mal den Namen `clone`, weil wir ja einen Klon des Objekts erzeugen.

Ein Parameter wird nicht gebraucht, da sich die Methode ja selbst kopiert. Die Rückgabe ist eine Referenz vom Typ `Datum`, das auf das neue Objekt verweist, in dem die Kopie liegt. Nun wissen Sie eigentlich schon alles, was Sie brauchen, um das Programm zu schreiben.

```
public class Datum {
    int jahr;
    int monat;
    int tag;
    public Datum clone() {
        Datum doppel = new Datum();
        doppel.tag = this.tag;
        doppel.monat = this.monat;
        doppel.jahr = this.jahr;
        return doppel;
    }
}
```

Listing 9.9: Datum kopiert sich selbst.

Die Methode `clone` arbeitet in folgenden Schritten:

- Zunächst muss natürlich ein Duplikat entstehen. Also benötigen Sie ein neues Objekt der Klasse `Datum`. Und das erstellen Sie mit dem Befehl `new`.
- Nun werden die Attribute des neuen Objekts mit Kopien der eigenen Attribute gefüllt. Im Listing sehen Sie, dass mit der Referenz `this` auf die eigenen Attribute zugegriffen wird. Sie müssen an dieser Stelle `this` nicht unbedingt verwenden. Java ist sehr verständnisvoll, wenn Sie es weglassen. Solange es keine Zweideutigkeiten gibt, wird der Compiler sowieso die Attribute des eigenen Objekts verwenden.
- Per `return` wird die Referenz des neu angelegten Objekts an den Aufrufer

zurückgegeben.

Da wir die Methode `clone` in der Klasse `Klasse` als nicht `static` definiert haben, wird die Methode über ein Objekt aufgerufen. Mein Geburtstag wird in der folgenden Anweisung tatsächlich kopiert. Ändere ich anschließend mein Geburtsjahr, bleibt Grzimek so alt, wie er vorher war. Probieren Sie es aus!

```
Datum meinGeburtstag = grzimeksGeburtstag.clone();
meinGeburtstag.jahr =  1960;
System.out.println(grzimeksGeburtstag.jahr);
```


Der Methodenname `clone` ist übrigens keine spontane Idee von mir, sondern wird auch von Java verwendet, um Objekte zu kopieren. Was es damit genauer auf sich hat, werden Sie im Laufe dieses Buches noch herausfinden. Es lohnt sich also, auch den Rest zu lesen.

Gleiche Objekte und gleiche Referenzen

Sie ahnen vermutlich schon, dass auch der Vergleich zweier Referenzvariablen mit dem doppelten Gleichheitszeichen nicht feststellt, ob der Inhalt der Objekte gleich ist.


Ein Vergleich `ref1==ref2` ist nur dann `true`, wenn die Referenzen `ref1` und `ref2` auf dasselbe Objekt verweisen.

Wenn das Kopieren von Objekten so elegant durch eine Klassenmethode gelöst werden kann, sollte das beim Vergleich auch möglich sein. Darum schreiben wir nun eine Methode, die wir `equals` nennen und die die Gleichheit zweier Objekte feststellen soll.

Der Rückgabewert der Methode muss natürlich `boolean` sein. Der Parameter ist eine Referenz auf ein anderes Datum, mit dem dieses Objekt verglichen werden soll. Innerhalb der Methode werden die Attribute mit denen des Vergleichsobjekts verglichen.

```
public boolean equals(Datum vgl) {
    return vgl.jahr == this.jahr
           && vgl.monat == this.monat
           && vgl.tag == this.tag;
}
```

Listing 9.10: Die Methode `equals`

Die Methode wird über eines der beiden Objekte aufgerufen und das andere Objekt wird als Parameter übergeben. So vergleichen Sie beispielsweise `meinGeburtstag` mit `grzimeksGeburtstag`:

```
if (meinGeburtstag.equals(grzimeksGeburtstag)) {
```



Umd warum gerade `equals`? Ich bin auch gegen überflüssige Anglizismen, aber stellen Sie sich einfach vor, wir täten es für den netten Marketing-Direktor unserer Firma. Der steht auf so etwas. Okay, Sie haben mich durchschaut. Ich nehme keine Rücksicht auf Marketing-Direktoren. Der englische Namen deutet daraufhin, dass es in Java eine gewisse Tradition für `equals` gibt. Und auch hier gilt, dass Sie diesen Namen bestimmt noch einmal lesen werden.

Objekte als Parameter

Sie können Objekte als Parameter an Methoden übergeben. Auch in diesem Fall werden die übergebenen Daten in die Parameter kopiert. Aber wie bei der Zuweisung gilt bei Objekten, dass nicht der komplette Objektinhalt kopiert wird, sondern nur die Referenz. Das ist bei großen Objekten natürlich ein Effizienzgewinn, hat aber den Nachteil, dass innerhalb der Methode auf das Objekt des Aufrufers referenziert wird. Das bedeutet, dass Manipulationen an übergebenen Objekten plötzlich auf den Aufrufer durchschlagen.

Wir spielen das noch einmal am Geburtstag von Herrn Grzimek durch.

```
public class Grzimek {
    public static void aendereJahr(Datum datum) {
        datum.jahr = 1909;
    }
    public static void main(String[] args) {
        Datum meinGeburtstag = new Datum();
        meinGeburtstag.jahr = 1960;
        aendereJahr(meinGeburtstag);
        System.out.println(meinGeburtstag.jahr);
    }
}
```

Listing 9.11: Manipulationen am Geburtsjahr

Die Referenz null

Eine Objektvariable enthält die Referenz auf ein durch `new` angelegtes Objekt. In manchen Fällen gibt es eine Variable, der noch keine Referenz zugewiesen ist. Solche Variablen besetzt der erfahrene Programmierer mit dem Wert `null`.

```
Datum datum = null;
```

Der Begriff `null` sagt aus, dass noch keine Referenz für diese Variable angelegt wurde. Falls Sie versuchen, nun über die Variable `datum` auf deren Attribute oder deren Methoden zuzugreifen, wird das Programm abstürzen und eine Nullpointer-

Exception melden.

Hier gilt wie so oft: Ein Ende mit Schrecken ist besser als ein Schrecken ohne Ende. So wissen Sie wenigstens, wo das Problem ist. Noch besser ist es natürlich, wenn Sie vor der Verwendung einer unsicheren Referenz wissen, ob diese vielleicht `null` ist:

```
if (datum==null) {
    // Hier stimmt etwas nicht!
} else {
    // Die Variable datum ist gültig.
}
```

Ausgabeunterstützung mit toString

Sie können jede Klasse mit `System.out.print` ausgeben. Dabei wird typischerweise die Adresse ausgegeben, auf die die Referenz verweist. Der Ehrlichkeit halber sei aber darauf hingewiesen, dass Sie damit vermutlich nicht viel anfangen können. Der Grund, warum jede Klasse »ausdruckbar« ist, liegt daran, dass jede Klasse die Methode `toString` implementiert.

Wenn Sie in Ihrer Klasse die Methode `toString` implementieren, dann können Sie bestimmen, wie die Daten Ihrer Klasse auf dem Bildschirm aussehen. Die folgende Methode sorgt dafür, dass das Datum in der Schreibweise erscheint, wie wir es in Deutschland gewohnt sind:

```
public String toString() {
    return ""+tag+"."+monat+"."+jahr;
}
```


Warum das funktioniert, erfahren Sie im Zusammenhang mit dem Überschreiben von Methoden. Bleiben Sie gespannt!

Wie stirbt ein Objekt?

Wenn Sie als Programmierer eine Ressource anfordern, müssen Sie sie in der Regel auch wieder freigeben. Das gilt für beinahe alle Ressourcen in beinahe allen Programmiersprachen. Klar, wer Dreck macht, soll ihn auch wieder wegschaffen.

Es gibt eine Ausnahme: Wenn Sie in Java Speicher mit `new` anfordern, müssen Sie ihn nicht entsorgen. Das erledigt Java für Sie. Der Mechanismus wird *Garbage Collection* genannt. Er sorgt dafür, dass Objekte, auf die es keine Referenzen mehr gibt, rechtzeitig entfernt werden, bevor ein Speicherproblem auftaucht.


Die Garbage Collection räumt übrig gebliebene Objekte für Sie weg. Sie kümmern sich bitte um wichtigere Dinge. Beispielsweise um den Hausmüll.

Da Sie sich nicht mehr darum kümmern müssen, ob jedes Element auch bestimmt wieder freigegeben wird, können Sie unbeschwert Zwischenobjekte anlegen. Das führt zu einem recht sorglosen Programmierstil, der an mancher Stelle zu einer deutlichen Verkürzung und besseren Übersicht der Programme führt.



Der Preis für diesen leichtfüßigen Umgang mit dem Speicher ist eben die Garbage Collection. Wie die echte Müllabfuhr, die auch nicht erst kommt, wenn sich Ihr Dreck meterhoch türmt, kommt auch die Garbage Collection regelmäßig. Allerdings kommt sie nicht nur mittwochvormittags, sondern läuft ständig im Hintergrund als Leichtgewichtsprozess, den man *Thread* nennt. Sollte Ihr Programm voll arbeiten und die ungeteilte Aufmerksamkeit des Computers benötigen, wird die Garbage Collection kurzzeitig auf Eis gelegt. Sie macht sofort weiter, wenn sich wieder alle etwas beruhigt haben. Bevor sich der Prozessor vor Langeweile in der Nase bohrt, weil das Programm beispielsweise gerade auf eine Benutzereingabe wartet, schaut die Garbage Collection, ob sie irgendwo noch ein paar Objekte findet, auf die keine Referenz mehr verweist. Auf diese Weise kommt es nie zu dramatischen Situationen, in denen die Müllberge den Fortlauf des Programms behindern.



Sie können den Jungs von der Müllabfuhr das Leben etwas erleichtern, indem Sie Referenzen auf null setzen, wenn Sie die zugehörigen Objekte nicht mehr benötigen. Dann sehen sie schon früher, dass das Objekt weg kann.

Selbstverweisende Klassen

In einer Klasse können Sie alle möglichen Datentypen ablegen. Das können natürlich auch Referenzen auf andere Objekte sein. Aber besonders verrückt scheint es, wenn man Referenzen auf die eigene Klasse einbaut.



Falls Sie diesen Abschnitt sehr kompliziert finden, können Sie ihn gern überspringen. Sie lernen hier die Möglichkeiten kennen, die Referenzen bieten, und den Umgang mit dynamischen Datenstrukturen. Aber Sie kommen auch erst erst einmal ohne diese Kenntnisse zurecht.

Eine verkettete Liste

Die folgende Klasse hat eine solche Selbstreferenz und Sie werden sehen, dass man damit durchaus sinnvolle Dinge tun kann.

```
class Liste {
    String name;
    Liste next = null;
}
```

Listing 9.12: Eine Klasse verweist auf sich selbst.

Diese Klasse heißt nicht ohne Grund `Liste`, da Sie mit dem Attribut `next` nicht nur auf sich selbst zeigen können, was zugegebenermaßen etwas schwachsinnig wäre. Stattdessen können Sie auf ein anderes Objekt der Klasse `Liste` verweisen.



Abbildung 9.3: Das `next`-Attribut verkettet die Elemente der Liste.

```
class ListenBauer {
    public static void main(String[] args) {
        Liste anfang = new Liste();
        anfang.name = "Alpha";
        anfang.next = new Liste();
        Liste lauf = anfang.next; // letztes Element
        lauf.name = "Beta";
        lauf.next = new Liste();
        lauf = lauf.next; // Einen Schritt weiter
        lauf.name = "Caesar";
        lauf = anfang;
        while (lauf != null) {
            System.out.println(lauf.name);
            lauf = lauf.next;
        }
    }
}
```

Listing 9.13: Wir bauen eine kleine Liste.

Wenn Sie das Programm ausführen, werden Sie sehen, dass Alpha, Beta und Caesar hintereinander auf dem Bildschirm erscheinen. Das Geheimnis besteht darin, dass die drei entstandenen Listenelemente durch ihre `next`-Attribute miteinander verkettet sind.



Wenn Sie mit solchen Listen arbeiten, dürfen Sie niemals die Referenz auf das erste Element verlieren, sonst schwebt die Liste völlig losgelöst durch den Speicherraum. Im Listing heißt diese Variable `anfang`. Oft spricht man von einem *Anker* auf die Liste.

Die verkettete Liste wirkt so ein wenig wie Arrays für Arme. Tatsächlich haben solche Listen Vorteile:

- Die Liste kann beliebig lang werden, ohne dass Sie ihre Länge bereits zu Anfang festlegen müssen.
- Das nachträgliche Löschen eines Listenelements ist extrem flott gemacht. Sie verweisen einfach mit dem Attribut `next` auf das übernächste Element.

```
lauf.next = lauf.next.next;
```

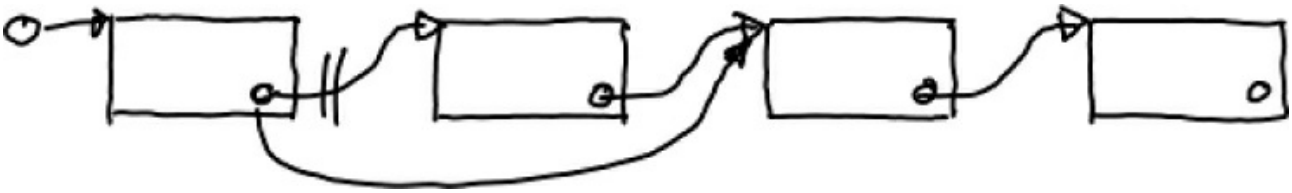


Abbildung 9.4: Das `next`-Attribut wird auf das übernächste Element gelenkt.

- Auch das Einfügen an einer beliebigen Stelle ist sehr effizient. Sie legen ein neues Element an und verketten dann das Element in der Liste. Im folgenden Beispiel ist `lauf` die Referenz auf die Einfügeposition.

```
Liste neu = new Liste();
neu.name = "Neues Element";
neu.next = lauf.next;
lauf.next = neu;
```

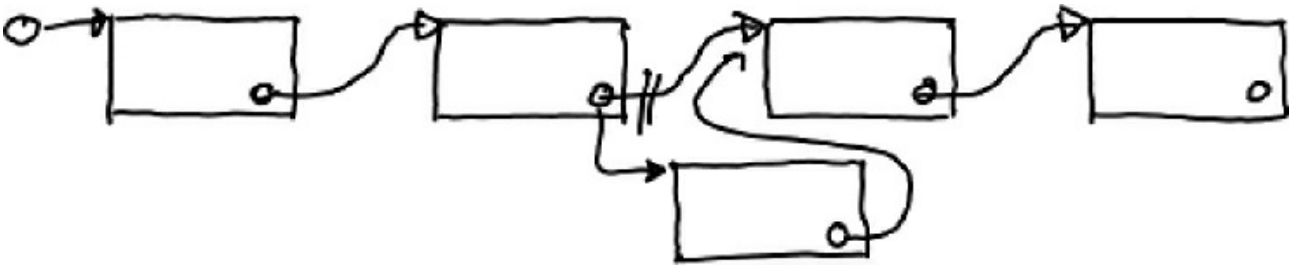


Abbildung 9.5: Das `next`-Attribut wird umgebogen.

Wenn Sie innerhalb eines Arrays Elemente löschen oder einfügen wollen, müssen Sie alle Elemente rechts von dieser Stelle eine Position nach links beziehungsweise nach rechts versetzen. Bei großen Arrays kann das durchaus mit Aufwand verbunden sein.

Kein Licht ohne Schatten. Wenn Sie das 574. Element aus einem Array auslesen wollen, stellen Sie 574 einfach zwischen die rechteckigen Klammern. Da in einem Array alle Elemente dicht auf dicht kuschelig nebeneinander stehen, reicht eine einfache Multiplikation von 574 mit der Elementgröße, damit Java das Element findet. In einer Liste muss man 574 Mal zum jeweils nächsten Element hüpfen. Da werden die Beine schon einmal müde.

Die Kette wird zum Ring

Sie können verkettete Listen auch zu einem Ring zusammenknoten, indem Sie

einfach das letzte Element mit dem ersten verbinden. Auf diese Weise laufen Sie durch Zugriff auf das jeweils nächste Element immer im Kreis.

```
static class Monat {
    String name = "";
    int nr = 0;
    int tage = 0;
    Monat next = null;
}
```

Das demonstriert Listing 9.14 an einem immerwährenden Kalender. Der letzte Monat Dezember wird mit dem Januar verknüpft. Auch bei einem Ring benötigen Sie einen Anker, der allerdings nicht auf dem ersten Element stehen bleiben muss, da ja alle Elemente kreisförmig verbunden sind. Im Beispiel wird dazu die globale Variable `kalender` verwendet.

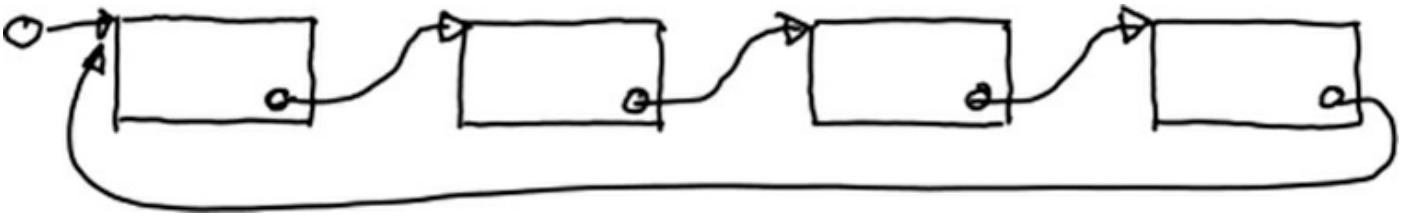


Abbildung 9.6: Das letzte `next`-Attribut baut den Ring.

Die Kalenderliste wird zur Initialisierung in einer Schleife über die beiden Array-Literale gefüllt, die den Monatsnamen und die Anzahl der Tage enthält.

Die Methode `naechster` liefert immer den nächsten Monat zurück. Die Methode `zeige` gibt den Inhalt des Monats aus.

Das Hauptprogramm erlaubt dem Benutzer, wiederholt einzugeben, wie viele Monate er sehen will. Sobald der Benutzer 0 eingibt, wird das Programm beendet.

```
public class Kalender {

    static Monat kalender = null;

    public static void main(String[] args) {
        String[] monatsname = { "Jan", "Feb", "Mrz",
                "Apr", "Mai", "Jun", "Jul", "Aug",
                "Sep", "Okt", "Nov", "Dez" };
        int[] monatstage = { 31, 28, 31, 30, 31, 30,
                31, 31, 30, 31, 30, 31 };
        int monate = monatstage.length;
        kalender = new Monat();
        kalender.name = monatsname[0];
        kalender.tage = monatstage[0];
        kalender.nr = 1;
        Monat erster = kalender;
        Monat lauf = kalender;
        for (int i = 1; i < monate; i++) {
            Monat neu = new Monat();
            neu.name = monatsname[i];
            neu.tage = monatstage[i];
            neu.nr = i + 1;
            lauf.next = neu;
            lauf = neu;
        }
        lauf.next = kalender; // Ring schließen
        int anzahl = 0;
        do {
            String str = javax.swing.JOptionPane
                    .showInputDialog("Wie viele Monate?");
            if (str == null || str.isEmpty()) {
                anzahl = 0;
            } else {
                anzahl = Integer.parseInt(str);
            }

            for (int i = 0; i < anzahl; i++) {
                Monat monat = naechster();
                zeige(monat);
            }
        } while (anzahl > 0);
    }

    static Monat naechster() {
        kalender = kalender.next;
        return kalender;
    }
```

```
    static void zeige(Monat monat) {
        if (monat != null) {
            System.out.print(monat.name);
            System.out.print(" - ");
            System.out.println(monat.tage);
        }
    }
}
```

Listing 9.14: Der immerwährende Kalender

Wenn Sie Freude daran haben, können Sie den immerwährenden Kalender noch um das Jahr ergänzen und dann über eine Wochentagsberechnung tatsächlich einen einfachen Kalender erstellen.

Über die folgende Befehlsfolge ermitteln Sie den Wochentag. Die Variable `tag` enthält eine 1 für Montag und eine 7 für einen Sonntag.

```
java.time.LocalDate heute
        = java.time.LocalDate.of(jahr, monat.nr, 1);
int tag = heute.getDayOfWeek().getValue();
```

Und wenn Ihnen dann noch langweilig ist, könnten Sie noch die Schaltjahre berücksichtigen.

Binäre Bäume

Referenzen ermöglichen sehr flexible Datenstrukturen. Die Klasse `Baum` im folgenden Listing enthält gleich zwei Referenzen auf sich selbst. Ein Objekt hat also nicht nur einen Nachfolger, sondern zwei. Hängt man mehrere der Objekte aneinander, ergibt sich ein Baum, der bei jeder Verzweigung zwei Nachfolger haben kann.

```
public class Baum {
    int wert;
    Baum kleiner = null;
    Baum groesser = null;
}
```

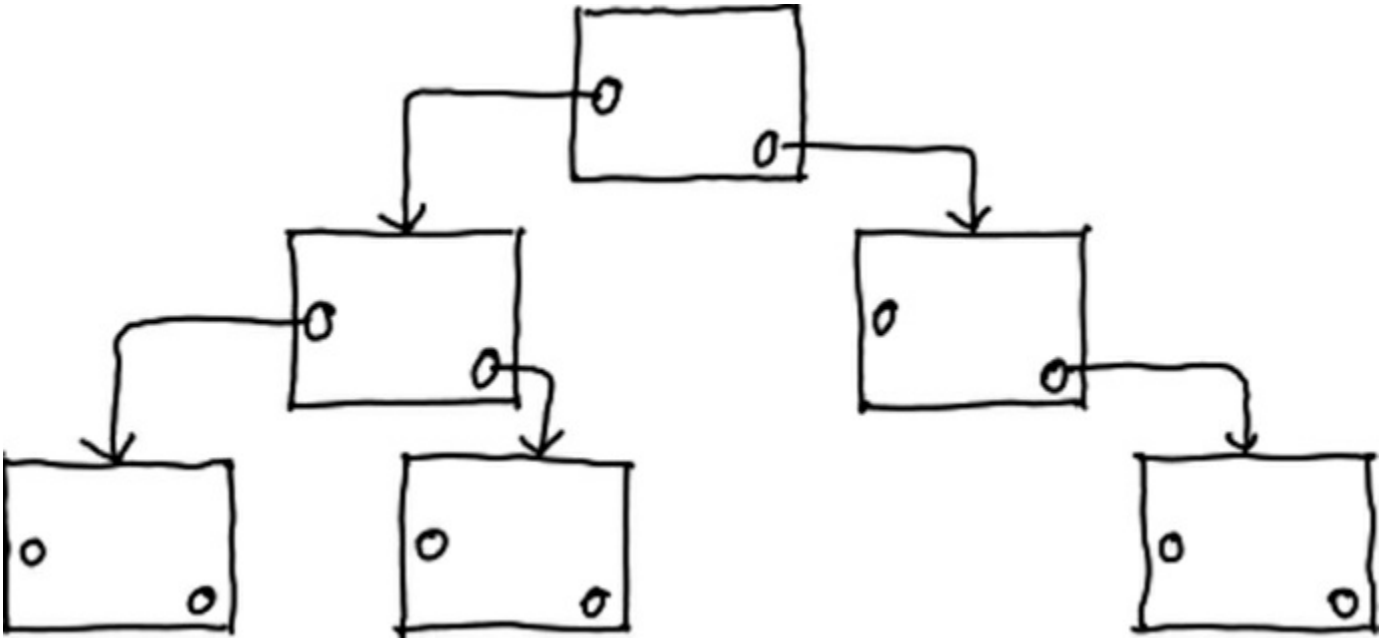


Abbildung 9.7: Zwei Referenzen ziehen einen binären Baum auf.

Einen solchen Baum kann man beispielsweise nutzen, um Daten sortiert abzulegen. Sie kennen sicher das Spiel Zahlenraten aus Ihrer Kindheit, mit der man sich lange Reisen verkürzte. Ein Kind denkt sich eine Zahl, das andere Kind rät. Ist das ratende Kind geschickt, teilt es den jeweils verbleibenden Zahlenraum immer durch 2 und kann so nach spätestens zehn Schritten eine Zahl zwischen 1 und 1000 herausfinden.

Wenn Sie Daten gleich in einem Baum ablegen, können Sie diese sehr schnell wiederfinden. Aber Sie können auch einfach den Baum ausgeben und haben die Elemente immer korrekt sortiert.

Allerdings werden Sie verzweifeln, falls Sie versuchen sollten, den Baum mit einfachen Schleifen auszugeben. Sie müssen im ersten Schritt zum kleinsten Element. Das ist noch einfach. Eklig wird es, wenn Sie dann einen Schritt zurück müssen und von dort aus immer wieder möglichst weit links und so fort. Mit einfachen Schleifen werden Sie da auf erhebliche Schwierigkeiten stoßen.

Aber welch ein Glück: Es gibt Rekursionen! Und diese ansonsten nicht so beliebten Selbstaufrufe der Methoden passen wie die Faust auf das Auge einer Baumstruktur. Die Methode in Listing 9.15 durchläuft einen binären Baum komplett und gibt den Inhalt aus und das in nur neun Zeilen!

```
static void ausgeben(Baum baum) {
    if (baum != null) {
        ausgeben(baum.kleiner);
        System.out.print(" " + baum.wert);
        ausgeben(baum.groesser);
    }
}
```

Listing 9.15: Einen binn Baum auslesen

Das Geheimnis liegt darin, dass die Methode der Struktur durch einen Selbstaufruf folgt. Ist sie ganz unten links angekommen, stößt sie auf ein `null` und kehrt in den vorigen Zustand zurück, gibt den Wert aus und geht in den rechten Ast, um von dort wieder tiefstmöglich abzusteigen.

Wenn Sie die Ausgabe einmal durchgespielt und verstanden haben, können Sie zu der etwas komplizierteren Aufgabe schreiten und ein Element in den Baum einfügen. Auch hier greifen Sie sinnvollerweise auf die Rekursion zurück.

```
static void einsortieren(Baum baum, int zahl) {
    if (baum.wert > zahl) {
        if (baum.kleiner != null) {
            einsortieren(baum.kleiner, zahl);
        } else {
            baum.kleiner = new Baum();
            baum.kleiner.wert = zahl;
        }
    } else {
        if (baum.groesser != null) {
            einsortieren(baum.groesser, zahl);
        } else {
            baum.groesser = new Baum();
            baum.groesser.wert = zahl;
        }
    }
}
```

Listing 9.16: In den binären Baum einfügen

Nun können Sie diese beiden Methoden austesten, indem Sie den Baum mit Zufallszahlen füllen und anschließend ausgeben.

```
public static void main(String[] args) {
    java.util.Random zufall = new java.util.Random();
    Baum baum = new Baum();
    baum.wert = zufall.nextInt(100);
    for (int i = 0; i < 12; i++) {
        int zahl = zufall.nextInt(100);
        einsortieren(baum, zahl);
    }
    ausgeben(baum);
}
```

Listing 9.17: Das passende Hauptprogramm zum Testen des Baums

Die Vorteile der Privatsphäre

Ergänzen wir die Klasse `Datum` um einen Wochentag. Der Einfachheit halber verwenden wir ein `int`-Attribut. Der Sonntag ist 0, der Montag 1 und so weiter.

Wenn nun eine andere Klasse auf den Wochentag zugreift, muss die Klasse `Datum` sicherstellen, dass der Wochentag zu den anderen Daten passt. Das scheint unmöglich, weil die Klasse ja nicht merkt, wenn jemand auf das Attribut `wochentag` zugreift. Sie können allerdings das Attribut privat deklarieren und dann eine öffentliche Methode erstellen, die den Wochentag herausrückt.

```
public class Datum {
    int jahr;
    int monat;
    int tag;
    private int wochentag;
    public int getWochentag() {
        // errechne den gültigen Wochentag
        return wochentag;
    }
}
```

Listing 9.18: Privater Wochentag mit Getter-Methode



Ein als `private` definiertes Attribut darf nur innerhalb des Objekts zugegriffen werden. Sie können auch Methoden privatisieren, wenn sie nur für den internen Gebrauch geschrieben sind.

Die Berechnung des Wochentags für ein Datum erweist sich als nicht trivial. Darum verwenden die meisten Programmierer dafür eine Bibliothek. Entsprechend ist es eigentlich Unsinn, den Wochentag jedes Mal neu zu berechnen, wenn sich das

Datum zwischenzeitlich nicht geändert hat. Damit stehen wir wiederum vor der gleichen Fragestellung: Wie bemerkt die Klasse, dass sich Tag, Monat oder Jahr geändert hat? Die Lösung besteht darin, diese Attribute zu privatisieren und mit Get- und Set-Methoden zu versehen.

Das Erstellen von Gettern und Settern unterstützt Eclipse:

- ✔ Legen Sie eine Klasse mit den gewünschten Attributen an und definieren Sie diese als `private`.
- ✔ Klicken Sie mit der rechten Maustaste in das Listing.
- ✔ Aus dem Menü wählen Sie Source|Generate Getters And Setters.
- ✔ Es erscheint ein Dialog wie in Abbildung 9.8. Darin wählen Sie die gewünschten Attribute an oder klicken gleich auf den Button Select All.
- ✔ Klicken Sie auf OK und im Editor erscheinen alle Getter und Setter der ausgewählten Attribute.

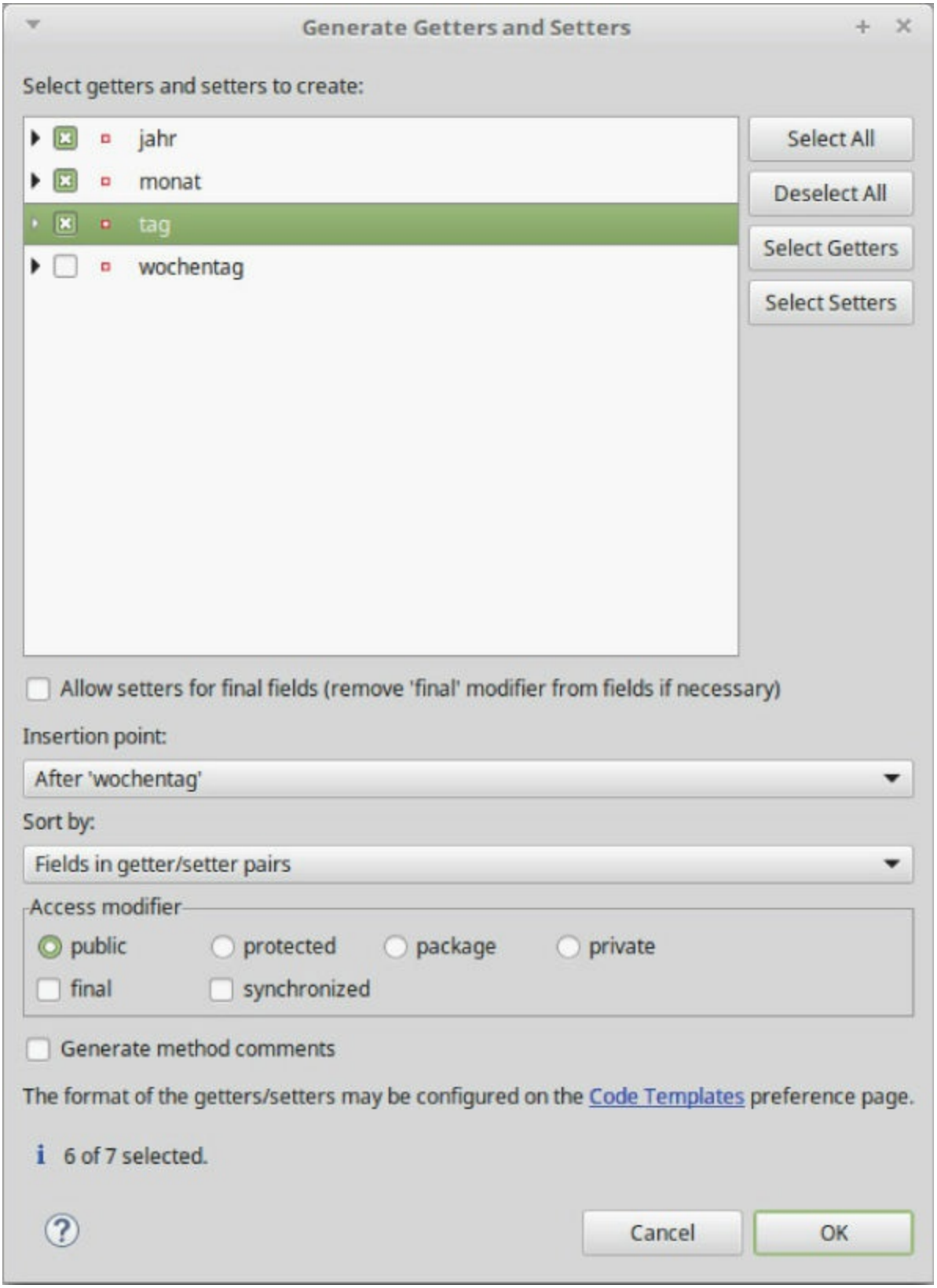


Abbildung 9.8: Eclipse generiert Getter- und Setter-Methoden.

Ihr Quellcode sieht danach etwa so aus:

```
public class Datum {
    private int jahr;
    private int monat;
    private int tag;
    private int wochentag;
    public int getJahr() {
        return jahr;
    }
    public void setJahr(int jahr) {
        this.jahr = jahr;
    }
    public int getMonat() {
        return monat;
    }
    public void setMonat(int monat) {
        this.monat = monat;
    }
    public int getTag() {
        return tag;
    }
    public void setTag(int tag) {
        this.tag = tag;
    }
    public int getWochentag() {
        // errechne den gültigen Wochentag
        return wochentag;
    }
}
```

Listing 9.19: Datum mit privaten Attributen und Getter- und Setter-Methoden

Es ist kein Zufall, dass Eclipse die Verwendung von Getter- und Setter-Methoden unterstützt, sondern ein empfohlenes Vorgehen. Sie erhalten so folgende Möglichkeiten.

- ✔ Sie können erreichen, dass Sie den Wert erst beim Zugriff ermitteln müssen.
- ✔ Sie können für Attribute den lesenden oder schreibenden Zugriff getrennt ausschließen, indem Sie die entsprechende Methode entfernen.
- ✔ Sie können bei Zugriffen prüfen, ob der Zugriff sinnvoll oder zulässig ist.

Da ein nachträgliches Umstellen auf Getter und Setter in einem größeren Projekt umständlich sein kann, setzt man Attribute typischerweise auf `private` und erstellt

Getter und Setter. Es ist also nicht pure Gehässigkeit gegenüber anderen Klassen, die zur Verwendung von `private` führen.

Um nun zu erreichen, dass der Wochentag nur berechnet wird, wenn es wirklich erforderlich ist, sorgen Sie dafür, dass bei jedem Aufruf eines Setters der Wochentag ungültig gemacht wird, indem er beispielsweise auf -1 gesetzt wird. Bei Anfrage nach dem Wochentag prüfen Sie, ob der Wert negativ ist. Nur in diesem Fall muss eine Neuberechnung erfolgen.

```
class Datum {
    private int tag, monat, jahr, wochentag=-1;
    void setTag(int tag) { // Setter-Methode
        this.tag = tag;
        this.wochentag = -1;
    }
    int getWochentag() { // Getter-Methode
        if (wochentag<0) { // nur rechnen, wenn nötig
            // Wochentag berechnen und dann....
            wochentag =  ... ;
        }
        return wochentag;
    }
}
```

Listing 9.20: Das Datum errechnet den Wochentag nur in Notwehr.

Wenn Sie die Setter-Methoden anschauen, sehen Sie, dass der Parameter den gleichen Namen trägt wie das Attribut, also beispielsweise `jahr`. Wird nur `jahr` als Name verwendet, wird der Parameter verwendet. Um das Attribut zu erreichen, wird mit `this` zugegriffen.



In der Software-Entwicklung gibt es das Prinzip der maximalen Geheimniskrämerei. Damit erreicht man, dass die anderen Programmierer im Team nicht alle Details wissen müssen, und vor allem, dass sie sich nicht auf Implementierungsdetails verlassen, die später eventuell wieder geändert werden.

Auch Klassen können privat sein!

Nicht nur Attribute können privat sein. Auch Methoden, die nur für den internen Gebrauch in der Klasse benötigt werden, können und sollten privat deklariert werden. Darüber hinaus ist es sogar möglich, private Klassen anzulegen.

Eine private Klasse ist immer dann sinnvoll, wenn sie ausschließlich von einer einzigen Klasse benötigt wird. Private Klassen bekommen keine eigene Datei, sondern stehen in der Datei der Klasse, die sie als Hilfsklasse verwendet. Auf diese

Weise werden größere Projekte übersichtlicher, weil man sofort weiß, dass diese Klasse nicht von mehreren Seiten angesprochen wird.

```
public class HatPrivatKlasse {

    private class Wochentag { // von außen unerreichbar
        int wochentag = -1;
        String name = "";
    }
    private int tag, monat, jahr;
    private Wochentag wochentag = new Wochentag();
    public String getWochenttagName() {
        return wochentag.name;
    }
}
```

Listing 9.21: Private Klasse

Im Listing 9.21 wird innerhalb der Klasse `HatPrivatKlasse` eine interne private Klasse namens `Wochentag` angelegt. Sie soll sich um die Besonderheiten des Wochentags kümmern, der neben seiner Nummer ja auch einen Namen wie »Samstag« haben könnte. Der Wochentag wird innerhalb der Klasse `HatPrivatKlasse` als privates Attribut der privaten Klasse `Wochentag` angelegt. Die Klasse `HatPrivatKlasse` bietet für den Zugriff darauf die Methode `getWochenttagName` an. Ansonsten hat niemand an dem Wochentag zu fummeln.

Verschachtelte Klassen

Was passiert eigentlich, wenn die innere Klasse nicht als `private` definiert wird? Tatsächlich können Sie eine Klasse innerhalb einer Klasse auch als `public` definieren.

Als Beispiel wird eine Klasse `Aussen` angelegt, die eine innere Klasse `Innen` enthält.

```
public class Aussen {
    private int aussenInt = 9;

    public class Innen {
        int innenInt = Aussen.this.aussenInt;
    }
}
```

Listing 9.22: Verschachtelte Klassen

Die innere Klasse hat immer Zugriff auf alle Elemente der Klasse, in der sie eingeschlossen ist. Der Zugriff erfolgt über `Aussen.this.aussenInt`. Mit diesem

Wert wird die innere Variable `innenInt` vorbelegt.

Die Klasse `AussenTest` legt zunächst ein Objekt der Klasse `Aussen` an. Über dieses kann ein Objekt der Klasse `Innen` angelegt werden. Zu guter Letzt ist es auch möglich, auf das Attribut des Objekts `innen` zuzugreifen, da alles `public` ist.

```
class AussenTest {
    public static void main(String[] args){
        Aussen aussen = new Aussen();
        Aussen.Innen innen = aussen.new Innen();
        System.out.println(innen.innenInt);
    }
}
```

Listing 9.23: Verschachtelte Klassen

Konstruktionsbedingt

Ihnen ist vielleicht schon aufgefallen, dass der Klassenname hinter dem Befehl `new` immer ein paar Klammern besitzt, wie sie normalerweise bei einem Methodenaufruf verwendet werden.

```
Datum datum = new Datum();
```

Tatsächlich wird bei der Erzeugung eines Objekts eine Methode aufgerufen. Diese initialisiert das Objekt und wird *Konstruktor* genannt. In den bisherigen Klassen haben wir uns um einen Konstruktor nicht geschert. Das war auch gar nicht notwendig, denn wenn eine Klasse keinen Konstruktor hat, erstellt Java einen Standardkonstruktor. Sie wollen selbst einen Konstruktor erstellen? Dann sind Sie hier richtig.

Konstruktoren sind eine besondere Form der Methoden und sind dadurch erkennbar, dass sie einerseits den Namen der Klasse tragen und zweitens keinen Rückgabetyp besitzen, auch nicht `void`.

Konstruktoren ohne und mit Parameter

Erstellen wir doch für unsere Lieblingsklasse `Datum` einen Konstruktor. Er kümmert sich sofort darum, dass alle Attribute mit 0 vorbesetzt werden.

```
class Datum {
    private int jahr, monat, tag;
    Datum() {
        jahr = 0; monat = 0; tag = 0;
    }
    // ...
}
```

Listing 9.24: Eine Klasse mit Konstruktor

Ein Standardkonstruktor ist ein Konstruktor ohne Parameter. Daraus können Sie blitzschnell folgern, dass es wohl auch andere Konstruktoren gibt und diese wohl Parameter besitzen. Und was soll ich sagen? Sie haben recht!

Konstruktoren mit Parametern

Ein Konstruktor kann auch Parameter haben, die beim Aufruf von `new` bedient werden. Damit können Standardwerte an das Objekt übergeben werden. Der folgende Konstruktor besetzt das Datum mit Jahr, Monat und Tag gleich bei der Entstehung.

```
    Datum(int jahr, int monat, int tag) {
        this.jahr  = jahr;
        this.monat = monat;
        this.tag   = tag;
    }
```

Listing 9.25: Konstruktor mit Parametern

Um diesen Konstruktor zu verwenden, werden die Parameter beim Aufruf von `new` verwendet.

```
Datum heute = new Datum(1989, 11, 9);
```

Überladen von Konstruktoren

Sie können beide Konstruktoren nebeneinander in der Klasse definieren. Sie kennen den Effekt schon vom Überladen von Methoden, dass Java die aufzurufende Methode nicht allein aufgrund des Namens, sondern anhand einer Kombination aus Namen und Parametertypen auswählt.



Wenn Sie nur einen Konstruktor mit Parametern anbieten, hat die Klasse keinen Standardkonstruktor mehr und muss mit Konstruktorparametern erzeugt werden. Sie zwingen damit den Aufrufer, das Objekt zu initialisieren.

Haben Sie mehrere Konstruktoren erstellt und wollen aus einem Konstruktor einen anderen Konstruktor aufrufen, dann verwenden Sie `this` als Methodenaufruf. Da ja im Konstruktor das Objekt erst in der Entstehung ist, muss dieser Aufruf zu Anfang des Konstruktors erfolgen.

```
public Datum(int jahr) {
    this.jahr = jahr;
}
public Datum(int tag, int monat, int jahr) {
    this(jahr);
    this.tag = tag; this.monat = monat;
}
```

Listing 9.26: Ein Konstruktor ruft einen anderen auf.

Der Singleton und der private Konstruktor

Auch ein Konstruktor kann natürlich privat deklariert werden. Obwohl es zunächst nach einem schlechten Scherz klingt, gibt es tatsächlich eine praktische Anwendung für einen privaten Konstruktor, nämlich den Singleton.



Ein Singleton ist eine Klasse, von der es nur ein Objekt geben soll. Beim ersten Zugriff auf den Singleton wird das Objekt erzeugt. Alle späteren Zugriffe erfolgen auf dieses bereits angelegte Objekt, typischerweise durch Aufruf der statischen Methode `getInstance()`.

Ein Singleton folgt etwa folgendem Muster:

```
public final class Singleton {
    private static Singleton singleton = null;
    private Singleton() {
        // der tut nix, der will nur spielen
    }
    public static synchronized Singleton getInstance() {
        if (singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

Listing 9.27: Ein typischer Singleton

Listing 9.27 zeigt folgende Besonderheiten:

- ✔ Das statische und natürlich private Attribut `singleton` referenziert auf das einzig existierende Objekt der Klasse und ist mit `null` vorbesetzt.
- ✔ Der Konstruktor ist privat und verhindert so, dass eine fremde Klasse ein Objekt dieser Klasse erzeugt. Er muss nicht unbedingt mit Anweisungen gefüllt sein.

- Die statische Methode `getInstance` findet beim ersten Aufruf das Attribut `singleton` mit `null` vorbesetzt vor. Sie ruft dann den privaten Konstruktor per `new`, besetzt das Attribut `singleton` und gibt dieses zurück.

  Ab dem zweiten Aufruf wird `getInstance` das bereits angelegte Attribut `singleton` zurückgeben.
- Der Modifizierer `synchronized` sorgt dafür, dass selbst dann, wenn Sie in Ihrem Programm parallele Abläufe einsetzen, die Methode nicht von zwei Threads parallel ausgeführt werden kann. Das Thema Threads finden Sie in Kapitel 28.

Damit also jemand den Zugriff auf das einzige Objekt der Klasse erhält, muss er die statische Methode `getInstance` statt des Befehls `new` aufrufen.

```
Singleton dasEinzigWahre = Singleton.getInstance();
```

Natürlich sollten Sie Ihrer Singleton-Klasse einen etwas originelleren Namen geben.

Statische Attribute und Methoden

Normalerweise werden Attribute und Methoden über das Objekt aufgerufen. Jedes Objekt hat eigene Attribute. Es wollen ja schließlich nicht alle Menschen am gleichen Tag Geburtstag haben. Wenn eine Methode über ein Objekt aufgerufen wird, ändert die Methode die Attribute dieses Objekts. Attribute und Methoden gehören also immer zum Objekt. Schießlich werden sie ja auch über den Namen des Objekts gerufen.

Im Gegensatz dazu gibt es aber auch Attribute, die für die ganze Klasse gelten. Beispielsweise würde ein Attribut `anzahlWochentage` für alle Objekte der Klasse `Datum` gleich sein. Darum wäre es auch unsinnig, sie für jedes Objekt einzeln zu führen. Durch das Schlüsselwort `static` kann definiert werden, dass dieses Attribut exakt einmal für die ganze Klasse existiert.

```
class Datum {
    // ...
    static int  anzahlWochentage = 7;
    // ...
}
```

Da sich die Anzahl der Wochentage in absehbarer Zeit nicht ändern wird, könnten Sie sogar den Modifier `final` hinzufügen und so aus dem Attribut eine Konstante machen.

Der Zugriff erfolgt dann konsequenterweise auch nicht über ein Objekt der Klasse `Datum`, sondern über die Klasse selbst.

```
while (i < Datum.anzahlWochentage) {
```

Weitere Anwendungen könnten Zähler sein, die protokollieren wie viele Objekte einer Klasse angelegt wurden. Dazu wird die Variable vom Konstruktor immer um 1 erhöht.

```
class Datum {
    // ...
    static int anzahlObjekte = 0;
    public Datum() {
        anzahlObjekte++;
    }
    // ...
}
```

Listing 9.28: Zähler für die angelegten Objekte

Gehörte `anzahlObjekte` zu jedem einzelnen Objekt, wäre der Inhalt immer 1. So gehört das Attribut aber zur Klasse und spiegelt eben die Anzahl der angelegten Objekte wider.

Es gibt auch statische Methoden. Wir haben wieder und wieder die Methode `main` verwendet. Diese Methode muss statisch sein, da sonst ein Henne-Ei-Problem entstehen würde: Die Methode gibt es erst nach dem Erzeugen eines Objekts und ein Objekt gibt es erst nach dem Programmstart. Außerdem darf es nur genau eine Hauptmethode geben. Und damit ist `main` eine typische Anwendung für eine statische Methode.



Bei der Ausgabe auf dem Bildschirm haben wir immer `System.out.println` aufgerufen. `System` beginnt mit einem Großbuchstaben. Also handelt es sich doch wohl auch um eine Klasse und nicht um ein Objekt. Das Attribut `out` ist offenbar statisch und bietet die Methode `println` an.



Statisch heißt bei Attributen, dass das Attribut exakt einmal pro Klasse existiert. Es wird also nicht für jedes Objekt neu angelegt. Bei Methoden bedeutet es, dass sie gerufen werden können, ohne dass zuvor ein Objekt erzeugt worden ist.

Statische Methoden existieren auch vor dem Erzeugen des ersten Objekts, da sie ohne ein existierendes Objekt funktionieren müssen. Sie können nur auf statische Attribute zugreifen und dürfen nur andere statische Methoden aufrufen. Auch dies kennen Sie schon von der Methode `main`.

Statischer Konstruktor

Manchmal müssen statische Attribute aufwendiger initialisiert werden. Ein

Konstruktor wird aber nur aufgerufen, wenn ein Objekt erzeugt wird. Da sind statische Attribute längst aus dem Quark. Um die statischen Attribute zu initialisieren, legen Sie einen `static`-Block an. Alle darin stehenden Anweisungen dienen der Initialisierung der statischen Attribute.

```
public class StaticSpielerei {
    static int statischerWert;
    static {
        statischerWert = 5;
    }
}
```

Listing 9.29: Statischer Konstruktor

Der statische Block wird als Erstes ausgeführt, also noch bevor ein Objekt erzeugt werden kann.

Wrapper-Klassen für Primitive

Die armen primitiven Typen waren nun ganz traurig, dass sie nicht die Klasse hatten wie die Objekte. Darum dürfen sie bei ganz vielen Dingen nicht mitspielen, die Objekten vorbehalten sind. Das ist ja so ungerecht. Aber Java wäre nicht Java, wenn es da nicht eine Lösung für arme, benachteiligte Primitive gäbe.

Für jeden primitiven Typ stellt Java eine Klasse zur Verfügung, die als Wrapper funktioniert. Wrapper kennen Sie vielleicht als Pfannkuchenrollen aus Fastfood-Restaurants. Und so ähnlich funktionieren diese Klassen auch in Java. Sie werden als Hüllen verwendet, wenn die primitiven Typen über Referenzvariablen angesprochen werden müssen.

Die Namen der Wrapper-Klassen ähneln der Typenbezeichnung des zugehörigen primitiven Typs, beginnen jedoch, wie das bei Klassen üblich ist, mit einem Großbuchstaben:

Primivtyp	Wrapper
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

Tabelle 9.1: Primitive Typen und ihre Klassen-Wrapper

Die Wrapper-Klassen sind zuweisungskompatibel zu den jeweiligen Primitivtypen. Diese Fähigkeit nennt man *Autowrapping* oder *Autoboxing*.

```
Integer intwrap = 5;
int i = 4;
intwrap = i;
i = intwrap;
```

Besonders wertvoll werden die Wrapper-Klassen dann, wenn für bestimmte Zwecke Objekte gefordert werden. Beispielsweise ist dies beim Java Collection Framework der Fall, auf das Sie in Kapitel 15 stoßen werden.



Sie können den Inhalt eines Wrapper-Objekts nur bei dessen Erzeugung besetzen. Eine spätere Änderung ist unmöglich. Brauchen Sie einen anderen Wert, so legen Sie ein neues Objekt an.

Neben der freundlichen Aufnahme ihrer Primitiven in höhere Klassen bieten die Wrapper-Klassen auch noch ein paar statische Methoden, die das Leben einfacher machen.

- Die Parse-Methoden sind in der Lage, aus Strings, wie sie beispielsweise bei grafischen Eingabeelementen entstehen Zahlenwerte auszulesen. Dabei bietet `Integer` die Methode `parseInt`, `Double` die Methode `parseDouble` an.

  ```
  int zahl = Integer.parseInt(string);
  double doppel = Double.parseDouble(string);
  ```

- Die Bestimmung von Minimum oder Maximum von zwei Werten werden als Methode `min` beziehungsweise `max` zur Verfügung gestellt.

  ```
  zahl = Integer.min(12,6);
  zahl = Integer.max(12,6);
  ```

Die Klassen stellen auch ein paar Konstanten zur Verfügung. Insbesondere die maximal und minimal möglichen Werte der jeweiligen Primitiven können aus den Wrapper-Klassen entnommen werden.

```
zahl = Integer.MAX_VALUE;
zahl = Integer.MIN_VALUE;
```

Aufzählungstyp mit enum definieren

Sehr elegant wirkt unsere bisherige Wochentagsmodellierung nicht. Statt Zahlen für die Tage zu verwenden, wäre es viel besser, wenn ein Montag als Montag und nicht als 1 codiert würde. Oder war der Montag doch eine 0? Mit dem Befehl `enum`

können wir einen eigenen Aufzählungstyp definieren, der Abhilfe schafft. Das folgende Beispiel zeigt die Wochentage und wie auf sie zugegriffen wird.

```
class Datum {
   private int tag, monat, jahr;

   enum Wochentag { MO, DI, MI, DO, FR, SA, SO }
   // ...
   Wochentag wochentag = Wochentag.MO;
   if (wochentag == Wochentag.DI) {
```

Listing 9.30: Eine Wochentagsaufzung

Nach der Definition der Aufzählung `Wochentag` können Variablen vom Typ `Wochentag` angelegt werden. Da die Elemente einer Aufzählung intern durch Nummern repräsentiert werden, können sie bei der Zuweisung wie primitive Typen verwendet werden.

Natürlich hätten Sie das auch mit ein paar Konstanten hinbekommen. Aber eine `enum`-Aufzählung können Sie darüber hinaus aufblasen, so ähnlich wie eine Klasse. So können Sie den Wochentag mit `enum` fast so behandeln wie die private Klasse aus Listing 9.21. Sie können also sowohl die Nummer als auch den Namen speichern:

```
public class WochentagEnum {

    private int tag, monat, jahr;

    enum Wochentag { MO(0, "Montag"), DI(1, "Dienstag"),
        MI(2, "Mittwoch"), DO(3, "Donnerstag"),
        FR(4, "Freitag"), SA(5, "Samstag"), SO(6, "Sonntag");

        private int tagnr;
        private String tagname;

        Wochentag(int nr, String name) {
            tagnr = nr;
            tagname = name;
        }

        public String toString() {
            return tagname;
        }
    }

    public static void main(String[] args) {
        Wochentag wochentag = Wochentag.DI;
        System.out.println(wochentag);
    }
}
```

Listing 9.31: Wochentage mit Nummern und Namen

Es gibt ein paar syntaktische Besonderheiten:

- ✔ Die Konstanten der Aufzählung erhalten Parameter.
- ✔ Wie eine Klasse hat `enum` mit `tagnr` und `tagname` nun private Attribute.
- ✔ Der Kontruktor nimmt die bei der Definition angegebenen Parameter entgegen und legt sie in den Attributen ab.
- ✔ Sie können innerhalb von `enum` weitere Methoden wie hier beispielsweise `toString` definieren.

In der Hauptfunktion `main` wird wie gehabt ein Objekt von `Wochentag` erzeugt. Der Aufruf der Methoden funktioniert genau wie bei einer normalen Klasse.

Aufgaben

Setter-Methoden für das Datum

Eigentlich haben Sie jetzt genug Anregungen für Ihre eigenen Experimente, um die nächsten Tage die Tastatur zum Glühen zu bringen. Falls nicht, schreiben Sie je eine `set`-Methode für Jahr, Monat und Tag, die prüfen, ob das Datum korrekt ist.

Normalerweise geben `set`-Methoden keinen Wert zurück. Hier könnten Sie aber in einem Rückgabewert vom Typ `boolean` angeben, ob der übergebene Wert akzeptiert wurde.

Die Prüfung, ob der Tag zwischen 1 und 31 liegt, ist natürlich sehr simpel. Aber da soll es doch Monate geben, die sich anders verhalten. Vielleicht könnte da dieses Array helfen.

```
int tageImMonat[] =
        { 31, 28, 31, 30, 31, 30,
          31, 31, 30, 31, 30, 31 };
```

Listing 9.32: Array mit den Tagen pro Monat

Berücksichtigen Sie gern auch Schaltjahre!

Sie sehen, es gibt reichlich Ideen. Und Sie entwickeln dafür bestimmt noch mehr. Wenn Ihnen irgendwann langweilig wird, schlagen Sie das Buch wieder auf! Da bin ich sofort wieder für Sie da.

Kalenderdaten addieren und subtrahieren

Sie sind nun in der Lage, eigene Methoden zu einer Klasse hinzuzufügen. Schreiben Sie doch eine Methode für die Addition. Spontan werden Sie das für eine Falle halten, denn was soll schon dabei herauskommen, wenn man zwei Kalenderdaten addiert? Das ist richtig. Aber die Addition eines Datums mit einer Anzahl von Tagen ist durchaus sinnvoll, wenn das Ergebnis wieder ein Datum ist.

Wenn Sie übermütig sind, können Sie auch gern noch die Subtraktion programmieren. Davon gibt es genau genommen zwei Varianten: Im ersten Fall subtrahieren Sie Tage vom Datum ab und erhalten ein Datum. In der zweiten Variante bilden Sie die Differenz zweier Kalenderdaten und erhalten die Differenz in Tagen.

Das Problem beim Rechnen mit Kalenderdaten ist die unterschiedliche Länge der Monate. Hier könnte das Array aus der letzen Aufgabe hilfreich sein.

Eine kleine, private Kiste

Bauen Sie sich eine kleine Kistenklasse. Die Klasse verwaltet eine rechteckige Kiste. Alle Kisten haben eine Höhe, eine Breite und eine Tiefe. Das Volumen berechnet sich durch die Multiplikation dieser drei Werte.

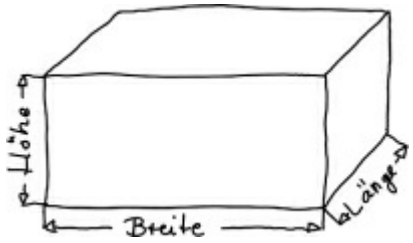


Abbildung 9.9: Eine Kiste hat drei Dimensionen.

Die Kiste soll eine Methode `getVolumen` besitzen. Wie können Sie erreichen, dass die Berechnung des Volumens möglichst selten erfolgt? Die Berechnung des Volumens soll innerhalb der Methode `getVolumen` erfolgen. Werden Höhe, Tiefe oder Breite geändert, stimmt das Volumen nicht mehr. Änderungen an diesen Werten müssen Sie also abfangen.

Brechen Sie mal

Wenn Sie dividieren, entsteht eine Zahl mit Nachkommastellen. Aber das kann ungenau sein, wenn bei der Darstellung Perioden entstehen. Definitiv verhindern können Sie dies, wenn Sie statt mit `float` oder `double` einfach mit Brüchen arbeiten. Schreiben Sie eine Klasse `Bruch`, die einen Zähler und einen Nenner enthält.

Die Klasse sollte natürlich über die vier Grundrechenarten verfügen. Dafür könnten Sie als Namen `add`, `sub`, `mul` und `div` verwenden.

Verwenden Sie als Konstruktor ausschließlich einen mit zwei Integer als Zähler und Nenner.

Bauen Sie ein System zum automatischen Kürzen ein. Dazu werden Sie eine Methode zum Berechnen des größten gemeinsamen Teilers (ggT) benötigen. Der gute alte Euklid soll so etwas schon mal programmiert haben. In Kapitel 4 haben Sie den Herrn bereits kennengelernt.

```
int ggt(int a, int b) {
    while (b!=0) {
        int h = a % b;
        b = a:
        a = h;
    }
    return a;
}
```

Listing 9.33: Größter gemeinsamer Teiler, abgeschrieben bei Euklid

Wenn Sie es rekursiv lieber mögen, gibt es eine kürzere Variante:

```
int ggt(int a, int b) {
    if (b == 0)
        return a;
    else
        return ggt(b, a % b);
}
```

Auch die Klasse `Bruch` könnte eine Methode `equals` gut gebrauchen können.

Punkte und Linien

Bauen Sie sich eine eigene Klasse `Punkt`, die einen Punkt auf dem Bildschirm oder in einem Koordinatensystem darstellt. So ein Punkt hat eine x- und eine y-Koordinate.

Nichts spricht dagegen, dass ein Objekt andere Objekte als Attribute hat. So könnten Sie eine Klasse `Linie` erstellen, die durch zwei Punkte festgelegt wird. Aber passen Sie auf, dass Sie den Punkt einer Strecke mit `new` erzeugen, bevor Sie ihn erstmalig benutzen! Zum Test Ihrer Klassen können Sie dann ja einmal `PunktLinienTest` laufen lassen.

```
public class PunktLinienTest {
    public static void main(String[] args) {
        Linie linie = new Linie();
        linie.von = new Punkt();
        linie.bis = new Punkt();
        linie.von.x = 100;
        System.out.println(linie.von.x);
    }
}
```

Listing 9.34: Testet Punkt und Linie

Konstruierte Linien

Sie haben doch sicher die Klasse `Punkt` und die Klasse `Linie` aus der Aufgabe eben erstellt. Wie ich Ihnen schon erklärte, ist es natürlich kritisch, wenn jemand auf die Koordinate eines Linienpunkts zugreift, bevor der Punkt mit `new` erzeugt wurde. Durch den Konstruktor können Sie in der Klasse `Linie` nun selbst dafür sorgen, dass der Anfangs- und Endpunkt automatisch erzeugt wird.

Sie können für die Linie auch einen Konstruktor bauen, der gleich vier Zahlenwerte als Parameter akzeptiert, mit denen Sie Anfangs- und Endpunkt sofort belegen können.

Kapitel 10

Klassen erweitern

IN DIESEM KAPITEL

Klassen werden erweitert, das Tier wird Mensch.

Methodenaufrufe richten sich an das Objekt: Polymorphie in der Mensa.

In abstrakten Klassen und Interfaces ist Musik drin.

Wenn Sie eine Software für eine Bibliothek schreiben sollen, dann werden Sie zunächst ein Modell mit Klassen erstellen. Dabei fällt Ihnen vermutlich auf, dass der Unterschied zwischen einem Angestellten und einem Kunden nicht wirklich groß ist. Das führt nicht unbedingt zu der Erkenntnis, dass alle Menschen gleich sind. Wir sind halt doch eher Informatiker als Juristen. Und es gibt Unterschiede: Ein Angestellter bekommt Gehalt und der Kunde zahlt höchstens noch Strafgebühren.

Aber was ist mit den Ähnlichkeiten? Muss ich wirklich sowohl für den Kunden als auch für den Angestellten Name, Adresse, Telefonnummer jeweils extra anlegen? Da hilft die objektorientierte Programmierung mit erweiterten Klassen. Sie legen eine Basisklasse `Person` an, die alle Gemeinsamkeiten enthält. Dann können Sie die Person um ein Gehaltskonto zu einem Angestellten erweitern und die Person um eine Ausleihliste zum Kunden erweitern. Wie das geht und was man damit für einen Spaß haben kann, erfahren Sie in diesem Kapitel.

Noch mal Klasse: Der Mensch als Objekt

Betrachten Sie das Edelste, was Sie kennen: mich. Vielleicht sind Sie nicht sofort einverstanden, weil Sie mich nicht so genau kennen. Dann sind wir uns vielleicht darin einig, dass Sie das Edelste überhaupt sind. Ich stimme sofort zu, denn Sie haben ja mein Buch gekauft.

Wenn wir uns trotz eventueller Vorbehalte Ihrerseits als Objekte ähnlicher Art betrachten, müssten wir ja durch eine bestimmte Klasse definiert sein. Biologisch gesehen sind wir Lebewesen und gehören zu den Tieren. Genauer ausgedrückt, sind wir Menschen spezielle Primaten, also so eine Art Affen, selbst wenn wir hoffen,

dass es etwas gibt, das uns Menschen vom Affen unterscheidet. Betrachten wir eine Klasse `Tier`:

```
class Tier {
    int anzahlBeine;
    int anzahlArme;
    private int alter=0;
    void altert() {
        alter = alter + 1;
    }
}
```

Listing 10.1: Klassifizierte Tiere

Damit nicht jemand einfach sein Alter wieder heruntersetzt, haben wir das Attribut `alter` privat gesetzt.

Was den Mensch vom Tier unterscheidet

Nun will ich für mich selbst ein Objekt anlegen. Die Referenz heißt damit Willemer. Pardon, nach Konvention kleingeschrieben, also `willemer`. Wir legen also ein Tier mit dem Namen `willemer` an.

```
Tier willemer = new Tier();
```

Damit sind die tierischen Eigenschaften des Buchautors geklärt. Tatsächlich verdiene ich manchmal etwas Geld. Und ich werde mich nun nicht mit Ihnen darüber streiten, ob ich es verdiene. Jedenfalls landet das Geld auf meinem Bankkonto. Dagegen haben Nashörner in der Regel kein Bankkonto. Das liegt vermutlich daran, dass ein Nashorn nicht durch die Tür der Sparkasse passt.

Die Klasse `Tier` kann also kein zusätzliches Attribut für das Bankkonto erhalten. Wir brauchen eine Sonderklasse `Mensch`, die alle Attribute und Methoden eines Tieres hat und zusätzlich ein Bankkonto. Objektorientiert ausgedrückt hieße das, dass der Mensch die Klasse `Tier` um ein Konto erweitert.

```
class Mensch extends Tier {
    int ktoNr;
    int blz;
}
```

Listing 10.2: Der Mensch erweitert das Tier um ein Bankkonto.

Damit ist bewiesen, was wir immer schon ahnten: Der Mensch ist nur ein Tier mit einem Bankkonto. Immerhin kann ich nun auch als Mensch definiert werden. Der Programmierer erzeugt dazu einen neuen Menschen mit `new`, auch wenn ich denke, dass die natürliche Art der Menschenerzeugung deutlich unterhaltsamer ist.

```
Mensch willemer = new Mensch();
willemer.anzahlBeine = 2;
willemer.altert();
willemer.blz = 0815;
```

Obwohl es nicht explizit in der Klasse `Mensch` erwähnt wird, hat das Objekt `willemer` nun auch das Attribut `anzahlBeine` und leider auch die Methode `altert`. Daneben aber auch das Attribut `ktoNr`.

Dieser Vorgang wird in der objektorientierten Programmierung als *Vererbung* bezeichnet und unter Java mit dem treffenderen Schlüsselwort `extends` markiert. Die Klasse `Tier` und ihre Eigenschaften bleiben erhalten und werden von der Klasse `Mensch` erweitert. Die neue Klasse `Mensch` beschreibt nun nur noch, worin sich der Mensch vom Tier unterscheidet, nämlich durch das Bankkonto.



Das Objekt einer abgeleiteten oder erweiterten Klasse ist auch ein Objekt der Basisklasse. Also ist der Mensch eben auch ein Tier. Eine Vererbung kann deshalb als »Ist-auch-ein«-Beziehung bezeichnet werden.

Referenzen auf erweiterte Objekte

Da `willemer` nicht nur `Mensch`, sondern auch `Tier` ist, kann das Objekt auch einer Referenz auf `Tier` zugewiesen werden.

```
Mensch willemer = new Mensch();
Tier tier = willemer;
```

So schnell kann man seine Menschenwürde verlieren. Na ja, bisher besteht diese ja eh nur aus einem Bankkonto. Umgekehrt ist das nicht so einfach. Das Tier kann nicht einfach einem Menschen zugewiesen werden. Dazu braucht es dann doch ein paar Millionen Jahre der Evolution oder den Segen der Bank. Der Mensch ist zwar auch ein Tier. Das Tier ist aber eben nicht auch ein Mensch. Wenn Sie es trotzdem versuchen, mag der Compiler es gar nicht.

```
Tier nashorn = new Tier();
Mensch willemer = nashorn; // geht gar nicht!
```



Ein Objekt der erweiterten Klasse kann einer Referenz der Basisklasse zugewiesen werden. Umgekehrt geht es nicht!

Wenn Sie die technische Begründung lieber mögen, denken Sie daran, dass die Referenz `tier` bei den zugewiesenen Objekten, ob Mensch, ob Tier alle Attribute und Methoden findet, auf die es vielleicht zugreifen will. Dagegen wird die Referenz vom Typ `Mensch` auf das Attribut `blz` bei einem Objekt vom Typ `Tier` ins Leere greifen und es knallt.

Object als der Urahn aller Objekte

Java stellt für alle Klassen automatisch eine heimliche Basisklasse zur Verfügung. Sie heißt sinnigerweise `Object`. Wenn Sie also eine Referenz auf `Object` anlegen, können Sie ihr ein beliebiges Objekt zuweisen, egal ob Tier, Mensch oder Datum. Umgekehrt geht das nicht, außer mit Gewalt: Casting! Sie erinnern sich? Wir haben es benutzt, um beispielsweise den Inhalt von `double`-Variablen in eine Variable vom Typ `short` zu stopfen.

```
Object object = tier; // kein Problem!
Tier tier = (Tier) object; // nur mit Casting!
```

Mit Casting kann man sogar aus dem Willemer ein Nashorn machen. Darum ist Casting so brandgefährlich! Im Beispiel sehen wir ja deutlich, dass `object` ursprünglich auf ein Tier referenzierte. Darum wird das Casting auf eine Tier-Referenz auch problemlos gelingen. Aber der Compiler würde auch nicht nörgeln, wenn Sie das ursprüngliche Tier nun auf ein Datum casten. Wenn Sie dann allerdings versuchen, auf den Monat des Datums zuzugreifen, wird es Probleme geben, denn so ein Tier hat einfach keinen Monat.

Konstruktoren in der Vererbung

Mensch und Tier hatten in den Beispielen keine Konstruktoren. Ich kann Ihnen verraten, dass beim `new` auf einen Menschen zunächst der Konstruktor von `Tier` und dann der von `Mensch` durchlaufen wird. Sollten Sie mir nicht glauben, erstellen Sie einfach je einen Standard-Konstruktor und geben Sie dort eine Bildschirmmeldung aus.

Wenn Sie allerdings einen Konstruktor mit Parametern haben, funktioniert dieser Automatismus nicht mehr. Wir schaffen in der Klasse `Tier` einen Konstruktor, der als Parameter die Anzahl der Beine des Tiers bekommt.

```
public class Tier {
    int anzahlBeine;
    int anzahlArme;
    public Tier(int beine) {
        anzahlBeine = beine;
    }
    private int alter=0;
    void altert() {
        alter = alter + 1;
    }
}
```

Listing 10.3: Basisklasse hat nur einen Konstruktor mit Parameter.

Sobald Sie diesen Konstruktor einbauen, übersetzt die Klasse `Mensch` nicht mehr. Der Grund liegt darin, dass die Klasse `Mensch` von Java einen Standardkonstruktor spendiert bekommt, der seinerseits den Standardkonstruktor der Basisklasse aufrufen will. Der ist aber nicht mehr da.

Nun müssen Sie den Konstruktor der Basisklasse selbst rufen. Dies können Sie über den Aufruf von `super` tun.

```
public class Mensch extends Tier {
    int ktoNr;
    int blz;
    public Mensch() {
        super(2);
    }
}
```

Listing 10.4: Aufruf des Konstruktors der Basisklasse per `super`

Der Aufruf des Basiskonstruktors per `super` darf nur einmal zu Anfang des Konstruktors erfolgen.

Die Mensa der Universität Norgaardholz

Mens sana in corpore sano. Für diejenigen, die von Latein verschont geblieben sind: Wer in die Mensa geht, braucht einen gesunden Körper. Unter diesem Motto betreiben Sie die Mensa der Universität Norgaardholz.

Diese Mensa hat einen neuen Koch und der kann zwei Speisen kochen: Kartoffelbrei und Suppe. Eine Vielfalt, die wie geschaffen ist für eine Mensa.

Kartoffelpampe

Damit die Speisekarte lebhafter wirkt, verkauft man den Kartoffelbrei, der vom Küchenpersonal nur Akademikerkitt genannt wird, als Kartoffelschnee und als Purée de pommes de terre. Das Zeug ist aus dem altbekannten Pulver schnell mit Wasser angerührt und vor allem preiswert. Für nur 50 Cent pro Portion kann man Studenten satt machen. Na ja, zumindest wollen sie danach keine zweite Portion.

Als erfahrener Kenner der objektorientierten Programmierung sehen Sie vermutlich schon die Abhängigkeiten. Die Basisklasse ist Kartoffelbrei, die wir der Kürze halber einfach `Kitt` nennen wollen. `Kitt` stellt für seine Erweiterungen die Grundsubstanz und vor allem den Preis bereit. Als Erweiterungen gibt es den Kartoffelschnee und das Purée de pommes de terre, das sich vom Kartoffelbrei nur durch den Namen unterscheidet.

```
class Kitt {
    public double kostet() {
        return 0.50;
    }
}
```

Listing 10.5: Eine Klasse für den Kartoffelbrei

Der Kartoffelschnee wird als Klasse einfach `Schnee` genannt. Er schmeckt ja auch kaum anders.

```
class Schnee extends Kitt {
    public String getName() {
        return "Kartoffelschnee";
    }
}
```

Listing 10.6: Kartoffelschnee ist eigentlich Kartoffelbrei.

Natürlich wird die Klasse `Purree` analog erstellt. Sie wissen noch, dass sich ein Objekt der Klasse `Schnee` oder `Purree` problemlos der Referenz auf `Kitt` zuweisen lässt, nicht wahr? Das könnte bald wichtig werden.

Suppe fassen!

Aus einem anderen Pulver wird die Suppe erstellt. Sie erinnern sich? Das war die zweite Köstlichkeit, die der Koch der Mensa zubereiten kann. Sie färbt sich beim Anrühren rot und wird darum als Budapester Tomatensuppe angepriesen. Nach zwei Tagen offenen Stehens verfärbt sich dieses Wunderwerk der Lebensmittelchemie ins Bräunliche, sodass sie dann als Gulaschsuppe angeboten werden kann, und wieder zwei Tage später wird sie grün, was an Erbsensuppe erinnert. Gourmets behaupten sogar, dass sie im grünen Zustand etwas würziger schmeckt.

Aus dieser Situation ist es leicht zu verstehen, dass es montags immer Tomatensuppe, mittwochs immer Gulaschsuppe und freitags die Erbsensuppe gibt. Die restlichen beiden Tage lassen sich mit dem Kartoffelbrei auffüllen.

Die Basisklasse heißt `Suppe`, die allerdings aufgrund der ausgefeilten Chemie mit 70 Cent schon etwas teurer ist als der Kartoffelbrei. Auch hier unterscheidet sich der Name. Die Erstellung der Klassen würden Sie auch allein hinbekommen. Aber ich bin ein netter Autor und helfe Ihnen.

```
class Suppe {
    public double kostet() {
        return 0.70;
    }
}
```

Listing 10.7: Die Basissuppe

Die Erweiterung nennen wir `Tomatensuppe`. Die Gulaschsuppe und die Erbsensuppe bekommen Sie allein hin.

```
class Tomatensuppe extends Suppe {
    public String getName() {
        return "Tomatensuppe";
    }
}
```

Listing 10.8: Die Basissuppe

`Gulaschsuppe` und `Erbsensuppe` sind natürlich analog. Nun wollen wir ein kleines Programm schreiben, das die Kosten der Suppen pro Woche ermittelt. Dazu bilden wir ein Array von Suppen und weisen den Elementen die verschiedenen Suppen zu.

```
class Speiseplan {
    public static void main(String[] args) {
        Suppe karte[] = new Suppe[3];
        karte[0] = new Tomatensuppe();
        karte[1] = new Gulaschsuppe();
        karte[2] = new Erbsensuppe();
        double kosten = 0.0;
        for (Suppe gedeck : karte) {
            kosten += gedeck.kostet();
        }
        System.out.println("\nKosten: " + kosten);
    }
}
```

Listing 10.9: Das Hauptprogramm berechnet die Kosten der Suppen.

Wir erstellen eine Speisekarte der Suppentage in der Mensa. Dazu legen wir ein Array für Referenzen der Basisklasse `Suppe` an. Nacheinander werden die verschiedenen Suppen erzeugt. Die Referenzen im Array zeigen auf die verschiedenen abgeleiteten Suppen.

Anschließend können wir eine Schleife über das Array laufen lassen und rufen die Methode `kostet` für jede Suppe auf.

Ein Objekt weiß, was es tut: Polymorphie

Aber es gibt Aufruhr unter den Studenten. Sie behaupten, dass sich nach EU-Vorschrift eine Erbsensuppe nur dann Erbsensuppe nennen darf, wenn wenigstens vereinzelt vollständige Erbsen vorzufinden sind. Daraufhin beschließt die Küche, in den Topf vor der Essensausgabe noch eine Dose Erbsen hineinzugeben. Das verteuert jede Portion Erbsensuppe allerdings um einen Cent. Im Programm können Sie das regeln, indem Sie die Methode `kostet`, die Sie in der Basisklasse definiert haben, in der erweiternden Klasse `Erbsensuppe` neu implementieren. Man spricht vom Überschreiben einer Methode.

```
class Erbsensuppe extends Suppe {
    public String getName() {
        return "Erbsensuppe";
    }
    public double kostet() {
        return 0.71;
    }
}
```

Listing 10.10: Die Erbsensuppe hat eigene Kosten.



Überschreiben hört sich sehr ähnlich an wie überladen, ist aber etwas völlig anderes! Eine gleichnamige Methode mit unterschiedlichen Parametern überlädt die andere Methode. Eine Methode überschreibt die Methode ihrer Basisklasse, wenn Name und Parameter gleich sind.

Lassen Sie noch einmal das Programm laufen und Sie werden sehen, dass sich der Preis um einen Cent erhöht hat. Tatsächlich wird die Methode `kostet` über das Objekt aufgerufen und da das Objekt der Klasse `Erbsensuppe` die Methode der Klasse `Suppe` überschrieben hat, schlägt sich dies hier im Ergebnis durch.



Man spricht von später oder dynamischer Bindung, weil nicht der Compiler festlegt, welche Methode aufgerufen wird, sondern weil erst zur Laufzeit erkannt werden kann, welche Methode aufgerufen werden muss. Dieser Vorgang wird Polymorphie genannt.

Sie können übrigens den Compiler darauf hinweisen, dass Sie die Absicht haben, eine Methode der Basisklasse zu überschreiben, indem Sie die Annotation `@Override` vor die Methode der erweiternden Klasse stellen. Das ist durchaus wichtig, denn sollten Sie sich verschrieben haben oder einer der Parameter nicht passen, wird der Compiler husten und Sie bemerken Ihren Fehler sofort.



Annotationen gibt es seit Java 5. Sie sind sehr praktisch, weil überprüft wird,

ob die Methode wirklich eine Methode der Basisklasse überschreibt. Sollte die Signatur der Basismethode oder der abgeleiteten Klasse nicht übereinstimmen, wird der Compiler sofort anfangen, sich zu räuspern. Diese freundliche Unterstützung des Compilers sollte man in Anspruch nehmen.

Aufruf der Basisklasse

So recht zufrieden bin ich mit der Methode `kostet` der Erbsensuppe noch nicht. Eigentlich ist der Preis ja einen Cent höher als der Suppenpreis. Wenn jemand den Suppenpreis erhöht, wird die Erbsensuppe plötzlich billiger, und das, obwohl die teuren Erbsen drin sind. Was tun? Am besten wäre es, wenn die Erbsensuppe die Methode `kostet` ihrer Basisklasse aufrufen und da dann einfach einen Cent aufschlagen würde. Das geht! Um eine Methode der Basisklasse aufzurufen, muss dem Methodenaufruf statt eines Objektnamens das Wort `super` vorangestellt werden.

```
class Erbsensuppe extends Suppe {
    public String getName() {
        return "Erbsensuppe";
    }
    @Override public double kostet() {
        return super.kostet() + 0.01;
    }
}
```

Listing 10.11: Die Kosten der Erbsensuppe orientieren sich an der Suppe.

Die Methode `kostet` ruft nun zunächst die Methode `kostet` ihrer Basisklasse `Suppe` auf, erhöht den so erhaltenen Preis um einen Cent und gibt ihn an den Aufrufer zurück.

Eine abstrakte Suppe

Bisher berechnet unser Speiseplan nur die Kosten für die Suppe. Aber das Programm soll ja eigentlich einen Wochenspeiseplan ausgeben. Dazu erweitern wir ihn ein wenig. Er muss ja nur noch die Methode `getName` jeder Suppe aufrufen und den gewonnenen String auf dem Bildschirm ausgeben und schon sind wir damit fertig.

```
class Speiseplan {
    public static void main(String[] args) {
        Suppe karte[] = new Suppe[3];
        karte[0] = new Tomatensuppe();
        karte[1] = new Gulaschsuppe();
        karte[2] = new Erbsensuppe();
        double kosten = 0.0;
        for (Suppe gedeck : karte) {
            System.out.println("Heute: " + gedeck.getName());
            kosten += gedeck.kostet();
        }
        System.out.println("\nKosten: " + kosten);
    }
}
```

Listing 10.12: Nun soll auch die Speisekarte gedruckt werden.

Beim Versuch, das Programm zu übersetzen, kommt es dann zur Katastrophe. Die Suppe kennt keine Methode `getName`! Und darum scheitert der Aufruf über `gedeck`. Schade! Aber was bei `kostet` funktioniert hat, müsste doch bei `getName` auch klappen. Wir ergänzen einfach eine Methode `getName` in der Basisklasse `Suppe`. Was drinsteht, ist egal, wird ja eh überschrieben!

```
class Suppe {
    public double kostet() {
        return 0.70;
    }
    public String getName() {
        return "nix";
    }
}
```

Listing 10.13: Eine Pseudomethode für die Namensausgabe

Insgesamt ist es aber etwas unbefriedigend, dass wir eine Methode schreiben, die eh nie ausgeführt wird. Eigentlich ist die ganze Klasse `Suppe` etwas merkwürdig, weil es ja keine Suppe an sich, sondern nur Tomaten-, Gulasch- und Erbsensuppe gibt. Eine Suppe an sich hat keine Geschmacksrichtung und ist darum eher etwas Abstraktes.

Zugegebenermaßen sind auch die anderen Suppen weitgehend geschmacklos. Aber es soll ja keiner merken, dass das alles dieselbe Suppe ist. Eine abstrakte Suppe können Sie in Java durch das Schlüsselwort `abstract` ausdrücken. In einer abstrakten Klasse gibt es mindestens eine abstrakte Methode, die sich darin auszeichnet, dass sie nichts tut.

```
abstract class Suppe {
    public double kostet() {
        return 0.70;
    }
    abstract public String getName();
}
```

Listing 10.14: Die Suppe wird abstrakt.

Die Methode `getName` ist auf ihren Kopf verkürzt. Damit tut die Klasse `Suppe` genau das, was wir wollen. Sie enthält eine Methode, die aber nicht ausgeführt werden darf, sondern von ihren Erweiterungen überschrieben werden muss. Die Klasse `Suppe` wird damit abstrakt.

- Es darf kein Objekt der Klasse `Suppe` erzeugt werden, sondern nur von ihren Erweiterungen, sofern sie die abstrakte Methode implementieren.
- Sie müssen jede Methode, die keinen Methodenblock hat, mit dem Schlüsselwort `abstract` versehen.
- Jede Klasse, die mindestens eine abstrakte Methode hat, ist ebenfalls abstrakt und auch ihr wird das Wort `abstract` vorangestellt.

Übersetzen Sie und Sie werden feststellen, dass das Programm ganz einwandfrei läuft.



Eine abstrakte Klasse ist eine Klasse, die nicht als Basis für Objekte, sondern als Basis für erweiterte Klassen dient. Der Compiler wird verhindern, dass ein Objekt der abstrakten Klasse erzeugt wird.

Allerdings darf es durchaus eine Referenz auf eine abstrakte Klasse geben. Sie kann auf alle Objekte erweiterter Klassen verweisen.

Die Mahlzeit als vollkommen abstraktes Interface

Es wäre schön, wenn unser Programm einen kompletten Wochenplan erstellen könnte, in dem sowohl Suppe als auch Kartoffelbrei berechnet und ausgegeben wird. Und vermutlich sind Sie selbst schon darauf gekommen, wie man dies lösen kann: Wir brauchen eine Basisklasse, von der sich Suppe und Kartoffelbrei ableiten. Diese Basisklasse wollen wir `Mahlzeit` nennen. Natürlich muss `Mahlzeit` die Methoden `kostet` und `getName` kennen.

Aber `Mahlzeit` ist schon sehr abstrakt. Es gibt weder einen Preis noch gibt es einen Namen. Den Preis werden Kartoffelbrei und Suppe definieren und den Namen jedes einzelnen Gerichts. Man könnte `Mahlzeit` so definieren.

```
abstract class Mahlzeit {
    abstract public double kostet();
    abstract public String getName();
}
```

Listing 10.15: Mahlzeit als Basisklasse für Suppe und Kartoffelbrei

Abstrakte Klassen, die nur abstrakte Methoden und keine Attribute enthalten, bezeichnet man als *Interface*. Das Interface dient als Schablone für andere Klassen und gibt Methoden vor, die später per Polymorphie aufgerufen werden können. In Java wird ein Interface folgendermaßen definiert:

```
interface Mahlzeit {
    public double kostet();
    public String getName();
}
```

Listing 10.16: `Mahlzeit` als Interface für Suppe und Kartoffelbrei



Ein Interface macht nichts selbst, sondern schreibt nur anderen vor, was sie zu tun haben.

Ein Interface wird allerding nicht erweitert, sondern implementiert. Darum wird die Klasse `Suppe` auch statt `extends` das Schlüsselwort `implements` verwenden.

```
abstract class Suppe implements Mahlzeit {
    public double kostet() {
        return 0.70;
    }
}
```

Listing 10.17: Interfaces werden implementiert, nicht erweitert

Die Suppe hat nun auch keine abstrakte Methode `getName` mehr. Das braucht sie auch nicht, weil sich `getName` aus der Implementierung des Interface `Mahlzeit` von selbst versteht.

Nun ist alles bereit. Wir können das Hauptprogramm `Speiseplan` anpassen.

```
class Speiseplan {
    public static void main(String[] args) {
        Mahlzeit karte[] = new Mahlzeit[5];
        karte[0] = new Tomatensuppe();
        karte[1] = new Purree();
        karte[2] = new Gulaschsuppe();
        karte[3] = new Schnee();
        karte[4] = new Erbsensuppe();
        double kosten = 0.0;
        for (Mahlzeit gedeck : karte) {
            System.out.println("Heute: " + gedeck.getName());
            kosten += gedeck.kostet();
        }
        System.out.println("\nKosten: " + kosten);
    }
}
```

Listing 10.18: Der Speiseplan mit voller Kostenermittlung

Das Interface und die Mehrfachvererbung

Der Koch kommt auf die Idee, auch Kartoffelsuppe anzubieten, und Sie ahnen schon, wie er sie herstellen wird. Er kippt das Pulver von der Suppe mit dem Pulver des Kartoffelbreis zusammen und rührt daraus die Kartoffelsuppe zusammen.

Nun könnten wir versuchen, die Klasse `Kartoffelsuppe` einfach so zu definieren, dass sie sowohl die Klasse `Suppe` als auch die Klasse `Kitt` erweitert. Aber das wird Ihr Compiler entsetzt von sich weisen. Und dazu hat er einen guten Grund! Denn welche der beiden Methoden `kostet` sollte denn ein Objekt der Klasse `Kartoffelsuppe` aufrufen? Der Compiler würfelt nicht, er braucht klare Vorgaben. Und letztlich stellen wir fest, dass in der Klasse `Kartoffelsuppe` zwar die Methode `kostet` verfügbar sein muss, dass aber weder die Methode der Klasse `Suppe` noch die der Klasse `Kitt` passt, weil die Kartoffelsuppe weder 50 noch 70 Cent, sondern eher etwa 60 Cent kosten wird.

Um solche Mehrdeutigkeiten zu vermeiden, schreibt Java vor, dass eine Klasse nur eine Basisklasse erweitern darf. Das ist in anderen Programmiersprachen wie C++ oder Python durchaus anders – dieses Konzept wird Mehrfachvererbung genannt. Aber auch in Java können Sie eine Art von Mehrfachvererbung erreichen, denn Sie dürfen beliebig viele Interfaces implementieren. Da ein Interface keine Methode implementiert, kann es nicht zu Mehrdeutigkeiten kommen.

Für die Kartoffelsuppe bedeutet dies, dass Sie entweder die Klasse `Suppe` oder die Klasse `Kitt` zum Interface umgestalten müssten.

```
interface Suppe extends Mahlzeit {
    public double kostet();
}

class Kartoffelsuppe extends Kitt implements Suppe {
    public double kostet() {
        return 0.60;
    }
    public String getName() {
        return "Kartoffelsuppe";
    }
}
```

Listing 10.19: Die Suppe wird zum Interface.

Da das Interface `Suppe` aber hier bislang nichts Eigenständiges gegenüber dem Interface `Mahlzeit` zu bieten hat, ist es in diesem speziellen Fall wohl klüger, die Kartoffelsuppe einfach als eine eigenständige Kreation zu feiern und als eigene Implementierung einer Mahlzeit umzusetzen.

```
class Kartoffelsuppe implements Mahlzeit {
    public double kostet() {
        return 0.60;
    }
    public String getName() {
        return "Kartoffelsuppe";
    }
}
```

Listing 10.20: Einfache Lösung

Statische Mitglieder des Interface

Ein Interface ist ein gern genutzter Ort, um Konstanten zu definieren, die für die implementierenden Klassen gültig sind.

So könnte das Interface `Mahlzeit` vielleicht ein paar Konstanten zur Beurteilung der Köstlichkeiten zur Verfügung stellen, die anschaulicher sind als die oft verwendeten Schulnoten.

```
static final int koestlich = 1;
static final int lecker = 2;
static final int essbar = 3;
static final int nahrhaft = 4;
static final int wuergbar = 5;
static final int gesund = 6;
```

Listing 10.21: Geschmacksnoten

Seit Java Version 8 ist es auch möglich, in Interfaces statische Methoden einzubauen. Normale Methoden sind aber nach wie vor nicht erlaubt.

Der Beruf wird geschützt

Sie haben in diesem Kapitel gelernt, dass Klassen aufeinander aufbauen und ganze Hierarchiebäume darstellen können. In dieser Welt sind die Schutzsphären `public` und `private` nicht ausreichend.

Kommen wir noch einmal auf den Menschen zurück, der sich mit seinem Bankkonto vom Nashorn unterscheidet. Und Sie erinnern sich, dass ich behauptet habe, dass auch Autoren Menschen seien. Das war bereits schwer zu glauben. Aber nun versteige ich mich in die Aussage, dass selbst der Koch in der Mensa ein Mensch ist. Daraus würde sich folgende Erweiterungsfolge ergeben:

```
class Tier {
    // ...
}
class Mensch extends Tier {
    int ktoNr;
    String beruf = null;
    // ...
}
class Koch extends Mensch {
    public Koch() {
        beruf = "Koch";
    }
}
```

Im Konstruktor richtet nun jeder Koch seine Berufsbezeichnung ein. Wenn Sie die Studenten fragen, würden diese dem Koch allerdings eine andere Bezeichnung zuordnen. Und das könnten sie auch tun, denn das Attribut `beruf` ist nicht privat.

Setzen Sie allerdings das Attribut aus Sorge vor übler Nachrede auf `private`, kann die Klasse `Koch` nicht mehr übersetzt werden, weil der Koch plötzlich nicht mehr an die Attribute von `Mensch` herankommt.

Für genau diese Fälle stellt Java den Modifizierer `protected` zur Verfügung. Die so geschützte Berufsbezeichnung ist von außen nicht zugreifbar. Die Studenten kommen nicht heran. Aber alle Klassen, die die Klasse `Mensch` erweitern, können ihre eigene Berufsbezeichnung jederzeit ändern.

Das ist durchaus sinnvoll, denn Köche und Menschen sind sich ja nicht fremd, sondern ein Koch ist eben auch Mensch.


Der Modifizierer `protected` erlaubt den Zugriff nur abgeleiteten Klassen.

Anonyme Klassen und das Lambda

Eines Tages wird die Suppe am Freitag nicht wie erwartet grün, sondern verblasst zu einem fahlen Gelb. Irgendwer hat vermutlich heimlich am letzten Wochenende die Töpfe gespült. Der Koch entscheidet, dass die Farbe am ehesten eine Käsesuppe darstellen könnte. Anstatt wie sonst eine eigene Klasse anzulegen, entschließt er sich zu einer anonymen Klasse. Das bedeutet, dass er die Erweiterung der abstrakten Klasse `Suppe` direkt vor Ort durchführt.

```
public class Speiseplan {
    public static void main(String[] args) {
        Mahlzeit karte[] = new Mahlzeit[5];
        karte[0] = new Tomatensuppe();
        karte[1] = new Purree();
        karte[2] = new Suppe() {
            @Override
            public String getName() {
                return "Käsesuppe";
            }
        };
        karte[3] = new Schnee();
        karte[4] = new Erbsensuppe();
        double kosten = 0.0;
        for (Mahlzeit gedeck : karte) {
            System.out.println("Heute: " + gedeck.getName());
            kosten += gedeck.kostet();
        }
        System.out.println("\nKosten: " + kosten);
    }
}
```

Listing 10.22: Die Käsesuppe tritt anonym auf.

Am Mittwoch wird also eine neue Suppe erzeugt, obwohl Suppe ja eigentlich eine

abstrakte Klasse ist. Es kann kein Objekt geben, solange die Methode `getName` nicht implementiert ist. In diesem Fall wird das vor Ort nachgeholt. Dazu wird hinter der Suppendefinition eine geschweifte Klammer geöffnet, in der die fehlende Methode `getName` steht.

Damit ist eine namenlose Klasse entstanden, die `Suppe` erweitert. Da von der Klasse ansonsten nichts weiter erwartet wird und dieser Fall nicht oft auftreten wird, lohnt es sich nicht, dafür eine eigene Klasse anzulegen.



Sie können sich von Eclipse helfen lassen. Wenn Sie nach dem Wort `Suppe` die geschweifte Klammer setzen und die Return-Taste drücken, erzeugt Eclipse automatisch die schließende Klammer, meldet aber noch den Fehler, dass es Methoden gibt, die nicht implementiert sind, da `Suppe` ja eine abstrakte Methode enthält. Klicken Sie nun auf das rote Symbol am Anfang der Zeile, zeigt Eclipse ein Menü an, in dem es als Lösung »Add unimplemented Methods« anzeigt. Wenn Sie darauf doppelt klicken, erzeugt Eclipse den Rahmen für `getName`.

Sie werden sich fragen, warum ich Sie mit solchen Zaubertricks belästige. Man kann doch nun wirklich für die Klasse einen Namen erfinden und dafür den Rahmen einer Klasse herstellen. Tatsächlich wird diese Praxis bei grafischen Oberflächen gern verwendet, wenn es darum geht, die Aktionen von Buttons oder Mäusen zu fangen.

Grafische Oberfläche behandelt Events

Gerade bei grafischen Oberflächen werden die Polymorphie und das Erweitern von Klassen ausgiebig genutzt. Das folgende Beispiel zeigt einen Button, dessen Methode `addEventHandler` eine Klasse erwartet, die ein Interface `EventHandler` implementiert. Wenn Sie nicht für jeden Button eine neue Klasse anlegen wollen, verwenden Sie eine anonyme Klasse, die `EventHandler` implementiert und damit dessen Methode `handle` realisiert. Diese wird dann ausgeführt, wenn auf den Button geklickt wird.

```
Button button = new Button("OK");
button.addEventHandler(new EventHandler() {
    @Override
    public void handle(Event e) {
        label.setText("Klick");
    }
});
```

Listing 10.23: Pseudocode für eine GUI, die ein Event behandelt

Listing 10.23 verwendet vereinfachte Namen und Methoden, um die Übersicht zu behalten. Tatsächlich finden Sie ähnliche Konstrukte sowohl in Swing als auch in

FX für die Behandlung von Buttons, Mäusen oder ähnlichen Ereignislieferanten.

Lambda

Ereignisbehandler sind meist Interfaces mit nur einer Methode, die das Ereignis behandeln. Damit erhält eine solche Klasse auch einen neuen Charakter. Eine solche Klasse ist weniger das Modell eines Objekts als vielmehr ein funktionales Modul. Für Interfaces mit nur einer Methode stellt Java ab Version 8 *Lambda* zur Verfügung.

Damit kann alles weggelassen werden, was sich von selbst versteht.

- ✔ Der Klassenname der Implementierung interessiert nicht, da alles sofort aufgelöst wird.
- ✔ Der Name der Methode ist ebenfalls überflüssig, da es nach Voraussetzung nur eine gibt.
- ✔ Der Typ des Parameters braucht auch nicht explizit erwähnt zu werden, da er durch die Methoden-Definition des Interface festgelegt ist.

Nach dem Entfernen aller überflüssiger Klammern und Annotationen passt nun alles in eine Zeile:

```
button.addEventHandler( (e) -> label.setText("geklickt") );
```

Das Interessante ist, dass hier ein Objekt auf eine Methode reduziert wird. Darum bezeichnet man diese Denkweise als *funktionale Programmierung*. Sie können also nun beim nächsten Entwicklertreffen damit Eindruck schinden, indem Sie erklären, dass durch das Lambda die funktionale Programmierung in Java Einzug gehalten hat.

Aufgaben

Nun sind Sie an der Reihe! Schreiben Sie ein paar Programme und schauen Sie, ob Sie alles verstanden haben!

Abstrakte Musik

Als Übungsaufgabe für erweiterte und abstrakte Klassen ziehen wir die Musik heran. Selbst, wenn Sie kein Instrument beherrschen, können Sie nun endlich ein Orchester leiten.

In einem Orchester oder in einer Band gibt es immer Instrumente, die den Bass übernehmen. Der Bass ist kein eigenes Instrument, sondern eine Tonlage und kann beispielsweise von einer Tuba, einem Kontrabass, einer Bassgitarre oder einem Moog-Synthesizer übernommen werden.

Damit ist der Bass eine ganz typische abstrakte Klasse, weil er selbst keine Töne

produzieren kann. Ein Instrument wie die Tuba ist ein Bass, von dem es dann tatsächlich Exemplare gibt.

Ein Bass kann tiefe Töne erzeugen, was wir hier einfach mal als Brummen bezeichnen wollen. Wie sich der Klang tatsächlich anhört, bestimmt das Instrument. Eine Tuba klingt ganz anders als ein Kontrabass oder ein E-Bass.

Schreiben Sie eine abstrakte Klasse `Bass`, die die Implementierung von `brumm` vorschreibt, und zwei Erweiterungen wie beispielsweise `Tuba` oder `BassGitarre`. Dann können Sie in der Hauptklasse `MainBass` einen Bass erzeugen, der dann brummt.

```
public class MainBass {
    public static void main(String[] args) {
        Bass tief = new Tuba();
        tief.brumm();
    }
}
```

Listing 10.24: Wir brummen ein wenig.

Die Implementierungen der Methode `brumm` sollen einfach auf dem Bildschirm eine Meldung ausgeben, aus der hervorgeht, welches Instrument die Arbeit übernimmt.

Sie können `Bass` als abstrakte Klasse oder als Interface realisieren. Versuchen Sie beides! Was ändert sich an den abgeleiteten Klassen?

Die Firma als Spielplatz für erweiterte Personen

Modellieren Sie das Umfeld einer Firma und insbesondere die Menschen, die mit einer Firma zu tun haben. Als Basisklasse wählen Sie hier eine Person und da es bei Firmen in erster Linie ums Geld geht, entwickeln Sie eine Methode `erhaeltZahlung`.

Nun erweitern Sie die Personen zu Kunden, Angestellten, Lieferanten.

Angestellte haben Namen und Bankverbindungen wie jede Person, gehören aber zu einer Abteilung. Als Zahlung erhalten sie ein Gehalt, über das auch die Personalabteilung informiert werden muss. Darum muss in der Methode `erhaeltZahlung` eine Methode `informierePersonalabteilung` aufgerufen werden.

Ein Lieferant bekommt eine Bezahlung für eine Lieferung. Und er hat natürlich eine Lieferantennummer. Hier wird die Methode `bezahleRechnung` der Warenwirtschaft aufgerufen.

Kunden haben eine Kundennummer. Auch der Kunde erhält manchmal Zahlungen. Beispielsweise nach Reklamationen. Diese Zahlung interessiert zwar die Personalabteilung nicht, aber die Warenwirtschaft schon.

Nun können Sie die Polymorphie austesten, indem Sie beispielsweise ein Kundenobjekt einer Personenreferenz zuweisen und über die Person `erhaeltZahlung` aufrufen.

```
Kunde kunde = new Kunde();
Person person = kunde;
person.erhaeltZahlung();
```

Wenn Sie alles richtig gemacht haben, müsste sich die Methode `erhaeltZahlung` der Klasse `Kunde` melden. Versuchen Sie das doch auch einmal mit Methodenparametern, indem Sie den Betrag übergeben.

Schreiben Sie ein Programm, indem Sie Objekte der Klassen `Person`, `Angestellter`, `Kunde` und `Lieferant` in einem Array von Personen ablegen. Rufen Sie dann `erhaeltZahlung` auf, wie unten gezeigt.

```
Person []person = new Person[4];
person[0] = new Angestellter();
person[1] = new Kunde();
person[2] = new Lieferant();
person[3] = new Person();
for (int i=0; i<4; i++) {
   person[i].erhaeltZahlung(12.5);
}
```

Listing 10.25: Polymorphes Zahlen

Oft sind Lieferanten keine Personen, sondern Unternehmen. Und selbst Kunden können Unternehmen sein. Sie haben zwar weiterhin die Eigenschaften einer Person. Aber darüber hinaus haben sie beispielsweise eine Gesellschaftsform. Außerdem haben wir bei einem Unternehmen einen oder mehrere Ansprechpartner, die wiederum einen Namen und eine Durchwahl haben.

Ein Lieferant könnte also zwei Klassen erweitern: das Unternehmen und die Person. Wenn nun sowohl die Klasse `Person` als auch die Klasse `Unternehmen` eine Methode `getTelefonnummer` implementiert, ist bei einem Aufruf durch einen Lieferanten nicht klar, welche der beiden Implementierungen verwendet würde. Also müssen Sie die Person oder das Unternehmen als Interface deklarieren.

Testen Sie @Override

Schauen Sie mal, ob dieses `@Override` wirklich funktioniert. Verwenden Sie absichtlich einen falschen Methodennamen und/oder falsche Parameter. Wann wird der Compiler meckern?

- ✔ Wenn die Parameternamen geändert werden?
- ✔ Wenn der Parametertyp zuweisungskompatibel, aber anders ist?

- ✔ Wenn der Rückgabetyp sich unterscheidet?

Probieren Sie es aus. Versuch macht kluch!

Erweiterte Referenzen

Welches Objekt kann welcher Referenz zugewiesen werden? Dazu gehen Sie von folgenden Erweiterungsketten aus:

```
class Person extends Tier {
class Kunde extends Person {
```

Nun kommen ein paar Anweisungen, die teils korrekt sind und teils vom Compiler und teils vom Laufzeitsystem unterbunden werden:

```
Object obj = kunde;
Object obj = (Object)kunde;
kunde = obj;
tier = kunde;
tier = (Tier)kunde;
tier = mensch;
mensch = tier;
kunde = tier;
person = kunde;
person = (Person)tier;
```

Überlegen Sie zunächst einmal selbst und lassen Sie sich bei Bedarf vom Compiler helfen. Der sagt die Wahrheit, die ganze Wahrheit und nichts als die Wahrheit!

Was passiert eigentlich, wenn wir ein Tier auf eine Person casten und über diese dann `erhaeltZahlung` aufrufen? Gibt es eine Fehlermeldung? Wann? Beim Kompilieren oder beim Ausführen?

Kapitel 11

Verteilung der Klassen auf Pakete

IN DIESEM KAPITEL

- Packages als Ordnungsmittel für Namensräume
- JAR-Dateien als Verpackung der Klassendateien und startbare Dateien
- Externe JAR-Dateien in das Projekt einbinden

Packages

Beim Erstellen einer Klasse fragt Eclipse jedes Mal nach einem Package und ist beleidigt, wenn es keinen Package-Namen bekommt. Obwohl ich Ihnen nicht erklärt habe, was es mit den Packages auf sich hat, haben Sie weitergelesen. Das zeugt von Charakter.

Wenn Projekte groß werden, verteilt man die Lasten gern auf die Kollegen und jeder bekommt seine Klassen. Wenn dann jemand eine Klasse `Datum` gut gebrauchen könnte, legt er sich schnell eine an. Der fantasiebegabte Programmierer aus dem Nachbarbüro benötigte auch eine Datumsklasse und der hat seine Klasse doch auch einfach `Datum` genannt. Jetzt ist der Knatsch groß, gibt es doch zwei Klassen mit demselben Namen.

Bevor Sie verhandeln, wer die Schuld trägt, sollten Sie über eine bessere Unterteilung der Aufgaben nachdenken. Sie haben Ihre Klassen einfach mit dem Klassennamen angesprochen. Das ist etwa so, wie es in einem kleinem Dorf mit drei Häusern und einem Klo üblich ist. Dort sprechen sich alle Nachbarn mit Vornamen an. Erst wenn viele Menschen zusammenkommen, dann nimmt man den Nachnamen hinzu.

In ähnlicher Weise sind die Klassen sozusagen die Vornamen und die Packages die Nachnamen. Mit einem feinen Unterschied: Beim Zugriff auf Klassen werden die Packages zuerst genannt, also so wie bei »Meiers Sepp«.

Eclipse macht Packages

Schauen wir mal, was Eclipse tut, wenn man seinem Drängen nach einem Package-

Namen nachgibt. Da ich auch etwas für meine Gesundheit tun sollte, wähle ich ein Beispiel aus dem Bereich des Sports. Die Klasse soll `Ball` heißen.

1. Legen Sie ein Java-Projekt mit dem Namen `Packages` an: File|New|Project|Java Project.
2. Klicken Sie das Projekt mit der rechten Maustaste an und erstellen Sie über das Kontextmenü eine neue Klasse: New|Class.
3. Im Dialog geben Sie als Namen `Ball` an.
4. Sie sehen oben in der Dialogbox ein gelbes Warnschild mit dem Hinweis, dass die Verwendung des Default-Packages nicht empfohlen wird.
5. Tragen Sie unter Package nun `tennis` ein.
6. Weil es so einen Spaß macht, erzeugen Sie nun wieder eine neue Klasse und verwenden Sie dieses Mal als Package `golf`.

Sie sehen nun in der Projektübersicht, wie die Klassen unter den Packages `golf` und `tennis` abgelegt sind. Wenn Sie nun heimlich auf Ihrer Festplatte im Projektpfad nachschauen, werden Sie feststellen, dass `golf` und `tennis` als Verzeichnisse angelegt wurden.

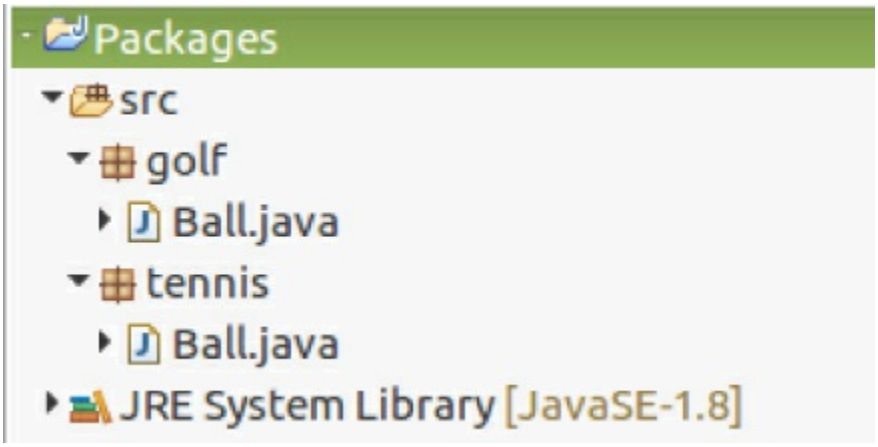


Abbildung 11.1: Das Projekt mit seinen Packages

Wenn Sie in das Listing der Klasse `Ball` schauen, finden Sie den Verweis auf das Package in der obersten Zeile.

```
package golf;

public class Ball {

}
```

Listing 11.1: Der Befehl `package` steht immer in der ersten Zeile.

Vielleicht würden Sie zum Golfspielen doch lieber einen Tennisball verwenden, weil man diesen winzigen Golfball mit dem beknackten Schläger ums Verrecken

einfach nicht trifft. Dann greifen Sie auf den Tennisball zu, indem Sie der Klasse `Ball` den Package-Namen `tennis` durch einen Punkt getrennt voranstellen. Verwenden Sie innerhalb des Packages `golf` die Klassenbezeichnung `Ball` ohne weitere Package-Angabe, wird immer der Golfball verwendet.

```
package golf;

public class Ball {
    public static void main(String[] args) {
        tennis.Ball meinBall = new tennis.Ball();
    }
}
```

Listing 11.2: Zugriff auf einen Tennisball



Wollen Sie auf die Klasse `Ball` im Package `tennis` von einem Package außerhalb des Packages `tennis` zugreifen, muss die Klasse `Ball` als `public` deklariert sein.

Nachträgliches Packen

Sie schreiben schon wieder eine Klasse `Ball`, müssen aber feststellen, dass sie weder etwas mit Golf noch mit Tennis zu tun hat. Also setzen Sie in die erste Zeile einen passenden Package-Namen wie `feuerwehr`.

```
package feuerwehr;
class Ball {
    ...
}
```

Listing 11.3: Die Klasse `Ball` soll dem Package `feuerwehr` zugeordnet werden.

Wenn Sie in Eclipse eine Klasse nachträglich zuordnen, wird Eclipse einen Fehler melden. Logisch: Die Klasse befindet sich ja nicht in dem Package. Wenn Sie nun auf die rote Marke links neben der Zeile klicken, schlägt Ihnen Eclipse zwei Lösungen vor:

- Verschieben der Klasse in das genannte Package. Das ist vermutlich das, was Sie auch erreichen wollten. Falls das Package noch nicht existiert, legt Eclipse es sogar für Sie an.
- Entfernen der Zeile mit dem Befehl. Dann arbeiten Sie weiterhin mit dem Default-Package.



Der Befehl `package` darf nur einmal in einer Quelltextdatei auftreten und steht immer ganz vorn.

Importe von Packages

Um Package-Namen weltweit eindeutig zu machen, verwendet man gern die URL der eigenen Website. Diese ist zwangsläufig eindeutig. Um der Hierarchie-Logik der Package-Namen zu folgen, dreht man die URL allerdings um. So könnte die Firma Tacoss ihr Finanzbuchhaltungs-Package etwa `de.tacoss.fibu` nennen. Dieser Name ist weltweit eindeutig, weil es nur eine URL namens `www.tacoss.de` gibt. Wird für die Packagestruktur wie oben beschrieben die URL verwendet, würden die Zugriffe etwas länglicher werden. Nehmen wir an, im Package `fibu` gibt es eine Klasse `Datum`.

```
package de.tacoss.fibu;

public class Datum {
   int tag, monat, jahr;
}
```

Listing 11.4: Datum in einer vollständigen Package-Struktur

Und natürlich wird der Pfad für den Zugriff auch immer länger.

```
class TestTacoss {
  public static void main(String[] args) {
    de.tacoss.fibu.Datum termin = new de.tacoss.fibu.Datum();
  }
}
```

Listing 11.5: Zugriff auf ein längeres Datum

Wenn Sie diesen Klassennamen immer wieder tippen müssen, werden Sie mehr Zeit damit verbringen, Tippfehler zu korrigieren, als Programme zu schreiben. Damit der lange Package-Name nicht ständig wiederholt werden muss, können Sie das Ganze abkürzen, indem Sie die Klasse `Datum` explizit importieren.

```
import de.tacoss.fibu.Datum;

class TestTacoss {
  public static void main(String[] args) {
    Datum termin = new Datum();
  }
}
```

Listing 11.6: Zugriff auf ein längeres Datum mit `import`

Sollten Sie auf mehrere Klassen des Packages `fibu` zugreifen wollen, können Sie entweder alle Klassen aufzählen oder mit einem Stern verkürzen.

```
import de.tacoss.fibu.*;
```

Und schon zeichnet sich am Horizont das Problem ab, dass zwei Klassen in den verschiedenen Packages den gleichen Namen tragen könnten. Was nun? Zunächst wird der Compiler wieder ein paar Bemerkungen dazu machen, wie etwa: »The type Datum is ambigious«. Und dann werden Sie wohl den Typ mit Package-Namen spezifizieren müssen, also `de.tacoss.fibu.Datum` verwenden müssen.



Ohne die explizite Angabe des Packages können Sie auf alle Klassen des eigenen Packages, auf die des Packages `java.lang` und auf das Package ohne Namen zugreifen. Ansonsten müssen Sie den kompletten Package-Namen vor die Klasse stellen oder einen entsprechenden `import`-Befehl absetzen.

Importlisten unter Eclipse generieren lassen

Sie bekommen mit dem Java-Interpreter eine große Menge an Programmbibliotheken mitgeliefert. Darin verbergen sich die tollsten Klassen, die Sie in Ihren Programmen verwenden können. Natürlich sind diese in Packages organisiert und müssen darum mit entsprechenden Importen versehen werden. Dabei können Sie sich allerdings von Eclipse helfen lassen. Nehmen wir an, Sie wollen ein `Label` benutzen. Dazu schreiben Sie in Ihrem Programm die folgende Zeile:

```
Label label = new Label("Ich bin ein Label");
```

Sofort werden die Wörter `Label` rot unterkringelt und es erscheint ein rotes Fehlersymbol links neben der Zeile. Darauf klicken Sie bitte! Es erscheint eine Vorschlagliste. Auf den vorderen Plätzen schlägt Eclipse vor, einen entsprechenden Import-Befehl hinzuzufügen.

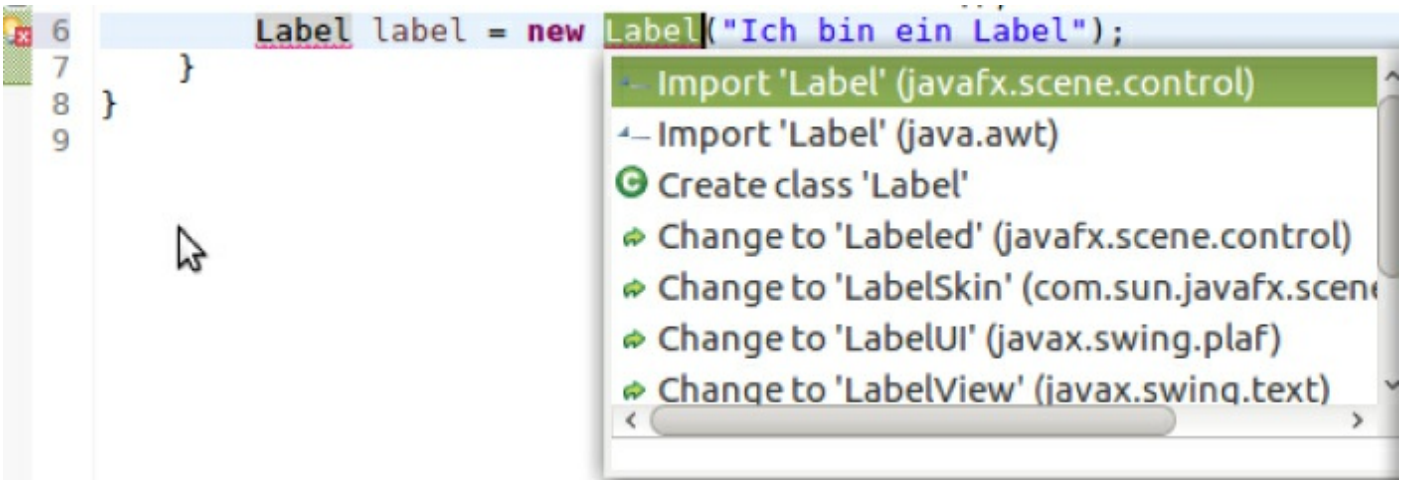


Abbildung 11.2: Automatische Importsuche

`Label` kann aus zwei verschiedenen Quellen bezogen werden. In diesem Fall haben Sie die Wahl zwischen AWT und FX. Sie sollten also schon grob wissen, welchen `Label` sie benötigen. Wählen Sie einen von beiden durch Doppelklick, sehen Sie, wie sich das Rot auflöst und eine Import-Zeile entsteht. Auf diese Weise kann man sich also seine Importe schnell zusammenklicken.

Wenn Sie allerdings später feststellen, dass Sie den `Label` gar nicht brauchen, werden Sie nach dessen Entfernung vielleicht nicht mehr in Erinnerung haben, dass Sie den Import ja noch am Anfang des Listings stehen gelassen haben, den Sie eigentlich gar nicht mehr benötigen. Das ist kein Drama, schließlich frisst der Import ja kein Brot. Allerdings sind überlange Importlisten unübersichtlich.



Eclipse klappt längere Importlisten sogar von sich aus ein. Um sie noch einmal lesen zu können, müssen Sie dann auf ein kleines Pluszeichen vor dem ersten Import klicken.

Importlistpflege durch Eclipse

Importe, die nicht mehr gebraucht werden, unterstreicht Eclipse gelb. Die können Sie also ohne Schaden löschen. Sie können die Liste aber auch von Eclipse automatisch aufräumen lassen, indem Sie die Tastenkombination Strg + ⇧ + O (O wie Ordnung) eingeben.

Unnötige Importe verschwinden. Sollten Sie aber im Listing Klassen ohne den zugehörigen Import haben, wird Eclipse diesen Import automatisch erzeugen, wenn er eindeutig ist, oder Ihnen seine Vorschläge unterbreiten, wenn die Klassen in mehreren Packages verfügbar sind.

Packages, Verzeichnisse und die Kommandozeile

Intern organisiert Java die Packages in Verzeichnissen. Diese Organisation können Sie auch unter Eclipse schon erkennen. Wenn Sie allerdings auf der Kommandozeile mit dem JDK entwickeln, müssen Sie die Verzeichnisse selbst anlegen. Die folgende Klasse `Datum` gehört zum Package `fibu`.

```
package fibu;
public class Datum {
}
```

Wenn Sie den Java-Compiler `javac` aufrufen, wird er die Datei *Datum.java* im Unterverzeichnis *fibu* suchen. Der Aufruf lautet dann:

```
javac fibu/Datum.java
```

Der Java-Compiler akzeptiert den Schrägstrich als POSIX-Standard für Verzeichnistrenner sogar unter Windows. Windows-Benutzer können aber gern auch einen Backslash verwenden.

Auch für den Start einer Klasse über das JRE müssen Sie das Package-Verzeichnis angeben, in dem sich die Klasse befindet. Wenn Sie in der Klasse `Datum` beispielsweise zu Testzwecken eine `main`-Methode eingebaut haben, können Sie die Klasse durch einen Aufruf über das Package-Verzeichnis starten.

```
java fibu/Datum
```


Packages-Strukturen werden als Verzeichnisse abgebildet.

Öffentlichkeit der Klassenzugriffe

Das Weglassen der Deklaration von `public` bei einer Klasse hat Konsequenzen. Ohne `public`- oder `private`-Deklaration ist eine Klasse nur innerhalb des eigenen Packages zugreifbar. Das ist bislang nicht weiter aufgefallen, weil alle unsere Beispiele keine Packages verwendet haben und damit zum Package `default` gehört haben, dessen Name nicht angegeben werden muss.

- ✔ public: Für alle Klassen zugreifbar
- ✔ private: Nur innerhalb der Klasse zugreifbar
- ✔ protected: Nur von erweiternden Klassen zugreifbar
- ✔ kein Modifizierer: Nur innerhalb des eigenen Packages zugreifbar

Ausführbare Dateien und Bibliotheken: JAR

Um eine Anwendung zu schreiben, werden Sie mehr als eine Klasse benötigen. Jede Klasse wird vom Java-Compiler in eine eigene Datei mit der Endung *.class* übersetzt. Um Ihr Programm aufzurufen, müssen Sie die Klasse, die die Methode `main` enthält, mit dem Java-Interpreter, also mit dem Befehl `java` aufrufen.

Das ist nicht besonders praktisch. Darum lassen sich die *class*-Dateien zu einer Datei zusammenfassen, die die Endung *.jar* trägt. Eine Datei mit der Endung *.jar* ist in ihrem Aufbau vergleichbar mit einer ZIP-Datei. Sie enthält alle benötigten *.class*-Dateien einer Anwendung oder auch einer Bibliothek. Hinzu kommen Ressourcen wie beispielsweise Bilder oder Konfigurationsdateien. Eine JAR-Datei kann dann, je nach Charakter, als ausführbares Programm oder als Bibliotheksdatei verwendet werden.

Erstellen einer JAR-Datei

Der Befehl `jar` erzeugt JAR-Dateien und kann sie auch wieder entpacken. Der Name erinnert nicht ohne Grund an den UNIX/Linux-Befehl `tar`, der ebenfalls Dateien in einer Datei zusammenpacken kann. Der Befehl `jar` verwendet sogar ganz ähnliche Kommandos wie der Befehl `tar`.

Solange die virtuelle Maschine von Java nur eine Class-Datei bekommt, sucht sie einfach nach deren `main`-Methode und startet sie. Eine JAR-Datei enthält in der Regel eine größere Zahl von Class-Dateien, die theoretisch alle eine `main`-Methode enthalten können.

Das Manifest

Um der virtuellen Maschine ein paar Tipps auf den Weg zu geben, enthält eine JAR-Datei eine Manifest-Datei im Verzeichnis *META-INF*. Darin steht unter anderem, welche Klasse die relevante `main`-Methode enthält. Der entsprechende Eintrag lautet:

```
Main-Class: Main
```

Der Befehl `jar` legt beim Erzeugen einer JAR-Datei automatisch eine leere Manifest-Datei an. Das hilft Ihnen für eine ausführbare JAR-Datei wenig. Darum müssen Sie die Datei selbst erstellen, bevor `jar` zupackt.

Eclipse und JAR

JAR-Dateien werden nur noch selten von Hand zusammengestellt. Schließlich nimmt Ihnen Eclipse diese Arbeit ab und macht dabei keine Flüchtigkeitsfehler.

1. Klicken Sie Ihr Projekt mit der rechten Maustaste an und wählen Sie den Punkt Export.
2. Es erscheint ein Dialog. Hier wählen Sie unter dem Punkt Java den Punkt Runnable JAR File. Klicken Sie auf Next.
3. Unter Launch Configuration klappt eine Liste auf, in der alle Klassen erscheinen, die im aktuellen Workspace eine statische Methode `main` enthalten. Suchen Sie diejenige aus, die der Startpunkt Ihrer JAR-Datei sein soll.
4. Unter Export Destination tragen Sie den Pfad und den Namen der JAR-Datei ein. Etwas einfacher wird es, wenn Sie über den Button Browse einen Ort suchen.
5. Mit dem Button Finish wird die Datei erzeugt.

JAR-Dateien ausführen

Eine JAR-Datei können Sie wie eine Klassendatei von der Konsole mit dem Befehl `java` starten. Sie müssen allerdings noch die Option `-jar` hinzufügen.

```
java -jar meinlieblingsprogramm.jar
```

Auf Computern mit einer grafischen Oberfläche können Sie eine JAR-Datei auch durch doppeltes Anklicken starten. Dazu sind ein paar Voraussetzungen erforderlich:

- ✔ Es muss eine Java Runtime Environment installiert sein.
- ✔ Die Datei muss gegenüber dem System als ausführbar gekennzeichnet werden.
- ✔ Der Dateityp *.jar* muss so zugeordnet werden, dass er von dem Programm `java` des Java Runtime Systems geöffnet wird.

Windows

- Mit der rechten Maustaste auf die JAR-Datei klicken und ÖFfnen Mit anklicken.
- Die Java Runtime Environment wird gleich vorgeschlagen. Ansonsten muss nach javaw.exe gesucht werden. Üblicherweise befindet sich die Java-Umgebung unter *C:\Programme\Java*. Im installierten Verzeichnis nach *bin\javaw.exe* suchen.
- Achten Sie darauf, dass »immer mit diesem Programm öffnen« ausgewählt ist.

Linux

Linux hat die Besonderheit, dass eine Datei nicht allein aufgrund ihrer Namenserweiterung gestartet wird. Jeder Datei muss einzeln das Recht auf Ausführbarkeit zugewiesen werden.

- Die JAR-Datei mit der rechten Maus anklicken und den Punkt Eigenschaften auswählen.
- Unter der Rubrik Zugriffsrechte einen Haken an die Ausführbarkeit der Datei setzen.
- Unter der Rubrik ÖFfnen Mit die passende JRE auswählen. Auf Als Vorgabe Festlegen klicken. Dadurch starten alle ausführbaren JAR-Dateien auf einen Doppelklick hin.
- Mit Schliessen den Dialog beenden.

Macintosh

Nach dem Installieren von Java ist der Macintosh normalerweise bereits so eingerichtet, dass die JAR-Datei auf einen Doppelklick hin startet. Verantwortlich zeichnet das Programm Jar Launcher.



In der Entwicklungsumgebung Xcode gibt es ein Programm Jar Bundler, das aus der JAR-Datei eine native Mac-Anwendung macht. Der Nachteil ist natürlich, dass das komplette JRE zum Programm hinzugebunden wird.

JAR-Datei als Bibliothek einbinden

Obwohl die Entwicklungsbibliothek – also das JDK – schon reichlich mit Bibliotheken gesegnet ist, werden immer wieder praktische Bibliotheken geschrieben, an die niemand zuvor gedacht hat. Sie werden im Rahmen des Buches beim Umgang mit Datenbanken, mit JSON oder Jersey darauf stoßen, dass JAR-Dateien benötigt werden.

Aber auch bei größeren Software-Projekten könnte es sich als sinnvoll erweisen,

mit mehreren JAR-Dateien zu arbeiten und dafür eigene Bibliotheken einzubinden.

Nachdem Sie die JAR-Dateien beispielsweise aus dem Internet heruntergeladen haben, befinden sie sich typischerweise im Download-Verzeichnis. Dort nützen sie Ihnen aber nichts, sie müssen in das Projekt eingebunden werden. Dazu gibt es grundsätzlich zwei Vorgehensweisen:

- Sie kopieren die JAR-Dateien in das Projekt. Damit befinden sie sich genau da, wo sie auch gebraucht werden. Sie befinden sich intern im Projekt.
- Sie verweisen im Projekt nur auf den externen Ort, an dem die JAR-Dateien liegen. Das erspart Speicher, wenn die JAR-Dateien für mehrere Projekte benötigt werden.

JAR-Dateien zum Projekt kopieren

Es empfiehlt sich, im Projekt ein Verzeichnis für die einzubindenden Bibliotheken anzulegen. Als Namen wird dazu üblicherweise *lib* verwendet.

1. Sie legen ein Verzeichnis an, indem Sie Ihr Projekt mit der rechten Maustaste anklicken und im Menü New|Folder wählen.
2. Im folgenden Dialog geben Sie den Namen des Verzeichnisses an. Der Name *lib* hat sich eingebürgert.
3. Sie kopieren die Dateien aus dem Download-Verzeichnis, indem Sie den Dateimanager Ihres Systems öffnen (Explorer, Finder, Caja, Nautilus, je nach System), die JAR-Dateien mit der rechten Maustaste anklicken und auf Kopieren oder Copy klicken.
4. Klicken Sie nun das Verzeichnis in Ihrem Projekt mit der rechten Maustaste an und wählen Sie Paste. Anschließend befindet sich eine Kopie der JAR-Datei in Ihrem Projekt.

Je nach System können Sie auch die Dateien direkt hinüberziehen. Dabei taucht eventuell die Rückfrage auf, ob Sie koperen wollen oder einen Link auf die Dateien verwenden möchten.

Nun sind die JAR-Dateien zwar glücklich im Projekt, aber noch weiß der Compiler gar nicht, dass er sie auch einbinden soll. Dazu müssen sie in den Build Path geholt werden.

1. Sie binden die JAR-Datei logisch ein, indem Sie das Projekt mit der rechten Maustaste anklicken und Properties wählen.
2. Es erscheint ein Dialog, in dessen linker Spalte Sie Java Build Path anwählen.
3. Auf der rechten Seite finden Sie einen Reiter Libraries, den Sie anklicken. Sie sehen nun den Dialog aus Abbildung 11.3.

4. Klicken Sie auf den Button Add Jars.
5. Im folgenden Dialog öffnen Sie Ihr Projekt durch Klicken auf das Dreieck links neben dem Projektnamen.
6. Der Ordner *lib* taucht auf, öffnen Sie auch diesen.
7. Es erscheinen Ihre JAR-Dateien. Wählen Sie sie aus und klicken Sie auf OK.
8. Die JAR-Dateien erscheinen in der Liste der Libraries. Diesen Zustand bestätigten Sie mit Apply And Close.

Nun erscheinen die JAR-Dateien außer im Verzeichnis *lib* auch im Projektverzeichnis *Referenced Libraries*.

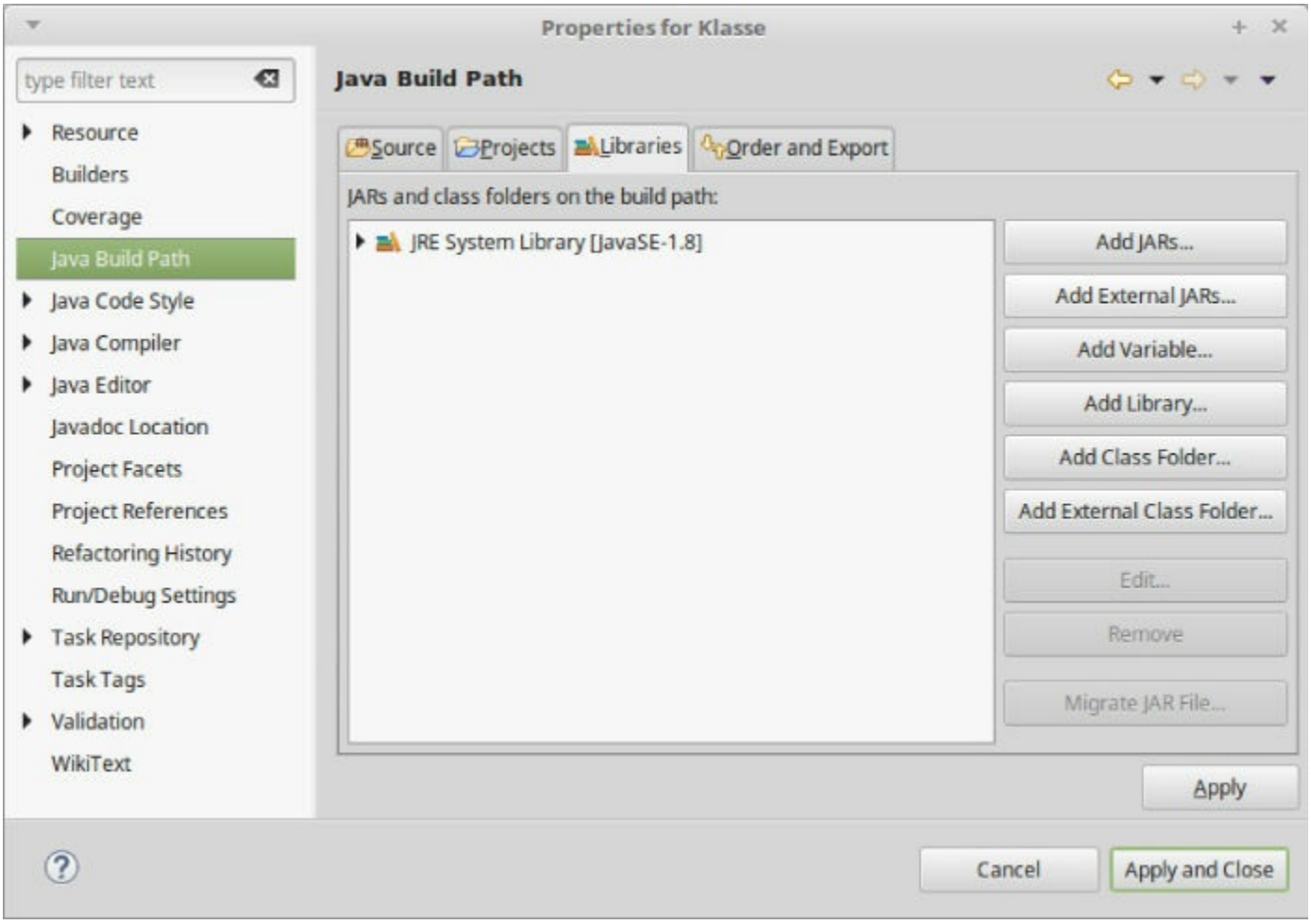


Abbildung 11.3: Hinzufügen von JARs zu den Libraries im Build Path

Wenn die JAR-Dateien nur referenziert werden sollen, gehen Sie ähnlich vor.

1. Sie klicken mit der rechten Maustaste auf das Projekt und gelangen über Properties|Java Build Path|Libraries in den Dialog von Abbildung 11.3.
2. Über den Button Add External Jars erreichen Sie eine Dateiauswahlbox, mit deren Hilfe Sie die JAR-Dateien aus den Dateien auf Ihrem Computer auswählen. Die Dateiem erscheinen in der Liste Ihrer JARs.
3. Sie bestätigen auch hier mit Apply And Close.

Auch diese JAR-Dateien erscheinen im Projektverzeichnis *Referenced Libraries*, allerdings steht hinter den Dateien ein Verweis auf das Verzeichnis, auf das referenziert wird.



Wollen Sie beim Debuggen in die JAR-Datei hineinsehen, um zu prüfen, was die Bibliothek mit Ihren Aufrufen macht, dann werden Sie auch die Quelltexte der JAR-Datei benötigen. In vielen Fällen können Sie diese an gleicher Stelle finden, wo Sie auch die JAR-Datei selbst fanden.

JAR aus dem Projekt entfernen

Einträge im Verzeichnis *Referenced Libraries* können nicht einfach per Delete gelöscht werden, sondern müssen auf dem gleichen Weg entfernt werden, wie sie entstanden sind.

1. Sie klicken Ihr Projekt mit der rechten Maustaste an und wählen Properties|Java Build Path|Libraries.
2. In der Liste finden Sie die JAR-Datei. Klicken Sie sie an!
3. Der Button Remove wird aktiv und kann angeklickt werden.
4. Bestätigen Sie den Vorgang mit dem Button OK.

Die Dateien sind aus dem Verzeichnis *Referenced Libraries* verschwunden. Falls Sie auch Dateien aus dem Verzeichnis *lib* entfernen wollen, können Sie diese direkt markieren und über das Kontextmenü per Delete entfernen.

Kommandozeile

Wollen Sie eine JAR-Datei nicht durch eine IDE einbinden, sondern beim Aufruf eines Programms von der Konsole aus einbinden, verwenden Sie die Option `-cp` beim Aufruf des Java-Laufzeitsystems. Gehen wir der Einfachheit halber davon aus, dass Sie ein Programm namens *Bonbonbude.java* geschrieben haben und dass die Datei *derby.jar*, die Sie einbinden wollen, im gleichen Verzeichnis wie das übersetzte Programm *Bonbonbude.class* liegt. Dann starten Sie das Programm folgendermaßen:

```
java -cp .;derby.jar Bonbonbude
```



Das Semikolon gilt nur für Windows. Unter Linux und Mac muss hier ein Doppelpunkt stehen.

Kapitel 12

Bermuda-Klassen

IN DIESEM KAPITEL

Identifikation der Klassen

Nicht weniger Code, aber besser verteilbar

Mehr Übersicht für spätere Erweiterung

Nachdem Sie nun so viel über Klassen gelernt haben, brennen Sie sicher darauf, diese Kenntnisse anzuwenden. Dazu kommen wir noch einmal auf das Spiel Bermuda zurück, das wir in Kapitel 8 erstellt haben. Tatsächlich wird dieses Kapitel das Programm um kein Feature erweitern, aber Sie werden lernen, wie man Klassen einsetzt, um ein Projekt übersichtlicher zu gestalten.

Das Zerschlagen des Programms in viele Klassen führt keineswegs dazu, dass weniger Quelltext entsteht. Der Hauptvorteil liegt darin, dass die Aufgabe in abgeschlossene Teilaufgaben zerlegt wird und so auf mehrere Programmierer verteilt werden kann.

Das Projekt erstellen

Sie starten Eclipse und legen ein neues Projekt an.

1. Rufen Sie über das Menü File|New|Java Project auf.
2. Es erscheint ein Dialog, das den Projektnamen einfordert. Wählen Sie beispielsweise `BermudaKlasse`, um es von dem bisherigen Bermuda-Projekt zu unterscheiden.

Klassen anlegen

Für das Projekt werden Sie mehrere Klassen benötigen. Eine Klasse legen Sie immer nach dem folgenden Verfahren an:

- ✔ Klicken Sie das Projekt mit der rechten Maustaste an und wählen Sie aus dem Kontextmenü New|Class.
- ✔ Geben Sie den Klassennamen ein.
- ✔ Wenn eine Klasse eine Methode `main` haben soll, können Sie sich den Rahmen

von Eclipse vorgeben lassen, indem Sie einen entsprechenden Haken setzen.

- ✔ Mit Finish schließen Sie die Erstellung ab.

Starten des Programms

Wenn Sie das Spiel in Eclipse starten wollen, gehen Sie erst folgendermaßen vor:

1. Klicken Sie das Projekt mit der rechten Maustaste an.
2. Wählen Sie aus dem Kontextmenü den Punkt Run AS|Java Application.

Das Programm startet. Ausgaben erscheinen im unteren Teilfenster mit dem Reiter Console. Bevor Sie Eingaben in die Konsole eintippen, klicken Sie einmal in dieses Teilfenster, damit die Eingaben nicht etwa in Ihrem Quelltext erscheinen.

Wenn das Programm einmal gestartet wurde, kann es auch über den großen, grünen Kreis mit dem weißen Dreieck gestartet werden oder über die Tastenkombination Strg + F11.

Lauffähige JAR-Datei erzeugen

Damit Sie das fertige Spiel im Freundeskreis verteilen können, werden Sie eine ausführbare JAR-Datei generieren wollen. Dazu gehen Sie in folgenden Schritten vor:

1. Klicken Sie das Projekt mit der rechten Maustaste an und wählen Sie den Punkt Export.
2. Es erscheint ein Dialog. Hier wählen Sie unter dem Punkt Java den Punkt Runnable JAR File. Klicken Sie auf Next.
3. Unter Launch Configuration klappt eine Liste auf, in der alle Klassen erscheinen, die im aktuellen Workspace eine statische Methode `main` enthalten. Suchen Sie diejenige aus, die der Startpunkt Ihrer JAR-Datei sein soll.
4. Unter Export Destination tragen Sie den Pfad und den Namen der JAR-Datei ein. Etwas einfacher wird es, wenn Sie über den Button Browse einen Ort suchen.
5. Mit dem Button Finish wird die Datei erzeugt.

Da wir noch keine grafischen Oberflächen verwenden, können Sie das Programm nur von der Konsole aus starten. Wenn Ihre JAR-Datei *Bermuda.jar* heißt, lautet der Befehl zum Start:

```
java -jar Bermuda.jar
```

Die Klassen

Die erste Aufgabe besteht darin, die Aufgabenstellung zu durchleuchten und zu schauen, welche Objekte identifizierbar sind. Für diese wird dann je eine Klasse erstellt. Schauen Sie noch einmal in Kapitel 8 und suchen Sie selbst, bevor Sie sich die folgende Liste ansehen.

- `Position`: Für die Eingabe wurde extra ein Array gezimmert, damit die Methode für die Benutzereingabe die Position zurückgeben kann. Eine Klasse kann nun die beiden Attribute `x` und `y` nennen.
- `Schiff`: Das Schiff wurde in drei getrennten Arrays emuliert. Stattdessen bilden wir lieber eine Klasse, die dann eine Position und das Flag `gefunden` enthält.
- `Flotte`: Die Schiffe sind nicht unabhängig. Das fängt schon damit an, dass nicht zwei Schiffe die gleiche Position haben dürfen. Darum bilden wir eine Flotte, die im Kern das Array der Schiffe enthält.
- `Spielfeld`: Im Spielfeld werden die bisherigen Peilungen des Spielers gespeichert. Das entspricht damit dem aktuellen Spielstand.

Das Spielfeld benötigt für seine Arbeit den Zugriff auf die Flotte. Dies kann auf folgende Arten gelöst werden:

- Das Spielfeld legt ein Attribut der Klasse `Flotte` an. Bei Anfragen an die Flotte ruft es deren Methoden auf.
- Die Flotte wird als separates Objekt beispielsweise als Singleton gepflegt.
- Das Spielfeld erweitert die Klasse `Flotte`. Damit erbt es auch deren Methodenaufrufe. Diese Vorgehensweise hat ihren Charme darin, dass alle Anfragen über das Spielfeld laufen können.

Ich habe mich dazu entschlossen, die Klasse `Spielfeld` als Erweiterung der Klasse `Flotte` zu realisieren. Insbesondere, wenn später einmal eine grafische Oberfläche eingesetzt wird, ist es angenehmer, die Aufrufe auf eine Klasse zusammenzuführen.

Ein einfaches UML-Diagramm

Gern wird für die Darstellung von Klassenabhängigkeiten ein UML-Diagramm bemüht. UML-Diagramme zeigen die Zusammenhänge zwischen Klassen an.

Für jede Klasse wird ein Kasten erstellt, der folgende Informationen enthält:

- In der ersten Zeile steht der Name der Klasse.
- Darunter sind die Attribute aufgelistet, eventuell mit Datentyp.
- Darunter befinden sich die Methoden.

Die Verbindungen zwischen Klassen können dargestellt werden.

- Ein senkrechter, durchgängiger Pfeil weist von der erweiternden Klasse zu ihrer Basisklasse.
- Ein senkrechter, gestrichelter Pfeil weist von der Implementierung zu dem zugehörigen Interface.
- Ein waagerechter Pfeil, der mit einer offenen Raute beginnt, zeigt auf die Klasse, deren Objekte die Klasse enthält. Über dem Ende des Pfeils steht der Name des Attributs, darunter, wie oft das Objekt enthalten ist.

Die Sichtbarkeit der Klassenelemente kann bei Bedarf durch ein Zeichen angedeutet werden.

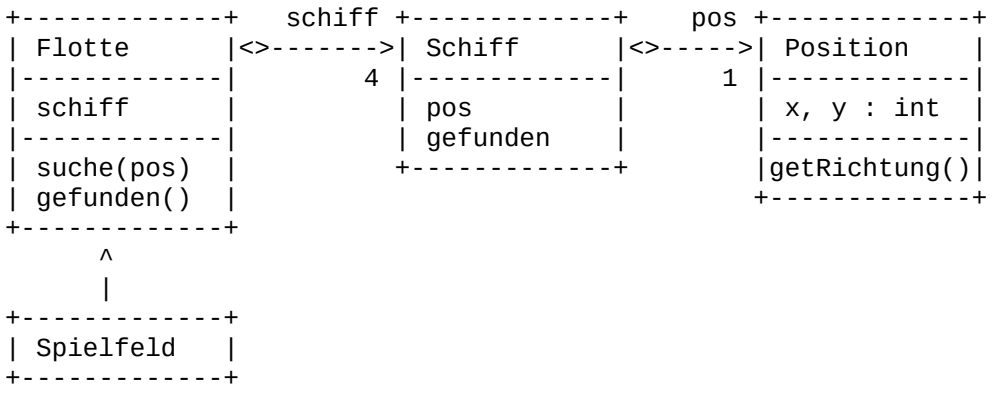
+ public

protected

- private

~ package

Für unser Beispiel ist die Sichtbarkeit nicht so spannend, da alle Attribute privat und alle Methoden öffentlich sind.



Auswechselbar: Der Ablauf

Damit sind die Objekte identifiziert. Nun betrachten wir den Ablauf des Spiels. In der Konsolenumgebung, die Sie bisher kennen, läuft das Spiel so ab:

- Das Spiel wird initialisiert.
- Das aktuelle Spielfeld wird angezeigt.
- Das Spiel blockiert, bis der Spieler eine Koordinate eingibt.
- Das Spiel prüft die Position, gibt das Ergebnis aus und schleift wieder zur Anzeige des Spielfelds.
- Nach einer Abschlussmeldung endet das Spiel.

Das Spiel Bermuda soll später aber mit einer grafischen Oberfläche ausgestattet werden. Dort wird beispielsweise das Spielfeld nicht laufend neu gezeichnet.

Stattdessen wird ein Fenster gezeichnet. Die Eingabe blockiert nicht, sondern tritt durch das Ereignis eines Mausklicks ein. Kurz: Es wird sich erheblich von der Konsolenversion unterscheiden.

Es kommen noch zwei Klassen hinzu, die allerdings bei einer Version für grafische Oberflächen komplett ersetzt werden müssen.

- ✔ `Darstellung`: Diese Klasse soll sich um die Darstellung des Feldes kümmern.
- ✔ `BermundaMain`: Hier steckt die Methode `main` und damit die Ablauflogik des Spiels.

Die Position

Beginnen wir mit einer einfachen Fingerübung, der Position. Eine Position enthält die Attribute `x` und `y`. Die Position wird nur durch ihren Konstruktor gesetzt und ansonsten durch die Methoden `getX` und `getY` ausgelesen. Als Hilfsmethoden werden wir später zwei Positionen auf Gleichheit überprüfen wollen. Dazu schreiben wir die Methode `equals`.

```
public class Position {

    private int x;
    private int y;

    public Position(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }

    @Override
    public boolean equals(Object obj) {
        Position pos = (Position) obj;
        return this.x == pos.x && this.y == pos.y;
    }
}
```

Listing 12.1: Die Klasse `Position`

Richtungsvergleich

Bei der Suche nach Schiffen in der Klasse `Flotte` wird ein Vergleich benötigt, der die Richtung einer Position zu einer anderen bestimmt. Da die reine

Richtungsbestimmung bereits von der Position erledigt werden kann, wird die Methode `getRichtung` hier implementiert. Die Richtung zwischen zwei Positionen lässt sich anhand der Koordinaten folgendermaßen bestimmen:

- Ist die gesuchte X-Koordinate gleich der eines Schiffes, muss das Schiff nördlich oder südlich liegen, je nachdem ob die Y-Koordinate größer oder kleiner ist.
- Ist die gesuchte Y-Koordinate gleich der eines Schiffes, muss das Schiff östlich oder westlich liegen, je nachdem, ob die X-Koordinate größer oder kleiner ist.
- Die Suchposition und ein Schiff liegen auf derselben Diagonalen, wenn die Differenz der beiden X-Koordinaten und der beiden Y-Koordinaten gleich ist. Das hört sich schräg an? Richtig! Wenn Sie diagonal laufen, bewegen Sie sich nämlich immer gleich viele Schritte in X- und in Y-Richtung. Das schlägt sich in einer gleichen Differenz nieder:

  Bei (7,9) und (3,5) ergibt die Differenz der Koordinaten 7-3=4 und 9-5=4.
- Auf der Gegendiagonalen ist die Differenz der Koordinaten ebenfalls gleich, allerdings negativ.

  Bei (7,5) und (3,9) ergibt die Differenz der Koordinaten 7-3=4 und 5-9=-4.

Der Quellcode ist eine Fallunterscheidung, die mit `if-else`-Staffeln realisiert wird:

```
public int getRichtung(Position vgl) {
    int richtung = -1;
    if (x == vgl.x && y < vgl.y) {
        richtung = NORD;
    } else if (x == vgl.x && y > vgl.y) {
        richtung = SUED;
    } else if (x < vgl.x && y == vgl.y) {
        richtung = OST;
    } else if (x > vgl.x && y == vgl.y) {
        richtung = WEST;
    }
    // Für die Diagonalen werden die Differenzen
    // berechnet
    int diffx = x - vgl.x;
    int diffy = y - vgl.y;
    if (diffx == diffy && x < vgl.x) {
        richtung = NORDOST;
    } else if (diffx == diffy && x > vgl.x) {
        richtung = SUEDWEST;
    } else if (diffx == -diffy && x < vgl.x) {
        richtung = SUEDOST;
    } else if (diffx == -diffy && x > vgl.x) {
    richtung = NORDWEST;
    }
    return richtung;
}
```

Listing 12.2: Bestimme die Richtung zu einer anderen Position.

Die Konstanten für die Richtungen werden ebenfalls hier angelegt. Da sie später als Indizes für ein Array verwendet werden, beginnen sie mit 0 und laufen ohne Lücke bis zur 7.

Vollständiges Listing

Nun fassen wir alles zusammen, damit Sie nicht Puzzle spielen müssen:

```java
public class Position {
    final int NORD = 0;
    final int SUED = 1;
    final int OST = 2;
    final int WEST = 3;
    final int NORDOST = 4;
    final int NORDWEST = 5;
    final int SUEDOST = 6;
    final int SUEDWEST = 7;

    private int x;
    private int y;

    public Position(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public int getRichtung(Position vgl) {
        int richtung = -1;
        if (x == vgl.x && y < vgl.y) {
            richtung = NORD;
        } else if (x == vgl.x && y > vgl.y) {
            richtung = SUED;
        } else if (x < vgl.x && y == vgl.y) {
            richtung = OST;
        } else if (x > vgl.x && y == vgl.y) {
            richtung = WEST;
        }
        // Für die Diagonalen werden die Differenzen
        // berechnet
        int diffx = x - vgl.x;
        int diffy = y - vgl.y;
        if (diffx == diffy && x < vgl.x) {
            richtung = NORDOST;
        } else if (diffx == diffy && x > vgl.x) {
            richtung = SUEDWEST;
        } else if (diffx == -diffy && x < vgl.x) {
            richtung = SUEDOST;
```

```
        } else if (diffx == -diffy && x > vgl.x) {
            richtung = NORDWEST;
        }
        return richtung;
    }

    @Override
    public boolean equals(Object obj) {
        Position pos = (Position) obj;
        return this.x == pos.x && this.y == pos.y;
    }
}
```

Listing 12.3: Das komplette Listing der Klasse `Position`

Schiff

Die Klasse `Schiff` modelliert ein einzelnes Schiff. Dieses hat eine Position und speichert in der booleschen Variablen `gefunden` ab, ob es bereits entdeckt wurde. Hier passiert also nicht besonders viel.

```
public class Schiff {
    private Position pos;
    private boolean gefunden = false;

    public void setPos(Position pos) {
        this.pos = pos;
    }

    public Position getPos() {
        return pos;
    }

    public boolean isGefunden() {
        return gefunden;
    }

    public void setGefunden(boolean gefunden) {
        this.gefunden = gefunden;
    }
}
```

Listing 12.4: Eine Klasse für das Schiff

Flotte

Ein Schiff allein macht bei Bermuda keinen Spaß. Es müssen schon mehrere davon da sein, also eine Flotte. Die Flotte besteht aus den vier Schiffen, die wir in einem Array ablegen. Im Konstruktor können wir die Schiffe erstellen.

Die erste herausragende Aufgabe besteht darin, dafür zu sorgen, dass es keine doppelten Positionen gibt. Dazu werden zwei Schleifen ineinandergestellt. Die äußere Schleife erzeugt nacheinander die Positionen. Das wird mithilfe der Bibliothek `Random` ausgewürfelt. Die innere Schleife durchläuft alle bisher gesetzten Schiffe und prüft, ob die neue Position schon besetzt ist. Erst wenn die neue Position wirklich neu ist, geht es mit der nächsten Position weiter.

Die Prüfung, ob bereits alle Schiffe entdeckt wurden, gehört natürlich ebenfalls in diese Klasse.

```
import java.util.Random;

class Flotte {
    final static char SCHIFF = 'X';
    final static int MAXSCHIFF = 4;

    private Schiff schiff[] = new Schiff[MAXSCHIFF];

    public Flotte() {
        // Schiffe vorbereiten
        schiff = new Schiff[MAXSCHIFF];
        // Schiffe verteilen
        Random zufall = new Random();
        for (int i = 0; i < MAXSCHIFF; i++) {
            schiff[i] = new Schiff();
            boolean kollidieren = false;
            do {
                Position pos = new Position(
                    Math.abs(zufall.nextInt()) % Spielfeld.MAXX,
                    Math.abs(zufall.nextInt()) % Spielfeld.MAXY);
                schiff[i].setPos(pos);
                kollidieren = false;
                for (int j = 0; j < i; j++) {
                    if (schiff[j].getPos().equals(pos)) {
                        kollidieren = true;
                    }
                }
            } while (kollidieren);
        }
    }
```

```
    public boolean alleSchiffeGefunden() {
        boolean gefunden = true;
        for (int i = 0; i < MAXSCHIFF; i++) {
            if (schiff[i].isGefunden() == false) {
                gefunden = false;
            }
        }
        return gefunden;
    }
}
```

Listing 12.5: Die Klasse `Flotte`

Die Klasse `Flotte` benötigt die Dimensionen des Spielfelds. Es wäre ja blöd, wenn ein Schiff außerhalb des Bermudadreiecks verschwindet.

Eine neue Schiffspeilung

In Listing 8.7 wurde die Peilung durchgeführt, indem vom Suchpunkt aus alle Richtungen des Spielfelds durchschritten wurden, bis entweder ein Schiff oder der Spielfeldrand auftauchte.

Ist es aber nicht möglich, allein anhand der Koordinaten festzustellen, welche Schiffe von der Suchposition aus zu sehen sind? Dann könnte die Klasse `Flotte` die Suche nach den Schiffen erledigen.

Dazu vergleicht das Programm die vom Spieler ausgewählte Koordinate mit denen aller Schiffe und bestimmt daraus die Anzahl der Richtungen.

1. Wir erstellen ein Array mit acht `int`-Elementen, für jede Richtung eines.
2. Die Spielerposition wird in einer Schleife mit jedem Schiff verglichen. Die Methode `getRichtung` ermittelt die Richtung als Wert zwischen 0 und 7.
3. Ist die Richtung gültig, wird eine 1 im Richtungsfeld eingetragen. Es wird nicht aufsummiert. So kann für jede Richtung nur 0 oder 1 gespeichert werden. Drei Schiffe in Richtung Nord ergeben so eine 1 im Array-Element für die Richtung Nord.
4. Die Anzahl der Richtungen wird ermittelt, indem die Werte des Richtungs-Arrays aufsummiert werden.

```
public char suche(Position suchPos) {
    final int RICHTUNGEN = 8;
    int richtung[] = new int[RICHTUNGEN];
    for (int i = 0; i < MAXSCHIFF; i++) {
        Position pos = schiff[i].getPos();
        if (pos.equals(suchPos)) {
            schiff[i].setGefunden(true);
            return SCHIFF;
        } else {
            int richt = suchPos.getRichtung(pos);
            if (richt>=0) {
                richtung[richt] = 1;
            }
        }
    }
    int anzahl = 0;
    for (int i=0; i<RICHTUNGEN; i++) {
        anzahl += richtung[i];
    }
    char ergebnis = (char) ('0' + anzahl);
    return ergebnis;
}
```

Listing 12.6: Richtungszählung in der Klasse `Flotte`

Vollständiges Listing

```
import java.util.Random;
class Flotte {
    final static char SCHIFF = 'X';
    final static int MAXSCHIFF = 4;
    private Schiff schiff[] = new Schiff[MAXSCHIFF];

    public Flotte() {
        // Schiffe vorbereiten
        schiff = new Schiff[MAXSCHIFF];
        // Schiffe verteilen
        Random zufall = new Random();
        for (int i = 0; i < MAXSCHIFF; i++) {
            schiff[i] = new Schiff();
            boolean kollidieren = false;
            do {
                Position pos = new Position(
                        Math.abs(zufall.nextInt()) % Spielfeld.MAXX,
                        Math.abs(zufall.nextInt()) % Spielfeld.MAXY);
                schiff[i].setPos(pos);
                kollidieren = false;
                for (int j = 0; j < i; j++) {
                    if (schiff[j].getPos().equals(pos)) {
                        kollidieren = true;
                    }
                }
            } while (kollidieren);
        }
    }
    public char suche(Position suchPos) {
        final int RICHTUNGEN = 8;
        int richtung[] = new int[RICHTUNGEN];
        for (int i = 0; i < MAXSCHIFF; i++) {
            Position pos = schiff[i].getPos();
            if (pos.equals(suchPos)) {
                schiff[i].setGefunden(true);
                return SCHIFF;
            } else {
                int richt = suchPos.getRichtung(pos);
                if (richt>=0) {
                    richtung[richt] = 1;
                }
            }
        }
        int anzahl = 0;
        for (int i=0; i<RICHTUNGEN; i++) {
            anzahl += richtung[i];
        }
```

```
        char ergebnis = (char) ('0' + anzahl);
        return ergebnis;
    }
    public boolean alleSchiffeGefunden() {
        boolean gefunden = true;
        for (int i = 0; i < MAXSCHIFF; i++) {
            if (schiff[i].isGefunden() == false) {
                gefunden = false;
            }
        }
        return gefunden;
    }
    public Position getSchiffPos(int i) {
        return schiff[i].getPos();
    }
}
```

Listing 12.7: Das komplette Listing der Klasse `Flotte`

Die Methode `getSchiffPos` wird benötigt, um alle Schiffe anzuzeigen, wenn der Spieler aufgibt.

Das Spielfeld

Das Spielfeld bestimmt vor allem die Dimension in Konstanten. Es verwaltet auch das zweidimensionale Array, das die bisherigen Rateversuche des Spielers enthält. Auf Anfrage liefert oder setzt es das Zeichen, das sich darin befindet.

Das Spielfeld sollte feststellen können, ob eine Position gültig, also noch innerhalb des Spielfelds ist.

Da es die Klasse `Flotte` erweitert, können die Suchanfragen auch an das `Spielfeld` gerichtet werden.

```
public class Spielfeld extends Flotte {
    final static int MAXX = 9;
    final static int MAXY = 7;

    private char[][] spielfeld = new char[MAXX][MAXY];

    public Spielfeld() {
        for (int x = 0; x < MAXX; x++) {
            for (int y = 0; y < MAXY; y++) {
                spielfeld[x][y] = '+';
            }
        }
    }

    void enttarneSchiffe() {
        for (int i=0; i<MAXSCHIFF; i++) {
            Position pos = getSchiffPos(i);
            set(pos, SCHIFF);
        }
    }

    public char get(Position pos) {
        return spielfeld[pos.getX()][pos.getY()];
    }

    public void set(Position pos, char zeichen) {
        spielfeld[pos.getX()][pos.getY()] = zeichen;
    }

    public boolean istImSpielfeld(Position pos) {
        int x = pos.getX();
        int y = pos.getY();
        return x>=0 && y>=0 && x<Spielfeld.MAXX && y< Spielfeld.MAXY;
    }
}
```

Listing 12.8: Die Klasse `Spielfeld`

Das Hauptprogramm

Die Methode `main` steht in der Klasse `BermudaMain`. Darin befindet sich eine Schleife, die so lange kreist, bis der Spieler abbricht oder gewinnt.

In der Schleife wird das Spielfeld angezeigt, die Koordinate eingegeben und anschließend nach den Schiffen gesucht.

```
public class BermudaMain {
    static public void main(String[] args) {
        Spielfeld spiel = new Spielfeld();
        Konsole view = new Konsole();

        boolean spielAbbruch = false;
        while (!spielAbbruch && !spiel.alleSchiffeGefunden()) {
            view.zeige(spiel);
            Position pos = view.eingeben();
            if (spiel.istImSpielfeld(pos)) {
                char zeichen = spiel.suche(pos);
                spiel.set(pos, zeichen);
            } else {
                spielAbbruch = true;
            }
        }
        if (spielAbbruch) {
            System.out.println("Hier waren die Schiffe");
            spiel.enttarneSchiffe();
            view.zeige(spiel);
        } else {
            System.out.println("Gewonnen");
        }
    }
}
```

Listing 12.9: Hauptprogramm

Die Klasse Konsole

In der Klasse `Konsole` werden die Besonderheiten für die Ein- und Ausgabe einer textbasierten Anwendung isoliert. Entsprechend finden Sie dort die Methode `eingeben`, die eine Position zurückliefert, und die Methode `zeige`, die das Spielfeld auf dem Bildschirm ausgibt.

```
import java.util.Scanner;

public class Konsole {
    public Position eingeben() {
        int x = -1, y = -1;
        System.out.println("\nPosition eingeben ");
        Scanner sc = new Scanner(System.in);
        String eingabe = sc.next();
        if (eingabe.length() > 1) {
            char cx = eingabe.charAt(0);
```

```
            if (cx >= 'A' && cx <= 'A' + Spielfeld.MAXX) {
                x = (int) (cx - 'A');
            }
            char cy = eingabe.charAt(1);
            if (cy >= '1' && cy <= '1' + Spielfeld.MAXY) {
                y = (int) (cy - '1');
            }
        }
        return new Position(x, y);
    }

    public void zeige(Spielfeld spiel) {
        System.out.println();
        System.out.print("  ");
        for (int x = 0; x < Spielfeld.MAXX; x++) {
            System.out.print(" " + (char) ('A' + x));
        }
        for (int y = 0; y < Spielfeld.MAXY; y++) {
            System.out.println();
            System.out.print("" + (y + 1) + " ");
            for (int x = 0; x < Spielfeld.MAXX; x++) {
                Position pos = new Position(x, y);
                System.out.print(" " + spiel.get(pos));
            }
        }
    }
}
```

Listing 12.10: Die Klasse `Konsole` kümmert sich um Ein- und Ausgabe.

Teil III

Erweitern Sie den Horizont



Kapitel 13

Hoffentlich Exception-versichert

IN DIESEM KAPITEL

- Ausnahmen statt Abfragen
- Ausnahmen fangen und werfen
- Die Stammbäume der Ausnahmen
- Die Katze und der Ausnahmezustand

Unter Java führen Fehler zu Ausnahmen, auch wenn manche Programmierer wohl nicht nur in Ausnahmesituationen Fehler machen. Diese Ausnahmen heißen *Exceptions* und werden als Klassen realisiert, die sich vererben lassen. Man kann sie fangen und werfen, leider nicht wegwerfen, aber immerhin weiterwerfen. Sie verstehen nur Bahnhof? Kein Problem, lesen Sie einfach weiter.

Ausnahmsweise Fehler

Stellen Sie sich schon mal darauf ein: Als Programmierer sind Sie immer schuld und man kann es beweisen. Sie können weder den Computer noch den Compiler verantwortlich machen. Ihre einzige Chance ist es, den Anwender bei seinem schlechten Gewissen zu packen und ihm klarzumachen, dass er schon allein deshalb verantwortlich ist, weil er als Unbedarfter einen Computer benutzt und dabei eben alles kaputt macht. Aber dieses Spiel spielen alle anderen Programmierer und Hotlines auch und je länger es läuft, desto mehr Anwender kommen dahinter, dass sie verladen werden. Und dann werden wieder Sie verantwortlich gemacht. Es wird also Zeit, dass wir über Fehlerbehandlungen reden.

Java verwendet für die Fehlerbehandlung wie erwähnt am liebsten Exceptions, also »Ausnahmen«. Mit der Ausnahme ist eine Ausnahmesituation gemeint. Der normale Ablauf eines Programms wurde also gestört. Vielleicht hat der Nachwuchs seinen Brei in den Schlitz des CD-Laufwerks geschmiert oder der entlaufene Hamster hat das Netzwerkkabel durchgebissen. Blöd, wenn das Programm gerade Dateien vom CD-Laufwerk lesen will oder aus dem Internet Informationen benötigt.

Damit es trotz so einer Ausnahmesituation nicht zum Programmabsturz kommt, gibt es zwei grundlegende Konzepte, wie Sie in Programmen mit Fehlersituationen

umgehen. Sie prüfen entweder jede Operation auf eventuelle Fehlermeldungen, die sich in den Rückgabewerten der Methode äußern, oder Sie sichern einen Ablauf gegen Unterbrechungen und fangen Probleme erst auf, wenn sie eingetreten sind.

Ein Beispiel aus dem wirklichen Leben: Stellen Sie sich vor, Sie sollen verreisen und wollen mit dem Auto zum Flughafen. Dabei können viele Dinge schiefgehen.

- Der Zündschlüssel bricht ab.
- Der Tank wurde von einem durstigen Marder ausgetrunken.
- Der Nachbarsjunge hat die Reifen am Boden festgeklebt.
- An der nächsten Ampel bricht Ihre Hinterachse. Als Ursache kommen Durchrostung oder Übergewicht infrage.

Sie können all diese Fälle abfragen. Aber es ist eigentlich egal, was schiefgelaufen ist. In all diesen Fällen werden Sie gleich reagieren: Sie zücken Ihr Handy und rufen ein Taxi.

Die Probleme, die Sie von Ihrem Flugzeug fernhalten, könnten Sie einfach so behandeln: Versuchen Sie, zum Flughafen zu fahren. Wenn etwas schiefgeht, schnappen Sie sich Ihr Handy und rufen ein Taxi. »Versuchen« heißt »try« und »schnappen« heißt »catch«.

```
try {
    Fahre mit dem Auto zum Flughafen
} catch (Ausnahme) {
    Rufe ein Taxi!
    Notiere die Ausnahmeursache für die Twitter-Follower
}
```

Listing 13.1: Exception-Handling für das Autofahren



Der Block von `try` umfasst alle Anweisungen, die eine Exception verursachen können. Der Block von `catch` wird direkt angesprungen, wenn eine Exception auftritt.

Stellt eine Methode einen Fehler fest, erzeugt sie ein Exception-Objekt und wirft dieses an den Aufrufer. Dieser kann es fangen, an seinen eigenen Aufrufer weiterwerfen oder ignorieren. Wenn sich niemand um die Ausnahme kümmern will, dann gelangt sie zur virtuellen Maschine, die dann die Notbremse zieht und etwas kryptisch meldet, welche Ausnahme von wem geworfen wurde. Das Programm endet.

Eingabe einer Zahl per Dialogbox

Verbinden wir das Angenehme mit dem Nützlichen und bauen uns eine schicke Dialogbox für Benutzereingaben. Grafische Eingabeelemente liefern einen String zurück. Wenn Ihr Programm aber eine Fließkommavariable erwartet, müssen Sie den String in eine `double`-Variable umwandeln.

```
import javax.swing.*;
public class DiaInput {
    public static void main(String[] args) {
        String eingabe="";
        double zahl = 0.0;
        eingabe = JOptionPane.showInputDialog("Bitte eine Zahl!");
        zahl = Double.parseDouble(eingabe);
        System.out.println((zahl*zahl));
    }
}
```

Listing 13.2: Eine schicke Dialogbox für die Fehlererzeugung



Abbildung 13.1: Eingabedialog

Fehleranfällige Zahlenkonvertierung

Wie das mit den grafischen Benutzeroberflächen genau funktioniert, zeige ich Ihnen ab Kapitel 16. Die Eingabebox erscheint durch den Aufruf von `JOptionPane.showInputDialog`. Diese Methode liefert den String zurück, den der Nutzer eingegeben hat. Das Programm benötigt allerdings eine Zahl, also müssen Sie den String von der Methode `Double.parseDouble` in eine Fließkommavariable umwandeln lassen. Zu guter Letzt wird das Quadrat der Zahl auf dem Bildschirm ausgegeben.

Um eine hinreißende Exception auszulösen, reicht es völlig, eine Zahl wie 12,5 einzutippen. Dabei liegt die Betonung auf dem in Deutschland üblichen Komma statt eines Punktes als Dezimaltrenner. Schon erscheint auf dem Bildschirm folgende Klage:

```
Exception in thread “main”
java.lang.NumberFormatException:
 For input string: “12,5”
  at ...
  at DiaInput.main(DiaInput.java:6)
```

Es hat eine Exception gegeben, also eine Ausnahme. Die Fehlermeldung besagt, dass sie im Thread `main` auftritt, also in der Hauptmethode. Der Ausnahmetyp ist

eine `NumberFormatException`. Das soll heißen, dass die Zahl das falsche Format hat. Dort steht auch, was Sie eingegeben haben: 12,5.

Es folgen die Methoden, die Sie aufgerufen haben, und letztlich auch die Zeile, in der der selbst verfasste Auslöser steht. Die Zeile 6. Hier steht im Quellcode:

```
zahl = Double.parseDouble(eingabe);
```

Das ist also die Zeile, wo es knallt. Und die Ursache ist, dass dem Programm die Eingabe von 12,5 nicht gefällt. Hätten Sie nicht auf mich gehört und 12.5 statt 12,5 eingegeben, wäre nichts passiert und Langeweile hätte sich breitgemacht. Das Komma muss ja schließlich ein Punkt sein. Damit kennen wir also die Ursache der Exception.

 Eine Exception meldet, warum und wo sie ausgelöst wurde.

Da die Benutzer im deutschsprachigen Raum die Eingabe von Dezimalzahlen nun mal mit Komma statt mit Punkt gewöhnt sind und dieser Fehler daher häufiger auftreten könnte, wäre es ziemlich unpraktisch, wenn ein Programm dabei jedes Mal sterben würde. Also werden wir etwas dagegen tun. Wir setzen einen `try`-Block um die Zeile 6. Knallt es, dann fängt das Programm den Fehler und meldet, dass ihm die Zahleneingabe nicht gefallen hat. Als Parameter von `catch` geben Sie die Art der Ausnahme an, die Sie fangen wollen, in diesem Fall also eine Referenz auf die Klasse `NumberFormatException`. Noch besser wäre es natürlich, wenn der Benutzer die Möglichkeit bekäme, die Zahl neu einzugeben. Also machen wir eine kleine Schleife um die Eingabe.

```
import javax.swing.*;
public class DiaInput {
    public static void main(String[] args) {
        String eingabe="";
        double zahl = 0.0;
        boolean eingabeProblem = false;
        do {
            eingabeProblem = false;
            eingabe = JOptionPane.showInputDialog
                    ("Bitte eine Zahl!");
            try {
                zahl = Double.parseDouble(eingabe);
            } catch(NumberFormatException e) {
                JOptionPane.showMessageDialog(null,
                        "War wohl keine tolle Zahl!");
                eingabeProblem = true;
            }
        } while (eingabeProblem);
        System.out.println((zahl*zahl));
    }
}
```

Listing 13.3: Der Fehler wird gefangen.



Wie Sie vielleicht gesehen haben, habe ich im Programm noch die Methode `showMessageDialog` untergebracht. Damit können Sie Strings als Dialog anzeigen lassen. Das sieht natürlich eleganter aus als `System.out.print`, allerdings kann die Dialogbox nur Strings anzeigen.

Testen Sie das Programm mal mit verschiedenen Eingaben und schauen Sie, ob es nun besser geworden ist.

Falls Sie mit Eclipse arbeiten, können Sie ja einmal den Mauszeiger über den Methodenaufruf `showInputDialog` halten. Es erscheint nach kurzer Zeit ein Fenster mit Informationen über diese Methode. Dort werden Sie feststellen, dass auch `showInputDialog` eine Exception werfen kann.

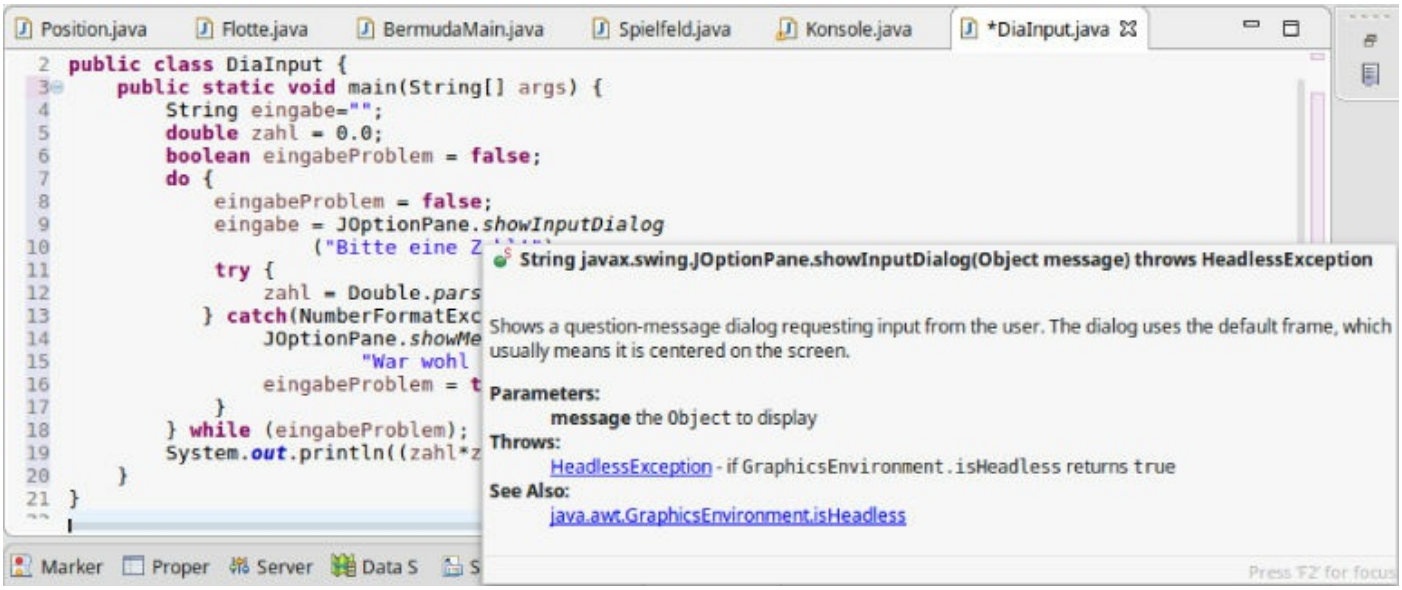


Abbildung 13.2: Anzeige der Methodeninformation

Mehrere Exceptions fangen

Dort steht: `throws HeadlessException`. Die Methode schmeißt also kopflos mit Exceptions um sich. Vielleicht sollten wir doch noch einmal genauer hinschauen und auf den Link der Exception klicken. Dann stellen Sie fest: `HeadlessException` wird ausgelöst, wenn das Programm keine Umgebung findet, um starten zu können, also beispielsweise, wenn das Programm auf einem Rechner ohne grafische Oberfläche gestartet wird. Wenn wir auch diese Exception fangen wollen, müsste in unserem Code der Import der `HeadlessException` und das Fangen dieser Exception ergänzt werden.

```
import java.awt.HeadlessException;
...
try {
   eingabe = JOptionPane.showInputDialog("Bitte eine Zahl!");
} catch(HeadlessException e) {
   ...
}
```

Listing 13.4: Hurra! Noch eine Exception.

Damit haben wir bereits zwei Exception-Lieferanten im Programm. Sie können mit `try` Jagd auf mehrere Exceptions machen. Dazu bauen Sie einfach zwei `catch`-Zweige.

```
import java.awt.HeadlessException;
...
try {
    eingabe = JOptionPane.showInputDialog("Bitte eine Zahl!");
    zahl = Double.parseDouble(eingabe);
} catch(NumberFormatException e) {
    ...
} catch(HeadlessException e) {
    ...
}
```

Listing 13.5: Jagd auf zwei Exceptions

Alle anderen Ausnahmen

Was passiert eigentlich, wenn der Benutzer in der Dialogbox auf den Button Abbrechen klickt? Oh, das sieht sehr nach einer `NullPointerException` aus. Das scheint unser Programm doch zu irritieren: Es steigt aus und stirbt. Der Aufruf von `showInputDialog` gibt den Wert `null` zurück, wenn man auf Abbrechen klickt. Und darauf reagiert `parseDouble` etwas stinkig.

Während ich hier fröhlich eine Exception nach der anderen hervorzaubere, fürchten Sie wahrscheinlich schon, dass Sie für jede denkbare Exception einen `catch` schreiben müssten. Tatsächlich könnten Sie einen `catch` setzen, der alle Exceptions fängt, indem Sie als Parameter einfach nur `Exception` verwenden.

```
} catch(Exception e) {
```

Allerdings gehört `NullPointerException` zu den Ausnahmefällen, die Sie nicht fangen, sondern vermeiden sollten. In älteren Java-Versionen lässt sie sich gar nicht fangen. Die `NullPointerException` deutet nämlich darauf hin, dass Sie mit einer Referenz auf `null` weitergearbeitet haben. Das Fehlen der Abfrage, ob die Methode `null` zurückgibt, ist aber nichts, was überraschend eintritt, sondern zeigt, dass die Programmierlogik nicht besonders gelungen ist, da der Programmierer vor der Übergabe einer Referenz an `parseDouble` nicht prüft, ob diese auf null steht. Das ist schlampige Arbeit und sollte nicht über eine Exception-Behandlung beschönigt werden.

Der catch-Parameter

Der Parameter von `catch` ist ein Objekt einer Klasse, die `Exception` erweitert. Diese wiederum erweitert die Klasse `Throwable`, und deren Methoden können wir gut für die weitere Fehlerbearbeitung gebrauchen.

getMessage

Zunächst bietet die Exception die Methode `getMessage`, die einen kurzen String

zurückgibt, welche Ausnahme aufgetreten ist. Für die fehlerhafte Kommazahl 12,5 statt 12.5 lautete dieser:

```
For input string: “12,5”
```

Sie könnten `getMessage` im `catch`-Zweig aufrufen:

```
} catch(NumberFormatException e) {
    String fehler = e.getMessage());
    JOptionPane.showMessageDialog(null, fehler);
}
```

Listing 13.6: Auswerten der Exception mit `getMessage`

Damit können Sie den Fehler den Anwendern Ihrer Programme in einer schicken Dialogbox anzeigen. Dafür werden sie Sie lieben. Mag sein, dass es dem englischen Text, den Java als Antwort auf eine Exception liefert, noch etwas an Charme mangelt, aber es ist immer noch besser, als wenn das Programm meldet: »Ich weiß zwar auch nicht, warum, aber irgendwie geht es mir dreckig!«

toString

```
} catch(NumberFormatException e) {
    String fehler = e.toString());
    JOptionPane.showMessageDialog(null, fehler);
}
```

Listing 13.7: Auswerten der Exception mit `toString`

Alternativ können Sie die Methode `toString` verwenden. Sie liefert, den gleichen String, den `getMessage` liefert und fügt noch den Namen der Exception-Klasse hinzu. Da `toString` auch die Standardkonvertierung zum String darstellt, könnten Sie den Exception-Parameter von `catch` direkt per `System.out.print` oder per Dialogbox auf den Bildschirm ausgeben. Für den Kommaeingabefehler lautet der String dann:

```
java.lang.NumberFormatException: For input string: “12,5”
```

printStackTrace

Noch weniger spannend für die Benutzer, aber sicher für den Programmierer ist die Methode `printStackTrace`. Sie zeigt alle Methodenaufrufe an, die zu der Ausnahmesituation geführt haben. Die Exception kann auf diese Weise melden, wo der Absprung zur fehlerhaften Stelle erfolgte.

```
} catch(NumberFormatException e) {
    e.printStackTrace();
}
```

Listing 13.8: Auswerten der Exception mit `printStackTrace`

Die Meldung erscheint auf der Fehlerkonsole und ist exakt die, die bei einem

Programmabsturz erscheint, wenn man die Exception nicht fängt.

```
java.lang.NumberFormatException: For input string: “12,5”
  at sun.misc.FloatingDecimal.readJavaFormatString
                          (FloatingDecimal.java:1242)
  at java.lang.Double.parseDouble(Double.java:527)
  at DiaInput.main(DiaInput.java:12)
```

Zum Abschluss schauen wir uns das Programm noch einmal als Ganzes an:

```
import java.awt.HeadlessException;
import javax.swing.*;
public class DiaInput {
    public static void main(String[] args) {
        String eingabe="";
        double zahl = 0.0;
        boolean eingabeProblem = false;
        do {
            eingabeProblem = false;
            try {
                eingabe = JOptionPane.showInputDialog(
                        "Bitte eine Zahl!");
                if (eingabe!=null) {
                    zahl = Double.parseDouble(eingabe);
                }
            } catch(NumberFormatException e) {
                String fehler = e.toString();
                JOptionPane.showMessageDialog(null, fehler);
                eingabeProblem = true;
            } catch(HeadlessException e) {
                System.err.println(
                        "Das Programm braucht eine GUI.");
                eingabeProblem = true;
            }
        } while (eingabeProblem);
        System.out.println((zahl*zahl));
    }
}
```

Listing 13.9: Exceptions gefangen, `NullPointerException` verhindert

Viel Quelltext für die Fehlerbehandlung, aber fehlertolerante Programme sind einfach wichtig. Falls Sie nun meinen, dass man damit idiotensichere Programme schreiben könnte, werden Sie irgendwann feststellen, dass es solche Programme nicht gibt, weil Idioten so erfinderisch sind.

Ein gelungener Wurf

Wenn eine Methode, die Sie verwenden, eine Exception wirft, müssen Sie diese nicht zwingend selbst fangen und bearbeiten. Sie können sie einfach weiterwerfen. Das ist beispielsweise dann sinnvoll, wenn Sie innerhalb einer Methode zwar die Exception fangen können, aber dort nicht der geeignete Ort zur Behandlung ist.

Weiterwerfen

Um eine Exception weiterzuwerfen, reicht es aus, zu deklarieren, dass Ihre Methode genau diese Exception wirft. Die Befehle `try` und `catch` benötigen Sie nun nicht mehr. Die müssen dann im aufrufenden Programmteil gesetzt werden.

Die Methode `diaDoubleInput` aus Listing 13.10 ruft `parseDouble`, das ja eine `NumberFormatException` auslösen könnte. Da die Methode gleich zu Anfang erklärt, dass sie selbst diese Exception werfen kann, benötigt sie kein `try` und `catch`. Das muss nun `main` tun, wenn es die Methode `diaDoubleInput` aufrufen will.

```
public class DiaInput {
    public static double diaDoubleInput()
            throws NumberFormatException {
        String eingabe="";
        double zahl = 0.0;
        eingabe = JOptionPane.showInputDialog(
                "Bitte eine Zahl!");
        zahl = Double.parseDouble(eingabe);
        return zahl;
    }
    public static void main(String[] args) {
        double zahl = 0.0;
        try {
            zahl = diaDoubleInput();
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(null,
                    "War wohl keine tolle Zahl!");
        }
        System.out.println((zahl*zahl));
    }
}
```

Listing 13.10: Weiterwerfen der Exception

Selber werfen

Wenn andere Leute einfach Exceptions auslösen dürfen, dann dürfen Sie das auch. Gerechtigkeit für alle! Tatsächlich ist das Auslösen von Exceptions nicht schwer. Aber erst einmal brauchen Sie eine eigene Exception-Klasse, die Sie werfen können, und die sollte sich von `Exception` ableiten. Nichts leichter als das: Wir definieren eine Klasse, die nichts tut, außer die Klasse `Exception` zu erweitern.

```
class  TutNixException extends Exception { }
```

Die tut wirklich nix! Darum sind die geschweiften Klammern leer. Für den Anfang ist es nur wichtig, dass sie einen anderen Namen hat als die bereits existierenden Exceptions, und da können wir bei diesem Namen doch recht sicher sein. Diese Exception wollen wir auslösen. Für das Auslösen gibt es den Befehl `throw`. Und wenn eine Methode werfen möchte, dann kündigt sie dies mit dem Befehl `throws` an. Nur so kann der Compiler prüfen, ob die aufrufende Methode auch artig die Exception fängt oder weiterleitet.

```
public void tuDochWas() throws TutNixException {
    throw new TutNixException();
}
```

Listing 13.11: Eine Methode droht mit Exception-Wurf.

Tatsächlich können wir natürlich keine Klasse werfen, sondern nur ein Objekt der Klasse. Darum steht hinter `throw` auch ein `new`.

Die Methode `tuDochWas` hat angemeldet, dass sie die `TutNixException` werfen könnte. Also muss der Aufrufer die Exception fangen.

```
class TuWas {
    public void tuDochWas() throws TutNixException {
        throw new TutNixException();
    }
   public static void main(String[] args) {
       TuWas tu = new TuWas();
       try {
            tu.tuDochWas();
       } catch (TutNixException e) {
            System.out.println(e.getMessage());
       }
   }
}
```

Listing 13.12: Wir fangen die eigene Exception.

Falls Sie das ausprobieren, werden Sie feststellen, dass da nur null ausgegeben wird. Wir müssen noch die Methode getMessage überschreiben. Schade, nun tut TutNixException doch etwas.

```
class  TutNixException extends Exception {
    @Override
    public String getMessage() {
        return "Ich tu doch nichts!";
    }
}
```

Listing 13.13: TutNixException tut was.

Sie könnten in Ihrer Exception-Klasse auch noch ein paar Informationen mehr verstecken oder ein paar Methoden schreiben. Schließlich wird das passende `catch` genau dieses Objekt fangen und kann dann diese Informationen weiterverarbeiten.

... und schließlich

Anschließend an die `catch`-Blöcken kann man auch einen `finally`-Block setzen. Der wird immer ausgeführt, ganz egal, ob ein Fehler gefangen wurde oder nicht.

```
try {
    // viele, viele Anweisungen
    // mit vielen, vielen Fehlermöglichkeiten
} catch(Excetiontyp1) {
    // Behandlung des Exceptiontyp1
} catch(Excetiontyp2) {
    // Behandlung des Exceptiontyp2
} finally {
   // was in jedem Fall passiert
}
```

Listing 13.14: Ein Schlusswort

Auf den ersten Blick scheint `finally` überflüssig, weil der Code auch einfach hinter den letzten `catch`-Block geschrieben werden könnte. Allerdings garantiert Java das Durchlaufen dieses Blocks auch dann, wenn im `try`-Block ein `return`, `break` oder `continue` die Bearbeitung abbricht. Darum setzt beispielsweise der Datenbankprogrammierer gern das Schließen der Datenbankverbindung in einen `finally`-Block. So wird die Datenbank auch dann geschlossen, wenn wirklich alles schiefgegangen ist.

Der Stammbaum der Exceptions

Sie haben gesehen, dass es im Sprachumfang von Java eine ganze Reihe von Standardausnahmen gibt, die alle auf `Exception` basieren, und dass man die Klasse `Exception` durch eigene Exceptions erweitern kann. Lassen Sie uns einmal einen Blick auf den Stammbaum der Familie Exception werfen.

- ✔ `RuntimeException` (typischerweise durch Programmierfehler)
  - `NullPointerException` (Zugriff auf `null`-Referenzen)
  - `IllegalArgumentException` (falscher Parameter)
    - `IllegalFormatException` (falsches Format)
    - `InvalidKeyException` (ungültiger Schlüssel)
    - `PatternSyntaxException` (Syntaxfehler)
  - `IndexOutOfBoundsException` (außerhalb der Array-Grenzen)
  - `NegativeArraySizeException` (negative Array-Größe)
- ✔ `IOException`
  - `FileNotFoundException` (Datei nicht gefunden)
  - `EOFException` (plötzliches Dateiende)
  - `UnknownHostException` (Host nicht ansprechbar)
  - `UnknownServiceException` (Dienst nicht ansprechbar)
- ✔ `AWTException` (durch das AWT ausgelöst)
- ✔ `ClassNotFoundException` (Klassenname unbekannt)
- ✔ `TimeoutException` (Time-out von blockierenden Operationen)

Die `FileNotFoundException` ist also beispielsweise eine Erweiterung von `IOException`.

Schauen Sie noch einmal nach der `RuntimeException`. Diese Exception kennen Sie schon. Sie gehört zum Sprachumfang und ist die Basisklasse aller »unchecked« Exceptions. Man bezeichnet sie als »unchecked«, weil sie nicht explizit importiert werden müssen. Außerdem müssen Methoden, die eine `RuntimeException` auslösen, auch nicht zwingend mit `try` und `catch` umgeben werden oder im Methodenkopf deklariert werden.

Die »unchecked« Exceptions gehen in der Regel auf Programmierfehler zurück, die durch den Anwender nicht korrigierbar sind. Darum sollten Sie sie auch nicht fangen, sondern lieber die Ursache beheben.

Schauen wir uns dagegen die `IOException` an. Sie gehört wie alle anderen Exceptions, die nicht von der `RuntimeException` abgeleitet sind, zu den »checked« Exceptions. Die »checked« Exceptions können meist durch entsprechende Aktionen des Anwenders behoben werden. Darum sollte das Programm diese auch unbedingt fangen und bearbeiten.

Die `IOException` wird ausgelöst, wenn der Umgang mit den Dateien nicht so klappt, wie es das Programm erwartet hätte. Sie werden sie noch etwas ausgiebiger

beim Umgang mit Dateien kennenlernen. Diese Exception und ihre Erweiterungen müssen importiert werden. Die `IOException` finden Sie im Package `java.io`, also bei den Bibliotheken, die sie auslösen oder erzeugen.

Die Katzen-Exception

Bei mir zu Hause herrscht Ausnahmezustand. Die Gemahlin will eine Katze. Offensichtlich findet sie Katzen süßer als mich. Das ist natürlich Pech! Vielleicht sollte ich an mir arbeiten. Aber auch so ist mein Verhältnis zu diesen Tierchen nicht ganz ungebrochen. Da gibt es ein paar schlechte Erfahrungen. Ich konnte immerhin durchdrücken, dass es eine Ausnahmebehandlung gibt, wenn mich die Katze mehrfach kratzt oder ernsthaft schädigt. Dann könne man sie immer noch zurück ins Tierheim bringen. So hieß es.

Kann Java helfen?

Nun sitze ich hier mit meinem einzigen Freund, dem Computer und bemitleide mich. Also werde ich das Drama erst einmal in Java formulieren. Wir haben also eine Katze und solange alles gut ist, freut sich die Gemahlin am liebevollen Schnurren und der Vernichtung der Mäuse. Sobald sie aber Ärger macht, gibt es eine `KatzenException`. Ist der Ärger sehr groß, wandert das Tier ins Tierheim.

```
Katze katze = new Katze();
while (katze.lebtNochBeiEuch()) {
    try {
        katze.schlaeft();
        katze.frisst();
        katze.schnurrt();
    } catch(KatzenException k) {
        katze.mussInsHeim();
    }
}
```

Listing 13.15: Lebenslauf einer Katze

Natürlich habe ich zunächst eine `KatzenException` angelegt, die die Klasse `Exception` erweitert.

```
class KatzenException extends Exception { }
```

Dennoch meldet Java beim Kompilieren ein Problem: Keine der Methoden würde eine `KatzenException` werfen.

Die `KatzenException` ist eben keine *unchecked* Exception. Einige Exceptions wie die Erben der `RuntimeException` werden von Java nicht weiter geprüft. Unsere selbst gebaute Exception ist nicht so privilegiert. Java prüft, ob die gefangene Exception überhaupt von einer der Methoden ausgelöst wird. Wir könnten nun bei

der Methode `katze.schlaeft` eine solche Ankündigung einbauen.

```
void schlaeft() throws KatzenException {
```

Aber die Gemahlin akzeptiert eben nur erhebliche Störungen als gewichtige Gründe, die Katze ins Tierheim abzuschieben – schlafen gehört leider nicht dazu. So werden die beiden Hauptaktivitäten einer Katze, aus denen ja auch ihr Name zusammengesetzt ist – Kacken und Kotzen –, für eine Abschiebung der Katze nicht reichen. In beiden Fällen gibt es Maßnahmen, die vermutlich meine Aufgabe sein werden.

```
} catch (KatzeKotztException k) {
    katze.mussZumTierarzt();
    willemer.macheDreckWeg();
} catch (KatzeKacktException k) {
    willemer.kaufeKatzenklo();
    willemer.macheDreckWeg();
}
```

Die Ausnahmen `KatzeKotztException` und `KatzeKacktException` sind natürlich Sonderfälle der allgemeineren `KatzenException` und erweitern diese.

Sollte die Katze Böses tun, also beispielsweise mich kratzen, steigen meine Chancen, sofern dies häufiger auftritt. Die Ausnahme `KatzeKratztException` wird in der Katze ablegen, wie oft es zu einem Übergriff gekommen ist.

```
} catch (KatzeKratztException k) {
     if (k.getKatze().getAttackenZahl()>5) {
          katze.mussInsHeim();
    }
}
```

Die Grundlagen sind gelegt. Dann können wir ein Katzenleben programmieren. Wir werden eine Klasse `Katze` und ein paar Exceptions brauchen.

Das Katzenleben

Das Katzenleben wird so lange währen, bis die Katze abgeschoben wurde oder sie stirbt. Gehen wir mal optimistisch davon aus, dass die Katze ins Tierheim kommt, bevor sie stirbt. Das wäre erstens früher und zweitens bräuchte ich im Falle des Ablebens der Katze vermutlich ein wasserdichtes Alibi. Und wer weiß schon, wann dies passiert. Dann sähe das Katzenleben folgendermaßen aus:

```
class KatzenLeben {
    public static void main(String[] args) {
        Mensch willemer = new Mensch("Willemer");
        Katze katze = new Katze();
        while (katze.lebtNochBeiEuch()) {
            try {
                katze.schlaeft();
                katze.frisst();
                katze.schnurrt();
            } catch (KatzeKotztException k) {
                katze.mussZumTierarzt();
                willemer.macheDreckWeg();
            } catch (KatzeKacktException k) {
                willemer.zeigeKatzenklo();
                willemer.macheDreckWeg();
            } catch (KatzeKratztException k) {
                if (k.getKatze().getAttackenZahl()>5) {
                    katze.mussInsHeim();
                }
            } catch(KatzenException k) {
               // sollte eigentlich nicht auftreten
               k.printStackTrace();
            }
        }
    }
}
```

Listing 13.16: Das Katzenleben als Klasse

Die `KatzenException` ist nun in die konkreten Erweiterungen `KatzeKotztException`, `KatzeKacktException` und `KatzeKratztException` aufgelöst. Die Basisklasse sollte also nicht mehr auftreten. Dies kann man leicht sicherstellen, indem man die Klasse `KatzenException` als `abstract` definiert. Da aber in den Methoden angedroht wird, die `KatzenException` oder eine ihrer Erweiterungen zu werfen, muss auch die `KatzenException` gefangen werden.

Der Mensch

Nun brauchen wir die Klasse `Mensch`, um meine Aufgaben zu spezifizieren. Das ist nicht kompliziert. Der Mensch bekommt im Konstruktor einen Namen, macht den Dreck weg und zeigt der Katze das Katzenklo.

```
class Mensch {
    String name;
    public Mensch(String pName) {
        name = pName;
    }
    void macheDreckWeg() {
        System.out.println("Willemer macht Dreck weg.");
    }
    void zeigeKatzenklo() {
        System.out.println(name + " zeigt Katze das Katzenklo.");
    }
}
```

Listing 13.17: Die Klasse `Mensch`

Die Ausnahmen

Nun kommen die Ausnahmen. Die Mutter aller Ausnahmen ist in unserem Fall die `KatzenException`. Letztlich benötigen wir sie aber nur, um eine spezielle Exception zu erweitern. Um abzusichern, dass sie nur in einer Erweiterung verwendet wird, machen wir sie abstrakt.

```
abstract class KatzenException extends Exception { }
```

Da muss nichts weiter passieren. Wenn noch eine Meldung erforderlich wäre, könnte man `getMessage` überschreiben. Aber für unser Beispiel ist das nicht notwendig. Nun geht es an die Exceptions für das Unwohlsein der Katze. Sie sind einfach Sonderfälle einer `KatzenException`. Sie müssen auch nichts Besonderes leisten. Ihr Wert liegt darin, dass sie sich von der `KatzenException` unterscheiden und so getrennt von dieser gefangen werden können.

```
class KatzeKacktException extends KatzenException { }
class KatzeKotztException extends KatzenException { }
```

Wenn die Katze kratzt, dann soll der Angriff gezählt werden. Das könnte man natürlich in einem statischen Attribut der Exception tun. Aber wir wollen so gerecht sein, dass jeder Angriff der jeweiligen Katze zugeordnet werden kann, falls es einmal mehrere davon geben sollte. Also merkt sich jede Exception die auslösende Katze. Sie wird dem Konstruktor als Parameter übergeben. Damit `catch` die Katze identifizieren kann, gibt es eine Methode `getKatze`.

```
class KatzeKratztException extends KatzenException {
    private Katze katze = null;
    public KatzeKratztException(Katze pKatze) {
        katze = pKatze;
    }
    public Katze getKatze() {
        return katze;
    }
}
```

Listing 13.18: Die relevante Exception

Die Katze

Als letzte Klasse brauchen wir nun die Klasse `Katze`. Sie beschreibt die Tätigkeiten und löst auch die Exceptions aus.

```
class Katze {
    private boolean istNochDa = true;
    private long zyklus = 0;
    private int attacken = 0;
    // lebtNochBeiEuch ist die Schleifenbedingung für das
    // Katzenleben.
    boolean lebtNochBeiEuch() {
       zyklus++; // Die Schleifenrunde wird gezählt
       return istNochDa;
    }
    void schlaeft() throws KatzenException {
        // Jedes 7. Mal muss sie sich übergeben.
        if (zyklus%7==0) {
            throw new KatzeKotztException();
        }
    }
```

```
    void frisst() throws KatzenException {
        // Jedes 3. Mal muss sie sich entleeren.
        if (zyklus%3==0) {
            throw new KatzeKacktException();
        }
    }
    void schnurrt() throws KatzenException {
        // Die Katze tut nur lieb. Jedes 3. Mal kratzt sie.
        if (zyklus%13==0) {
            attacken++;
            System.out.println("Autsch!");
            throw new KatzeKratztException(this);
        }
    }
    // Bilanzauswertung
    int getAttackenZahl() {
        return attacken;
    }
    // Reparatur
    void mussZumTierarzt() {
        System.out.println("Zum Tierarzt");
    }
    // Die Hoffnung!
    void mussInsHeim() {
        istNochDa = false;
        System.out.println("Katze weg!!!!");
    }
}
```

Listing 13.19: Die Klasse Katze

Nun kann man alles übersetzen und schauen, was passiert. Na bitte. Das Programm wiederholt ständig, dass Willemer den Dreck wegmachen soll und irgendwann ist die Katze weg!

Und was soll ich sagen? Die Realität hat sich tatsächlich an das Programm gehalten. Irgendwann zeigte ich der Gemahlin meine Narben und erwartete nun den Abtransport der Katze. Sogar eine Lebewohl-Karte hatte ich schon gedruckt. Da kommt meine Frau mit einem Kratzbaum nach Hause.

Ich soll mich nicht so anstellen und lieber das Verhältnis mit der Katze dadurch verbessern, dass ich mit ihr mehr spiele. Ich überdenke den Vorschlag. Wie wäre es mit einer Runde Katzenpolo?

Nochmal clone

In Kapitel 9 hatten wir zum Kopieren eines Objekts selbst eine Methode `clone` geschrieben und ich hatte behauptet, dass der Name *clone* nicht willkürlich gewählt wäre. Sie haben mir geglaubt und das war sehr nett von Ihnen.

Nun darf ich Ihnen mehr erzählen, ohne fürchten zu müssen, Sie zu verschrecken. Die Methode `clone` wird in Java bereits in der Klasse `Object` implementiert und wird darum an alle Klassen vererbt.

Allerdings ist `clone` `protected`. Sie können also `clone` nicht von einem Hauptprogramm aufrufen, um ein Objekt wie beispielsweise ein Datum zu kopieren. Sie könnten aber in einer selbst geschriebenen `clone`-Methode von `Datum` einfach die `clone`-Methode von `Object` rufen und damit wäre die Arbeit schnell erledigt.

Bei der Klasse `Datum` funktioniert das prima, weil sie keine Referenzen enthält. Nun werden Sie mich fragen, warum ich Sie in Kapitel 9 habe schwitzen lassen, wenn das so einfach ist. Ich zeige Ihnen einfach einmal den Quelltext und dann werden Sie mich wieder liebhaben:

```
public class Datum implements Cloneable {
    int jahr;
    int monat;
    int tag;

    public Datum clone() {
        Datum kopie = null;
        try {
           kopie = (Datum) super.clone();
        } catch(CloneNotSupportedException e) {
        }
        return kopie;
    }
}
```

Listing 13.20: `Datum` mit `clone` und `Exception`

Mit `super.clone` rufen Sie die Methode `clone` der Basisklasse auf. Die einzige Basisklasse von `Datum`, die eine Methode `clone` anbietet, ist eben `Object`, die ja die Basisklasse aller Klassen ist. Tatsächlich wirft diese Methode aber die `CloneNotSupportedException`, die Sie nun fangen müssen. Damit die Methode `clone` die Arbeit übernimmt, muss allerdings die Klasse `Datum` das Interface `Cloneable` implementieren. Dieses hat keine Methoden. Es soll nur signalisieren,

dass die Klasse zulässt, dass ihr Inhalt »flach« kopiert wird. Das alles wäre für den Anfang vermutlich etwas viel auf einmal gewesen.

Und nun möchten Sie noch wissen, was eine »flache« Kopie ist. Bei einer solchen Kopie werden alle Attribute kopiert, ohne die Referenzen aufzulösen. Bei Primitiven wird einfach eine Kopie gefertigt. Bei Referenzen wird diese Kopie auf dasselbe Objekt verweisen wie das Original. Das Objekt, auf das verwiesen wird, wird allerdings nicht kopiert. Es existieren nach der Kopie also zwei Referenzen auf ein Objekt.

Aufgaben

Erstellen Sie eine Klasse `Bruch`, die einen Bruch bestehend aus Zähler und Nenner realisiert. Die Klasse soll die typischen Grundrechenarten als Methoden zur Verfügung stellen, beispielsweise `add`, `sub`, `mul` und `div`.

Dabei können zwei dumme Situationen entstehen.

- ✔ Die Zahl im Nenner könnte 0 werden. Damit ist ein Weiterarbeiten unsinnig.
- ✔ Bei Multiplikationen könnte die maximale Größe des Integers für Zähler oder Nenner überschritten werden. Das Ergebnis wäre dann unbrauchbar.

In beide Fällen soll eine selbst gebastelte Exception ausgelöst werden.

Kapitel 14

Fertige Bausteine zur Selbstbedienung

IN DIESEM KAPITEL

Zufallszahlen für Lotto und Computerspiele

Wann endet der Java-Kalender?

Eine Klasse für Mathe-Fans

Selbst gebasteltes vermittelt so schön den Hauch von weihnachtlicher Stimmung. Alle freuen sich, wenn sie die selbst getöpferten Aschenbecher von Tante Frida geschenkt bekommen, und da macht es auch nichts, wenn man schon seit Jahren Nichtraucher ist. Und nur vereinzelt wird in der Familie die Hoffnung geäußert, dass Ton bald zu den seltenen Erden gerechnet wird und dann nicht mehr frei am Markt erhältlich ist.

Bei Programmierern ist es anders. Die greifen lieber auf eine fertige Bibliothek zurück, als selbst zu programmieren. Das selbst geschriebene mag ja individueller sein. Aber es enthält vermutlich auch viele individuelle Fehler. Java bringt einen ganzen Sack voller Bibliotheken mit, bei denen sich der Programmierer nur selbst bedienen muss.

Wie der Zufall es will

Spiele sind ein ganz typisches Anwendungsgebiet für Zufallszahlen. So benötigen wir sie ja auch für die Schiffspositionen im Bermuda-Spiel, das wir in Kapitel 7 angefangen haben. Aber auch für Simulationen sind Zufallszahlen nützlich. So könnten Sie per Simulation recht einfach berechnen, wie wahrscheinlich es ist, dass beim Lotto ein Pärchen gezogen wird. Das ist mit den Methoden der Stochastik schwieriger zu ermitteln. Außerdem können wir daraus eine prima Übung machen.

Wir erzeugen Zufälle

Wenn Sie also etwas mehr Abwechslung im Leben wollen, erzeugen Sie ein Objekt der Klasse `java.util.Random`. Das stellt Ihnen ein paar Methoden zur Verfügung, die alle mit `next` beginnen und einer Typbezeichnung enden. Bei Aufruf erhalten Sie immer wieder eine weitere Zufallszahl.

```
import java.util.Random;
// ...
Random z = new Random();
int    i = z.nextInt()
long   l = z.nextLong()
float  f = z.nextFloat()
double d = z.nextDouble()
```

Die ganzzahligen Methoden liefern einen Wert über den gesamten Zahlenraum, auch im negativen Bereich. In der Regel wird eine Zufallszahl aus einem bestimmten Intervall benötigt. Diese wird typischerweise durch den Einsatz der Modulo-Rechnung eingegrenzt. Wenn Sie nur positive Werte gebrauchen können, verwenden Sie die Betragsfunktion `Math.abs`. Das folgende Programmbeispiel liefert das Ergebnis eines Würfelwurfs.

```
import java.util.Random;

public class WuerfelWerfen {
    public static void main(String[] args) {
        Random zufall = new Random();
        int wurf = Math.abs(zufall.nextInt()) % 6 + 1;
        System.out.println(wurf);
    }
}
```

Listing 14.1: Simulation eines Würfels

Die Fließkomma-Methoden liefern einen Wert, der größer oder gleich 0, aber kleiner als 1 ist. Gleiches liefert die Methode `Math.random`, die letztlich auf der Utility-Klasse aufsetzt. Dieser Wert wird dann mit dem gewünschten Wertebereich multipliziert und, falls gewünscht, werden die Nachkommastellen abgetrennt.

```
import java.util.Random;

public class FliessWuerfel {
    public static void main(String[] args) {
        Random zufall = new Random();
        double wurf = Math.floor(
                          Math.abs(
                              zufall.nextDouble()) * 6 + 1);
        System.out.println(wurf);
    }
}
```

Listing 14.2: Simulation eines Würfels mit Fließkommazahlen

Da allerdings immer wieder ganze Zahlen benötigt werden, die zwischen 0 und einer bestimmten Zahl liegen, können Sie `nextInt` die Obergrenze als Parameter übergeben.

```
import java.util.Random;

public class WuerfelWerfen {
    public static void main(String[] args) {
        Random zufall = new Random();
        int wurf = zufall.nextInt(6) + 1;
        System.out.println(wurf);
    }
}
```

Listing 14.3: Würfeln leicht gemacht

Lottozahlenpärchen

Für mich ergab sich eine der ersten praktischen Einsatzmöglichkeiten eines Computers im Zusammenhang mit einer Mathematikhausaufgabe, in der ich die Wahrscheinlichkeit ermitteln sollte, dass bei einer Lottoziehung mindestens zwei Zahlen benachbart sind. Der Blick in das Mathematikbuch versprach einen langen Abend. Das musste man doch viel einfacher programmieren können. Es klappte tatsächlich sehr gut. Eine Nacht wurde simuliert, damit das Ergebnis auch möglichst dicht an dem mathematisch korrekt ermittelten Ergebnis sein würde.

Um die Geschichte vollständig zu erzählen, war das Ergebnis durchaus gut, mit einer gewissen Abweichung, die ich der Ungenauigkeit des Taschenrechners in die Schuhe schob. Der Mathematiklehrer war fasziniert und ließ mich an der Tafel vorführen, wie ich zu diesem Ergebnis gekommen war. Simulationen haben manchmal kurze Beine.

Lottozahlen erzeugen

Nachdem ich es schon einmal programmiert habe, sind Sie nun dran.

- ✔ Eine Lottoziehung besteht aus sechs Zahlen, die zwischen 1 und 49 liegen und keine doppelten enthalten.
- ✔ Sie benötigen ein Array für sechs ganze Zahlen.
- ✔ Das befüllen Sie mit Zufallszahlen. Kopieren Sie einfach das Beispiel mit dem Würfelwurf von oben und verwenden Sie Modulo 49 statt 6.
- ✔ Beim Lottospielen dürfen bei einer Ziehung keine gleichen Zahlen auftreten. Wie verhindern Sie das? Sie prüfen beim Ziehen jeder Zahl, ob Sie diese Zahl schon einmal gezogen haben. Solange dies der Fall ist, ziehen wir erneut.

Nun versuchen Sie, zunächst einmal selbst eine Lösung zu finden, bevor Sie im

Listing spicken.

```
import java.util.Random;
public class LottoZiehung {
    public static void main(String[] args) {
        final int MAX = 6;
        int lotto[] = new int[MAX];
        Random zufall = new Random();
        for (int i=0; i<MAX; i++) {
            boolean doppelt = false;
            do {
                lotto[i] = Math.abs(zufall.nextInt()) % 49 + 1;
                doppelt = false; // wieder zurücksetzen!!!
                for (int j=0; j<i; j++) {
                    if (lotto[i]==lotto[j]) {
                        doppelt = true;
                    }
                }
            } while(doppelt);
        }
        for (int i=0; i<MAX; i++) {
            System.out.println(lotto[i]);
        }
    }
}
```

Listing 14.4: Ziehung der Lottozahlen



Wenn Sie das Zurücksetzen der booleschen Variablen doppelt vergessen, entsteht bei der ersten Dopplung eine Endlosschleife! Warum erzähle ich Ihnen das? Weil es mir passiert ist.

Paare identifizieren

Die Lottozahlen sind gezogen und befinden sich im Array. Aber wie ermitteln wir die Pärchen?

- Erste Möglichkeit: Man vergleicht jede Zahl mit ihrem Nachbarn und schaut, ob die Differenz 1 ist. Beinahe! So würde es funktionieren, wenn die Zahlen sortiert wären. Sind sie aber nicht! Wir könnten die Zahlen sortieren. Dazu müssten wir das Array mehrfach durchlaufen. Aber es muss auch einfacher gehen.
- Zweite Möglichkeit: Gehen Sie einfach genauso vor wie bei der Suche nach doppelt gezogenen Werten! Sie vergleichen einfach jede Zahl mit jeder anderen. Wenn zwei Zahlen eine Differenz von 1 haben, ist es ein Pärchen.

```
boolean paar = false;
for (int i=0; i<MAX; i++) {
    // Schleife über die vorher gezogenen Zahlen
    for (int j=0; j<i; j++) {
        if (Math.abs(lotto[i]-lotto[j]) == 1) { // Pärchen!!!
            paar = true;
        }
    }
}
if (paar) {
    anzahlDoppelte++;
}
```

Listing 14.5: Pärchenzählung

Nun müssen wir nur noch eine große Schleife um die Ziehung machen und sie 1.000.000 Mal durchlaufen lassen, um ein statistisch zuverlässiges Ergebnis zu erhalten. Alles klar! Erster Testlauf. Die Wahrscheinlichkeit ist die Zahl der Pärchen geteilt durch die Zahl der Ziehungen. Und die Wahrscheinlichkeit ist ... 0! 0? Das kann nicht sein. Wie sieht die Rechnung für die Wahrscheinlichkeit aus?

```
System.out.println(anzahlDoppelte/anzahlZiehungen);
```

Ach natürlich: Der klassische Fehler! Eine ganze Zahl durch eine ganze Zahl gibt immer nur eine ganze Zahl. Und so steht da 0 statt 0,irgendwas. Dagegen hilft eine Multiplikation mit 1.0, damit eine Fließkommazahl daraus wird.

Hier finden Sie nun das komplette Programm.

```
import java.util.Random;

public class LottoPaare {

    public static void main(String[] args) {
        final int MAX = 6;
        int lotto[] = new int[MAX];
        long anzahlZiehungen = 0;
        long anzahlDoppelte = 0;
        Random zufall = new Random();
        for (int ziehungen=0; ziehungen<1000000; ziehungen++) {
            for (int i=0; i<MAX; i++) {
                boolean doppelt = false;
                do {
                    lotto[i] = Math.abs(zufall.nextInt())
                            % 49 + 1;
                    doppelt = false;
```

```
                    for (int j=0; j<i; j++) {
                        if (lotto[i]==lotto[j]) {
                            doppelt = true;
                        }
                    }
                } while(doppelt);
            }
            // Lottozahlen sind gezogen
            boolean paar = false;
            for (int i=0; i<MAX; i++) {
                for (int j=0; j<i; j++) {
                    if (Math.abs(lotto[i]-lotto[j]) == 1) {
                        paar = true;
                    }
                }
            }
            if (paar) {
                anzahlDoppelte++;
            }
            anzahlZiehungen++;
        }
        System.out.println("Verhältnis: "
            + 1.0*anzahlDoppelte/anzahlZiehungen);
    }
}
```

Listing 14.6: Komplette Lottosimulation

Nun funktioniert die Simulation. Und erstaunlicherweise treten tatsächlich fast in der Hälfte aller Fälle Pärchen auf.

Das Datum, der Kalender und die Neuerungen

Kaum etwas verändert sich so schnell wie die Zeit. Bei Java scheint das für den Neueinsteiger besonders zu gelten. Dort gibt es diverse Klassen und Typen, die Datum und Uhrzeit realisieren. Sie befinden sich in verschiedenen Packages.

- ✔ `java.time`
- ✔ `java.util.Calendar`
- ✔ `java.util.Date`

- `java.sql.Date`

Warum ist die Definition von Datum und Uhrzeit so schwer? Weil das Thema wirklich etwas kompliziert ist. Ein paar Beispiele gefällig?

- Es ist jetzt 22:17. Aber wo? In Sydney oder in Texas? Oder deutsche Sommerzeit?
- Warum hat der Februar als einziger 28 Tage? Abgesehen von den Jahren, in denen er 29 Tage hat.
- Warum folgt auf den 4.10.1582 der 15.10.1582?
- Wie viele Tage liegen noch zwischen Neujahr und Weihnachten in einem Schaltjahr?

Die Fragen zeigen, dass der Umgang mit Datum und Uhrzeit an sich nicht trivial ist. Schon die Menschen haben es nicht leicht damit und das hat etwas mit Traditionen zu tun. Oder warum gibt es zwölf statt zehn Monate?

Java schleppt auch noch ein paar alte Traditionen mit sich herum, die aus verschiedensten Quellen kommen. UNIX und Linux betrachten einen Zeitpunkt als die Zahl der Sekunden seit dem 1.1.1970. MS-DOS fand den 1.1.1980 relevanter.

Moderne Zeiten seit Java 8

Mit Java 8 wurde eine neue Bibliothek für Datum und Zeit eingeführt. Sie befindet sich in der Bibliothek `java.time`. Die bisherigen Klassen aus `java.util` bleiben Ihnen erst einmal erhalten. Es wäre ja auch unangenehm, alle älteren Programme umzuschreiben.

Neue Entwicklungen sollten Sie wenn möglich bereits auf der Basis der neuen Klassen aus `java.time` schreiben. Falls Sie bei Arbeiten an älterem Code in Eclipse gewarnt werden, dass bestimmte Methoden der alten Klassen »deprecated« seien, können Sie vermuten, dass diese in einer der nächsten Java-Versionen vielleicht gar nicht mehr unterstützt werden.



Die Reihenfolge eines Datums ist in Deutschland Tag, Monat, Jahr. In vielen Ländern ist dies umgekehrt. Das ist auch sinnvoll. Wenn Sie Dateinamen in der Form JJJJ-MM-TT verwenden, werden die Dateinamen bei alphabetischer Sortierung auch gleich historisch korrekt einsortiert. Darum wird dieses Format auch von der ISO 8601 vorgeschlagen. Bei Uhrzeiten ist das Format hh:mm:ss. Alle Kalenderdaten verstehen sich auf der Basis der UTC-Zone und dem gregorianischen Kalender.

Ein lokales Datum: LocalDate

Das Datum wird durch die Klasse `LocalDate` repräsentiert. Wie Sie jedes Jahr an

Silvester erfahren, beginnt in Australien das neue Jahr bereits ein paar Stunden früher als bei uns. Damit ist Datum und Uhrzeit also grundsätzlich ortsabhängig. Für die Frage, ob der Kunde in Zahlungsverzug ist oder wann Sie Ihre Freunde zur Geburtstagsfeier einladen, interessiert Sie allerdings nur das hiesige Datum und nicht das in Sydney. In den meisten Fällen ist also `LocalDate` vollkommen ausreichend.

Sie erstellen ein Objekt der Klasse `LocalDate` nicht über den Konstruktor, sondern bemühen eine der statischen Methoden von `LocalDate`:

- Wenn Sie das Geburtsdatum von Bernhard Grzimek anlegen wollen, verwenden Sie die Methode `of`, der Sie Jahr, Monat und Tag als Parameter übergeben.
  ```
  LocalDate datum = LocalDate.of(1909, 4, 24);
  ```
- Sind Sie dagegen mehr am heutigen Datum interessiert, verwenden Sie die Methode `now`, die keine Parameter benötigt.
  ```
  LocalDate heute = LocalDate.now();
  ```

Eine nachträgliche Änderung eines einmal erstellten Datums ist nicht erwünscht. Stattdessen wird wie bei Strings durch die Methoden ein neues Objekt erzeugt, das wiederum nicht änderbar ist. Es gibt also keine Methode wie etwa `setYear`.

Dafür ist das Auslesen der Bestandteile eines Datums möglich: In den folgenden Beispielen wird davon ausgegangen, dass das Objekt `datum` der Klasse `LocalDate` initialisiert ist.

- Die Methode `getYear` liefert die Jahreszahl als `int` zurück.
  ```
  int jahr = datum.getYear();
  ```
- Um den Monat zu ermitteln, verwenden Sie die Methode `getMonthValue`. Sie liefert eine Zahl im Bereich von 1 bis 12 zurück. Das ist insofern bemerkenswert, weil diverse Datumsklassen den Monat von 0 bis 11 zählen.
  ```
  int monat = datum.getMonthValue();
  ```
- Es gibt eine zweite Methode, die den Monat liefert, allerdings nicht als ganzzahligen Wert, sondern als Objekt des Aufzählungstyps `Month`.
  ```
  Month month = datum.getMonth();
  ```
  Über das Objekt der Klasse `Month` können Sie die Bezeichnung des Monats ermitteln.
  ```
  month.getDisplayName(TextStyle.FULL, Locale.GERMAN);
  ```
  Dies liefert den String »April«. Wird die Kurzform »Apr« gewünscht, verwenden Sie `TextStyle.SHORT`.
- Um den Tag zu ermitteln, rufen Sie `getDayOfMonth` auf. Die Methode liefert den Tag zwischen 1 und 31 zurück.

```
int tag = datum.getDayOfMonth();
```

- Sie können auch den Wochentag mit der Methode `getDayOfWeek` ermitteln. Sie liefert ein Objekt der Klasse `DayOfWeek`. In seinem Herzen ist `DayOfWeek` eine Enumeration. Die einzelnen Tage können als großbuchstabige, englische Konstanten mit dem Wert verglichen werden.

```
DayOfWeek tag = datum.getDayOfWeek();
if (tag == DayOfWeek.WEDNESDAY) {
    System.out.println("Mittwoch");
}
```

Mit Kalenderdaten rechnen

Die Klasse `LocalDate` stellt mehrere Methoden bereit, um neue Termine zu berechnen.

- Um den Termin in zwei Wochen zu ermitteln, verwenden Sie die Methode `plusDays`.

```
LocalDate termin = datum.plusDays(14);
```

- Mit dem Aufruf von `datum.plusWeeks(2)` wären Sie allerdings zum gleichen Ergebnis gekommen.

```
LocalDate frist = datum.plusWeeks(2);
```

- Sie ahnen nun vermutlich schon, dass es auch `plusMonths` gibt. Ich brauche also nicht auszuführen, was die Methode genau tut.
- Und nun können Sie schon im Chor mitsingen: Es gibt eine Methode `plusYears`.

Auch die Subtraktion ist möglich, Sie müssen die Silbe `plus` nur durch `minus` ersetzen.

Die Differenz zwischen zwei Kalenderdaten errechnen Sie über die Klasse `Period`. Sie wollten schon immer wissen, wie sich der Geburtstag von Bernhard Grzimek und Edsger Wybe Dijkstra unterscheidet.

```
public void calcDijkstraMinusGrzimek() {
    LocalDate grzimek = LocalDate.of(1909, 4, 24);
    LocalDate secondDate = LocalDate.of(1930, 5, 11);
    Period period = Period.between(grzimek, secondDate);
    int tage = period.getDays(); // 18
    int monate = period.getMonths(); // 9
    int jahre = period.getYears(); // 4
    boolean aelter = period.isNegative(); // false
    System.out.println("Dijkstra ist "+tage+" Tage, "
            +monate+" Monate und "+jahre+" Jahre "
            +(aelter?"älter":"jünger")+" als Grzimek");
}
```

Listing 14.7: Methode zur Ermittlung des Altersunterschieds

LocalTime

Analog zu `LocalDate` gibt es `LocalTime`, die sich mit der Uhrzeit befasst. Auch hier gibt es die Methoden `of` und `now`, um eine Uhrzeit anzulegen.

Auch bei der Zeit gibt es Addition und Subtraktion mit den vertrauten `plus`- und `minus`-Methoden, also `addHours`, `addMinutes` und `addSeconds`.

Instant

Wenn es eher um Zeitintervalle als um Uhrzeiten geht, ist die Klasse `Instant` in der Kombination mit `Duration` eine gute Wahl. Auch die Objekte der Klasse `Instant` lassen sich mit der Methode `now` erstellen.

```
Instant start = Instant.now();
// Hier arbeitet sich das Programm einen Wolf.
Instant stopp = Instant.now();
Duration dauer = Duration.between(start, stopp);
long seconds = dauer.getSeconds();
long absoluteResult = dauer.abs().toMinutes();
```

Listing 14.8: Eine kleine Stoppuhr

LocalDateTime

Als Kombination von `LocalDate` und `LocalTime` gibt es `LocalDateTime`. Sie vereinigt Datum und Uhrzeit. Zur Erzeugung funktioniert auch hier die Methode `now` völlig überraschungsfrei.

Um einen bestimmten Zeitpunkt mit der Methode `of` zu erzeugen, kann man so vorgehen, dass ein `LocalDate`- und ein `LocalTime`-Objekt als Parameter übergeben wird, oder Sie geben nacheinander Jahr, Monat, Tag, Stunde und Minute als ganze Zahlen an.

Zeitzonen

Mit den `Local`-Klassen haben wir die Zeitzonen dieser Erde galant ignoriert. Um eine Zeitzone zu realisieren, wird die Klasse `ZoneId` verwendet. Auch diese wird mit der Methode `of` initialisiert. Dabei erwartet sie einen String mit der Zonenbezeichnung als Parameter.

Ein zeitzonenabhängiger Zeitpunkt der Klasse `ZonedDateTime` wird wiederum über die Methode `of` erstellt. Als Parameter wird ein Objekt der Klasse `LocalDateTime` und ein Bezeichner für die Zeitzone vom Typ `ZoneId` erwartet.

```
ZoneId losAngeles = ZoneId.of("America/Los_Angeles");
ZoneId berlin = ZoneId.of("Europe/Berlin");
LocalDateTime dateTime = LocalDateTime.of(2016, 02, 20, 12, 0);
ZonedDateTime berlinDateTime = ZonedDateTime.of(dateTime, berlin);
```

Listing 14.9: Zeitpunkte in Zeitzonen

Calendar

Vor der Java-Version 8 gab es natürlich auch schon Kalender und Uhrzeiten. Und dementsprechend finden Sie in diversen Programmen diese älteren Varianten.

Java hält schon etwas länger eine eigene Klasse `Calendar` für Kalenderdaten bereit. Listing 14.10 zeigt, wie Sie Datumsvariablen erstellen, die Sie dann auf Grzimeks Geburtsdatum setzen können, also auf den 24. April 1909.

```
import java.util.Calendar;
public class Kalender {
   public static void main(String[] args) {
         Calendar datum = Calendar.getInstance();
         // Parameter: Jahr, Monat, Tag, Stunde, Minute
         datum.set(1909, 3, 24, 8, 37);
    }
}
```

Listing 14.10: Grzimeks Geburtstag

Sie werden vielleicht ein paar Dinge rügen.

- Erstens steht da eine 3 statt der 4 für den April. Tatsächlich ist das richtig, da die Monatszählung bei `Calendar` mit 0 beginnt. Bei allen anderen Angaben beginnt die Zählung bei 1. Das macht man absichtlich, damit in Fachbüchern etwas drinsteht, auf das der Leser nicht intuitiv kommt.
- Zweitens werden Sie fragen, woher ich weiß, dass Bernhard Grzimek um 8:37 Uhr geboren wurde. Ich könnte behaupten, ich hätte seine Mutter gefragt, aber verschwenden wir keine Zeit: Ich habe mir die Uhrzeit ausgedacht.
- Und drittens: Was soll das mit der `getInstance`-Methode der Klasse? Ist das

ein Singleton? Nein, ein Singleton kann es schon deshalb nicht sein, weil man in einem Programm ja eventuell mehr als ein Datum setzen möchte. Aber tatsächlich wird hier auch über die Methode `getInstance` eine Referenz geholt. In diesem Fall ist aber der Grund ein anderer. Hier wird die Implementierung der abstrakten Basisklasse `Calendar` beschafft. Die kann nämlich regional sehr unterschiedlich sein, so wie es eben unterschiedliche Kalender in der Welt gibt.

Und hier kommt Papst Gregor ins Spiel. Der gregorianische Kalender `GregorianCalendar` erweitert nämlich als nicht abstrakte Klasse die abstrakte Klasse `Calendar`. Der Aufruf von `Calendar.getInstance` liefert ein Objekt von `GregorianCalendar`.



Lassen Sie uns nun über die weltpolitisch wichtigste Tat von Papst Gregor sprechen. Nein, Sie sind hier nicht im katholischen Religionsunterricht. Gregor ließ 1582 im damals gültigen julianischen Kalender ein paar Tage im Oktober ausfallen und sorgte so dafür, dass der Frühlingsanfang wieder auf den 21. März fiel. Daran sehen Sie wieder die Ungerechtigkeit der Welt. Wenn Sie ein paar Tage ausfallen lassen, bekommen Sie dafür Ärger mit dem Arbeitgeber. Papst Gregor tut es und prompt wird der gregorianische Kalender nach ihm benannt.

Sehr originell sind die Getter- und Setter-Methoden von `Calendar`. Die meisten Attribute eines Kalenders sind ja ganze Zahlen, die durch `int` repräsentiert werden. Statt nun für jedes Attribut eine eigene Getter- und Setter-Methode zu implementieren, gibt es eine Methode `get`, die als Parameter eine Konstante erhält, die besagt, welches Attribut geliefert werden soll. Umgekehrt hat die Setter-Methode zwei Parameter. Der erste besagt, welches Attribut gesetzt werden soll, und der zweite Parameter ist der neue Wert.

```
int get(int attribut)
void set(int attribut, int wert)
```

Die Datumskonstanten werden in der Klasse `Calendar` festgelegt. Die Bezeichnungen werden also an den Klassennamen `Calendar` angehängt.

Konstantenname	Bedeutung
ERA	Vor (0) oder nach (1) Christi Geburt
YEAR	Jahr
MONTH	Monat (0–11)
WEEK_OF_YEAR	Woche im Jahr
WEEK_OF_MONTH	Woche im Monat
DAY_OF_MONTH	Tag im Monat (1–31)
DAY_OF_YEAR	Tag im Jahr (1–366)

DAY_OF_WEEK	Wochentag (1–7)
DAY_OF_WEEK_IN_MONTH	x-ter Wochentag im Monat
AM_PM	AM (0) / PM (1)
HOUR	Amerikanische Stunde (0–12)
HOUR_OF_DAY	Stunde (0–23)
MINUTE	Minute
SECOND	Sekunde
MILLISECOND	Millisekunden
ZONE_OFFSET	Zeitzonenversatz in Millisekunden
DST_OFFSET	Sommerzeitversatz in Millisekunden

Tabelle 14.1: Datumskonstanten

Auf dieser Basis schreiben wir ein kleines Programm, um endlich festzustellen, an welchem Wochentag Grzimek geboren wurde.

```
import java.util.Calendar;
public class Wochentag {
    static final String[] wochentag
        = {"So", "Mo", "Di", "Mi", "Do", "Fr", "Sa"};
    public static void main(String[] args) {
        Calendar datum = Calendar.getInstance();
        datum.set(Calendar.YEAR, 1909);
        datum.set(Calendar.MONTH, 3);
        datum.set(Calendar.DATE, 24);
        System.out.println(wochentag[datum.get(
                    Calendar.DAY_OF_WEEK)-Calendar.SUNDAY]);
    }
}
```

Listing 14.11: Wochentagsberechnung

Es wird »Sa« ausgegeben, also Samstag. Das Ergebnis der `get`-Methode für DAY_OF_WEEK ist 7. Davon wird der erste Tag der Woche, Sonntag, also SUNDAY, abgezogen, weil dieser eben als 1 codiert ist. Mit diesem Index können wir dann auf das vorbereitete String-Array zugreifen.

Soll der Geburtstag ausgegeben werden, käme noch die folgende Anweisung hinzu:

```
System.out.println( "" + datum.get(datum.DATE)
                  + "." + (datum.get(datum.MONTH)+1)
                  + "." + datum.get(datum.YEAR) );
```

Achtung: Der Monat wird von 0 bis 11 dargestellt. Darum muss er um eine 1 erhöht werden.

Die Zeit in Millisekunden setzen und lesen

Mit der Methode `System.currentTimeMillis` liefert Ihnen Java die aktuelle Zeit in der Maßeinheit Millisekunden seit dem 1.1.1970 als `long`-Wert. Das Jahr 1970 wurde gewählt, weil diese Art der Datumsberechnung erstmalig in UNIX verwendet wurde und dieses Anfang der 1970er Jahre entwickelt wurde.

Vermutlich wissen Sie nicht, wie viele Millisekunden wir es gerade jetzt haben. Als Zeitangabe mag das unpraktisch sein. Aber mit ganzen Zahlen rechnen ist lustiger als mit Monaten, die unterschiedlich viele Tage haben.

Ein Objekt der Klasse `Calendar` kann mit der Methode `setTimeInMillis` gesetzt werden. Als Parameter erwartet die Methode die Anzahl der Millisekunden seit dem 1.1.1970.

```
Calendar datum = Calendar.getInstance();
long vieleMillis = System.currentTimeMillis();
datum.setTimeInMillis(vieleMillis);
```

Falls Sie nicht die aktuelle Zeit, sondern die Millisekunden eines `Calendar`-Objekts erfahren wollen, erhalten Sie diese über die Methode `getTimeInMillis`.

Datum erhöhen

Zum Addieren gibt es die Methode `add`. Sie verwendet ähnliche Parameter wie die Methode `set`. Geben Sie als ersten Parameter die Konstante an, die sagt, was addiert werden soll.

```
datum.add(Calendar.WEEK_OF_YEAR, 2);
datum.add(Calendar.DAY_OF_MONTH, 14);
```

Vergleiche

Sie ahnen schon, dass Sie zwei `Calendar`-Objekte mit der Methode `equals` auf Gleichheit überprüfen können. Mit den Methoden `before` und `after` können Sie die Reihenfolge von Daten abfragen.

```
if (datum.equals(ziel)) { // Sind die beiden Daten gleich?
    System.out.println("gleich");
} else if (datum.before(ziel)) { // Liegt datum vor ziel?
    System.out.println("vorher");
} else if (datum.after(ziel)) { // Liegt datum nach ziel?
    System.out.println("nachher");
} else {
    System.out.println("Was ist das?");
}
```

Der alte Kalender: Date

Bereits vor der Klasse `Calendar` gab es die Klasse `Date` aus dem Package

`java.util`. Die meisten Konstruktoren sind bereits als »deprecated« abgekündigt. Um ein `Date` zu erzeugen und zu initialisieren, ist ausschließlich die Übergabe in Millisekunden noch aktuell.

```
long millis = System.currentTimeMillis();
Date tag = new Date(millis);
```

Auch die Setter-Methoden sind »deprecated«, abgesehen von `setTime`, die die Zeit seit dem 1.1.1970 in Millisekunden akzeptiert. Auch auf Getter-Seite sieht es mau aus. Lediglich `getTime` ist noch nicht »deprecated«. Man kann also sagen, dass `Date` nur noch als Kompatibilitätshülle erhalten wurde, da die Klasse aufgrund ihres Alters weit verbreitet ist.

Die Datenbankschnittstelle JDBC (siehe Kapitel 24) verwendet eine eigene Klasse `Date` aus dem Package `java.sql`, die aber unter ähnlicher Schwindsucht leidet wie ihre Cousine aus dem Package `java.util`. Kein Wunder, sie erweitert die Klasse `java.util.Date`.

Umrechnungen

Wenn Sie Daten zwischen den verschiedenen Klassen austauschen müssen, ist die Millisekunden-Schnittstelle ein heißer Kandidat. Beispielsweise tauschen die folgenden Zeilen das Datum zwischen der Klasse `Date` und `Calendar` aus.

```
Date tag = new Date();
Calendar datum = Calendar.getInstance();
datum.setTimeInMillis(tag.getTime());
// anders herum
tag.setTime(datum.getTimeInMillis());
```

Für die bessere Akzeptanz der neuen Klassen sind die alten Klassen um Umrechnungsmethoden erweitert worden.

```
java.util.Calendar calendar  = // ...
java.util.Date date    = new Date();
java.time.Instant  instant = calendar.toInstant();
// ...
instant = date.toInstant();
date  = Date.from(instant);
```

Bei Umwandlungen in `ZonedDateTime`-Objekte reicht eine Deklaration von `Calendar` nicht aus, weil erst dessen Implementation `GregorianCalendar` die Methode `toZonedDateTime` besitzt.

```
java.util.GregorianCalendar gregor = (GregorianCalendar)calendar;
java.time.ZonedDateTime zonedatetime = gregor.toZonedDateTime();
gregor = GregorianCalendar.from(zonedatetime);
```

Rechnerisches aus der Math-Klasse

Spätestens wenn Sie Ihr erstes 3D-Spiel programmieren wollen, werden Sie feststellen, dass die Winkelfunktionen in der Schulmathematik nicht reiner Selbstzweck waren. Aber Java hilft Ihnen und implementiert in der Klasse `Math` diverse mathematische Funktionen.

Da `Math` seine Methoden statisch implementiert, ist es wenig sinnvoll, ein Objekt der Klasse `Math` anzulegen. Ohne Objekte keine Methode? Doch! Es sind halt statische Methoden, so wie die `main`-Methode, für die ja auch kein Objekt angelegt wird. Und auch die Konstanten sind statisch.

Winkelfunktionen

Wenn Ihnen der Begriff der trigonometrischen Winkelfunktionen nichts sagt und Sie die Namen Sinus, Cosinus und Tangens für Comic-Figuren halten, werden Sie alles, was hier beschrieben steht, wohl eher nicht benötigen und sollten gleich zum nächsten Abschnitt wechseln. Wir müssen uns ja nicht gegenseitig die Zeit stehlen.

Für den Aufruf der Winkelfunktionen verwendet Java die typischen Funktionsnamen, wie Sie sie auch vom Taschenrechner und aus der Mathematik kennen:

- ✔ `sin`, `cos` und `tan`: Alle diese Funktionen erwarten einen Parameter, der den Winkel angibt.
- ✔ Die Umkehrfunktionen werden in der Mathematik als arcsin, arccos und arctan abgekürzt. Java wird noch kürzer und nennt die passenden Methoden `asin`, `acos` und `atan`.

Diese Funktionen liefern einen Rückgabewert vom Typ `double` zurück und erwarten genau einen Parameter, der ebenfalls vom Typ `double` ist.

```
double wert = sin(winkel);
```

Bogenmaß

Die meisten Leute messen ihre Winkel in Grad. Dabei ist es etwas ungewöhnlich, dass die Gradzahl sinkt, je spitzer der Winkel ist. Vielleicht ist das der Grund, warum Java wie die meisten Programmiersprachen die Winkelwerte in Bogenmaß verwendet. Ein rechter Winkel ist für uns 90 Grad. Im Bogenmaß dagegen $\pi/2$. Sofern Sie sich also nicht umstellen und in Zukunft die Winkel im Bogenmaß verwenden möchten, müssen Sie wohl zwischen Bogenmaß und Gradmaß umrechnen. Die passende Formel ist übersichtlich. Dankenswerterweise liefert `Math` auch noch die Konstante `PI`, sodass Sie nicht einmal 3.1415926535 eintippen müssen.

```
bogenmass = gradmass/180.0*Math.PI;
gradmass = bogenmass*180.0/Math.PI;
```

Daraus lässt sich auf die Schnelle ein kleines Programm basteln, das sowohl die Bogenmaß-Umrechnung als auch den Sinus demonstriert.

```
public class TestSin {
    public static void main(String[] args) {
        double gradmass = 90.0;
        double bogenmass = gradmass/180.0*Math.PI;
        System.out.println("Der Winkel " + gradmass +
                " lautet in Bogenma ß " + bogenmass);
        System.out.println("und hat einen Sinus von " +
                Math.sin(bogenmass));
    }
}
```

Listing 14.12: Sinusberechnung und Bogenmaß

Das Ergebnis bietet wenig Anlass zur Überraschung. Der Winkel 90.0 lautet in Bogenmaß 1.5707963267948966 und hat einen Sinus von 1.0.

Exponenten und Logarithmen

Vergessen Sie die Hilfsmittel in Sachen Potenz aus Ihren Spam-Mails. Mit Java brauchen Sie die nicht mehr. Die Klasse `Math` stellt Ihnen für die Potenz die Methode `pow` für »power« zur Verfügung. Sie erwartet zwei Parameter vom Typ `double`: zuerst die Basis und dann den Exponenten. 2^5 sieht im Programm also folgendermaßen aus. Das Ergebnis sollte 32 lauten.

```
double erg = Math.pow(2.0, 5.0);
```

Häufig wird die Exponentialfunktion zur Basis e benötigt. Diese heißt in Java `exp` und hat natürlich nur einen Parameter, da sich die Basis durch die eulersche Zahl e von selbst ergibt.

```
double erg = Math.exp(5.0);
```

Die Umkehrung dazu ist der Logarithmus. Wie beim Taschenrechner heißt er auch in Java `log`. Die Methode erwartet einen Wert vom Typ `double` als Parameter und gibt einen Wert vom Typ `double` zurück.

```
double erg = Math.log(5.0);
```

Die Methode zur Berechnung der Wurzel heißt `sqrt`. Sie kennen diese Abkürzung vielleicht von den Schildern der Römer im Asterix-Heft. Bei Java steht `sqrt` als Abkürzung für »square root«, also Quadratwurzel. Die Römer haben die Abkürzung SQRT vielleicht für ihre Pioniere verwendet, weil die sich immer unter der Graswurzel durchbuddeln mussten. Die `Math`-Methode `sqrt` erwartet einen

Parameter vom Typ `double` und liefert einen Wert vom Typ `double` zurück.

```
double erg = Math.sqrt(2.0);
```

Absolut und Rundheraus

Betrag

Die Methode `abs` berechnet den Betrag, entfernt also das Minuszeichen, sofern ein solches Vorzeichen vorlag. Das haben wir schon bei den Zufallszahlen erfolgreich genutzt. Parameter und Rückgabewert sind vom Typ `double`.

Runden

Wenn Sie eine Zahl abrunden wollen, dann verwenden Sie die Methode `floor`. Floor ist der englische Begriff für Fußboden und rundet immer nach unten ab. Soll dagegen immer nach oben gerundet werden, verwenden Sie `ceil`, wie der englische Begriff ceiling für die Decke. Wenn Sie kaufmännisch runden wollen, verwenden Sie die Methode `round`. Der Kaufmann rundet ab 0,5 nach oben, alle Werte darunter nach unten.

```
System.out.println(Math.round(2.5999)); // ergibt 3.0
System.out.println(Math.round(2.4999)); // ergibt 2.0
System.out.println(Math.ceil(2.5999));  // ergibt 3.0
System.out.println(Math.ceil(2.4999));  // ergibt 3.0
System.out.println(Math.floor(2.5999)); // ergibt 2.0
System.out.println(Math.floor(2.4999)); // ergibt 2.0
```

Maximum und Minimum

Natürlich ist es kein Problem, Maximum und Minimum zweier Zahlen über den Fragezeichenoperator zu bestimmen. Etwas übersichtlicher sind allerdings die Methoden `min` und `max` aus `Math`.

```
double max = Math.max(6.9, 2.8);
double min = Math.min(6.9, 2.8);
```

Hilfsmethoden für Arrays

Wenn Ihnen Arrays gefallen, werden Sie sich freuen, dass Java die Arrays durch eine eigene Klasse namens `Arrays` unterstützt. Die Klasse stellt ein paar sehr praktische Methoden zur Verfügung.

Um diese Methoden zu benutzen, binden Sie am Anfang Ihres Programms mit dem Befehl `import` die Klasse `java.util.Arrays` ein.

```
import java.util.Arrays;
```

Anschließend stehen die folgenden Methoden zur Verfügung.

Vergleichen mit equals

```
boolean Arrays.equals(Object[] array, Object[] nocheinarray)
```

Die Methode `equals` vergleicht Objekte. Das kennen Sie schon. Diese hier vergleicht Arrays. Das klappt bei nichtprimitiven Elementen aber nur, sofern die Array-Elemente eine Methode `equals` implementieren.

```
import java.util.Arrays;

public class ArrayEquals {
    public static void main(String[] args) {
        int feld1[] = { 4, 3, 7, 1 };
        int feld2[] = { 4, 3, 7, 1 };
        System.out.println(Arrays.equals(feld1, feld2));
    }
}
```

Listing 14.13: Arrays auf Gleichheit prüfen

Füllen mit fill

```
void Arrays.fill(Object[] array, Object element)
```

Der Aufruf füllt alle Elemente des Arrays mit dem Objekt per Zuweisung.

```
import java.util.Arrays;

public class ArrayFill {
    public static void main(String[] args) {
        int feld[] = new int[5];
        Arrays.fill(feld, 7);
        for (int zahl : feld) {
            System.out.print(" - ");
            System.out.print(zahl);
        }
    }
}
```

Listing 14.14: Arrays mit Werten füllen

Sortieren mit sort

```
void Arrays.sort(Object[] array)
```

Das Array wird sortiert. Die Elemente müssen allerdings die Schnittstelle `Comparable` implementieren. Für unsere einfachen Zahlen ist das gewährleistet.

```
import java.util.Arrays;

public class ArraySort {
    public static void main(String[] args) {
        int feld[] = { 4, 3, 7, 1 };
        Arrays.sort(feld);
        for (int zahl : feld) {
            System.out.print(" - ");
            System.out.print(zahl);
        }
    }
}
```

Listing 14.15: Arrays sortieren

Durchsuchen mit binarySearch

```
int Arrays.binarySearch(Object[] array, Object such)
```

Wenn das Sortieren geklappt hat, können Sie im Array auf binäre Art nach einem Element suchen. Das bedeutet, dass die Methode das Element in der Mitte des Arrays mit dem gesuchten Wert vergleicht. Ist der gesuchte Wert kleiner, wird die linke Hälfte weiter untersucht, andernfalls die rechte Hälfte. Diese Vorgehensweise ist natürlich schneller, als das Array Element für Element zu durchsuchen.

```
import java.util.Arrays;

public class ArraySearch {
    public static void main(String[] args) {
        int feld[] = { 1, 3, 4, 7 };
        int pos = Arrays.binarySearch(feld, 4);
        System.out.println(pos);
    }
}
```

Listing 14.16: In einem sortierten Array suchen

Das Beispiel liefert den Index 2, weil die 4 an dritter Stelle steht.

Kapitel 15

Datenelemente sammeln und anordnen

IN DIESEM KAPITEL

- Arrays, die wachsen und schrumpfen können
- Verkettete Listen, Warteschlangen und Stapel
- Mengen und assoziative Speicher
- Generics, die Collections erst typsicher machen
- Algorithmen, die eigentlich Methoden sind
- Wrapper für die Primitiven, die auch in den Container wollen

Daten sind Herdentiere. So ein übersichtliches Kleinrudel von Lottozahlen kann man prima in ein Array einsperren. Wenn es aber mehr werden, dann greift der erfahrene Java-Programmierer zur Collection. Nein, nicht in die Garbage Collection, sondern er verwendet einen Datenbehälter, den ihm das Java Collection Framework bereitstellt. Ein solcher Behälter ist nicht unbedingt größer als ein Array. Interessant ist die Flexibilität, weil man nie sicher ist, wie viele Daten denn noch kommen könnten.

Java Collection Framework

Das Java Collection Framework schafft die Möglichkeit, Objekte effizient abzulegen. Es verfügt über verschiedene Organisationsformen, je nachdem ob es darum geht, Daten möglichst schnell aufzufinden oder ein häufiges Löschen und Einfügen zu optimieren. Begleitend gibt es Methoden, die Standardoperationen auf den Containern implementieren.

Aus der Formulierung »Objekte« dürften Sie schon gefolgert haben, dass die primitiven Typen nicht in die Behälter passen. Sollten Sie genau dies unbedingt benötigen, können Sie die primitiven Typen in einer Klasse verstecken oder eine Wrapper-Klasse verwenden. Auf Wrapper werde ich in diesem Kapitel noch zurückkommen.

Interfaces

Die Grundlage des Java Collection Frameworks sind einige Interfaces. Wie Sie sicher noch wissen, schreibt ein Interface einer Klasse, die es implementiert, vor,

welche Methoden es zu implementieren hat. Damit bestimmt das Interface, wie man mit einem Datenbehälter umgehen kann und welche Methoden Ihnen bei welchem Behälter zur Verfügung stehen. Die Interfaces werden darum noch ausführlicher behandelt.

Das flexible Array: ArrayList

Eine `ArrayList` ähnelt einem Array. Es nimmt also mehrere gleichartige Elemente auf. Wie bei einem Array liegen die Elemente direkt nebeneinander und sind durchnummeriert. Der Zugriff auf das dritte Element ist genauso schnell wie der Zugriff auf das 27.569ste Element.

Sie können sich also eine `ArrayList` wie eine Reihe Bauklötze vorstellen, die direkt nebeneinander stehen. Abbildung 15.1 zeigt nummerierte Bauklötze. Jeder Bauklotz steht für ein Kästchen, in dem sich ein Objekt, oder genau genommen eine Referenz auf ein Objekt befindet.

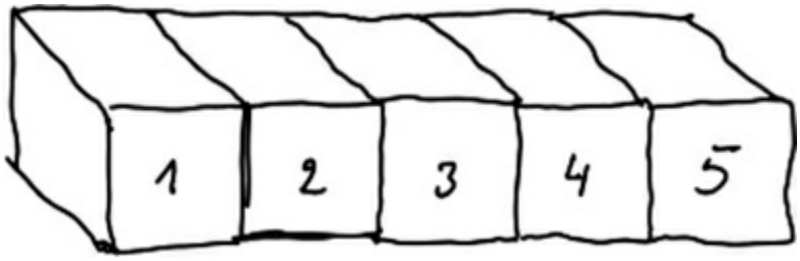


Abbildung 15.1: Die `ArrayList` in Bauklotzform

Man findet einen Bauklotz sehr schnell über seine Nummer, weil man nur die Breite eines Bausteins mit der Position multiplizieren muss und schon findet man auch bei sehr vielen Bauklötzen einen bestimmten Bauklotz. Immer vorausgesetzt, man hat einen beliebig langen Zollstock.

Die Flexibilität der `ArrayList` liegt darin, dass Sie hinten immer wieder neue Elemente anhängen können oder das jeweils letzte Element löschen können. Wenn Sie in der Mitte Elemente einfügen oder löschen, müssen alle Elemente nach dieser Position verschoben werden. Sollten Sie immer wieder aus der Mitte entfernen und löschen müssen, ist die `ArrayList` vermutlich nicht die Idealbesetzung der Rolle. Aber seien Sie sicher, Sie finden noch etwas Passendes im Java Collection Framework.

Die ArrayList als dehnbares Array

Das klassische Array hat den klaren Nachteil, dass der Programmierer vor seiner ersten Benutzung festlegen muss, wie groß es ist. Eine nachträgliche Änderung ist nicht mehr möglich. Diese Begrenzung hebt die `ArrayList` auf.


Immer dann, wenn ein Array benötigt wird, das später größer werden könnte,

ist eine `ArrayList` die bessere Wahl. Wie bei einem Array können die Elemente einer `ArrayList` über einen Index schnell zugegriffen werden. Dieser Index beginnt bei 0 und steigt lückenlos auf.

Als Beispiel für eine `ArrayList` verwenden wir die Boxen in einem Pferdestall. Damit der Hufschmied das richtige Pferd schnell findet, sind die Boxen durchnummeriert. Zunächst definieren Sie eine Klasse `Pferd`, damit der Stall sinnvoll gefüllt werden kann.

```
class Pferd {
    String name;
    int geburtsJahr;
    Pferd(String name, int geboren) {
        this.name = name;
        geburtsJahr = geboren;
    }
    void String toString() {
        return name+": "+geburtsjahr;
    }
}
```

Listing 15.1: Ein Pferd, das die Box voll macht

Um einen Pferdestall mit zwölf Boxen zu modellieren, kann ein Array mit zwölf Plätzen angelegt werden.

```
Pferd []box = new Pferd[12];
```

Das klassische Array belegt den Speicher für zwölf Elemente und ist nicht erweiterbar. Da sich auch Pferde vermehren wie die Karnickel, kann es irgendwann dazu kommen, dass zusätzliche Boxen benötigt werden. In der Praxis müssten Sie anbauen. Bei Arrays sind Sie angeschmiert, aber bei einer `ArrayList` hängen Sie einfach neue Elemente an. Bevor Sie aber die Vorzüge der `ArrayList` nutzen können, müssen Sie sie zunächst importieren.

```
import java.util.ArrayList;
// ...
ArrayList box = new ArrayList();
```

Allerdings haben Sie damit lediglich den Stall angelegt. Es ist noch keine Box vorhanden. Hier verhält sich die `ArrayList` anders als das Array. Die Elemente werden nacheinander durch die Methode `add` hinzugefügt.

```
Pferd fury = new Pferd("Fury", 1955);
box.add(fury));
```

Die Methode `add` erhält als Parameter die Referenz auf das Objekt, das im Datenbehälter gespeichert werden soll. Das Beispiel erzeugt ein neues Pferd, das

die `ArrayList box` um ein Element vergrößert. Das würde also einem Pferdestall entsprechen, in dem die Boxen erst erstellt werden, wenn die Pferde einziehen. Dieses Modell ist tatsächlich sinnvoll, weil man erst dann Speicherplatz belegt, wenn die Daten entstehen.

Wenn Sie 12 Pferdeboxen mit Pferden füllen wollen, könnte das folgende Schleife erledigen:

```
for (int i=0; i<12; i++) {
    box.add(new Pferd("Fury", 1955+i));
}
```

Nun befinden sich 12 Pferde namens Fury im Stall, die in den Jahren nach 1955 geboren wurden. So schnell kann Pferdezucht gehen. Allerdings sollten Sie sich über den Wert Ihres Stalls nicht täuschen. Pferde in diesem Alter haben typischerweise ihren Besuch beim Abdecker bereits hinter sich.

Zugriff analog zum Array

Also sollten Sie einen Blick in den Stall werfen, wer da noch drinsteht. Um ein Element aus der `ArrayList` herauszuholen, verwenden Sie die Methode `get`. Als Parameter erwartet sie den Index der Box. Und wie beim Array beginnt dieser mit 0. Auf die hübschen rechteckigen Klammern, die Sie beim Array vielleicht lieb gewonnen haben, müssen Sie allerdings verzichten.

Darum bietet die `ArrayList` die Methode `get` zum Auslesen und die Methode `set` zum Zuweisen eines Elements an. Hier sehen Sie die syntaktischen Unterschiede im Überblick:

```
element = array[i];          element = arraylist.get(i);
array[i] = element;          arraylist.set(i, element);
```

Listing 15.2: Zugriff auf ein Array (links) und eine `ArrayList` (rechts)

Die Methode `get` erhält den Index als Parameter und liefert das Element als Rückgabewert. Etwas ungewohnter ist die Methode `set`. Hier wird der Index als erster Parameter übergeben und das Element folgt an zweiter Stelle.

Die Methode `size` gibt an, wie viele Boxen angelegt wurden. Wenn Sie also schauen wollen, wer alles in Ihrem Stall steht, können Sie damit die Grenze für Ihre `for`-Schleife bauen.

```
for (int i=0; i<box.size(); i++) {
    System.out.println(box.get(i));
}
```

Eine Kuh im Pferdestall

Der Nachbar ist Bauer und hat eine Kuh, die er gerade nicht unterbringen kann, weil sie von den anderen Kühen gemobbt wird. Also fragt er, ob die Kuh Asyl in

Ihrem Pferdestall bekommt. Da Sie großzügig sind, erlauben Sie es. Und was soll ich sagen? Die `ArrayList` ist genauso großzügig!

```
box.set(5, new Kuh("Berta");
```

Selbst wenn Sie mit der Schleife von oben eine Inventur machen, gibt es keinen Ärger. Das liegt daran, dass `ArrayList` standardmäßig Objekte vom Typ `Object` aufnimmt. Da `Object` die Basisklasse aller Klassen ist, kann man beliebige Objekte in den Container hineinstecken. Beim Ausgeben per `System.out.println` wird automatisch die Methode `toString` aufgerufen, die bereits bei `Object` definiert ist. Also klappt alles. Holen wir aber die Pferde aus dem Stall, müssen wir Farbe bekennen, dass das eingestellte Tier ein Pferd ist. Das erreichen wir durch ein Casting.

```
for (int i=0; i<box.size(); i++) {
    Pferd pferd = (Pferd)box.get(i);
}
```

Das klappt so lange, bis aus der Kuh in Box 5 ein Pferd gemacht werden soll. Nun schlägt das Runtime System Purzelbäume. Das ist nachvollziehbar, weil sich der Hufschmied auch weigern würde, die Kuh zu beschlagen. Und im Springreiten macht sich eine Kuh auch nicht gut.

Typ-Parameter alias Generics

Damit solch ein Mischmasch zwischen Kühen und Pferden nicht passiert, können Sie Typ-Parameter einsetzen, die auch gern *Generics* genannt werden. Dazu wird der `ArrayList` mit spitzen Klammern erzählt, welche Typen sie aufnehmen soll. Wir definieren nun einen Stall, der nur Pferde aufnimmt.

```
ArrayList<Pferd> stall = new ArrayList<Pferd>();
for (int i=0; i<12; i++) {
    stall.add(new Pferd("Fury", 1955+i));
}
for (int i=0; i<stall.size(); i++) {
    Pferd pferd = stall.get(i); // Ohne Casting!
}
stall.set(5, new Kuh("Berta")); // Fehler!!!
```

In der Schleife zum Auslesen der Pferde benötigen wir schon einmal kein Casting mehr. Das ist gut. Und der Versuch, eine Kuh im Pferdestall unterzubringen, stößt auf den Widerstand des Compilers. Das ist noch besser. Ansonsten schleppt noch jemand Katzen oder Nashörner an. Die einen kratzen, die anderen würden die Einrichtung demolieren. Beide würde der Hufschmied aber aus unterschiedlichen Gründen nicht beschlagen wollen.



Der Typ der `ArrayList` wird in spitzen Klammern hinzugefügt. Der hier gezeigte Stall kann Pferde aufnehmen. Es wäre aber auch denkbar, einen Kaninchenstall zu modellieren, indem Sie in die spitzen Klammern den Klassennamen `Kaninchen` schreiben. Wenn Sie keinen Typ angeben, arbeitet die `ArrayList` mit dem Typ `Object`, der Mutter aller Klassen. Dann kann der Compiler allerdings nicht prüfen, ob Sie versehentlich ein Kaninchen in die Box stecken und ein Pferd herausholen.



Wenn Sie die Methode `set` verwenden, wird im Gegensatz zu der Methode `add` kein neues Element hinzugefügt, sondern ein existierendes Element ersetzt. Wenn Sie also bei `set` ein nicht existierendes Element ansprechen, erhalten Sie eine `IndexOutOfBoundsException`. Mit der Methode `size` können Sie jederzeit feststellen, wie viele Elemente in der `ArrayList` vorhanden sind, um diese Exception zu vermeiden.

Einfügen und Löschen von Elementen

Die Box mit der Nummer 3 ist frei geworden. Der Stallbursche, der die Boxen reinigen soll, findet eine leere Box zwischen den Pferden doof. Prinzipiell können Sie Elemente einer `ArrayList` entfernen. Dazu steht Ihnen sogar die Methode `remove` zur Verfügung. Sie akzeptiert als Parameter den Index des zu entfernenden Elements oder das Element selbst.

```
stall.remove(3);
```

Damit wird die Box aus dem Stall entfernt und alle folgenden Boxen werden um eine Nummer aufgerückt. Nun ist zwar der Stallbursche zufrieden, aber die Pferdebesitzer müssen sich neue Nummern merken. Wenn der Hufschmied erst einmal die falschen Pferde beschlagen hat, wird dem Stallburschen wenig Sympathie entgegenschlagen. Sie können sich merken, dass das Löschen eines Elements der `ArrayList` zwar wunderbar funktioniert, aber dennoch nur im Ausnahmefall erfolgen sollte.

Eine neue Box an erster Position

Um das gleiche Chaos anzuzetteln, könnten Sie ein neues Pferd in die erste Box stecken und alle folgenden aufrücken lassen. Dazu müssen Sie der Methode `add` nur als ersten Parameter die gewünschte Box-Nummer hinzufügen. Hier ist es die 0, da das erste Element bei der `ArrayList` die Nummer 0 hat.

```
box.add(0, new Pferd("Lasagne", 2004));
```



Die `ArrayList` kann sehr schnell Elemente über ihren Index auslesen und eintragen. Das Einfügen oder Löschen von Einträgen aus der Mitte verursacht ein Aufrücken aller Folgeelemente und ist nicht so das Ding der `ArrayList`.

Dehnungsfuge: Die Kapazität

Wenn Sie immer wieder nacheinander Daten an die `ArrayList` hängen, muss jedes Mal neuer Platz geschaffen werden. Zu Anfang passiert da nichts, weil die `ArrayList` dafür einen gewissen Platz reserviert hat. Aber irgendwann ist der aufgebraucht. Nun muss eine größere `ArrayList` angelegt, die Daten umkopiert und die alte `ArrayList` der Garbage Collection überantwortet werden.

Dabei muss jedes Mal eine komplett neue `ArrayList` angelegt werden, weil die Elemente ja gern hintereinander liegen sollen. Den Stress tut sich die `ArrayList` nicht für jedes neue Element an. Stattdessen beschafft sie sich einen Freiraum, in dem sie wachsen kann. Diesen Freiraum nennt man Kapazität.

Beim Anlegen der `ArrayList` beschafft sie sich eine Kapazität von zehn Elementen. Sobald das elfte Element hinzukommt, wird wieder eine größere `ArrayList` angelegt, die Daten werden umkopiert und die alte `ArrayList` wird der Garbage Collection übergeben. Immerhin tut dies die `ArrayList` dankenswerterweise von sich aus – ganz ohne die Hilfe des Programmierers. Aber wenn Sie von vornherein schon wissen, dass die `ArrayList` größer wird, seien Sie ein Kumpel und geben Sie der `ArrayList` einen Tipp. Geben Sie ihr die Zielkapazität bei Aufruf des Konstruktors als Parameter mit.

```
ArrayList<Datum> feld = new ArrayList<Datum>(1000);
```

Nun hat die `ArrayList` von vornherein eine Kapazität von 1000. Sollte sich erst nach der Konstruktion herausstellen, dass eine höhere Kapazität erforderlich ist, können Sie mit der Methode `ensureCapacity` nachhelfen. Ihr Parameter legt die neue Kapazität fest.

```
void ensureCapacity(int minCapacity)
```

Sollten Sie irgendwann feststellen, dass eine `ArrayList` eine viel zu große Kapazität besitzt, können Sie diese durch den Aufruf von `trimToSize` auf die aktuelle Größe reduzieren lassen.

```
void trimToSize()
```

Ein altes flexibles Array: Der Vector

Wenn Sie eines Tages Ihr Google ausgelesen haben, werden Sie sich vielleicht beklagen, dass ich Ihnen hier laufend etwas über `ArrayList` erzähle, während das ganze Internet nur vom `Vector` redet.

Bevor die `ArrayList` im Zusammenhang mit dem Java Collection Frame entstand, gab es bereits die Klasse `Vector`, die das gleiche Ziel verfolgte, nämlich ein dynamisches Array zu implementieren.

Damit der `Vector` nicht so einsam ist, wurde er nachträglich integriert, sodass man eine `ArrayList` einfach gegen einen `Vector` im Listing ersetzen kann.



Einen feinen Unterschied gibt es. Die Methoden des `Vectors` sind synchronisiert. Das bedeutet, dass sie in Umgebungen mit mehreren Threads ununterbrechbar laufen. Das kann natürlich auch Konsequenzen hinsichtlich ihrer Laufzeit haben. Da die Definition ununterbrechbarer Abläufe normalerweise sowieso von der Anwendung gemacht werden muss und auch eine `ArrayList` synchronisierbar ist, bringt ein `Vector` in der Praxis keine Vorteile. Aber es ist schon einmal gut, dass man ihn in alten Programmen nicht ersetzen muss.

Den `Vector` finden Sie noch in vielen älteren Quelltexten. In diesen werden auch manchmal Methoden verwendet, die die `ArrayList` nicht anbietet.

Vector	ArrayList
`setElementAt(Element, index)`	`set(index, Element)`
`removeElementAt(index)`	`remove(index)`

Abgesehen vom Namen müssen Sie bei `setElementAt` beachten, dass die Parameter vertauscht sind. Da der `Vector` inzwischen das Interface `List` genau wie die `ArrayList` implementiert, können Sie auch beim `Vector` `set` und `remove` verwenden.

Ein sehr hübsches Beispiel für den Umgang mit Vektoren ist ein Mau-Mau-Kartenspiel unter `www.must.de/default.html?Java-Vector-Beispiel.html`. Der Computer spielt nur gegen sich selbst und gewinnt dadurch natürlich immer.

In Ketten: Die LinkedList

In einer `ArrayList` liegen die Daten wie Bauklötze direkt nebeneinander und sind durchnummeriert. Für den schnellen Zugriff auf die Elemente ist das sehr praktisch. Soll allerdings ein Element dazwischengeschoben oder aus der Mitte herausgenommen werden, muss der gesamte Rest der Liste verschoben werden. Werden Listen also oft geplündert oder nachgefüllt, kann es effizienter sein, eine `LinkedList` einzusetzen.

In einer `LinkedList` sind die Elemente nicht wie in einem Array nebeneinander angeordnet, sondern werden wie in einer Perlenkette aneinandergehängt. Wenn wir also bei unseren Bauklötzen bleiben, wäre es so, als wären diese über Seile und Haken miteinander verbunden.

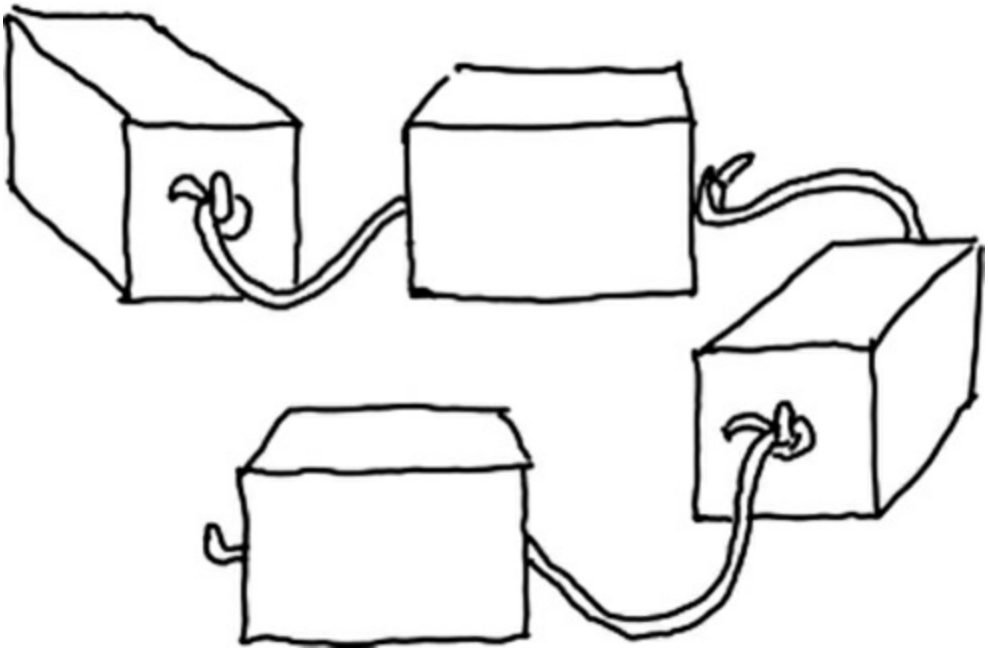


Abbildung 15.2: Die `LinkedList` in Bauklotzform

Wenn Sie nun direkt auf den 960sten Klotz zugreifen wollen, wird das eine langwierige Angelegenheit. Wenn Sie aber eh Schritt für Schritt durch eine Liste gehen und dann an einer Stelle ein Element entfernen wollen, nehmen Sie es nur vom Haken des Vorgängers und verbinden diesen mit dem Nachfolger. Das entfernte Element überlassen Sie getrost den fleißigen Leuten von der Müllabfuhr. Die ganze Arbeit des Umknüpfens wird dann auch noch von der `LinkedList` übernommen. Den Programmierer interessieren die Details nicht.

Aber das ist nicht die ganze Wahrheit. Eigentlich sind die Elemente einer `LinkedList` doppelt verkettet. Doppelt heißt in diesem Zusammenhang, dass das Programm sich auf dieser Kette effizient vor- und zurückhangeln kann.

Das Anlegen einer `LinkedList` birgt gegenüber dem Verfahren bei der `ArrayList` keine Überraschungen.

```
import java.util.List;
import java.util.LinkedList;

LinkedList<Element> liste = new LinkedList<>();
```

Listing 15.3: Anlegen einer `LinkedList`

Iterieren über Datenbehälter

Das Durchlaufen einer `LinkedList` ähnelt dem Wechsel der Perlen bei einem Rosenkranz. Den Perlenwechsel vollführt der geübte Katholik mit seinen Fingern. Die LinkedList bemüht dazu typischerweise den Iterator. Falls Sie bisher nur den Imperator aus Star Wars kennen, werden Sie feststellen, dass ein Iterator viel netter ist. Er ist wie ein Zeigestock, der von Bauklotz zu Bauklotz geht. Über ihn greift man auf die Daten zu, löscht Elemente und fügt neue hinzu.

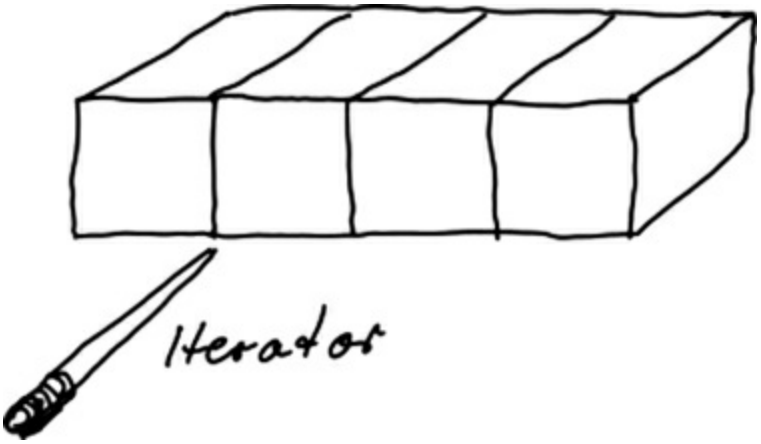


Abbildung 15.3: Iteratoren zeigen auf Container-Elemente.

Die Bauklötze im Bild stellen offensichtlich eine `ArrayList` dar. Tatsächlich können Sie auch für die `ArrayList` einen Iterator verwenden. Allerdings ist er dort von geringerem Nutzen. Für die `LinkedList` ist der Iterator dagegen der typische Weg für das Durchlaufen seiner Elemente.

1. Im ersten Schritt holen Sie sich vom Container einen Iterator über den Aufruf der Methode `iterator`. Der Iterator steht zunächst vor dem ersten Element.
2. Der Iterator kann über seine Methode `hasNext` feststellen, ob sein Container noch Elemente enthält.
3. Über die Methode `next` liefert der Iterator das nächste Element und bewegt sich fort.

Eine Schleife zum Auslesen eines Containers sieht also so aus:

```
import java.util.List;
import java.util.LinkedList;
import java.util.ListIterator;
import java.util.Iterator;

LinkedList<Element> liste = new LinkedList<>();
Iterator<Element> iterator = liste.iterator();
while (iterator.hasNext()) {
    Element element = iterator.next();
    // Das Element kann bearbeitet werden.
}
```

Listing 15.4: Etwas Bewegung für den Iterator



Dieses Verfahren funktioniert mehr oder weniger direkt mit jedem Datenbehälter des Java Collection Frames. Um genau zu sein, betrifft es alle Implementierungen des Interface `Collection`. Das Interface `List`, das von `ArrayList` und `LinkedList` implementiert wird, kennt allerdings noch einen

spezialisierten `ListIterator`. Während der normale Iterator nur das Vorwärtsbewegen, Auslesen und Löschen einzelner Elemente erlaubt, kann sich ein `ListIterator` durch die zusätzlichen Methoden `hasPrevious` und `previous` in beide Richtungen bewegen.

Die Methoden `nextIndex` und `previousIndex` liefern den Index des Elements. Das folgende Beispiel zeigt, wie eine Liste zunächst vorwärts und dann wieder rückwärts durchlaufen wird.

```
ListIterator<Element> iterator = list.listIterator();
while (iterator.hasNext()) {
    Element element = iterator.next();
}
while (iterator.hasPrevious()) {
    Element element = iterator.previous();
}
```

Listing 15.5: Der Iterator läuft vor und zurück.

Probleme mit dem Rückwärtsgang

Falls Sie ein Programm im Stil von Listing 15.5 schreiben, funktioniert das Vor- und Zurücklaufen super. Falls Sie aber nur das Rückwärtslaufen versuchen, wird die Freude nur kurz andauern, weil der Start-Iterator nach seiner Erzeugung vor dem ersten Element steht. Wenn Sie dann rückwärtslaufen, ist da eben nichts.

Sie könnten natürlich einmal durch die Liste vorwärtslaufen, um sie dann rückwärts auszugeben. Eine nette Übung, die Sie gern programmieren können. Aber eigentlich wäre es schick, wenn man den `ListIterator` irgendwie hinter das letzte Element setzen könnte. Und tatsächlich können Sie dem Konstruktor des `ListIterators` als Parameter übergeben, an welcher Stelle er in der Liste stehen soll.

```
ListIterator<Element> iterator
        = liste.listIterator(liste.size());
```

Listing 15.6: Die Startposition als Konstruktorparameter

Darüber hinaus kennt die `LinkedList` einen `descendingIterator`. Raten Sie mal, was der tut! Descending heißt Herabsteigen, sozusagen ein Moses-Iterator, der vom Berg Sinai herabsteigt. Das ist natürlich Quatsch. Es ist eher zu vermuten, dass dieser Iterator rückwärtsläuft.

Der `descendingIterator` kennt allerdings nur die Methode `next`, kein `previous`. Da der Iterator aber absteigt, ist das »next« für ihn rückwärts.

```
Iterator<Element> iterator = liste.descendingIterator();
while (iterator.hasNext()) {
    Element k = iterator.next();
    // Mache etwas mit dem Element
}
```

Listing 15.7: Endlich geht es rückwärts!



Mit der Methode `previous` werden einzelne Schritte zurückgegangen. Um eine Liste rückwärts zu durchlaufen, verwendet man den durch die Methode `descendingIterator` gelieferten Iterator.

Veränderung kann tödlich sein

Über den Iterator können Sie Elemente einfügen oder löschen. Dazu gibt es die Methoden `add` und `remove`. Allerdings erwarten diese Varianten keinen Index als Parameter, da der Iterator ja weiß, wo er steht. Die Methode `add` benötigt natürlich das Element, das eingefügt werden soll, als Parameter.

Wenn Sie mit einem Iterator auf einer Liste arbeiten, achten Sie darauf, dass die Liste nur über diesen Iterator verändert wird. Wird die Liste am Iterator vorbei direkt verändert, ist das Runtime Environment so irritiert, dass es eine ganz hässliche `ConcurrentModificationException` von sich wirft. Löschen und verändern Sie die Liste also nur über den Iterator! Das gibt keinen Ärger, weil der Iterator diese Löschung ja mitbekommt. Gefährlich ist es, hinter dem Rücken des Iterators den Container zu ändern, über den er streicht.



Concurrent heißt eigentlich »nebenläufig« und darum wird dieser Fehler oft mit Threads in Verbindung gebracht. Dabei hat es damit nichts zu tun. Java geht davon aus, dass Sie den Container nicht verändern, solange Sie ihn iterieren.



Ändern Sie nie den Container hinter dem Rücken seines Iterators, sondern nur über den Iterator. Wird der Container direkt verändert, werden sofort alle Iteratoren ungültig und müssen neu angelegt werden.

Sollte es doch unabwendbar sein, den Container direkt oder über einen anderen Iterator zu löschen, dann holen Sie sich einen neuen Iterator.

Die LinkedList als Queue

Warteschlangen sind nicht nur im Supermarkt und an Kinokassen beliebt. Auch bei der Datenverarbeitung wird diese Form menschlichen Miteinanders gern als Vorbild verwendet. Und damit nicht jeder Programmierer seine eigene Warteschlange programmieren muss, können Sie sie im Collection Framework kostenlos erhalten. Stellen Sie sich bitte hinten an. Danke!

Die Informatiker nennen eine Warteschlange einen FIFO-Speicher. *FIFO* steht für First In First Out, was übersetzt etwa bedeutet: »Wer zuerst kommt, mahlt zuerst.« Das Java Collection Framework bietet einen FIFO als `Queue` an.

Wenn wir bei den Holzklötzen als Modell bleiben, müssten wir einen Kasten zu Hilfe nehmen, in den man oben die Bauklötze hineinwerfen kann (add) und unten immer einen herausziehen kann (poll). Hat man den Bauklotz entnommen, rutschen alle anderen automatisch nach.

Die `Queue` bietet noch die Möglichkeit zu luschern[1] . Damit der Bauklotz zur Untersuchung nicht unbedingt entnommen werden muss, besitzt der Kasten eine kleine Klappe, in der man sehen kann, was der untere Bauklotz enthält (peek).

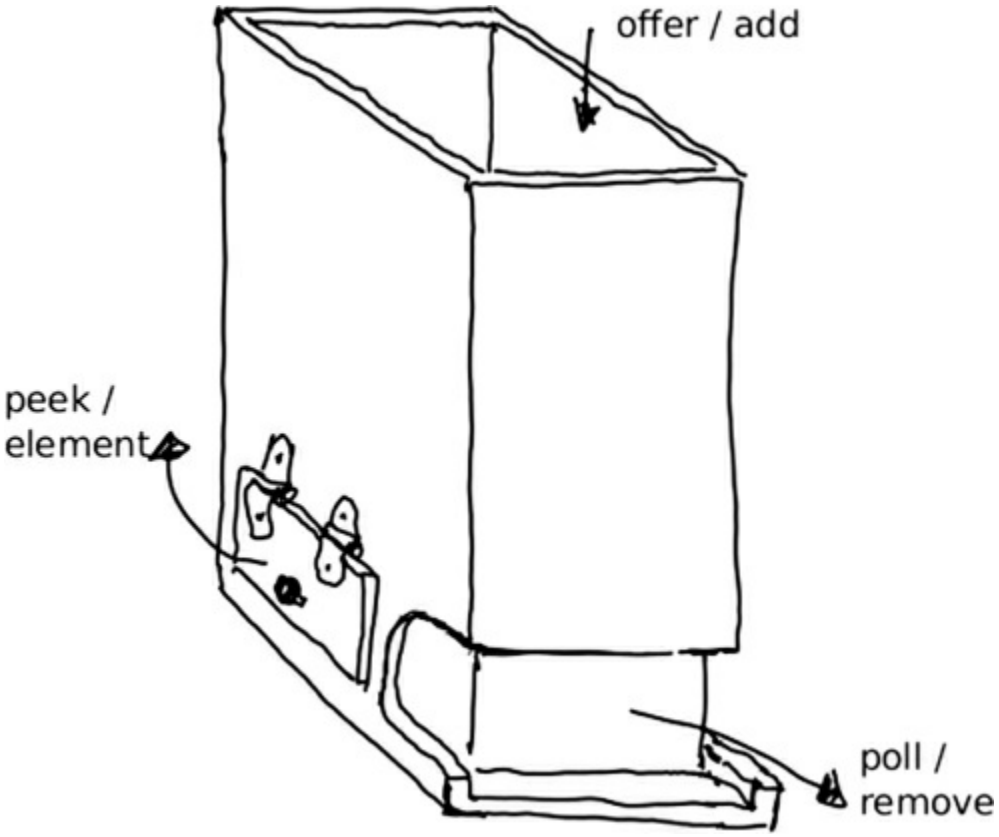


Abbildung 15.4: Möglichkeiten einer Queue



Eine `Queue` ist ein Interface, also keine Klasse. Für die Implementierung wird wieder die `LinkedList` bemüht, die neben dem Interface `List` eben auch `Queue` implementiert.

Es gibt zwei Sätze von Methoden für die Operationen einer Queue. Sie unterscheiden sich darin, wie der Erfolg der Operationen an den Aufrufer gemeldet werden.

Erfolgsmeldung per Rückgabewert

Bei den Methoden `offer` für das Einfügen, `peek` für das Lesen des ersten Elements und `poll` für das Entfernen wird der Erfolg der Operationen über die

Rückgabewerte gelöst.

Listing 15.8 füllt drei Strings in die Queue, schaut mal nach dem ersten und holt dann alle Strings wieder ab. Die Ausgabeschleife endet, wenn `null` zurückgegeben wird.

```
import java.util.LinkedList;
import java.util.Queue;
public class TestQueue {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<String>();
        // Schlange mit ein paar Werten füllen
        queue.offer("Alpha");
        queue.offer("Beta");
        queue.offer("Charlie");
        // Anschauen, ohne zu entfernen
        String liesdas = queue.peek();
        System.out.println("Mal luschern: " + liesdas);
        // Alles wieder herausholen
        while ((liesdas = queue.poll()) != null) {
            System.out.println("Ausgelesen: " + liesdas);
        }
    }
}
```

Listing 15.8: Wir testen keine Kuh, sondern eine Queue.

Dasselbe mit Exceptions

Die gleichen Funktionalitäten gibt es noch einmal unter anderen Namen. Der Unterschied liegt in der Art der Fehlerbehandlung. Bei Verwendung der Methoden `add` für das Einfügen, `element` für das Anschauen und `remove` für das Herausholen löst ein Fehler eine Exception aus. Ansonsten verhalten sie sich weitgehend wie `offer`, `peek` und `poll`.

Schauen wir uns den gleichen Ablauf in der Version mit Exceptions an:

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.NoSuchElementException;

public class TestExceptionQueue {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<String>();
        try {
            queue.add("Alpha");
            queue.add("Beta");
            queue.add("Charlie");
            String liesdas = queue.element();
            System.out.println("Mal luschern: " + liesdas);
            // Die Schleife endet durch die Exception.
            while (true) {
                System.out.println("Und abholen: " + queue.remove());
            }
        } catch(NoSuchElementException e) {
            System.out.println("gefangen! ");
        }
        System.out.println("fertig! ");
    }
}
```

Listing 15.9: Queue mit Exceptions

Auf dem Bildschirm erscheinen die folgenden Zeilen als Ergebnis:

```
Mal luschern: Alpha
Und abholen: Alpha
Und abholen: Beta
Und abholen: Charlie
gefangen!
fertig!
```

Zum Abschluss zeigt Tabelle 15.1 noch einmal, welche Methoden per Rückgabe und welche per Exception ihre Fehlerbehandlung durchführen.

Funktion	Rückgabe	Exception
Einfügen	offer	add
Abholen	poll	remove
Spicken	peek	element

Tabelle 15.1: Die Methoden der Queue

LinkedList als Stapel

Ein Stapel entsteht, wenn man gelesene Zeitungen übereinanderlegt. Diese Form der Ablage ist nicht nur einfach, sondern durchaus praktisch. Wenn sich der Leser einen Artikel noch einmal zu Gemüte führen möchte, wird dieser wohl eher in einer

neueren Ausgabe stehen. An die älteren Artikel wird er sich wohl nicht mehr so gut erinnern.

Auch wenn Sie umziehen, werden Sie Ihre Umzugskartons nach dem Prinzip eines Stapels organisieren. Sie werden ein paar Monate vorher das einpacken, was Sie sehr selten benötigen. Dagegen werden die Lebensmittel zuletzt eingepackt, weil Sie diese nach dem Umzug gleich wieder brauchen und als Erstes auspacken. Die zuerst eingepackten Kartons werden Sie zuletzt auspacken, weil Sie diese Dinge ja nur selten benötigen.


Dieses Verfahren nennen die Informatiker *LIFO* als Abkürzung für »Last In First Out«, was soviel bedeutet wie: »Die Ersten werden die Letzten sein.«

Für Informatiker ist der Stack eine wichtige Struktur. Der Stack taucht immer wieder auf, wenn verschachtelte Elemente verarbeitet werden sollen. Wenn ein Programm beispielsweise eine mathematische Formel auswerten will, wird es für jede öffnende Klammer einen neuen Eintrag auf den Stapel legen. Tritt eine schließende Klammer auf, wird der Eintrag ausgewertet und vom Stapel genommen. Ähnliches gilt, wenn Methoden Methoden aufrufen. Auch die Aufrufe werden auf einen Stapel gelegt, damit das Runtime-System weiß, wo der Aufruf herkommt. Diesen Stapel sehen Sie bei Exceptions als »Stack« angezeigt.

Der Stack zeichet sich durch die Operationen `push` für das Einfügen und `pop` für das Entnehmen von Elementen aus. Als dritte Operation steht noch `peek` zur Verfügung, wenn man mal schauen möchte, was ein `pop` so bringen würde. Diese drei Methoden werden durch das Interface `Deque` angeboten, das `LinkedList` implementiert.

```
import java.util.LinkedList;
import java.util.NoSuchElementException;

public class DequeStack {

    public DequeStack() {
        LinkedList<String> stack = new LinkedList<>();
        stack.push("Alpha");
        stack.push("Beta");
        stack.push("Charlie");
        try {
            while (true) {
                System.out.println(stack.pop());
            }
        } catch(NoSuchElementException e) {
        }
    }

    public static void main(String[] args) {
        new DequeStack();
    }
}
```

Listing 15.10: Der Stack auf Basis von Deque



Auf Ihren Besuchen im Internet werden Sie vielleicht auf einen Datenbehälter namens `Stack` stoßen, der die gleichen Methoden zur Verfügung stellt. Er basiert auf dem `Vector`. Oracle empfielt allerdings eine `LinkedList`. Falls Sie dennoch den Typ `Stack` verwenden, sollten Sie darauf achten, dass die entsprechende Exception dort `EmptyStackException` heißt.

Sammlung der Einzigartigen: Set

Ein `Set` enthält eine Menge von Elementen, von denen keines doppelt auftreten darf. Damit entspricht ein `Set` dem Mengenbegriff in der Mathematik. Oder wenn Sie die Mathematik nicht so lieben, dann verhält sich ein `Set` wie die Lottozahlen. Es ist nicht entscheidend, in welcher Reihenfolge die Zahlen gezogen wurden, und es gibt nur ein Exemplar jeder Zahl. In einem `Set` kann ein Objekt nur einmal vertreten sein.

`Set` ist ein Interface, das nur die Vorgaben macht, was eine ordentliche Menge so können muss. Für den Einsatz in einem Programm benötigen Sie eine

Implementierung von einem `Set`, um damit konkret arbeiten zu können. Im Angebot haben wir folgende Kandidaten:

- Garantiert aus geordnetem, biologischem Anbau:

  `TreeSet`, das naturverbundene Modell, liefert geordnete Listen. Er fordert von seinen Elementen, dass sie vergleichbar sind.
- Heute für Sie frisch eingetroffen:

  `HashSet` ist unser Modell für beste Performance, liefert aber keine Gewähr bezüglich der Reihenfolge. Seine Elemente müssen sich auf Hash verstehen.
- Wenn es individuell und gepflegt sein soll:

  Wie immer Sie Ihre Daten einfügen: `LinkedHashSet` merkt sich ihre Reihenfolge, fordert aber auch Elemente mit dem gewissen Hash.

Für den ersten Anlauf werden wir Mengen von Strings betrachten und dort sind die Anforderungen der verschiedenen Set-Implementierungen gegeben. Aber behalten Sie die Forderungen im Hinterkopf. Ich komme darauf noch zurück.

Ein Wort zu der Kapazität des Sets. Wenn Sie abschätzen können, wie groß die Menge wird, können Sie die Anzahl der erwarteten Elemente als Parameter des Konstruktors angeben. Der Vorgabewert für die Kapazität der Menge ist 16. Die folgende Definition legt ein Set `meinSet` an, das durch ein `HashSet` implementiert wird und eine Kapazität von 64 Elementen hat.

```
Set<String> meinSet = new HashSet<String>(64);
```

Hinter dem `new` können Sie ein `HashSet`, ein `TreeSet` oder ein `LinkedHashSet` angeben. Der Umgang mit dem Set wird sich nicht ändern, sondern nur die Art, wie das Set seine Elemente pflegt. Das wiederum kann für die Elemente Konsequenzen haben. Solange wir uns auf Strings oder Zahlen beschränken, ist das allerdings kein Problem.

Zahlen? Das sind doch keine Klassen! Ja, das stimmt. Aber mithilfe eines Wraps bekommen wir auch primitive Zahlen in die Datenbehälter.

Die Mengen im Vergleich

Lassen Sie uns mit den drei Set-Implementierungen ein wenig herumspielen. Dazu schreiben wir ein Programm, das die Wörter eines einfachen Satzes in einem Set ablegt. Dabei werden doppelte Wörter entfallen, weil ein Set jedes Element ja nur einmal enthält.

In Listing 15.11 sehen Sie, dass hinter der Deklaration von `HashSet` zwei Zeilen auskommentiert sind. Durch Änderung der Kommentarzeichen können Sie so nach und nach `HashSet`, `TreeSet` und `LinkedHashSet` ausprobieren. Nacheinander werden die folgenden Wörter dem Set zugefügt: das buch das das java beschreibt.

Rechtschreibfehler sind hier wie überall im Buch volle Absicht und Stilmittel.

```
import java.util.*;
public class EntDoppler {
    public static void main(String[] args) {
        // Wir legen die Set-Implementierung fest.
        Set<String> s = new HashSet<String>();
        // Set<String> s = new TreeSet<String>();
        // Set<String> s = new LinkedHashSet<String>();
        int doppelte = 0;
        // Der Fragezeichenoperator hilft, Dopplungen zu zählen.
        doppelte += s.add("das")?0:1;
        doppelte += s.add("buch")?0:1;
        doppelte += s.add("das")?0:1;
        doppelte += s.add("das")?0:1;
        doppelte += s.add("java")?0:1;
        doppelte += s.add("beschreibt")?0:1;
        System.out.println("Doppelte: " + doppelte);
        System.out.println("Anzahl Einzelne: " + s.size()
                           + " und zwar "  + s);
    }
}
```

Listing 15.11: Wir entdoppeln.

Diese Experimente habe ich einmal für Sie durchgeführt. Alle Set-Varianten sind sich einig, dass es zwei Doppelte gibt. Und auch die Anzahl der gefundenen Wörter ist gleich. Aber die Reihenfolge der Wörter ist unterschiedlich.

- Die Ausgabe von `Hashset` liefert diese Zeilen:
  ```
  Doppelte: 2
  Anzahl Einzelne: 4 und zwar [beschreibt, das, buch, java]
  ```
  Die Reihenfolge, die `HashSet` liefert, erscheint völlig chaotisch.
- `TreeSet` kommt zu dem folgenden Ergebnis:
  ```
  Doppelte: 2
  Anzahl Einzelne: 4 und zwar [beschreibt, buch, das, java]
  ```
  `TreeSet` liefert eine alphabetisch sortierte Reihenfolge.
- Das Programm liefert mit `LinkedHashSet`:
  ```
  Doppelte: 2
  Anzahl Einzelne: 4 und zwar [das, buch, java, beschreibt]
  ```
  `LinkedHashSet` behält die Reihenfolge der Einfügung bei.

Komplexere Elemente

Lassen Sie uns mal mit einem etwas komplexeren Beispiel herumspielen. Statt der Strings verwenden wir einen Landkreis. Den brauchen wie eh bald für den assoziativen Speicher `Map`. Der Landkreis besteht erst einmal aus dem Namen des Kreises, den wir einfach `wo` nennen. Als weiterer Wert interessiert uns die Einwohnerzahl.

```
public class Kreis {
    protected String wo;
    protected long einwohner;

    Kreis(String w, long e) {
        wo = w; einwohner = e;
    }

    String getWo() {
        return wo;
    }

    long getEinwohner() {
        return einwohner;
    }
}
```

Listing 15.12: `Kreis` ist die Datenstruktur zum Mengenabfüllen.

Es kann jeden Landkreis nur einmal geben. Dabei identifizieren wir die Einzigartigkeit des Landkreises an seinem Namen. Der Landkreis Pinneberg darf nur einmal auftauchen. Die Einwohnerzahl interessiert da nur sekundär.

TreeSet und Comparable

Wenn Sie Zettel mit Namen hätten und sollten die doppelten ausfiltern, würden Sie vermutlich die Zettel erst alphabetisch sortieren. Danach stehen die doppelten direkt nebeneinander und das Aussortieren ist leicht. Genau diesen Weg schlägt auch `TreeSet` ein.

Ob die Menge arbeitet, testen wir, indem wir einen Kreis doppelt eintragen. Er sollte in der Liste später nicht mehr auftauchen.

```
import java.util.Set;
import java.util.TreeSet;

public class TreeKreise {

    public TreeKreise() {
        Set<Kreis> kreise = new TreeSet<Kreis>();
        kreise.add(new Kreis("Schleswig-Flensburg",197903));
        kreise.add(new Kreis("Schleswig-Flensburg",197903));
        kreise.add(new Kreis("Recklinghausen",628817));
        for (Kreis k : kreise) {
            System.out.print("" + k.getWo());
            System.out.print(" (" + k.getEinwohner());
            System.out.println(" Einwohner)");
        }
    }
   public static void main(String args[]) {
        new TreeKreise();
    }
}
```

Listing 15.13: Kreise durch `TreeSet` verwaltet

Sie übersetzen die Klassen aus Listing 15.12 und Listing 15.13. Alles klappt, aber bei der Ausführung werden Sie auf eine Exception stoßen, die Ihnen erzählt, dass der Kreis das Interface `Comparable` nicht implementiert.

`TreeSet` fordert von allen seinen Elementen, dass sie vergleichbar sind. Wie sollten sie sonst sortiert werden können? Dies erreicht `TreeSet`, indem es von den Elementen fordert, dass sie `Comparable` implementieren. Das hat dann zur Folge, dass jedes Element die Methode `compareTo` zur Verfügung stellen muss.

Strings sind immer vergleichbar und da der Name des Landkreises der String ist, nach dem sortiert werden soll, müssen Sie nur erreichen, dass der Kreis diesen als Vergleich heranzieht. Dazu machen Sie ein paar kleine Änderungen an Listing 15.12.

```
public class Kreis implements Comparable {
    ...
    @Override
    public int compareTo(Object o) {
        Kreis p = (Kreis)o;
        return wo.compareTo(p.wo);
    }
}
```

Listing 15.14: Der Kreis wird vergleichbar.

Anschließend sollte das Programm erwartungsgemäß funktionieren.

HashSet und LinkedHashSet

Wenn Sie die Kreise in einem `HashSet` organisieren, wird `Comparable` nicht benötigt, da die Kreise hier nicht sortiert werden.

```
import java.util.Set;
import java.util.HashSet;

public class HashKreise {

    public HashKreise() {
        Set<Kreis> kreise = new HashSet<Kreis>();
        kreise.add(new Kreis("Schleswig-Flensburg",197903));
        kreise.add(new Kreis("Schleswig-Flensburg",197903));
        kreise.add(new Kreis("Recklinghausen",628817));
        for (Kreis k : kreise) {
            System.out.print("" + k.getWo());
            System.out.print(" (" + k.getEinwohner());
            System.out.println(" Einwohner)");
        }
    }
   public static void main(String args[]) {
        new HashKreise();
    }
}
```

Listing 15.15: Kreise durch Hash verwaltet

Nun wäre es natürlich interessant, was bei doppelten Einträgen passiert. Und? Sie werden überrascht feststellen, dass sich das blöde `HashSet` überhaupt nicht dafür interessiert. Dieses dummdreiste Ding nimmt ja doch Doppelte auf! Da hat der Willemer mal wieder einen sagenhaften Unfug erzählt!

Gleichheit heißt equals

Offensichtlich erkennt das Set die gleichen Kreise nicht als gleich! Da Gleichheit bei Objekten durch die Methode `equals` ermittelt wird, sollte das in fünf Minuten erledigt sein. Wir bauen einfach die Methode in die Klasse Kreis ein.

```
@Override
public boolean equals(Object p1) {
    Kreis p = (Kreis)p1; // Übles Casting unvermeidbar
    return wo.equals(p.wo) && einwohner==p.einwohner;
}
```

Listing 15.16: Die Klasse `Kreis` bekommt eine Methode `equals`.

Der Parameter der Methode muss ein `Object` sein und nicht `Kreis`, sonst überschreibt die Methode nicht. Die Basisklasse ist ja schließlich `Object` und die vergleicht eben mit `Object`.

Nun ist `equals` implementiert und jetzt müssten die gleichen Elemente verschwinden. Mal testen! Nein, es geht immer noch nicht. Die Methode `equals` arbeitet aber.

Die Gleichheit zweier Objekte stellen `HashSet` und `HashLinkedSet` über einen Hash fest. Daher auch der Name. Was aber ist ein Hash?

Hash ist toll

Hash ist nicht das, was Sie denken. Es ist auch nicht verboten. Jedenfalls nicht bei Programmierern. Und das Zeug, was Sie meinen, heißt auf Englisch Hush, dessen Aussprache dem deutschen Hasch entspricht. Erzählen Sie aber jetzt bitte nicht überall herum, dass man in Willemers Büchern die korrekte Aussprache von Drogen lernt. Sie hatten doch den Gedanken mit dem Hush!

Ein Hash, der übrigens »Häsch« ausgesprochen wird, dient dazu, Informationen so zu organisieren, dass man sie möglichst schnell wiederfindet. Die meisten Menschen sortieren die Daten, auf die sie schnell zugreifen wollen. `TreeSet` macht es ähnlich.

Das Hashverfahren geht einen anderen Weg. Dazu wird über die Daten mithilfe einer Hashfunktion ein Index gebildet, der für die Positionierung der Daten verwendet wird. Erstellen wir ein extrem vereinfachtes Beispiel: Sie wollen etwa 50 Städtenamen speichern und schnell wiederfinden.

- Sie legen ein Array von 100 Positionen an.
- Nun berechnen Sie aus den Namen einen Index, der im Bereich von 0 bis 99 liegt. Der Einfachheit halber verwenden Sie die Quersumme der Buchstabenwerte, auch wenn das vielleicht nicht gerade eine optimale Hashfunktion bildet.

- ✔ Geht die Quersumme über die Dimension des Arrays hinaus, verwenden Sie den Modulo-Wert, also den Restwert.

Nehmen wir an, wir würden Berlin ablegen wollen: B ist der zweite Buchstabe im Alphabet, also wird B mit 2 gleichgesetzt. E ist 5, R ist 18, L ist 12, I ist 9, N ist 14. Damit erhält Berlin die Quersumme 60 und die Referenz auf Berlin wird an der Position 60 in einem Pool für Referenzen von Städtenamen abgelegt. Sucht jemand Berlin, rechnet er die Quersumme und hat mit einem Schlag die 60 und damit die Daten.

Wenn die 60 bereits belegt ist, wird eine Liste für alle 60er angelegt und die Städte werden sequenziell angehängt. Wenn nun jemand Berlin sucht, greift er auf 60 zu. Nun prüft er, ob der Eintrag tatsächlich Berlin ist. Stellt er fest, dass da eine andere Stadt steht, schaut er die Liste durch und findet vermutlich in kurzer Zeit Berlin. Das ist meist immer noch schneller als das Suchen in sortierten Feldern, zumal man das Sortieren spart.

Eine ideale Hashfunktion bringt eine gleichmäßige Verteilung über den Raum und bewirkt so möglichst wenige Kollisionen. `HashSet` greift für das Einsortieren der Elemente auf die Methode `hashCode` zurück. Alle Klassen bieten diese Methode, weil `Object` als Basis aller Klassen diese enthält. Die Methode liefert einen Hashcode, der auf der Position des Objekts im Speicher basiert. Daraus ergibt sich ganz klar, dass zwei Objekte nur dann den gleichen Hashcode besitzen, wenn es sich um dasselbe, identische Objekt handelt,

Wenn Ihre Klasse allerdings die Methode `equals` implementiert, weil es mehrere Objekte geben kann, die gleich sind, ist es auch sinnvoll, dass gleiche Objekte den gleichen Hashcode liefern. Und nun wird deutlich, dass es eine enge Freundschaft zwischen `hashCode` und `equals` gibt. Denn zur Feststellung, ob die Stadt mit dem `hashCode` 60 tatsächlich Berlin ist, wird `equals` verwendet.

Wir hashen den Kreis

Woher bekommen Sie auf die Schnelle einen eindeutigen Hashwert? Natürlich! Der Name des Kreises ist ja eindeutig und über den String können Sie `hashCode` aufrufen und das Ergebnis einfach durchreichen.

```
@Override
public int hashCode() {
    return wo.hashCode();
}
```

Listing 15.17: Und `hashCode` muss auch überschrieben werden.

Enthaltene Elemente

Ein Landkreis besteht oft aus mehreren Städten und Dörfern, die unterschiedliche Postleitzahlen haben. Wenn Sie diese sammeln wollen, bietet sich dafür wieder ein

Set an. Da wir die Klasse `Kreis` erweitern, können wir dazu auch eine neue Klasse von `Kreis` ableiten, die `PlzKreis` heißen soll.

```
import java.util.HashSet;
public class PlzKreis extends Kreis {

    protected HashSet<String> plzset = new HashSet<String>();

    PlzKreis(String w, long e) {
        super(w, e);
    }

    public void addPlz(String plz) {
        plzset.add(plz);
    }

    public boolean enthaeltPlz(String plz) {
        return plzset.contains(plz);
    }

    // Hauptprogramm zum Testen der Klasse
    public static void main(String args[]) {
        PlzKreis plzKreis = new PlzKreis("Recklinghausen",628817);
        plzKreis.addPlz("45601");
        plzKreis.addPlz("45665");
        System.out.println(plzKreis.enthaeltPlz("45666"));
    }
}
```

Listing 15.18: Erweiterung von `Kreis` (Listing 15.12)

Nun weiß jeder Kreis, welche Postleitzahlen er hat. Es tauchen auch keine doppelten Einträge auf, egal, wie oft sie eingefügt werden. Wenn Sie wissen wollen, ob eine bestimmte Postleitzahl im Landkreis enthalten ist, verwenden Sie die Methode `contains`.

Aber Mengen wären nicht Mengen, wenn sie nicht in Mengen auftreten würden.

- ✔ Darum können Sie eine Menge mit der Methode `addAll` einfügen.
- ✔ Über die Methode `containsAll` können Sie feststellen, ob eine Menge ein Teil einer Menge ist.
- ✔ Mit der Methode `retainAll` können Sie eine Menge mit einer anderen schneiden und erhalten zur Überraschung aller die Schnittmenge.

Listing 15.19 zeigt, wie es geht.

```
import java.util.HashSet;

public class SetPlz {

    SetPlz() {
        HashSet<String> plzSet = new HashSet<String>();
        plzSet.add("45601");
        plzSet.add("45603");
        plzSet.add("45605");
        HashSet<String> mehr   = new HashSet<String>();
        mehr.add("45602");
        mehr.add("45603");
        mehr.add("45604");
        plzSet.addAll(mehr);
        System.out.println(plzSet.containsAll(mehr));
        mehr.remove("45603");
        System.out.println(plzSet.retainAll(mehr));

        for (String str : plzSet) {
            System.out.println(str);
        }
    }

    // Hauptprogramm zum Testen der Klasse
    public static void main(String args[]) {
        new SetPlz();
    }
}
```

Listing 15.19: Mengen mengenweise verknüpft

Zugriff per Schlüssel: Der assoziative Speicher

Auf ein Array-Element greifen Sie über dessen Nummer zu. Das wird erst dann schwierig, wenn Sie dafür eine Artikelnummer verwenden, die eine sechsstellige Zahl darstellt, obwohl Sie nur ein paar Hundert Artikel haben. Das gibt auch dann einen löchrigen Speicher, wenn Sie gar nicht mit Käse handeln.

Hier hilft eine Map. Mit einer Map können Sie einen beliebigen Schlüssel verwenden und die Map sieht zu, dass kein Speicher verschwendet wird. Sie sind

beim Schlüssel nicht einmal auf Zahlen beschränkt. Auch Strings funktionieren wunderbar.


Eine Map ermöglicht den Zugriff auf die Daten über einen Schlüssel. Man spricht von einem assoziativen Speicher.

Die Eigenheiten einer Map sind mit den Bauklötzen nicht ganz so einfach darstellbar. Abbildung 15.5 versucht es wenigstens. Die Map hat für jeden Eintrag immer zwei Bauklötze. Der vordere Bauklotz enthält den Schlüssel. Über eine Schnur hängt der zugehörige Datenbauklotz am Schlüssel. Der Datenbauklotz hat weder Öse noch Haken für weitere Anhänger. Die anderen Datenelemente des Containers sind also nur über deren Schlüssel erreichbar.

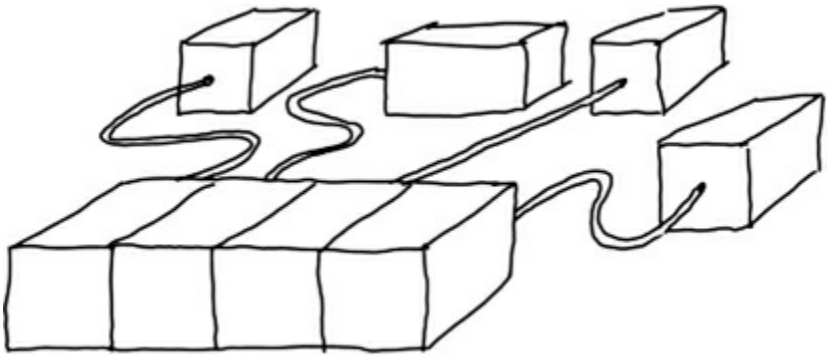


Abbildung 15.5: Die Map in Bauklotzform

Für die Demonstration der Map verwende ich wieder das Beispiel der Landkreise. Als Schlüssel drängt sich das Kfz-Kennzeichen geradezu auf, immerhin ist es ein eindeutiger Schlüssel für jeden Landkreis.

Jedes Element der Map hat also einen Schlüssel und einen Datensatz, der den Schlüssel selbst nicht enthalten muss. Darum können Sie den Kreis aus Listing 15.12 direkt wiederverwenden.

Elemente einfügen, abfragen und löschen

Als Implementierung von `Map` verwenden wir eine `HashMap`. Sie besitzt zwei Generics: die erste für den Schlüssel und die zweite für die Daten.

```
import java.util.HashMap;
// ...
HashMap<String, Kreis> kfzKreis = new HashMap<String, Kreis>();
```


Sie müssen nun ganz tapfer sein. `Map` ist kein Nachfahre von `List` und verwendet darum auch andere Methoden. Alles ist also etwas anders.

`Map` stellt Ihnen folgende Methoden zur Verfügung:

- **Elemente eintragen:** Die Methode zum Einfügen heißt `put`. Sie benötigt den

Schlüssel als ersten Parameter und das Element als zweiten. Wenn es für den Schlüssel bisher keinen Eintrag gab, wird er angelegt. Sollte es den Schlüssel schon geben, wird der Datenteil durch den zweiten Parameter ersetzt.

```
kfzKreis.put("SL", new Kreis("Schleswig-Flensburg",197903));
```

- **Existiert das Element?** Wenn Sie vor einem `put` wissen wollen, ob es den Eintrag schon gibt, fragen Sie die Methode `containsKey`.

```
if ( ! kfzKreis.containsKey("PI")) {
    System.out.println("Kennzeichen PI unbekannt");
}
```

- **Element auslesen:** Wenn Sie später einmal wissen wollen, ob das Kennzeichen tatsächlich für Sylt steht, können Sie über die Methode `get` danach fragen. Der Rückgabewert ist eine Referenz auf das Datenobjekt, hier also ein Kreis.

```
Kreis sl = kfzKreis.get("SL");
```

  Jetzt werden Sie feststellen, dass SL nicht für Sylt steht. Sylter Fahrzeuge haben natürlich NF als Kennzeichen. NF steht für Neidfaktor.

- **Element löschen:** Die Methode `remove` entfernt ein Element aus der `Map`. Und es ist nicht überraschend, dass dazu wiederum der Schlüssel als Parameter übergeben wird. Die Methode liefert noch einmal das Element als Rückgabewert oder `null`, wenn es für den Schlüssel keinen Eintrag gab.

```
kfzKreis.remove("SL");
```

Maps durchlaufen

Wenn Sie versuchen, eine `for`-Schleife über eine `Map` laufen zu lassen, wird der Compiler böse Kommentare abgeben, die darauf hinauslaufen, dass eine `Map` nicht iterierbar sei.

Sie können aber einen Trick anwenden! Die `Map` liefert Ihnen über die Methode `keySet` ein `Set`, das alle Schlüssel enthält. Ein `Set` können Sie aber wunderbar iterieren. Diese Iteration liefert Ihnen in einer Schleife einen Schlüssel nach dem anderen und Sie können mit der Methode `get` auf die `Map` losgehen, um die Elemente auszulesen.

```
Set<String> kennzeichenSet = kfzKreis.keySet();
for (String key : kennzeichenSet) {
    System.out.println(key + ": " + kfzKreis.get(key).getWo());
}
```



Um eine `Map` zu iterieren, holen Sie sich mit der Methode `keySet` das Set über alle Schlüssel. Dieses können Sie iterieren und erhalten nacheinander die Schlüssel. Über diese Schlüssel wiederum greifen Sie mit der Methode `get` auf die `Map` zu, um das Element zu holen.

Das folgende Beispielprogramm demonstriert den Umgang mit einer `Map`. Wir füllen eine `Map` mit ein paar Kreisen, holen den Kreis Pinneberg über dessen Kennzeichen PI, um ihn gleich anschließend zu entfernen. Zuletzt wird die `Map` über einen Iterator ausgelesen und aufgelistet. Los geht's!

```
import java.util.HashMap;
import java.util.Set;

public class HashMapTest {

   public static void main(String args[]) {
        HashMap<String, Kreis> kfzKreis
                = new HashMap<String, Kreis>();
        kfzKreis.put("SL",
                new Kreis("Schleswig-Flensburg",197903));
        kfzKreis.put("HG",
                new Kreis("Hochtaunuskreis",227425));
        kfzKreis.put("MH", new Kreis("Muehlheim",167344));
        kfzKreis.put("PI", new Kreis("Pinneberg",303481));
        System.out.println("Es gibt " + kfzKreis.size()
                + " Kennzeichen.");
        if (kfzKreis.containsKey("PI")) {
            System.out.println("PI ist das Kennzeichen von: "
                    + kfzKreis.get("PI").getWo());
        } else {
            System.out.println("Kennzeichen PI unbekannt");
        }
        kfzKreis.remove("PI");
        System.out.println("Eine Liste: ");
        Set<String> s = kfzKreis.keySet();
        for (String k : s) {
            System.out.println(k + ": "
                    + kfzKreis.get(k).getWo());
        }
    }
}
```

Listing 15.20: Eine Spielerei mit der HashMap

Sortierte Schlüssel in der TreeMap

Etwas bedauerlich ist es allerdings, dass die Liste so gar nicht sortiert ist. Das liegt natürlich daran, dass der Iterator durch ein `Set` läuft und das ist von Haus aus eben nicht sortiert.

Eine Sortierung erreichen Sie, indem Sie `TreeSet` statt `HashSet` verwenden, da `TreeSet` ja sortiert ist. Dazu übergeben Sie das von der Methode `keySet` gelieferten `Set` dem Konstruktor eines `TreeSet`. Die Zeile heißt dann:

```
TreeSet<String> s = new TreeSet<String>(tagesHits.keySet());
```

Der Rest kann bleiben, denn dem Iterator ist es letztlich ganz egal, ob der Set nun sortiert ist oder nicht.

Lebt denn der alte Hashtable noch?

Beim Stöbern im Internet oder in alten Java-Programmen stoßen Sie vielleicht auf den `Hashtable`. Der Container `Hashtable` ist keine Implementierung von `Map`, sondern sollte durch `HashMap` abgelöst werden. `Hashtable` ist älter als `HashMap` und nicht Bestandteil des Java Collection Frameworks, erfüllt aber ähnliche Aufgaben. Allerdings ist eine Überführung in das Java Collection Framework nicht so einfach möglich wie beim `Vector`, sodass von der Verwendung von `HashTable` abgeraten wird. Nur aus Kompatibilitätsgründen wird er noch unterstützt.

Die meistzitierten Unterschiede liegen darin, dass `Hashtable` synchronisiert ist, was bei `HashMap` durch das Programm gewährleistet werden kann, wenn es wirklich benötigt wird. Da dies aber nicht weiter schwierig ist, spricht dies nicht wirklich für eine Verwendung von `Hashtable`.

Enumeration

Ein Auslesen des `Hashtable` erfolgt nicht über einen Iterator, sondern über eine `Enumeration`, die ebenfalls nicht zum Java Collection Framework gehört. Listing 15.21 zeigt das Zusammenspiel zwischen `Hashtable` und `Enumeration`.

```
import java.util.Enumeration;
import java.util.Hashtable;

public class HashtableDemo {

    public static void main(String[] args) {
        Hashtable<String, Kreis> kfzKreis
                = new Hashtable<String, Kreis>();
        kfzKreis.put("SL",
                new Kreis("Schleswig-Flensburg",197903));
        kfzKreis.put("HG",
                new Kreis("Hochtaunuskreis",227425));
        kfzKreis.put("MH", new Kreis("Muehlheim",167344));
        kfzKreis.put("PI", new Kreis("Pinneberg",303481));
        System.out.println("Es gibt " + kfzKreis.size()
                + " Kennzeichen.");
```

```
        if (kfzKreis.containsKey("PI")) {
            System.out.println("PI ist das Kennzeichen von: "
                    + kfzKreis.get("PI").getWo());
        } else {
            System.out.println("Kennzeichen PI unbekannt");
        }
        kfzKreis.remove("PI");
        System.out.println("Eine Liste: ");
        Enumeration e = kfzKreis.keys();
        while (e.hasMoreElements()) {
            String k = (String) e.nextElement();
            System.out.print(k + ": "
                    + kfzKreis.get(k).getWo());
            System.out.print(" ("
                    + kfzKreis.get(k).getEinwohner());
            System.out.println(" Einwohner)");
        }
    }
}
```

Listing 15.21: Wie der `Hashtable` damals arbeitete

Die Interfaces

Die Datenbehälter des Java Collections Frameworks basieren auf verschiedenen Interfaces. Auf den ersten Blick interessiert das den Programmierer wenig. Allerdings geben Interfaces die Methoden vor, die ihre Implementierungen umsetzen müssen. Und dann wird es doch recht interessant, weil man als Programmierer leicht erfassen kann, welcher Datenbehälter welche Operationen zulässt.

Zu den Basis-Interfaces gehören `Collection` und `Map`.

- `Collection` beschreibt allgemein eine Datensammlung von Objekten. Von `Collection` leiten sich weitere Interfaces ab. All diese Interfaces müssen also die Methoden umsetzen, die die Collection vorgibt.
  - Der Name des Interface `List` lässt schon erahnen, dass es Listen mit einer Reihenfolge modelliert.

    Das Interface `List` wird durch die Klasse `ArrayList` implementiert, dessen Grundidee ein dynamisches Array ist, das bei Bedarf wachsen kann, und durch `LinkedList`, das als verkettete Liste implementiert ist. `LinkedList` implementiert darüber hinaus auch das Interface `Queue`.

- Eine andere Collection ist ein `Set`, das eine Menge darstellt. Seine Elemente haben keine Reihenfolge, treten aber nur einmal auf.

  Die Klasse `HashSet` implementiert sehr effizient das Interface `Set`. Daten werden unsortiert gesammelt. Die Klasse `TreeSet` wird eingesetzt, wenn eine sortierte Reihenfolge wichtig ist.

- Das Interface `Queue` realisiert Warteschlangen. Queues haben natürlich auch etwas mit Listen zu tun. Wie gut, dass man Interfaces kombinieren kann.

- Das Interface `Map` steht für Datenbehälter, die nicht einfach durchlaufen werden können. Statt dessen wird der Dateninhalt über einen Schlüsselbegriff, den Key, erreicht.

  Die Klassen `HashMap` und `TreeMap` implementieren das Interface `Map`.

Das Interface Collection und seine Methoden

Die Interfaces bilden die Basis für die eingesetzten Collections. Das Interface `Collection` beschreibt eine Zusammenstellung gleichartiger Daten. Erbberechtigt sind `List`, `Queue` und `Set` und einige weitere Interfaces. Im Erbgut finden sich ein paar Methoden wie `add` oder `remove`, die Sie hier zusammengestellt finden. Bei den Parametern verwende ich `E` als Stellvertreter für den Typ der verwalteten Datenelemente.

- `boolean add(E e)`

  Die Methode `add` fügt der `Collection` ein Element hinzu.

- `boolean remove(E e)`

  Mit `remove` kann ein übergebenes Objekt wieder entfernt werden.

- `void clear()`

  Die Methode `clear` löscht alle Elemente auf einmal.

- `boolean contains(E o)`

  Die Methode `contains` ermittelt, ob das übergebene Objekt in der `Collection` enthalten ist.

- `int size()`

  Die Methode `size` liefert die Anzahl der enthaltenen Elemente.

- `Object toArray()`

  Die Methode `toArray` erzeugt ein Array, das alle Elemente der `Collection` enthält.

- `Iterator iterator()`

  Die Methode `iterator` einer `Collection` liefert ihren Iterator. Ein Iterator ist eine sehr spezielle Referenz auf ein Element der `Collection`. Sie können damit

die Daten durchlaufen.

Die Methoden des Interface List

Das Interface `List` fügt dem Interface `Collection` noch ein paar Methoden hinzu, die zu der speziellen Organisation der Listen passen. `List` hat die besondere Eigenschaft, dass die Elemente in einer Reihenfolge abgelegt werden. Darum ist es möglich, über eine Positionsnummer auf die Elemente zuzugreifen. Ob das in jedem Fall effizient ist, hängt aber von der Implementierung ab.

- ✔ `E get(int index)`

  Die Methode `get` liefert das Element an der Position `index`.
- ✔ `E set(int index, E element)`

  Das Element an der Position `index` wird durch das übergebene Element ersetzt.
- ✔ `void add(int index, E element)`

  Ein neues Element wird an der Position `index` eingefügt.
- ✔ `E remove(int index)`

  Das Element an der Position `index` wird entfernt.
- ✔ `int indexOf(Object o)`

  Die Methode `indexOf` liefert die Position, an der das übergebene Objekt zu finden ist.
- ✔ `int lastIndexOf(Object o)`

  Die Methode `lastIndexOf` liefert die letzte Position, an der das übergebene Objekt in der Liste zu finden ist.
- ✔ `ListIterator<E> listIterator()`

  Die Liste liefert durch den Aufruf `listIterator` einen speziell auf die Liste zugeschnittenen Iterator. Der kann mehr als ein normaler Collection-Iterator.
- ✔ `ListIterator<E> listIterator(int index)`

  Durch Angabe der Position kann auch ein Iterator erlangt werden, der auf eine beliebige Stelle in der Liste verweist.
- ✔ `List<E> subList(int from, int to)`

  Die Methode `subList` liefert eine Liste, die eine Ansicht auf einen Teil der Liste darstellt. Begrenzt wird sie durch Anfangs- und Endposition.

Die Methoden des Interface Map

Maps eignen sich vor allem zum Auffinden von Elementen über einen Schlüssel. Schlüssel und Element werden separat als Parameter übergeben. Der Schlüssel muss eindeutig sein. Doppelungen sind also nicht erlaubt. Man könnte eine `Map` als Modellierung einer mathematischen Funktion betrachten. Hier sehen Sie die Methoden der `Map` im Überblick:

- ✔ `E put(K key, E e)`

  Die Methode fügt einen Schlüssel mit dem dazugehörigen Element ein.
- ✔ `E get(K key)`

  Die Methode liefert das zu dem Schlüssel gehörige Element.
- ✔ `E remove(K key)`

  Die Methode löscht das zu dem Schlüssel gehörige Element.
- ✔ `boolean containsKey(Object key)`

  Die Methode prüft, ob der angegebene Schlüssel enthalten ist.
- ✔ `boolean containsValue(Object value)`

  Die Methode prüft, ob das angegebene Element enthalten ist.
- ✔ `int size()`

  Die Methode liefert die Anzahl der Elemente.
- ✔ `boolean isEmpty()`

  Die Methode prüft, ob die `Map` leer ist.
- ✔ `Set keySet()`

  Die Methode `keySet` dient dazu, die Schlüssel einer `Map` in ein `Set` zu überführen. Dieses kann dann per Iterator bearbeitet werden.

Die Methoden des Java Collection Frameworks

Neben den Datenbehältern stellt das Java Collection Framework in der Klasse `Collections` auch einige statische Methoden zur Verfügung. Die Methoden arbeiten vor allem mit Implementierungen von `List`, die sie als Parameter erwarten.

Listing 15.22 ruft als Beispiel die Methode `shuffle` auf. Sie können dieses Programm als Rahmen verwenden, um die anderen Methoden zu testen.

```
import java.util.*;
public class TestShuffle {
    public static void main(String[] args) {
        String farben[] = {"rot", "blau", "gelb", "rosa"};
        List<String> farbliste = Arrays.asList(farben);
        Collections.shuffle(farbliste);
```

```
        for (String farbe : farbliste ) {
            System.out.println(farbe);
        }
    }
}
```

Listing 15.22: Das Programm mischt seine Aufrufparameter

Das Beispielprogramm in Listing 15.22 übernimmt Farb-Arrays in eine Liste. Dabei hilft die Array-Methode `asList`. Anschließend wird die Liste geschüttelt und auf dem Bildschirm ausgegeben.

- Unordnung stiften mit `shuffle`:

  Wenn ich mir mein Arbeitszimmer anschaue, scheint hier die Methode `shuffle` eingesetzt worden zu sein. Sie bewirkt, dass die Elemente des Datenbehälters gemischt werden. Nach jedem Aufruf der Methode ist die Reihenfolge anders.

- Umdrehen mit `reverse`:

  Sehr viel kontrollierter geht es bei der Methode `reverse` zu. Sie dreht die Reihenfolge der Elemente einer Liste um.

- Endlich Ordnung mit `sort`:

  Eine häufige Anwendung in der Programmierung ist die Sortierung. Die Klasse `Collections` stellt eine statische Methode `sort` zur Verfügung, die mit großer Wahrscheinlichkeit schneller ist als das, was wir uns als einzelne Programmierer selbst zusammenbauen.

Suche in der Liste: binarySearch

Die Methode `binarySearch` sucht mit den binären Suchverfahren in einer `Collection` nach einem Element. Naturgemäß findet die Methode nur etwas, wenn die Collection in sortierter Reihenfolge vorliegt. Das erreichen Sie, indem Sie zuvor `sort` aufrufen.

```
import java.util.*;
public class TestBinarySearch {
    public static void main(String[] args) {
        String farben[] = {"rot", "blau", "gelb", "rosa"};
        List<String> farbliste = Arrays.asList(farben);
        Collections.sort(farbliste);
        System.out.println(
                Collections.binarySearch(farbliste, "gelb"));
    }
}
```

Listing 15.23: Binäres Suchen in einer sortierten Struktur

Das Programm liefert als Ergebnis die Position, an der der String »gelb« nach der Sortierung steht. Dabei ist zu berücksichtigen, dass die Position mit 0 beginnt. Wurde der Suchbegriff nicht gefunden, ist der zurückgegebene Wert negativ.

Wie sortiert man Autos?

Wenn wir sortieren, sollten wir wissen, wonach wir sortieren und wie. Klar und einfach ist dies bei Zahlen und bei Zeichenketten. Na ja, zumindest bei den Zahlen.

Aber unangenehmer wird es bei selbst gebastelten Typen, also eigenen Klassen. Vielleicht erinnern Sie sich an das Autoquartett auf dem Schulhof. Der Porsche hat den Bentley zwar in Bezug auf Geschwindigkeit und Beschleunigung abgehängt, wenn aber der Bentley mit der Länge oder dem Gewicht auftrumpfte, war der Porsche erledigt. Und selbst ein Kleinwagen schlägt den Sportwagen, wenn nach der Anzahl der Sitze gefragt wird.

Bei einer Klasse mit mehreren Attributen muss also festgelegt werden, welches die natürliche Reihenfolge ist. Und das muss wohl die Klasse selbst tun.

Das Interface Comparable

Eine Klasse legt die Reihenfolge ihrer Objekte fest, indem sie das Interface `Comparable` erfüllt und dessen abstrakte Methode `compareTo` implementiert.

Die Klasse `Vergleichbar` in Listing 15.24 nimmt Vor- und Nachnamen auf. Um `compareTo` zu implementieren, greift sie auf `compareTo` der Strings zurück. Das erste Kriterium ist der Nachname, das zweite der Vorname.

```
public class Vergleichbar implements Comparable<Vergleichbar> {
    String vorname, nachname;
    public int compareTo(Vergleichbar vgl) {
        int vergleichsWert = 0;
        vergleichsWert = nachname.compareTo(vgl.nachname);
        if (vergleichsWert==0) {
            vergleichsWert = vorname.compareTo(vgl.vorname);
        }
        return vergleichsWert;
    }
    public String toString() {
        return vorname + " " + nachname;
    }
    Vergleichbar(String vor, String nach) {
        vorname = vor; nachname = nach;
    }
}
```

Listing 15.24: Eine vergleichbare Klasse

Der Vergleichswert, den `compareTo` zurückliefert, ist negativ, wenn der übergebene Wert kleiner ist, 0 bei Gleichheit und positiv, wenn der übergebene Wert größer ist.

Die Methode `toString` werden wir für die Kontrollausgabe benötigen. Das Hauptprogramm testet die Sortierreihenfolge. Es wird zunächst ein Array von `Vergleichbar` mit ein paar Vorbesetzungen angelegt. Es wird in eine Liste überführt, die dann sortiert und zum Schluss ausgegeben wird.

```
import java.util.*;

public class VergleichSort {
    public static void main(String[] args) {
        Vergleichbar nameArray[] = {
            new Vergleichbar("Helmut", "Schmidt"),
            new Vergleichbar("Karl", "Bart"),
            new Vergleichbar("Marta", "Haari"),
            new Vergleichbar("Peter", "Schulz"),
            new Vergleichbar("Ingrid", "Schulz")
        };
        List<Vergleichbar> names = Arrays.asList(nameArray);
        Collections.sort(names);
        System.out.println(names);
    }
}
```

Listing 15.25: Vergleichbares lässt sich sortieren.

Diverse Standardklassen implementieren `Comparable`. Das sind die Zahlenklassen wie `Integer` oder `Float`. Dazu gehören aber auch `Date` oder `String`. Letzteres habe ich ja für `Vergleichbar` ausgenutzt.

Nachsortieren mit dem Comparator

Wollen Sie eine Klasse anders sortieren, als `compareTo` vorgibt, oder soll eine Klasse sortiert werden, die `Comparable` gar nicht implementiert, kann der `Comparator` eingesetzt werden.

Das folgende Beispiel geht wieder auf `Vergleichbar` los. Diesmal soll allerdings isländisch sortiert werden. In Island steht der Vorname zuerst im Telefonbuch. Dafür wird ein `Comparator` namens `island` über `Vergleichbar` definiert. Er enthält eine Methode `compare`, die zwei Parameter der zu vergleichenden Klasse erwartet. Sie gibt -1 für kleiner, 0 für gleich und 1 für größer zurück. Der `Comparator` wird als zweiter Parameter an `sort` übergeben.

```
import java.util.*;

public class ComparatorTest {

    public ComparatorTest() {
        Vergleichbar nameArray[] = {
            new Vergleichbar("Helmut", "Schmidt"),
            new Vergleichbar("Karl", "Bart"),
            new Vergleichbar("Marta", "Haari"),
            new Vergleichbar("Peter", "Schulz"),
            new Vergleichbar("Ingrid", "Schulz")
        };
        List<Vergleichbar> names = Arrays.asList(nameArray);
        Comparator<Person> island =
                new Comparator<Person>() {
            public int compare(Person orig, Person vgl) {
                int vergleichsWert = 0;
                vergleichsWert = orig.vorname.compareTo(vgl.vorname);
                if (vergleichsWert==0) {
                    vergleichsWert =
                            orig.nachname.compareTo(vgl.nachname);
                }
                return vergleichsWert;
            }
        };
        Collections.sort(names, island);
        System.out.println(names);
    }
    public static void main(String[] args) {
    }
}
```

Listing 15.26: Vergleich per `Comparator`

Da die Variable `island` aber nur als zweiter Parameter für die Methode `sort` eingesetzt wird, kann man die Generierung des `Comparators` auch direkt dort machen. Es verkürzt das Listing ein wenig.

```
public ComparatorTest() {
    Person nameArray[] = {
            new Person("Helmut", "Schmidt"),
            new Person("Karl", "Bart"),
            new Person("Marta", "Haari"),
            new Person("Peter", "Schulz"),
            new Person("Ingrid", "Schulz")
    };
    List<Person> names = Arrays.asList(nameArray);
    Collections.sort(names, new Comparator<Person>() {
        public int compare(Person orig, Person vgl) {
            int vergleichsWert = 0;
            vergleichsWert = orig.vorname.compareTo(vgl.vorname);
            if (vergleichsWert==0) {
                vergleichsWert =
                        orig.nachname.compareTo(vgl.nachname);
            }
            return vergleichsWert;
        }
    });
    System.out.println(names);
}
```

Listing 15.27: Vergleich per `Comparator` etwas verkürzt

Ein ähnliches Vorgehen werden Sie bei den Ereignisbehandlungen der grafischen Oberflächen sehen. Ab Java Version 8 kann dieser Ausdruck durch den Einsatz von Lambda (siehe Abschnitt 334) noch etwas verkürzt werden.

Lambda-Ausdrücke und Streams

Es kommt in Java nicht selten vor, dass Klassen gebaut werden, um eine einzelne Funktion zu realisieren. Sie werden dies bei der Ereignisbehandlung der grafischen Oberflächen häufiger sehen. Das führt zu länglichen Konstrukten, die gern auch kürzer sein dürften. Hier setzen Lambda-Ausdrücke an.



So ganz simpel sind Lambda-Ausdrücke nicht zu verstehen. Als Anfänger können Sie den Abschnitt erst einmal überspringen und später wiederkommen, wenn Sie sich stark genug fühlen.

Der ganze Spaß beginnt erst bei Java Version 8. Dort kann die Methode einer Klasse, die nur eine Methode hat, per Lamba extrem kurz aufgerufen werden. Für die Klasse wird zunächst ein Interface geschaffen, das garantiert, dass es nur eine Methode gibt.

```
@FunctionalInterface
interface NurEineMethode {
    abstract int gebeProdukt(int a, int b);
}
```

Die Methode `gebeProdukt` ist die einzige Methode. Und damit der Compiler prüft, ob nicht doch irgendein Clown noch eine Methode dazu geschummelt hat, wird dies durch `@FunctionalInterface` gewährleistet.

Durch den Lambda-Ausdruck wird zunächst die Methode mit Leben gefüllt.

```
NurEineMethode lamobj = (int a, int b) -> { return a*b; };
```

Damit kann nun aufgerufen werden.

```
int a = lamobj.gebeProdukt(12, 4);
```

Nun werden Sie zu Recht einwerfen, dass Sie noch nie so viel Code geschrieben haben, um zwei Zahlen miteinander zu multiplizieren. Tatsächlich ist es aber so, dass Sie das Interface in den seltensten Fällen selbst schreiben müssen. Andere Klassen haben bereits dafür gesorgt.

Hinzu kommt, dass das Objekt `lamobj` nur hier zur Demonstration verwendet wird. In der Praxis wird Lambda dann eingesetzt, wenn das Objekt selbst nicht interessant ist. Beispiel gefällig?

Comparable im Lambda-Fieber

Wie man eine Liste über einen `Comparator` sortiert, haben Sie ja schon gesehen. Der `Comparator` gibt dem Programmierer die Möglichkeit, die Sortierung zu beeinflussen. So könnten Autos nach Verbrauch, PS oder Höchstgeschwindigkeit sortiert sein, je nach persönlichen Präferenzen. Als Beispiel wurden allerdings nicht Autos, sondern Personen verwendet, die nach dem isländischen Telefonbuch sortiert sind, also zuerst nach dem Vornamen und dann erst nach dem Nachnamen.

Ein Comparator ist eine Klasse mit einer Methode und eignet sich wunderbar zum Einsatz von Lambda und zeigt dessen Einsatz in einem etwas realistischeren Beispiel. Listing 15.28 zeigt das Listing ohne jegliche Sortierung.

```
import java.util.*;

public class ComparatorTest {

    // Person als zu sortierende lokale Klasse
    class Person {
        public String vorname, nachname;
        // Konstruktor: Vorname und Name als Parameter
        public Person(String v, String n) {
            vorname = v;
            nachname = n;
        }
        // Um die Person anzeigen zu können
        public String toString() {
            return vorname + " " + nachname;
        }
    }

    // Wir bilden ein Array von Personen, um später ...
    Person personenArray[] = {
            new Person("Helmut", "Schmidt"),
            new Person("Karl", "Bart"),
            new Person("Marta", "Haari"),
            new Person("Peter", "Schulz"),
            new Person("Ingrid", "Schulz")
            };
    // ... eine Liste daraus zu machen. Darüber wird sortiert.
    List<Person> personen = Arrays.asList(personenArray);

    // Der Konstruktor, damit nicht alles static läuft.
    public ComparatorTest() {
        // Zur Kontrolle ausgeben
        System.out.println(personen);
    }

    public static void main(String[] args) {
        new ComparatorTest(); // Starte den eigenen Konstruktor.
    }
}
```

Listing 15.28: Personenliste

Die Sortierung wird im Konstruktor `ComparatorTest` erfolgen. Zunächst wird ein `Comparator`-Objekt mit dem Namen `IslandComparator` erstellt, das dann von der Methode `sort` als Parameter verwendet wird.

```
public ComparatorTest() {
    Comparator<Person> IslandComparator =
            new Comparator<Person>() {
        // In compare wird wirklich verglichen!
        @Override
        public int compare(Person orig, Person vgl) {
            int vergleichsWert = 0;
            vergleichsWert = orig.vorname.compareTo(vgl.vorname);
            if (vergleichsWert==0) {
                vergleichsWert =
                        orig.nachname.compareTo(vgl.nachname);
            }
            return vergleichsWert;
        }};
    personen.sort(IslandComparator);
    // Zur Kontrolle ausgeben
    System.out.println(personen);
}
```

Listing 15.29: `Comparator` wird vor der Ausgabe eingebaut.

Sie sehen, wie ein Objekt `IslandComparator` angelegt wird, das nur als Parameter für `sort` benötigt wird. Es wird die komplette Deklaration des Comparators benötigt. All das fällt weg, wenn man statt dessen Lambda einsetzt.

```
public ComparatorTest() {
    // personen.sort((Person orig, Person vgl) -> {
    // Noch kürzer:
    personen.sort((orig, vgl) -> {
        // dies ist der Inhalt der Methode compare()
        int vergleichsWert = 0;
        vergleichsWert = orig.vorname.compareTo(vgl.vorname);
        if (vergleichsWert == 0) {
            vergleichsWert = orig.nachname.compareTo(vgl.nachname);
        }
        return vergleichsWert;
    });
    // Zur Kontrolle ausgeben
    personen.forEach((p) -> System.out.println(p));
}
```

Listing 15.30: `Comparator` wird vor der Ausgabe eingebaut.

Der `Comparator` wird direkt im Parameter des Aufrufs von `sort` als Lambda-Ausdruck erzeugt. Dabei bleibt von der gesamten Deklaration nur noch der aktive Inhalt der alten Methode `compare` übrig. Selbst die Parametertypen können entfallen, weil sie sich aus der Generics von `Person` ergibt.

Zum Schluss werden noch einmal alle Personen ausgegeben. Hier wird wiederum

das Lambda genutzt, um die Methode `forEach` zu füttern.

Streams

Mit Java 8 wurden auch die Streams eingeführt, die als Wrapper um die Collections gebildet werden und Lambda intensiv nutzen.

Um aus einer Collection einen Stream zu machen, wird einfach die Methode `stream` über die Collection aufgerufen. Dieser Stream hat selbst wieder einige Methoden, die es ermöglichen, Daten des Streams zu verändern und zu filtern. Als Rückgabewert liefern sie wiederum einen Stream, sodass es möglich ist, diese Methoden einfach aneinanderzuhängen.

```
public ComparatorTest() {
    personen.stream().filter((p) ->  p.nachname.equals("Schulz")).
                      forEach((p) -> System.out.println(p));
    System.out.println("------");
    personen.forEach((p) -> System.out.println(p));
}
```

Listing 15.31: Stream aus der Liste

In Listing 15.31 wird aus den Personen ein Stream generiert, der zunächst über die Methode `filter` auf die Personen reduziert wird, die mit Nachnamen Schulz heißen. Anschließend wird mit der Methode `forEach` auf den verbleibenden Personen im Stream eine Aktion ausgeführt. Hier ist es eine Ausgabe auf dem Bildschirm.

Filter

Die Methode `filter` erwartet als Parameter ein Objekt der Klasse `Predicate`, deren einzige Methode `test` als Rückgabewert `boolean` hat. Durch den Einsatz von Lambda müssen Sie diese Details gar nicht wissen.

Sie können aber ein Objekt der Klasse `Predicate` anlegen, falls Sie denselben Filter noch mehrfach benötigen sollten. Filter können Sie mehrfach hintereinandersetzen, da ihr Rückgabewert selbst wieder ein Stream ist.

Neben der Methode `filter` gibt es noch weitere Methoden, die einen Stream zurückgeben. Sie können in Ketten hintereinandergesetzt werden, bis der Stream alle Methoden durchgearbeitet hat.

- `limit`: Dies begrenzt die Anzahl der Elemente im Stream. Als Parameter wird ein Wert vom Typ `long` erwartet.
- `sorted`: Die Methode sortiert den Stream. Wird kein Parameter übergeben, muss die Elementklasse `Comparable` implementieren. Ansonsten müssten Sie einen Comparator definieren und diesen als Parameter übergeben. Und das funktioniert sehr gut als Lambda, wie Sie eben bereits gesehen haben.

- `distinct`: Sortiert die doppelten Elemente aus dem Stream aus. Damit die Methode doppeltes Auftreten erkennt, muss die Elementklasse die Methode `equals` und `hashCode` implementieren. Das Problem ist in Abschnitt 317 schon einmal beschrieben worden.

Terminatoren

Nein, mein Namensvetter Schwarzenegger hat damit nichts zu tun. Der kennt sich vermutlich auch gar nicht mit Java aus. Es handelt sich einfach um eine Methode, die an das Ende der Streams gesetzt wird und als Rückgabewert keinen Stream mehr liefert. Damit ist dann die Kette am Ende.

Dennoch können die Methoden interessante Ergebnisse liefern.

- `count`: Mit dieser Methode wird einfach durchgezählt, wie viele Elemente vorhanden sind. Und es ist naheliegend, dass die Methode einen Wert vom Typ `long` zurückgibt.
- `max`: Liefert den maximalen Wert des Streams. Um diesen zu ermitteln, wird ein Comparator als Parameter benötigt, der wiederum als Lambda implementiert werden kann. Rückgabewert ist ein `Optional`. Dieses liefert über die Methode `isPresent` die Information, ob es ein Objekt gibt, das man im Erfolgsfalle per `get` abholt. So sieht das dann aus:

```
Optional<Person> max =personen.stream()
        .max((p, v) -> (p.nachname.compareTo(v.nachname)));
if (max.isPresent()) {
    Person p = max.get();
    System.out.println(p.nachname + p.vorname);
}
```

- `min`: Die Überraschung ist groß, dass es auch ein Minimum gibt. Und ich verrate Ihnen nicht, wie man es ermittelt.

Konsument

Die Methode `forEach` erwartet als Parameter ein Objekt der Klasse `Consumer`. Deren einzige Methode `accept` gibt nichts zurück, sondern kann auf die Elemente des Streams zugreifen und deren Werte verändern. Sie könnten also das Alter der Personen erhöhen oder ihr Gehalt. Letzteres dürfte ihnen lieber sein. Sie können sie alle nach Hamburg umziehen lassen, sofern dieses Attribut in der Klasse `Person` vorliegt. Hier habe ich mir zu viele Details verkniffen, damit das Beispiel nicht unübersichtlich wird.

Collection

Ihre Änderungen im Stream wirken sich auf den Datenstrom, nicht aber auf den Datenbehälter aus, den Sie als Basis Ihres Streams verwenden. Mit der Methode

`collect` können Sie aus dem Stream wieder eine Liste erstellen.

```
personen = personen.stream()
                   .filter(....
                   .collect(Collectors.toList());
```

Hinweis

[1] norddeutsch: neugierig und heimlich nachschauen

Teil IV

Fenster- und Grafikprogrammierung



Kapitel 16

Mit Swing Fenster, Mäuse und Grafik programmieren

IN DIESEM KAPITEL

Von Fensterrahmen, Ereignissen und Mauseschwänzchen

Rechtecke, Kreise und alles in Farbe

Clipping abseits von Dover und Rügen

Fenster, durch die man nicht durchschauen kann, Menüs, die nicht satt machen, und Mäuse, die man schieben muss: Der Desktop ist voller alberner Metaphern. Die Programmierung von grafischen Benutzerschnittstellen ist also wie geschaffen für dieses Buch. Und immerhin stellt Java mit Swing sogar eine Programmierschnittstelle zur Verfügung, die dann auf fast allen Systemen läuft.

JFrame: Guck mal durchs Fenster

`JFrame` heißt die Rahmenklasse, in der unser Programm stattfinden wird. Swing stellt mit dieser Klasse ein komplett graues Fenster ohne weiteren Inhalt zur Verfügung. Unser erstes Programm erzeugt ein solches schmuckloses Fenster und macht es auf dem Bildschirm sichtbar.

`JFrame` ist von Natur aus bescheiden und fordert nur so viel Platz an, wie es wirklich benötigt. Da es am Anfang leer ist, ist es so klein, dass Sie einige Zeit mit Suchen verbringen würden, um es zu entdecken. Überdies ist das Fenster zunächst unsichtbar.

- Mit dem `import`-Befehl werden alle Swing-Bibliotheken eingebunden. Es würde reichen, `javax.swing.JFrame` zu importieren. Über kurz oder lang werden wir aber mehr benötigen.
- Das Beispielprogramm erzeugt zunächst ein Objekt der Klasse `JFrame`.
- Über die Methode `setSize` gibt das Programm dem Fenster eine Mindestgröße vor.
- Zuletzt schaltet sich das Programm per `setVisible` sichtbar, sonst hätte ja niemand etwas davon.

```java
import javax.swing.*; // Wir verwenden alle Swing-Klassen

public class ErsterSwing {
    public static void main(String[] args) {
        JFrame Fenster = new JFrame("");
        Fenster.setSize(400, 300);
        Fenster.setVisible(true);
    }
}
```

Listing 16.1: Das erste Swing-Programm

Die Klasse `JFrame` erledigt einige Aufgaben von sich aus. Sie können das Fenster bereits verschieben, vergrößern und schließen, ohne dass das Programm etwas dazu beitragen muss.

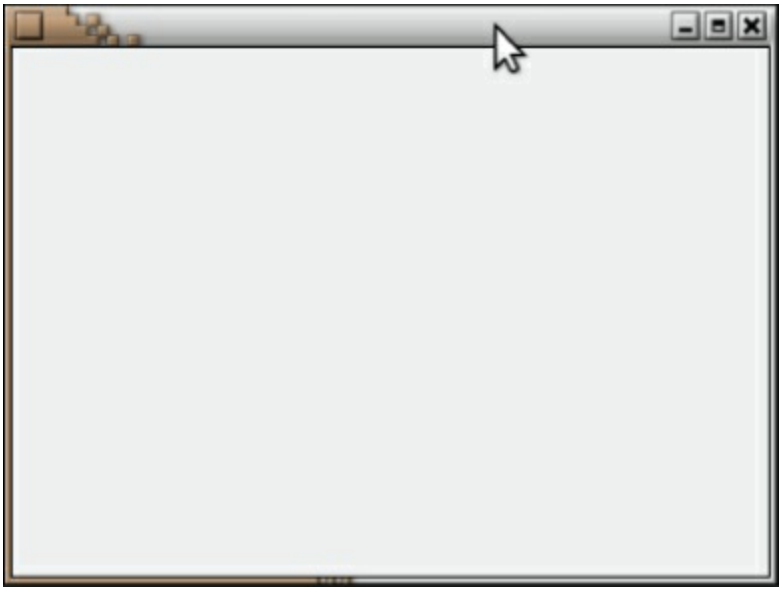


Abbildung 16.1: Ein Swing-Programm

Das Schließkreuz schließt nur das Fenster

Wenn Sie ein wenig damit herumspielen, werden Sie feststellen, dass das Fenster zwar tatsächlich über den Schließbutton schließt, aber dass das Programm deshalb noch lange nicht endet.

- ✔ Aus der Kommandozeile müssen Sie extra noch einmal Strg + C drücken.
- ✔ Wenn Sie mit Eclipse arbeiten, sehen Sie, dass der rote Button zum Terminieren immer noch aktiv ist. Klicken Sie diesen bitte an.

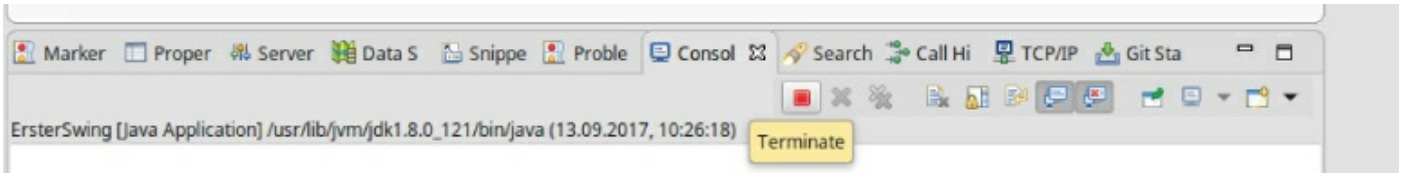


Abbildung 16.2: Terminate-Button in Eclipse

Die Mühe des Selbstmords überlässt Swing dem Programm. Vielleicht will das Programm ja gar nicht sterben! Die Drecksarbeit bleibt also wieder einmal am Programmierer hängen.

Erweiterung von JFrame

Wenn sich das Programm um die Fensterereignisse kümmern soll, ist es das Klügste, das Programmfenster als eine Erweiterung des Standardfensters zu betrachten. Erweiterung? Ja, genau! Das ist diese objektorientierte Geschichte. Die objektorientierte Programmierung spielt wohl kaum irgendwo seine Stärken so extrem aus wie bei der Programmierung grafischer Oberflächen. Darum stellen wir das Hauptprogramm kurz um und leiten es von `JFrame` ab. Schauen Sie genau hin! In der Funktion ändert sich nichts. Es sieht nur etwas anders aus. Damit Sie die Quelltexte leichter unterscheiden können, benenne ich das Programm in `MeinJFrame` um.

```
import javax.swing.*;
public class MeinJFrame extends JFrame {

    public MeinJFrame() {
        this.setSize(400, 300);
        this.setVisible(true);
    }
    public static void main(String[] args) {
        new MeinJFrame();
    }
}
```

Listing 16.2: Das Swing-Programm erweitert `JFrame`

Die Klasse `JFrame` übernimmt alle Arbeiten, die für den Umgang mit Fenstern erforderlich sind. Erst wenn eine Benutzeraktion für die Applikation relevant ist, wird sie die entsprechende Methode überschreiben und gerät damit in die Finger des Programmierers. Der Schließbutton des Fensters wird die erste Benutzeraktion, die wir fangen wollen.


Swing-Programme erweitern die Klasse `JFrame`.

Zeichnen und Malen

Wenn Sie schon einmal eine grafische Oberfläche zur Verfügung haben, möchten Sie auch zeichnen. Und das sollen Sie auch! Sie werden sich in die Kindergartenzeit zurückbegeben, Farben auswählen und dann Linien, Rechtecke und Kreise malen. Für das Vorschulalter gibt es das Buchstabenmalen. Dazu können Sie alle im System verwendeten Schriften verwenden. Auch diese dürfen

Sie bunt machen oder verschiedene Größen verwenden. Und dann stellen wir noch Bilder dar. Es wird bestimmt lustig.

Wir lassen zeichnen

Nun möchten Sie natürlich gleich mit dem Tuschkasten in der Hand auf das Fenster losgehen. Aber halt! So funktioniert das bei Swing nicht. Nicht Sie zeichnen wie auf eine Leinwand, wenn Ihnen danach ist, sondern das Fenster meldet sich, wenn es einen Anlass gibt, den Fensterinhalt zu zeichnen. Sobald das Fenster entscheidet, dass der Inhalt des Fensters neu gezeichnet werden muss, ruft es die Methode `paint` auf. Sie können nun in Ihrer Erweiterung von `JFrame` diese Methode `paint` überschreiben und dort werden Sie dann alle Malereien durchführen, die zur Wiederherstellung des Fensterinhalts notwendig sind.


In ein Fenster wird gezeichnet, indem die Methode `paint` überschrieben wird.

```
@Override
public void paint(Graphics g) {
    // Hier malen Sie mit der Grafikumgebung g
}
```

Listing 16.3: Die Methode `paint` malt.

Graphics

Der Methode `paint` wird eine Referenz auf `Graphics` übergeben. `Graphics` ist eine abstrahierte Grafikumgebung, die auf verschiedener Grafikhardware aufgesetzt sein kann. Es stellt Ihnen die Grafikumgebung und die Zeichenmethoden zur Verfügung, man könnte auch sagen: Tafel und Kreide.

Sofort zeichnen per repaint

Die Methode `paint` wird nicht direkt aufgerufen, sondern dann, wenn das System das Fenster auffordert, seinen Bildschirmausschnitt zu aktualisieren. Das geschieht, wenn der Fensterinhalt zerstört wurde und Sie diesen neu zeichnen sollen, auch wenn es ungerecht ist, weil Sie es ja gar nicht kaputt gemacht haben. Auf jeden Fall sind Sie nicht Herr des Geschehens.

Wenn Sie selbst eine Neuzeichnung auslösen wollen, weil beispielsweise das Bild auf einen Mausklick reagieren soll, können Sie die Methode `repaint` aufrufen. Damit wird dem System signalisiert, dass irgendetwas passiert ist, was ein Neuzeichnen erforderlich macht. Die Methode `paint` übernimmt die Kontrolle und erstellt aus den gespeicherten Daten die Figur.

Die Leinwand

Sie müssen wissen, wohin Sie malen können. Dazu benötigen Sie mindestens die Angaben über den Platz, den Ihnen das Fenster für Ihre Kunst einräumt.

Direkt auf das Fenster malen

Sie können direkt in das Fenster malen, allerdings nicht überall. Wenn Sie den Schiebebalken oder die Statuszeile beschmieren, wird das leicht als Graffiti angesehen, dem man zwar mit Wohlwollen künstlerische Aspekte abgewinnen kann, die aber eher ungewollt und unprofessionell wirkt.

Zunächst müssen Sie feststellen, welches der Bereich des Fensters ist, in dem Sie sich beliebig austoben dürfen.

- ✔ Der Nullpunkt des Fensters ist immer links oben.
- ✔ Die Methode `getWidth` liefert die Breite des Fensters in Pixeln.
- ✔ Die Methode `getHeight` liefert die Höhe des Fensters.
- ✔ Alternativ können Sie auch `getSize` aufrufen, das ein Objekt der Klasse `Dimension` mit den Attributen `width` und `height` besitzt.
- ✔ Die Methode `getInsets` liefert die Ausdehnung des Fensterrands, der nicht bemalt werden soll. Der Rückgabewert ist ein Objekt der Klasse `Insets` mit den Attributen `left`, `top`, `right` und `bottom`. Diese Werte müssen vom Rahmen abgezogen werden.

Auf diese Weise können Sie den Bereich bestimmen, in dem sich Ihre Grafiken wohlfühlen können.

```
import java.awt.Insets;
// ...
@Override public void paint(Graphics g) {
    // Zeichenbereich ermitteln und Schirm löschen
    Insets insets = getInsets();
    int links  = insets.left;
    int oben   = insets.top;
    int breite = getWidth() - insets.left - insets.right;
    int hoehe  = getHeight() - insets.top - insets.bottom;
```

Listing 16.4: Den Platz im Fenster suchen

JPanel als Leinwand

Eine Alternative ist es, ein `JPanel` als Leinwand anzulegen.

- ✔ Das `JPanel` kann vollständig bemalt werden. Links oben ist also bei 0,0. Die Breite errechnet sich über `getWidth` und die Höhe über `getHeight`.
- ✔ Das `JPanel` lässt sich wie ein normaler Button zwischen den Kontrollelementen anordnen, ohne dass sich die Zeichenkoordinaten ändern.
- ✔ Sie können in dem `JPanel` die Mausereignisse unabhängig vom restlichen

Fenster behandeln.

Um ein `JPanel` zu bepinseln, müssen Sie sich eine solche Leinwand anlegen. Dazu gehen Sie sinnvollerweise in den folgenden Schritten vor:

1. Sie erweitern Ihr eigenes `JPanel` und nennen es beispielsweise `PaintPanel`.
2. Sie überschreiben die Methode, die für das Zeichnen zuständig ist. Wie bei `JFrame` heißt die Methode `paint`. Der lieb gewonnene Parameter der Klasse `Graphics` ist auch da.
3. Diese Leinwand wird mit der Methode `add` dem `JFrame` hinzugefügt.

Das erweiterte `JPanel` wird in Listing 16.5 als private Klasse angelegt.

```
import javax.swing.*;
import java.awt.*;

public class GrafikJPanel extends JFrame {

    public static void main(String[] args) {
        new GrafikJPanel();
    }

    private class PaintPanel extends JPanel {
        // Die überschriebene Methode übernimmt das Zeichnen
        @Override protected void paintComponent(Graphics g) {
            int breite = getWidth();
            int hoehe  = getHeight();
            // ...
        }
    }
    public GrafikJPanel() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setTitle("Zeichnen mit Java");
        setSize(400, 300);
        // Hier wird das JPanel eingebaut
        add(new PaintPanel());
        setVisible(true);
    }
}
```

Listing 16.5: Zeichnung in das JPanel

Tafelwischen mit clearRect

Da die Methode `paint` aufgerufen wird, wenn das Fenster zuvor überdeckt war, wissen Sie nichts über den aktuellen Zustand Ihrer Leinwand. Bevor Sie also darauf zeichnen, sollten Sie zunächst einen sauberen Hintergrund herstellen. Das erledigt die Methode `clearRect`. Sie überdeckt das bezeichnete Rechteck in der Hintergrundfarbe. Man könnte sie auch als Methode `tafelPutzen` bezeichnen.

```
@Override protected void paintComponent(Graphics g) {
    // Berechne links, oben, breite und hoehe
    g.clearRect(int links, int oben, int breite, int hoehe)
```

Farbspielereien

Bevor Sie anfangen zu malen, führen Sie Ihren Pinsel in die Farbe. Das ist auch bei `Graphics` nicht anders. Mit der Methode `setColor` legen Sie die Farbe fest, in der ab sofort gemalt wird. Wollen Sie in einer anderen Farbe weitermalen, müssen Sie `setColor` erneut mit anderer Farbe aufrufen.

Als Parameter erwartet `setColor` ein Objekt der Klasse `Color`.

Falls Sie einmal in die Situation kommen, die aktuelle Farbe ermitteln zu wollen, geht Ihnen die Methode `getColor` zur Hand. Sie liefert eine Referenz auf ein Objekt der Klasse `Color` zurück.

Die Klasse `Color` stellt die Standardfarben als statische Konstanten zur Verfügung. Sie können mit den folgenden Bezeichnungen verwendet werden.

```
Color.black   Color.blue     Color.cyan     Color.darkGray
Color.gray    Color.green    Color.magenta  Color.yellow
Color.orange  Color.pink     Color.red      Color.lightGray
Color.white
```

Sollten die festgelegten Farben nicht passen, können Sie RGB-Farbcodes dem Konstruktor von `Color` so übergeben, wie Sie es vielleicht von HTML her kennen.

```
setColor(new Color(0x00ff00));
```

Die ersten beiden Stellen sind der Rot-Anteil. Der ist 00. Es folgt der Grün-Anteil, der mit FF der höchstmögliche Wert ist. Als dritter steht der Blau-Anteil auf 00, sodass wir ein kräftiges Grün erwarten dürfen.

Malen und Zeichnen

Die Zeichenmethoden der Klasse `Graphics` beginnen mit `draw`, wenn der Umriss gezeichnet werden soll, und mit `fill`, wenn der Inhalt gefüllt werden sollen.

Ein Beispiel

Das folgende Beispiel malt ein gelbes Oval mit einem schwarzen Rand, das sich in der Größe am Fenster orientiert. Innerhalb des Ovals malt die Anwendung einen Text (siehe Abbildung 16.3).

1. In `main` wird ein Objekt der eigenen Klasse erzeugt, die `JFrame` erweitert.

2. Der Konstruktor erledigt die Arbeiten am Rahmenfenster. Er setzt die Close-Operation, den Titel und die Größe und macht das Fenster schließlich sichtbar.
3. Das wirklich Spannende passiert in der überschriebenen Methode `paint`. Gleich nach dem Listing geht es weiter.

```
import javax.swing.*;
import java.awt.*;

public class KleineGrafik extends JFrame {

    public static void main(String[] args) {
        new KleineGrafik(); // Löse Konstruktor aus!
    }

    // Der Konstruktor bereitet das Fenster vor
    public KleineGrafik() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setTitle("Zeichnen mit Java");
        setSize(400, 300);
        setVisible(true);
    }

    @Override
    public void paint(Graphics g) {
        // Ermittle die Größe der nutzbaren Zeichenfläche
        Insets insets = getInsets();
        int links  = insets.left;
        int oben   = insets.top;
        int breite = getWidth()-insets.left-insets.right;
        int hoehe  = getHeight()-insets.top-insets.bottom;
        // Fenster freimachen
        g.clearRect(links, oben, breite-1, hoehe-1);
        // Male ein möglichst großes, gelb gefülltes Oval
        g.setColor(Color.yellow);
        g.fillOval(links, oben, breite-1, hoehe-1);
        // Zeichne einen schwarzen Rand um das Oval
        g.setColor(Color.black);
        g.drawOval(links, oben, breite-1, hoehe-1);
        // Beschrifte es nett!
        String meldung  = "Hallo Swing!";
        g.drawString(meldung, breite/2, hoehe/2);
    }
}
```

Listing 16.6: Eine kleine Grafik

1. Am Anfang der Methode `paint` wird mittels `getWidth`, `getHeight` und

`getInsets` die Dimension der Zeichenfläche bestimmt und in lokalen Variablen gespeichert.

2. Der Bildschirm wird mit `clearRect` geleert.
3. Mit `fillOval` wird die Fläche des Ovals in der zuvor festgelegten Farbe Gelb gemalt.
4. Die Farbe wird auf Schwarz umgestellt, um im Anschluss den Rand des Ovals mit `drawOval` zu zeichnen.
5. Mit `drawString` wird ein Text in die Grafik geschrieben.



Abbildung 16.3: Eine kleine Grafik

Linien, Rechtecke und Polygone

Es gibt sehr viele Zeichenmethoden. Fangen wir geradlinig an.

- ✔ Wenn Sie Punkte zeichnen müssen, können Sie sie als Linien mit gleichem Anfangs- und Endpunkt darstellen.
- ✔ `void drawLine(int xAnfang, int yAnfang, int xEnde, int yEnde)`

  Linien werden mit `drawLine` gezeichnet. Die Parameter sind x- und y-Koordinate des Anfangs- und Endpunkts. Die Koordinaten beziehen sich auf

den Nullpunkt links oben.

- ```
  void drawRect(int x, int y, int breite, int hoehe)
  void fillRect(int x, int y, int breite, int hoehe)
  ```

  Ein Rechteck bestimmt sich durch seinen Ausgangspunkt, seine Breite und seine Höhe. Mit der Methode `drawRect` wird die Außenlinie eines Rechtecks gezeichnet. Mit der Methode `fillRect` können Sie hingegen ein Rechteck erzeugen, das mit der zuletzt durch `setColor` definierten Farbe gefüllt ist.

- ```
  void drawRoundRect(int x, int y, int breite, int hoehe,
                     int arcBreite, int arcHoehe)
  void fillRoundRect(int x, int y, int breite, int hoehe,
                     int arcBreite, int arcHoehe)
  ```

  Weil Rechtecke einfach so furchtbar eckig sind, hat man in Java noch die Möglichkeit, Rechtecke für Weicheier zu erzeugen. Diese haben dann runde Ecken, und daran kann man sich auch nicht so leicht verletzen.

  Die letzten beiden Parameter geben an, welche Höhe und Breite die gerundete Ecke haben soll. Natürlich gibt es dazu passend auch die Methode, die die Fläche füllt, und die beginnt erwartungsgemäß mit `fill`.

- ```
  void draw3DRect(int x, int y, int breite, int hoehe, boolean vorn)
  void fill3DRect(int x, int y, int breite, int hoehe, boolean vorn)
  ```

  Recht erhaben wirkt `fill3DRect`. Es entsteht der Eindruck, als wäre das Rechteck von der Ebene nach vorn oder nach hinten aus der Fläche herausgehoben, je nachdem, ob der letzte Parameter auf `true` oder `false` gesetzt ist. Es gibt eine passende `draw`-Methode, die allerdings den Effekt eher verdeckt.

- ```
  void drawPolygon(int[] x, int[] y, int anzahl)
  void fillPolygon(int[] x, int[] y, int anzahl)
  ```

  Für unregelmäßige, geradlinige Figuren können noch Polygone gezeichnet und gefüllt werden. Als Parameter werden je ein Array für die x- und y-Koordinaten übergeben. Der letzte Parameter gibt an, wie viele Elemente jedes Array hat.

Das Programm in Listing 16.7 erzeugt die Rechteckvarianten und ein Polygon.

```java
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Insets;

import javax.swing.JFrame;

public class Geradlinig extends JFrame {
    public static void main(String[] args) {
        new Geradlinig(); // Löse Konstruktor aus!
    }

    // Der Konstruktor bereitet das Fenster vor
    public Geradlinig() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setTitle("Zeichnen mit Java");
        setSize(500, 300);
        setVisible(true);
    }

    @Override
    public void paint(Graphics g) {
        // Ermittle die Größe der nutzbaren Zeichenfläche
        Insets insets = getInsets();
        int links  = insets.left;
        int oben   = insets.top;
        int breite = getWidth()-insets.left-insets.right;
        int hoehe  = getHeight()-insets.top-insets.bottom;
        // Fenster freimachen
        g.clearRect(links, oben, breite-1, hoehe-1);
        // Lege die Variablen für die Viertelung fest
        int dist = 16;
        int x = links + dist;
        int y = oben + dist;
        int w = breite/2 - 2*dist;
        int h = hoehe/2 - 2*dist;

        g.setColor(Color.yellow);
        g.fillRect(x, y, w, h);
        g.setColor(Color.black);
        g.drawRect(x, y, w, h);
        String meldung  = "drawRect/fillRect";
        g.drawString(meldung, x, y);

        x = links + breite/2 + dist;
        g.setColor(Color.pink);
        g.fillRoundRect(x, y, w, h, dist*2, dist*2);
        g.setColor(Color.black);
```

```
        g.drawRoundRect(x, y, w, h, dist*2, dist*2);
        meldung  = "drawRoundRect/fillRoundRect";
        g.drawString(meldung, x, y);

        x = links + dist;
        y = oben + hoehe/2 + dist;
        g.setColor(Color.cyan);
        g.fill3DRect(x, y, w, h, true);
        g.setColor(Color.black);
        meldung  = "draw3DRect/fill3DRect";
        g.drawString(meldung, x, y);

        x = links + breite/2 + dist;
        y = oben + hoehe/2 + dist;
        g.setColor(Color.lightGray);
        int xl = x;
        int xr = links + breite - dist;
        int[] px = { xl, xr, xr, xl };
        int yo = y;
        int yu = oben + hoehe - dist;
        int[] py = { yo, yu, yo, yu };
        g.fillPolygon(px, py, px.length);
        g.setColor(Color.black);
        g.drawPolygon(px, py, px.length);
        meldung  = "drawPolygon/fillPolygon";
        g.drawString(meldung, x, y);
    }
}
```

Listing 16.7: Die Rechteckvarianten und ein Polygon

Nach dem Start erzeugt das Programm in Listing 16.7 ein Fenster wie in Abbildung 16.4.

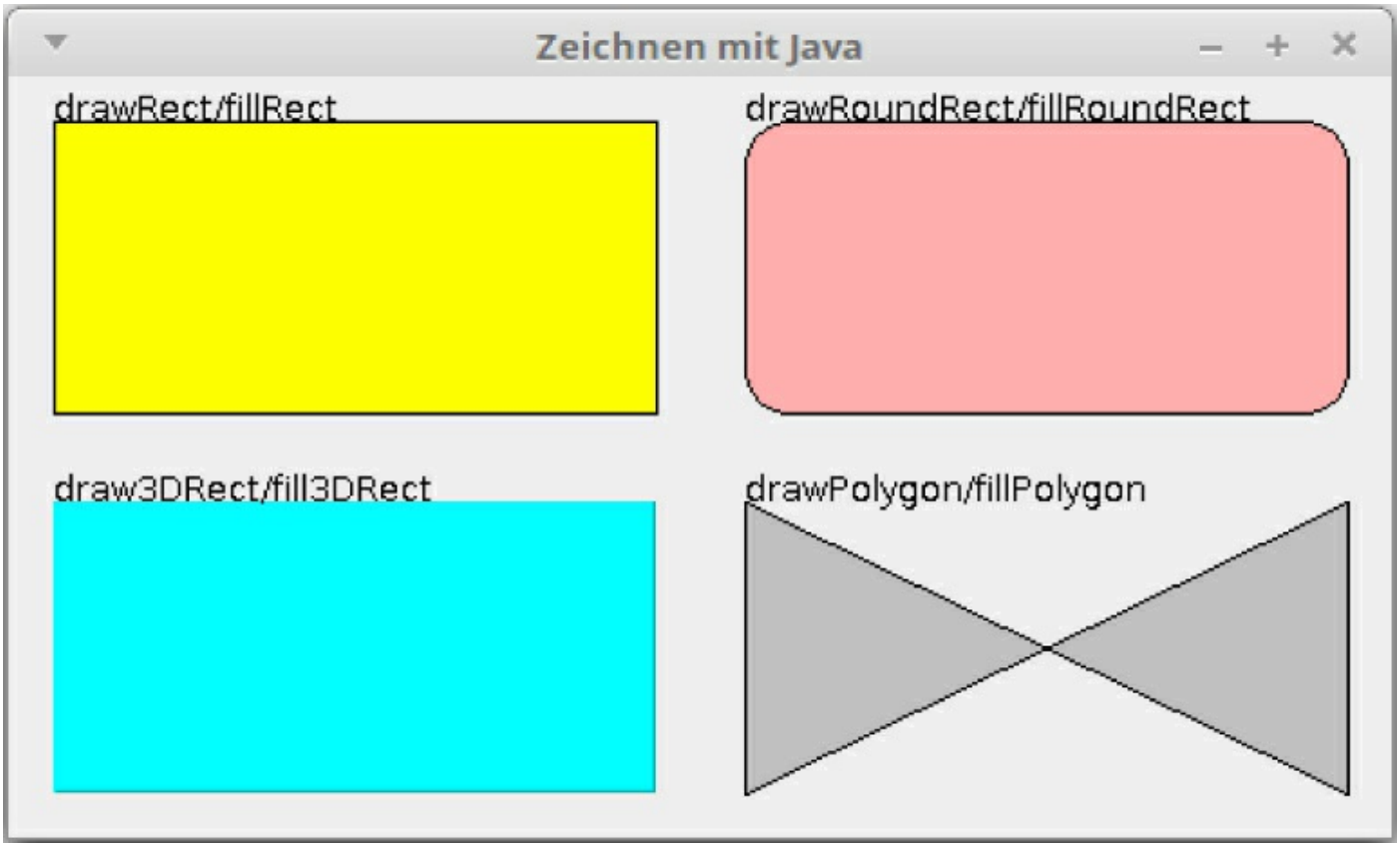


Abbildung 16.4: Eckig und geradlinig

Kreise, Ellipsen, Ovale und Kreisausschnitte

Zum Zeichnen von Kreisen und Ellipsen gibt es nur die Methoden `drawOval` und `fillOval`. Das ist in mehrerer Hinsicht bemerkenswert.

- ✔ Der Aufruf kann tatsächlich kein Oval zeichnen, sondern bestenfalls deren Spezialfälle: eine Ellipse oder einen Kreis. Aber Oval lässt sich einfach leichter tippen als Ellipse.
- ✔ Um eine Ellipse oder einen Kreis zu zeichnen, werden die Koordinaten des umfassenden Rechtecks verwendet. Ist das Rechteck ein Quadrat, wird die Ellipse zum Kreis.
- ✔ Die Parameter sind also zunächst x und y als linke, obere Ecke. Es folgen Breite und Höhe des umschließenden Rechtecks.
- ✔ Wenn Sie einen Kreis um einen Mittelpunkt zeichnen wollen, müssen Sie also x und y um die Hälfte der Breite erhöhen.

Und auch von der Methode zum Zeichnen von Ellipsen gibt es wieder eine unausgefüllte (`drawOval`) und eine ausgefüllte (`fillOval`) Variante. Der Code in Listing 16.8 zeichnet ein schwarzes Rechteck, in das mit den gleichen Parametern erst eine gefüllte gelbe Ellipse und darüber ein roter Umriss gezeichnet wird. Das Ergebnis sehen Sie in Abbildung 16.5.

```
g.setColor(Color.black);
g.drawRect(x, y, breite, hoehe);
g.setColor(Color.yellow);
g.fillOval(x, y, breite, hoehe);
g.setColor(Color.red);
g.drawOval(x, y, breite, hoehe);
```

Listing 16.8: Eine Ellipse in einem Rechteck

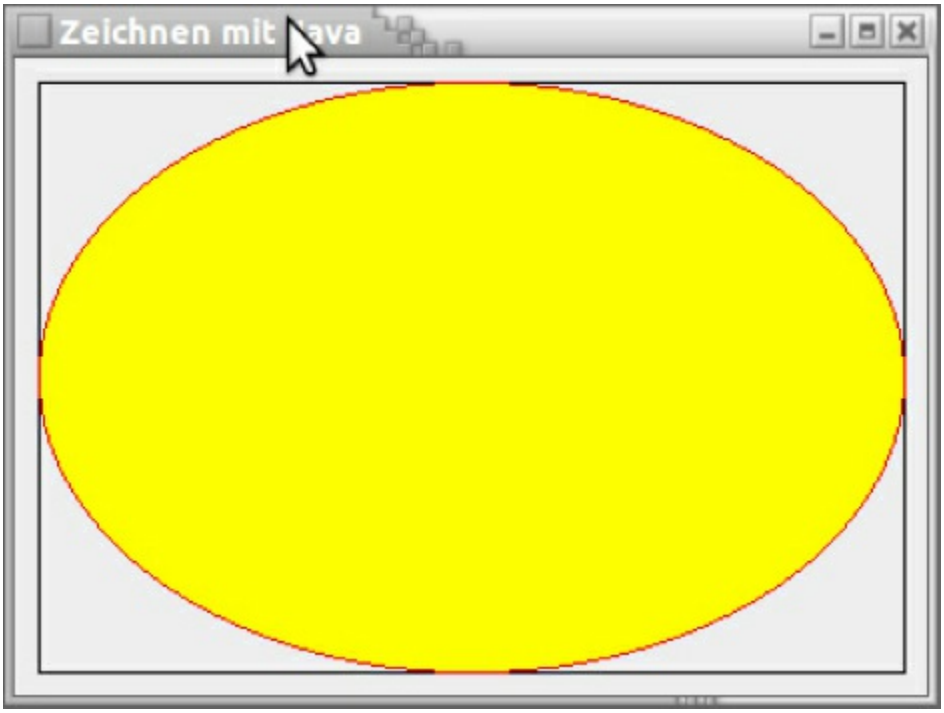


Abbildung 16.5: Abstrakte Kunst?

Das Programm führt zu dem Fenster in Abbildung 16.5.

Tortendiagramme und Kreisausschnitte

Wenn der Kreis nicht rund, sondern kaputt ist, reden wir von einem Kreisausschnitt. Wir bestaunen so etwas unter dem Namen Tortendiagramm bei jeder Wahlsendung, die die Sitzverteilung präsentiert. Wenn Sie das unter Java erstellen wollen, benötigen Sie die Aufrufe `drawArc` oder `fillArc` für die ausgefüllte Variante.

```
void drawArc(int x, int y, int breite, int hoehe,
             int anfWinkel, int ausschnitt)
void fillArc(int x, int y, int breite, int hoehe,
             int anfWinkel, int ausschnitt)
```

Der Unterschied zu `drawOval` liegt in den letzten beiden Parametern. Sie bezeichnen den Anfangswinkel und den Winkel des Ausschnitts, der überstrichen wird. Der Anfangswinkel beginnt mit der 0 an der äußersten rechten Position quasi bei 3 Uhr und bewegt sich entgegen dem Uhrzeigersinn. Ein voller Kreis ist 360 Grad. Na ja, ein Kreis ist es nur, wenn Höhe und Breite gleich sind. Aber das erwähnte ich bereits.

Anstatt ewig die Parteien als Torten darzustellen, könnten wir auf etwas Süßes zurückgreifen. Letzthin fiel mein Blick auf die Frühstückscreme, die ja so gesund ist. Die Zutaten besagten:

- 32 % Fett: Das ist natürlich nötig, damit sich das Zeug ohne Anwendung von Gewalt auf dem Brot verteilen lässt. Die Hauptzielgruppe sind schließlich Kinder.
- 55 % Zucker: Mehr Fett ging einfach nicht. Ansonsten würde die Substanz so flüssig, dass sie vom Brot fließt. Überdies klebt der Zucker auch noch ein wenig.
- 7 % Eiweiß: So etwas kann schon einmal passieren. Wenn man mit Nougat wirbt, müssen Haselnüsse hinein. Und in denen ist nun einmal Eiweiß enthalten.

Zählen Sie die Zahlen kurz zusammen und Sie werden feststellen, dass Fett, Zucker und Eiweiß zusammen keine 100 Prozent ergeben. Die restlichen Inhaltsstoffe mit 6 Prozent sind vermutlich hoch bekömmliche Chemikalien. Wir wollen sie darum vernachlässigen.

- Um eine Verteilungsübersicht zu erstellen, muss man die Zahlen erst auf 100 % hochrechnen. Dazu teilt man jeden Wert durch die Summe.
- Der Gesamtkreis ist 360 Grad. Also müssen wir den jeweiligen Anteil mit 360 multiplizieren. Auf diese Weise müssen nun alle Werte in Winkelbereiche umgerechnet werden.
- Anschließend müssen sie angeordnet werden. Dazu setzt man zuerst den Startwinkel auf 0 und erhöht ihn stückweise um die bereits verbrauchten Winkelbereiche.

So ergibt sich eine stattliche Torte mit weniger Kalorien als dem dargestellten Brotaufstrich (siehe Abbildung 16.6).

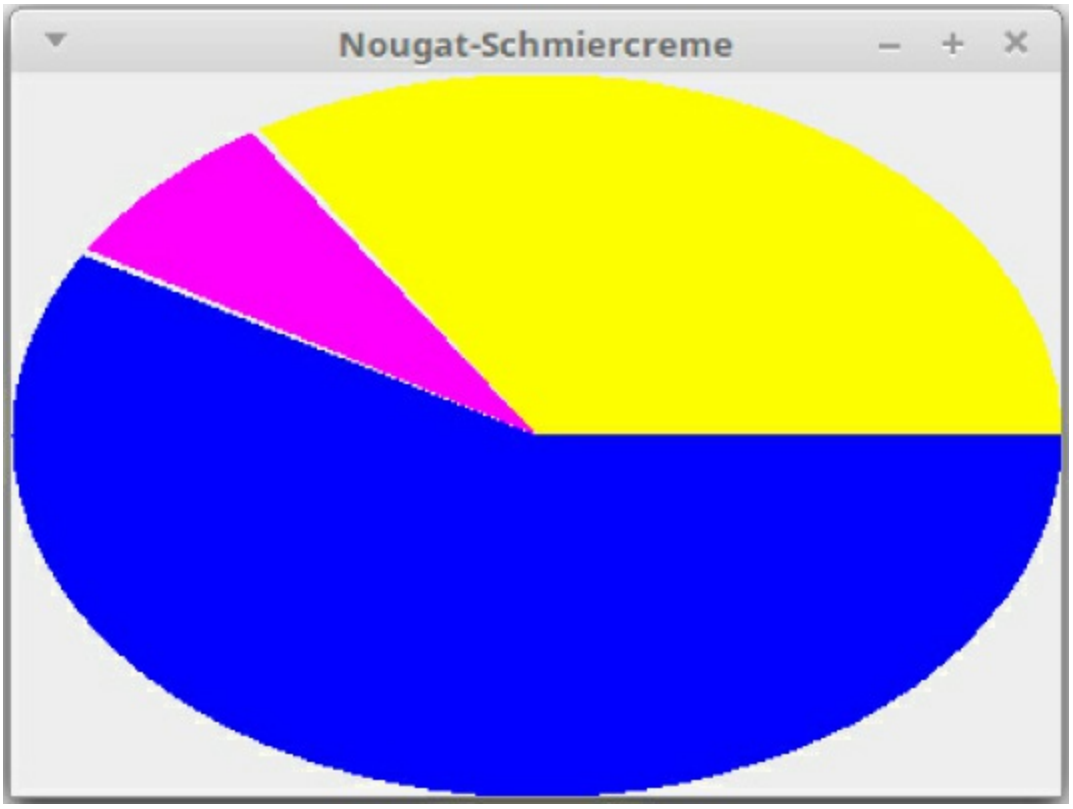


Abbildung 16.6: Fett ist gelb, Zucker ist blau, und Eiweiß ist magenta.

Die Torte in Abbildung 16.6 wurde durch das Programm in Listing 16.9 gebacken.

```
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Nougat extends JFrame {

    public static void main(String[] args) {
        new Nougat();
    }

    public Nougat() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setTitle("Nougat-Schmiercreme");
        setSize(400, 300);
        add(new NougatPanel());
        setVisible(true);
    }

    class NougatPanel extends JPanel {
        @Override
        public void paint(Graphics g) {
            int breite = getWidth();
            int hoehe = getHeight();
            int fett = 32;
            int zucker = 55;
            int eiweiss = 7;
            int summe = fett + zucker + eiweiss;
            final int vollkreis = 360;
            fett = fett * vollkreis / summe;
            zucker = zucker * vollkreis / summe;
            eiweiss = eiweiss * vollkreis / summe;
            g.setColor(Color.yellow);
            g.fillArc(0, 0, breite, hoehe, 0, fett);
            g.setColor(Color.magenta);
            g.fillArc(0, 0, breite, hoehe, fett+1, eiweiss);
            g.setColor(Color.blue);
            g.fillArc(0, 0, breite, hoehe, fett+eiweiss+2,
                    zucker);
        }
    }
}
```

Listing 16.9: Die Nougattorte

Texte malen

Auch Texte können gezeichnet werden. Dazu dient die Methode `drawString`. Als Parameter übernimmt sie den String und die Koordinaten, wo die linke untere Ecke des Strings positioniert werden soll. Das alles überrascht Sie jetzt vermutlich nicht, da in einigen Beispielprogrammen bereits Texte gemalt wurden.

Wie die anderen Zeichenmethoden verwendet auch `drawString` die Farbe, die zuvor durch `setColor` gesetzt wurde. Aber die Farbe ist bei Schriften ja doch meist Schwarz. Viel spannender sind die Schriftarten und -familien.

Font

Ähnlich wie beim Setzen der Farbe wird auch die Schrift vor dem Zeichnen eingestellt und zwar mit der Methode `setFont`. Der Parameter dieser Methode ist vom Typ `Font`, das heißt, um die Schriftattribute zu ändern, benötigen Sie ein Objekt der Klasse `Font`. Da der Font nur zum Setzen des Zeichensatzes verwendet wird, können Sie das `new` direkt in die Parameterklammer setzen und die Schrifteigenschaften an den Konstruktor übergeben.

```
setFont(new Font("Serif", Font.PLAIN, 24);
```

Hier wird eine Schrift ausgewählt, die Serif heißt und 24 Punkt groß ist. Da nach der gegenwärtigen Mode bei den Computerschriften fast nur noch serifenlose Schriften verwendet werden, können wir anhand der Serifen sehen, ob das Setzen der Schriftattribute geklappt hat.

Unter Java sind fünf Schriftarten auf jeden Fall verfügbar:

- ✔ Serif
- ✔ SansSerif
- ✔ Monospaced
- ✔ Dialog
- ✔ DialogInput

Durch den zweiten Parameter des Konstruktors von `Font` wird der Schriftschnitt bestimmt. Dazu bietet die Klasse `Font` statische Konstanten an:

- ✔ `Font.PLAIN`: normal
- ✔ `Font.BOLD`: fett
- ✔ `Font.ITALIC`: kursiv

Diese Werte können ganz einfach mit dem Pluszeichen kombiniert werden, um beispielsweise fett und kursiv zu mischen.

Der letzte Parameter des Konstruktors von `Font` ist die Schriftgröße.

Wenn Sie eines Tages vergessen haben sollten, welcher Font derzeit gesetzt ist, können Sie das mithilfe der Methode `getFont` erfahren. Die Methode gibt eine Referenz auf ein Objekt der Klasse `Font` zurück.

FontMetrics

Für die korrekte Darstellung der Texte benötigt man die Ausdehnung der Zeichenketten auf dem Bildschirm. Wie sollte man ansonsten einen Text mittig darstellen oder die Höhe einer Zeile bestimmen? Diese Informationen liefert die Klasse `FontMetrics`. Um die Dimensionen der aktuell gesetzten Schrift zu ermitteln, rufen Sie die Methode `getFontMetrics` auf.

```
FontMetrics getFontMetrics()
```

Man kann aber auch einen Font als Parameter übergeben und dessen `FontMetrics` ermitteln.

```
FontMetrics getFontMetrics(Font f)
```

Die Klasse `FontMetrics` wiederum stellt Methoden zur Verfügung, um die Maße von dargestellten Zeichenketten zu ermitteln. Mit diesen Vorkenntnissen bewaffnet, können wir nun ein kleines Programm erstellen, das uns diverse Maße des aktuell verwendeten Zeichensatzes anzeigt.

```
import javax.swing.*;
import java.awt.*;

public class Zeichensatz extends JFrame {
    public static void main(String[] args) {
        Zeichensatz fenster = new Zeichensatz();
    }

    public Zeichensatz() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setTitle("Zeichensatzinfo mit Java");
        add(new ZeichensatzPanel());
        setSize(600, 300);
        setVisible(true);
    }

    class ZeichensatzPanel extends JPanel {
        @Override
        public void paint(Graphics g) {
            int breite = getWidth();
            int hoehe = getHeight();
            g.setFont(new Font("Serif", Font.PLAIN, 20));
            // Nun holen wir die Schrift-Informationen
            FontMetrics fm = g.getFontMetrics();
            int zeile = 1; // Zeilenpositionierungszähler
            g.drawString("Fenstergrö ß e: " +
                    breite + " x " + hoehe,
                    0, zeile * fm.getHeight());
            zeile++;
            g.drawString("Zeilenhöhe: " + fm.getHeight(),
                    0, zeile * fm.getHeight());
            zeile++;
            g.drawString("Ascent: " + fm.getAscent(),
                    0, zeile * fm.getHeight());
            zeile++;
            g.drawString("Descent: " + fm.getDescent(),
                    0, zeile * fm.getHeight());
            zeile++;
            g.drawString("Durchschuss: " + fm.getLeading(),
                    0, zeile * fm.getHeight());
            zeile++;
            g.drawString("Breite von \"Willemer\": " +
                    fm.stringWidth("Willemer"),
                    0, zeile * fm.getHeight());
        }
    }
```

```
}
```

Listing 16.10: Informationen über einen Zeichensatz

Wenn Sie dieses Beispielprogramm mit dem Beispiel zum Auslesen der Zeichensätze Ihres Systems mischen, können Sie sich eine irre Flut von Informationen anzeigen lassen. Aber was bedeuten die einzelnen Werte?

- `int getHeight()`

  Die Methode liefert den Abstand zwischen zwei Zeilen. Im Beispiel wird dieser Wert mit der Zeilennummer multipliziert, um die nächste Zeile zu finden.
- `int getAscent()`

  Die Methode liefert den Abstand von der Grundlinie zur Oberlänge der Zeichen.
- `int getDescent()`

  Die Methode liefert den Abstand von der Grundlinie zur Unterlänge der Zeichen.
- `int getLeading()`

  Die Methode liefert den unbedruckten Abstand zwischen zwei Zeilen, den sogenannten Durchschuss.
- `int stringWidth(String str)`

  Die Methode ermittelt, wie viel Platz ein String in der Breite benötigt. Damit ist die horizontale Positionierung von Texten möglich.

Der Umgang mit Bildern und Images

Bilder werden in Java Images genannt. Auch diese können Sie in der Methode `paint` auf den Bildschirm zaubern. Sie können für Diashows oder für die Spielfiguren eines Computerspiels verwendet werden.

Sie könnten auf jedem Ihrer Programme das Konterfei Ihres Erbonkels darstellen. Vielleicht hilft's. Wenn nicht, müssen Sie Ihren Lebensunterhalt vielleicht doch mit Programmieren verdienen. Aber auch dann könnte es hilfreich sein, zu wissen, wie man in Java mit Bildern umgeht.

Zeichnen per drawImage

Die Methode `drawImage` zeichnet ein Image in den Grafikkontext `Graphics` und hat die folgenden Parameter:

- `Image`: Dies ist das Bild.
- `int`: Die x-Koordinate
- `int`: Die y-Koordinate
- `int`: (optional) Wenn Sie eine Breite angeben, wird das Bild auf diese skaliert.

- ✔ `int`: (optional) Analog können Sie die Höhe zur Skalierung angeben.
- ✔ `ImageObserver`: Wird meist auf `this` gesetzt.

Die Methode `drawImage` ist in der Lage, transparente Bestandteile eines Images so zu zeichnen, dass der Hintergrund in diesen Bereichen nicht überdeckt wird. Im Programm erfolgt der Aufruf in der Methode `paint`:

```
import java.awt.Image;

    // ...
    Image img = null;
    // ...
        // im Konstruktor
        img = // ...
    // ...

    @Override
    public void paint(Graphics g)  {
       // ...
       g.drawImage(img, xPos, yPos, this);
```

Listing 16.11: Aufruf von `drawImage`

Vor dem Zeichnen muss also ein Objekt der Klasse `Image` angelegt werden, das gezeichnet werden kann.

Erzeugung eines Objekts der Klasse Image aus einer Datei

Zentrum unserer Bemühungen wird darum ein Objekt der Klasse `Image` sein. Die Grafik befindet sich typischerweise zunächst in einer Bilddatei. Diese muss in ein `Image` geladen werden.

Im Toolkit von Java finden Sie eine Methode `getImage`, die eine Bilddatei in ein `Image` überführen kann.

```
Toolkit toolkit = Toolkit.getDefaultToolkit();
Image img = toolkit.getImage(dateiName);
```

Listing 16.12: Umwandlung über das Toolkit

Sie können aber auch ein `ImageIcon` erzeugen, dem Sie im Konstruktor den Namen der Bilddatei übergeben. Über die Methode `getImage` können Sie dann das `Image` herausholen.

```
ImageIcon icon = new ImageIcon(dateiName);
Image img = icon.getImage();
```

Listing 16.13: Umwandlung über `ImageIcon`

Dieser Aufruf wird das `Image img` an der Position darstellen, die durch `xPos` und `yPos` bestimmt ist. Das folgende Programm stellt ein Bild dar, das Sie über einen Dateiauswahldialog auswählen.

```
import javax.swing.*;
import java.awt.*;

public class BildAnzeige extends JFrame {
    Image img = null;
    String dateiName = null;
    public BildAnzeige() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setTitle("Bildanzeige mit Java");
        setSize(400, 300);
        setVisible(true);
        // Dateiauswahl bei Fensterstart
        JFileChooser openDialog = new JFileChooser();
        int wahl = openDialog.showOpenDialog(this);
        if (wahl==JFileChooser.APPROVE_OPTION) {
            // Dateiname und Image festhalten
            dateiName = openDialog.getSelectedFile()
                                  .getAbsolutePath();
            img = Toolkit.getDefaultToolkit()
                         .getImage(dateiName);
            repaint(); // neuzeichnen erzwingen
        }
    }
```

```
    @Override public void paint(Graphics g) {
        // Zeichenausschnitt ermitteln
        Insets insets = getInsets();
        int links  = insets.left;
        int oben   = insets.top;
        int breite = getSize().width-insets.left-insets.right;
        int hoehe  = getSize().height-insets.top-insets.bottom;
        g.clearRect(links, oben, breite-1, hoehe-1);
        // Bild anzeigen
        if (img!=null) {
            g.drawImage(img, links, oben, this);
        }
    }

    public static void main(String[] args) {
        new BildAnzeige();
    }
}
```

Listing 16.14: Bildbetrachter

Das Programm öffnet zunächst eine Dateiauswahlbox. Damit wählen Sie sich eines Ihrer Lieblingsbilder aus. Es wird in ein Image überführt und ein Neuzeichnen ausgelöst. In der Methode `paint` wird dann dieses Image durch die Methode `drawImage` dargestellt.



Abbildung 16.7: Ein kleiner Bildbetrachter

Bilder aus dem Projekt

In manchen Fällen wollen Sie aber Bilder vom Programm her bereits zur Verfügung stellen. Idealerweise werden sie dann auch mit der JAR-Datei ausgeliefert. Dazu müssen die Bilder natürlich einen festen Ort im Projekt bekommen.

1. Die Bilder müssen Sie im Bereich Ihrer Quelltexte anlegen. Bei Eclipse ist dies das Projektverzeichnis *src*.
2. Dort erzeugen Sie sinnvollerweise ein eigenes Verzeichnis in Form eines Packages für Ihre Bilder, beispielsweise mit dem Namen *images*.
3. Hier kopieren Sie mit der Maus das gewünschte Bild hinein. Im Beispiel ist dies *schiff.png*.
4. Im Programm erhalten Sie die URL auf die Ressource der Klasse folgendermaßen:
   ```
   URL url = getClass().getResource("/images/schiff.png");
   ```
5. Über diese URL können Sie ein `ImageIcon` erzeugen und daraus das Image gewinnen:

```
Image img = new ImageIcon(url).getImage();
```

Dieses sollte sowohl aus dem Projekt als auch aus der JAR-Datei funktionieren.

Skalieren

Wenn Sie ein Bild einer digitalen Kamera mit dem Bildbetrachter laden, werden Sie feststellen, dass Sie vermutlich nur die linke obere Ecke des Fotos sehen können. Das wird auch nicht sehr viel besser, wenn Sie das Fenster aufspannen. Die meisten Fotoapperate liefern heutzutage mehr Auflösung, als Ihr Monitor darstellen kann.

Sie können `drawImage` durch Angabe der Breite und der Höhe des Bildschirmausschnitts dazu zwingen, das Bild in den freien Bereich des Fensters zu quetschen. Dazu ermitteln Sie Breite und Höhe des Zeichenbereichs über die Insets oder über `getWidth` und `getHeight` des Panels.

Allerdings kommt es dann zu Verzerrungen, wenn das Fensterformat nicht zufällig mit denen des Fotos übereinstimmt. Zum Glück bietet Ihnen die Klasse `Image` ebenfalls die Methoden `getWidth` und `getHeight` an, sodass Sie nun dafür sorgen können, bei Vergrößerungen das Bild verzerrungsfrei darzustellen.

Bildausschnitt kopieren

In manchen Fällen wollen Sie die Grafik nicht aus einer Bilddatei erzeugen, sondern einen Ausschnitt des Fensters an eine andere Stelle kopieren. Dabei hilft Ihnen die Methode `copyArea`. Die ersten vier Parameter beschreiben das Rechteck, das kopiert werden soll. Die letzten beiden sind die Koordinaten des Ziels.

```
void copyArea(int x, int y, int breite, int hoehe, int dx, int dy)
```

Wenn Sie das Programm `BildAnzeige` in Listing 16.14 nach dem Aufruf von `drawImage` um die folgenden Zeilen ergänzen, erhalten Sie das Bild, das Sie in Abbildung 16.8 sehen.

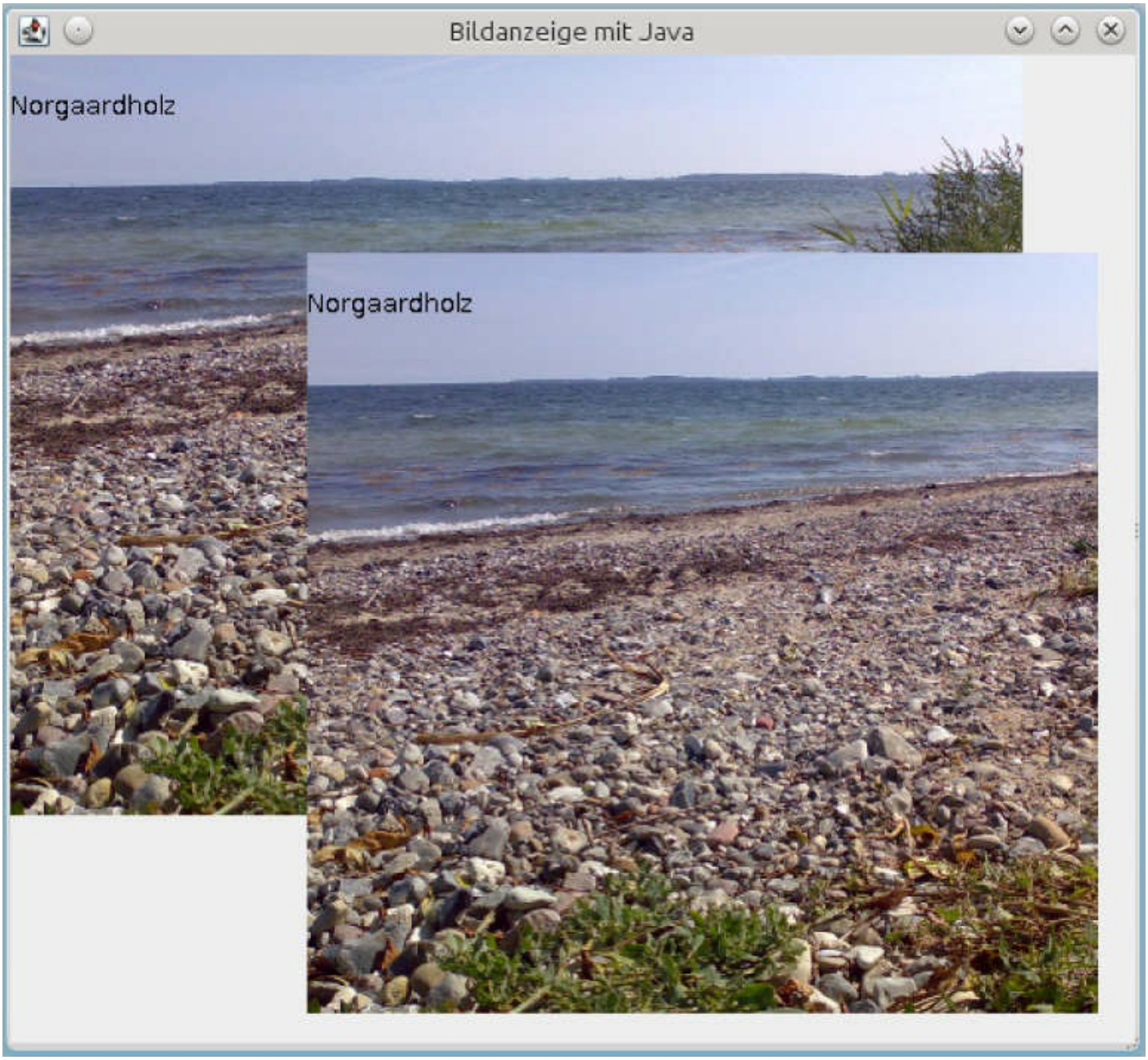


Abbildung 16.8: Bildausschnitt kopieren

```
g.setColor(Color.BLACK);
g.drawString("Norgaardholz", links, oben+30);
g.copyArea(links, oben, 400, 400, 150, 100);
```

Am Schriftzug »Norgaardholz« können Sie erkennen, dass nicht nur das Bild, sondern alles, was in dem Bereich dargestellt ist, kopiert wurde.

Clipping

Clipping ist eine Technik, die etwa so funktioniert wie das Abkleben mit Kreppklebeband, wenn Sie zu Hause mit Farbe Ihre Fensterrahmen streichen und später dennoch durch die Glasscheiben schauen wollen. Beim Programmieren ist es meist genau anders herum, denn hier geben Sie einen Rahmen vor, in dem das Programm zeichnen darf. Alle Zeichnungen, die aus diesem Rahmen herausragen, werden abgeschnitten.

Der Programmierer könnte natürlich alternativ ausrechnen, ob sich die gemalte Figur außerhalb des Zielgebiets ausdehnt. In vielen Fällen ist der Einsatz von Clipping viel einfacher. Die Methode `setClip` sorgt für das Setzen der Grenzen. In seiner einfachsten Form übergibt man der Methode ein Rechteck.

```
void setClip(int x, int y, int breite, int hoehe)
```

Den Clipping-Bereich kann man weiter verkleinern. Mit der Methode `clipRect` schneidet man den bisher gültigen Clipping-Bereich mit dem durch die Parameter angegebenen Rechteck.

```
void clipRect(int x, int y, int breite, int hoehe)
```

Wenn Sie vor dem ersten `drawImage` in das Programm `BildAnzeige` in Listing 16.14 den folgenden Befehl einbauen, wird der Ausgabebereich auf das angegebene Rechteck beschränkt.



Abbildung 16.9: Clippingbereich setzen

```
g.setClip(30, 30, 240, 120);
```

Das Ergebnis sähe dann so aus wie in Abbildung 16.9. Sie erkennen, dass nur der freigegebene Bereich angezeigt wird. Sogar der Textanfang ist abgeschnitten.

Die Methode `getClipBounds` ermittelt den Clipping-Rahmen.

```
Rectangle getClipBounds()
```

Das Ergebnis ist eine Referenz auf ein Objekt der Klasse `Rectangle`. Dessen Attribute `x`, `y`, `width` und `height` können direkt ausgelesen werden und ergeben den vom Clipping-Bereich umspannten Bereich.

Sie können auch vor dem Zeichnen prüfen, ob ein Rechteck innerhalb des Clipping-Bereichs liegt, indem Sie die Methode `hitClip` aufrufen.

```
boolean hitClip(int x, int y, int breite, int hoehe)
```

Mit der `Graphics2D`-Bibliothek kommt das Thema Clipping im nächsten Abschnitt

noch einmal zum Tragen, denn dort sind noch viel mehr Möglichkeiten verfügbar.

Modernisierung: Graphics2D

Die Grafikfähigkeiten in Java sind durch die `Graphics2D`-Bibliothek erweitert, wenn nicht revolutioniert worden.

- Statt mit dem Pinsel nacheinander Figuren auf die Leinwand zu pinseln, sind die Zeichenprimitive nun Objekte, die die Klasse `Shape` erweitern.
- Statt Methoden für jeden Grafiktyp (`drawRect`, `drawOval`) gibt es nun die Methoden `draw` und `fill`, denen das zu zeichnende Grafikobjekt übergeben wird.
- `Graphics2D` abstrahiert von den Bildpixeln der Grafikhardware, was zur Folge hat, dass die Koordinaten nicht mehr vom Typ `int`, sondern `double` sind.



Die alte Schnittstelle wurde so übernommen, dass alle bisherigen Methoden mit 2D harmonisieren. Neu an 2D ist die Betrachtung der Zeichenprimitive als eigenständige Objekte.

Wenn Sie Graphics2D benutzen möchten, casten Sie in `paint` den übergebenen `Graphics`-Parameter auf eine `Graphics2D`-Referenz.

```
@Override
public void paint(Graphics pg) {
    Graphics2D g = (Graphics2D) pg;
```

Über diese Referenz können Sie nun alle bisherigen Graphics-Methoden als auch die neuen 2D-Methoden aufrufen.

Die auffälligste Neuerung ist, dass es Klassen der geometrischen Elemente wie Linien, Kreise und Rechtecke gibt, die den überladenen Methoden `draw` oder `fill` übergeben werden. Das passt natürlich besser zu einer objektorientierten Sprache und ermöglicht, mit den geometrischen Figuren zu rechnen, um beispielsweise Schnittpunkte zu ermitteln. Spätestens hier machen sich die Koordinaten vom Typ `double` bezahlt.

```
Rectangle2D rect = new Rectangle2D.Double(x, y, breite, hoehe);
g.setColor(Color.yellow);
g.fill(rect);
g.setColor(Color.black);
g.draw(rect);
```

Listing 16.15: Zeichnen eines gefüllten Rechtecks mit Rand


Die 2D-Erweiterung erstellt erst die grafischen Figuren und zeichnet sie dann.

Aber was ist denn das für ein `Double`, das da am `Rectangle2D` klebt? `Rectangle2D` ist eine abstrakte Klasse. Die implementierenden Klassen heißen eben `Rectangle2D.Float` oder `Rectangle2D.Double`, die eine unterschiedliche Genauigkeit der Positionierung erlauben.

Neu ist, dass Positionen und Dimensionen nicht mehr ganzzahlig in den Pixeln ihrer Grafikhardware angegeben werden. Stattdessen werden Fließkommazahlen verwendet. Das ist sinnvoll, weil die Zeichenumgebung ja erst später auf die Grafikhardware umgerechnet wird und die Auflösung eines Monitors beispielsweise sehr viel geringer ist als die eines Druckers.

- `Point2D`: Erstellt einen Punkt mit den Koordinaten `x` und `y`.
  ```
  Point2D punkt = new Point2D.Double(double x, double y);
  ```
- `Line2D`: Eine Linie kann entweder von den Koordinaten `xVon`,`yVon` zu `xBis`,`yBis` verlaufen oder durch ihren Start- und Endpunkt definiert werden.
  ```
  Line2D linie = new Line2D.Double(double xVon, double yVon,
                                   double xBis, double yBis);
  Line2D linie = new Line2D.Double(Point2D von, Point2D bis);
  ```
- `Rectangle2D`: Ein Rechteck wird durch `x`, `y`, `breite` und `hoehe` festgelegt:
  ```
  Rectangle2D rechteck = new Rectangle2D.Double(x,y,breite,hoehe);
  ```
- `Ellipse2D`: Offenbar hat einer der Entwickler im Wikipedia gelesen, dass ein Oval etwas anderes ist als eine Ellipse. Die Position und Größe der Ellipse wird durch die des umschließenden Rechtecks definiert.
  ```
  Ellipse2D ellipse = new Ellipse2D.Double(x,y,breite,hoehe);
  ```

Die grafischen Objekte können nicht nur gezeichnet werden. Beispielsweise ermittelt die Methode `Line2D.intersectsLine`, ob sich die Linie mit einer anderen als Parameter übergebenen Linie schneidet. Für ein Rechteck oder eine Ellipse kann durch die Methode `contains` ermittelt werden, ob ein Punkt oder ein anderes Objekt enthalten ist.

Selbst erstellte Figuren

Sie können auch eigene geometrische Figuren bilden, die als `Area` bezeichnet werden.

- Sie erzeugen eine `Area`, indem Sie dem Konstruktor ein 2D-Objekt als Parameter übergeben.
- Sie können der `Area` über die Methode `add` eine andere `Area` hinzufügen, die dann zu einer neuen `Area` verschmilzt.

So kann eine Wappenform durch das Verschmelzen eines Rechtecks mit einem Kreis erzeugt werden. Listing 16.16 demonstriert dies, füllt das Wappen gelb und umgibt es mit einer schwarzen Linie (siehe Abbildung 16.10).

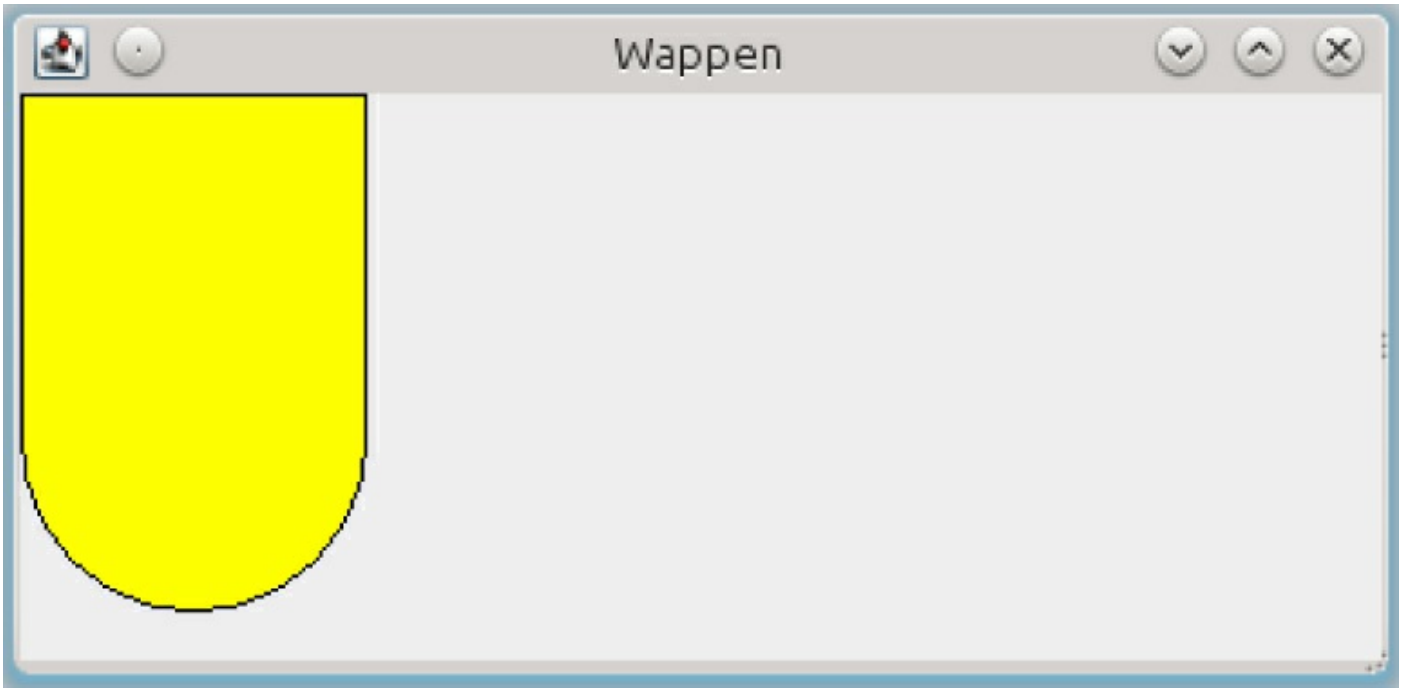


Abbildung 16.10: Eine `Area` für ein Wappen

```
import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;

public class Wappen extends JFrame {

    public static void main(String[] args) {
        Wappen fenster = new Wappen();
        fenster.setTitle("Wappen");
        fenster.setSize(400, 300);
        fenster.setVisible(true);
    }

    public Wappen() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
```

```
    @Override public void paint(Graphics pg) {
        Graphics2D g = (Graphics2D) pg;
        // Ermittle den Zeichenbereich
        Insets insets = getInsets();
        int links  = insets.left;
        int oben   = insets.top;
        int breite = getSize().width-insets.left-insets.right;
        int hoehe  = getSize().height-insets.top-insets.bottom;
        // Definiere eine Area als Rechteck
        Area area = new Area(
                new Rectangle2D.Double(links,oben,100,100));
        // Füge einen Kreis hinzu, sodass er unten zur Hälfte
        // herausschaut.
        area.add(new Area(
                new Ellipse2D.Double(links,oben+50,100,100)));
        // Nun kann das Gebilde gezeichnet werden
        g.setColor(Color.yellow);
        g.fill(area);
        g.setColor(Color.black);
        g.draw(area);
    }
}
```

Listing 16.16: Ein Wappen aus Rechteck und Kreis

- Es ist möglich, die `Area` an die Methode `clip` zu übergeben, woraufhin alle weiteren Zeichnungen außerhalb des Wappens unterbunden werden.
- Die `add`-Methode im Listing führt eine Vereinigung der Figuren herbei.
- Mit `substract` kann eine Figur von einer anderen abgeschnitten oder ausgestanzt werden.
- Mit der Methode `intersect` wird die Schnittmenge der Figuren erzeugt.

Mit ein wenig Übung können Sie sich Schablonen basteln und diese zum virtuellen Paintbrushing verwenden. Dazu fehlen Ihnen jetzt nur noch die schicken Farbverläufe. Da ist mit `setColor` noch lange nicht Schluss.

Farbverläufe und Strichstärke

Wie bisher kann die Methode `setColor` verwendet werden, um die Farbe der nachfolgenden Zeichenvorgänge festzulegen. `Graphics2D` bietet neben `Color` auch die Möglichkeit von `Paint`, das auch in der Lage ist, Farbverläufe zu erzeugen. Der folgende Code-Schnipsel erzeugt einen Farbverlauf von Blau nach Grün.

```
g.setPaint(new GradientPaint(
        0, 0, Color.blue, 50, 25, Color.green, true));
Rectangle2D rect = new Rectangle2D.Double(0, 0, w, h);
g.fill(rect);
```

Die Methode `setPaint` erwartet einen Parameter vom Typ `Paint`. `Paint` ist ein Interface, dass insbesondere durch die Klasse `GradientPaint` implementiert wird. Der Konstruktor verarbeitet folgende Parameter:

- Die ersten drei Parameter beschreiben einen Punkt an der Position 0,0, dessen Farbe Blau ist.
- Die nächsten drei Parameter beschreiben einen grünen Punkt an der Position 50,25.
- Der letzte Parameter gibt an, ob sich der Farbverlauf zyklisch wiederholen, also ständig zwischen Blau und Grün wechseln soll.

Damit ist ein sich wiederholender Farbverlauf von Blau nach Grün mit einer steilen Diagonalen definiert.

`Graphics2D` stellt auch `Stroke` zur Verfügung, mit dem die Dicke eines Striches verändert wird. Durch den folgenden Aufruf wird die Strichstärke 5. Damit können Sie richtig fett durchstreichen.

```
g.setStroke(new BasicStroke(5));
```

Das folgende Listing demonstriert diese Möglichkeiten.

```
import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;

public class FarbUebergang extends JFrame {

    public static void main(String[] args) {
        FarbUebergang fenster = new FarbUebergang();
        fenster.setTitle("Farbübergang");
        fenster.setSize(400, 300);
        fenster.setVisible(true);
    }

    public FarbUebergang() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public void paint(Graphics pg) {
        Graphics2D g = (Graphics2D)pg;
        double links = 10.0; double oben = 30.0;
        double breite = 100.0; double hoehe = 80.0;
        g.setStroke(new BasicStroke(10));
        Rectangle2D rect = new Rectangle2D.Double(links,
                oben, breite, hoehe);
        g.setColor(Color.red);
        g.draw(rect);
        g.setPaint(new GradientPaint(0, 0, Color.blue,
                50, 25, Color.green, true));
        g.fill(rect);
    }
}
```

Listing 16.17: Ein sich wiederholender Farbübergang

Abbildung 16.11 zeigt, wie der Farbübergang aussieht.

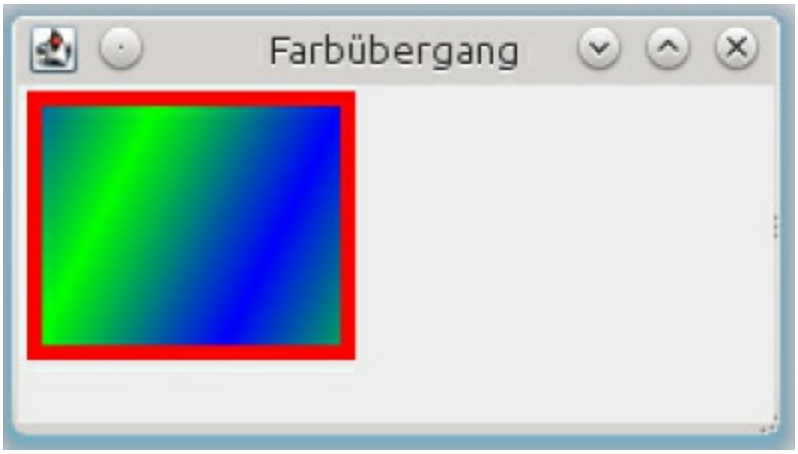


Abbildung 16.11: Farbübergang

Falls Sie nun fürchten, dass Sie plötzlich farbenblind seien: Keine Sorge, das liegt daran, dass der Buchinhalt nicht bunt ist. Programmieren Sie es selbst und Sie können die ganze Farbenpracht genießen.

Ereignisreichtum

Die Programmierung einer grafischen Oberfläche besteht aus zwei Phasen. Zunächst wird das Fenster zusammengesetzt, schön gemacht und gestartet. Und dann steht es dämlich herum, bis der Anwender anfängt, das Fenster mit Mausklicks und Tastendrücken zu bombadieren. Diese Ereignisse fängt das Programm und verändert das Fenster in der Art, dass der Benutzer damit zufrieden ist.

Das Fenster schließt sich

Die einfachste Methode, das Programm nach einem Klick auf den Schließbutton zu beenden, besteht darin, diese Aufgabe einfach `JFrame` in die Schuhe zu schieben. Früher nannte man es Faulheit, heute nennen Unternehmensberatungen es Delegieren. Freundlicherweise bietet `JFrame` dafür die Methode `setDefaultCloseOperation` an, der als Konstante `EXIT_ON_CLOSE` übergeben wird.

```
import javax.swing.*;
public class MeinJFrame extends JFrame {

    MeinJFrame() {
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(400, 300);
        this.setVisible(true);
    }
    public static void main(String[] args) {
        new MeinJFrame();
    }
}
```

Listing 16.18: Die Schlieox beendet das Programm.

Ereignis selbst bearbeiten

Die Zeile drückt aus, dass die Schließoperation von `JFrame` die Methode `exit` aufrufen soll. Der Aufruf von `exit` kann überall erfolgen und beendet das Programm. Nun zickt das Programm beim Beenden nicht mehr. Das ist schon nicht schlecht, aber besser wäre es, wenn das Programm zuvor über das Ende informiert wird, damit es seine Daten sichern kann.

Ein Stück Programm-Code muss also dazwischengeschaltet werden. Erst einmal müssen wir informiert werden, wenn der Anwender die Schließbox anklickt, und dann müssen wir unsere eigenen Anweisungen ablaufen lassen. Wir haben zwar noch keine Daten, die wir sichern könnten, aber wir wollen erst einmal selbst `exit` aufrufen. Später könnten wir an dieser Stelle dann den Benutzer fragen, ob er seine Arbeit wirklich in die Tonne treten will.

Lauschangriff

Um ein Ereignis zu empfangen, müssen zwei Dinge passieren:

- Dem Fenster muss mitgeteilt werden, dass das Programm an dem Ereignis interessiert ist. Diesen Wunsch meldet es mit der Methode `addWindowListener` an. Der Aufruf passt natürlich am besten in den Konstruktor. So weiß das Fenster schon, wie es sterben wird, noch bevor es richtig da ist. Manchmal haben Programmierer wohl doch eine Neigung zum Sadismus.
- Als Parameter erhält die Methode einen Listener, also einen Lauscher. Der Parametertyp ist das Interface `WindowListener`. Eine Implementierung von `WindowListener` ist der `WindowAdapter`.
    - Das Interface hat den Vorzug, dass die Fensterklasse selbst die Implementierung übernehmen kann. Da sie `JFrame` erweitert, kann sie nicht auch `WindowsAdapter` erweitern. Dagegen kann sie

`WindowsListener` implementieren, muss dann allerdings alle Methoden aufführen, auch wenn sie sich nur für eine interessiert.

```
import java.awt.event.*;
import javax.swing.*;
public class MeinJFrame extends JFrame
                        implements WindowListener {

    public MeinJFrame() {
        this.addWindowListener(this);
        this.setSize(400, 300);
        this.setVisible(true);
    }
    public static void main(String[] args) {
        new MeinJFrame();
    }
    @Override
    public void windowClosing(WindowEvent e) {
        this.dispose();
        System.exit(0);
    }
    @Override
    public void windowOpened(WindowEvent e) {
    }
    @Override
    public void windowClosed(WindowEvent e) {
    }
    @Override
    public void windowIconified(WindowEvent e) {
    }
    @Override
    public void windowDeiconified(WindowEvent e) {
    }
    @Override
    public void windowActivated(WindowEvent e) {
    }
    @Override
    public void windowDeactivated(WindowEvent e) {
    }
}
```

- Vom `WindowAdapter` kann eine Instanz gebildet werden, die lediglich die

Methode `windowClosing` überschreibt.

```
import java.awt.event.*;
import javax.swing.*;
public class MeinJFrame extends JFrame {

    public MeinJFrame() {
        this.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                dispose();
                System.exit(0);
            }
        });
        this.setSize(400, 300);
        this.setVisible(true);
    }
    public static void main(String[] args) {
        new MeinJFrame();
    }
}
```

Das ist nun etwas arg kurz geraten. Sie werden vielleicht erkennen, dass im Parameter von `addWindowListener` ein Objekt von `WindowAdapter` angelegt und gleich durch die offene geschweifte Klammer implementiert wird, inklusive des Überschreibens ihrer Methode `windowClosing`.

Vielleicht ist eine Langversion verständlicher. Das folgende Programm tut exakt das Gleiche, legt aber eine lokale Klasse an, die den `WindowAdapter` beerbt und `windowClosing` überschreibt. Davon wird ein Objekt angelegt und dieses an `addWindowListener` übergeben.

```
import javax.swing.*;
import java.awt.event.*;
public class MeinClosedFrame extends JFrame {

    class MeinWindowAdapter extends WindowAdapter {
        @Override
        public void windowClosing(WindowEvent ev) {
            System.exit(0); // macht das Programm kaputt
        }
    }

    MeinClosedFrame() {
        MeinWindowAdapter meinAdapter=new MeinWindowAdapter();
        addWindowListener(meinAdapter);
        this.setSize(400, 300);
        this.setVisible(true);
    }
    public static void main(String[] args) {
        new MeinClosedFrame();
    }
}
```

Bevor wir diesen Abschnitt abschließen, noch eins: Vielleicht haben Sie es schon gemerkt: Die Ereignisse, auf denen Swing basiert, stammen aus dem Paket `java.awt.event`, also vom AWT.

```
import java.awt.event.*;
```



AWT war die erste Version der grafischen Oberfläche von Java. Schick war irgendwie anders und etwas mager war das Ganze auch. Aber einige Basisfunktionalitäten waren durchweg in Ordnung. Warum sollte man sie wegwerfen? Außerdem neigt Java dazu, die damaligen Programmierer nicht im Regen stehen zu lassen. Sie könnten also auch heute noch reine AWT-Programme schreiben. Darum werden Sie auf die Abkürzung AWT noch häufiger stoßen.

Mausige Ereignisse

Fangen wir die Maus. Oder zumindest deren Ereignisse. »Moment mal!«, mögen Sie vielleicht einwerfen: »Das Klicken des Schließbuttons wurde doch durch die Maus ausgelöst!« Richtig!

- ✔ Klickt der Anwender die Fensterelemente an, kümmert sich das Fenster darum.
- ✔ Klickt er irgendwelche Kontrollelemente wie beispielsweise Buttons an, reagieren diese selbst.
- ✔ Klickt der Anwender in den freien Raum, reagiert niemand. Das ist nicht weiter tragisch, sofern dort nichts passiert. Erst wenn darin Aliens gemalt werden, die

sich bewegen und totgeklickt werden müssen, dann sollten Sie sich dafür interessieren.

Viele Programme beobachten die Maus gar nicht weiter. Nur wenn Sie die Mausaktionen auf dem freien Fenster verfolgen wollen, ist dieser Abschnitt für Sie überhaupt von Interesse.

Wenn sich das Programm selbst um die Maus kümmern möchte, weil es beispielsweise mit der Maus malen will, muss ein entsprechender Listener eingesetzt werden. Ja, es gibt da ein paar Ähnlichkeiten zum Schließen des Fensters. Der `MouseListener` arbeitet letztlich ganz ähnlich wie der `WindowListener` aus dem vorigen Abschnitt.

Sommersprossen per Mausklick

Als Beipiel soll das Programm `MausKlicker` bei jedem Mausklick eine Sommersprosse im Fenster erscheinen lassen. Warum sollen Fenster keine tragen dürfen?

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MausKlicker extends JFrame implements MouseListener {

    public static void main(String[] args) {
        MausKlicker fenster = new MausKlicker();
        fenster.setTitle("Sommersprossen mit Java");
        fenster.setSize(400, 300);
        fenster.setVisible(true);
    }
```

```
    public MausKlicker() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        addMouseListener(this);
    }

    int posX; // X-Koordinate der Mausposition
    int posY; // Y-Koordinate der Mausposition

    public void paint(Graphics g) {
        g.setColor(Color.RED);
        g.fillOval(posX-4, posY-4, 8, 8);
    }

    @Override
    public void mouseClicked(MouseEvent e) {
        posX = e.getX();
        posY = e.getY();
        repaint();
    }

    @Override
    public void mousePressed(MouseEvent e) {
    }
    @Override
    public void mouseReleased(MouseEvent e) {
    }
    @Override
    public void mouseEntered(MouseEvent e) {
    }
    @Override
    public void mouseExited(MouseEvent e) {
    }
}
```

Listing 16.19: Klick dir deine Sommersprossen!

- Das Programm `MausKlicker` implementiert den `MouseListener`.
- Der `MouseListener` wird aktiv, weil die Klasse über die Methode `addMouseListener` eingebunden wurde.
- Dabei interessiert sich das Programm für die Methode `mouseClicked`, die durch das Klicken der Maus aktiviert ist.

- Sobald die Methode ausgelöst wurde, wird die aktuelle Mausposition ermittelt. Dies erreicht man durch Aufruf der Methoden `getX` und `getY` der Parametervariablen der Klasse `MouseEvent`.
- Die Mausposition wird in den Variablen `posX` und `posY` gespeichert.
- Per Aufruf von `repaint` wird ein Neuzeichnen ausgelöst, was indirekt zum Aufruf der Methode `paint` führt.
- Darin wird an der durch `posX` und `posY` bezeichneten Stelle ein kleiner roter Kreis mit dem Durchmesser von 8 Punkten gezeichnet. Damit der Kreis um den Mittelpunkt herum wächst, muss dieser um 4 Punkte korrigiert werden.

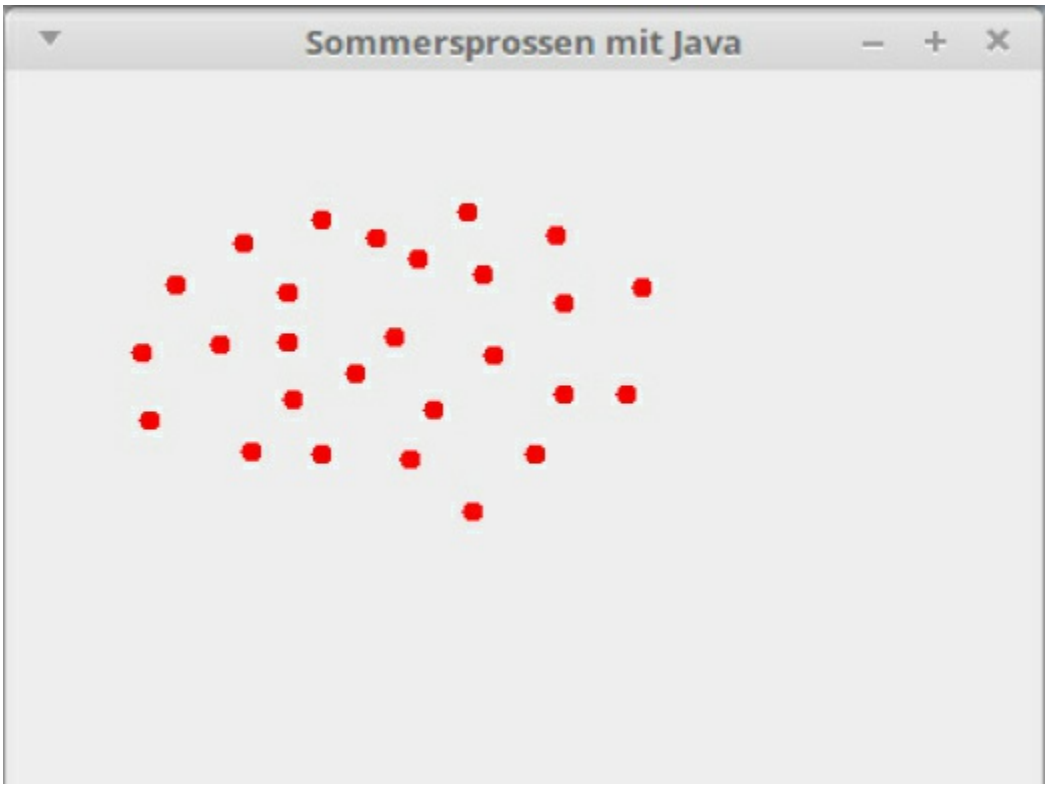


Abbildung 16.12: Sommersprossen selbst gemacht

Die drei Mauslauscher

- Das Interface `MouseListener` reagiert auf einmalige Ereignisse wie das Klicken der Maustaste.
- Dagegen ist das Interface `MouseMotionListener` für die Bewegungen der Maus zuständig, also quasi Dauerobservierung mit Bewegungsprofil.
- Für beides wurde dann unter dem Namen `MouseInputListener` ein Sammelbecken eingerichtet, und es müssen alle Methoden implementiert werden.

Um das Interface `MouseListener` zu implementieren, muss Ihre Klasse die folgenden Methoden definieren:

- `void mouseClicked(MouseEvent e)`

  Die Methode `mouseClicked` wird aufgerufen, wenn ein vollständiger Klick ausgeführt wurde. Ein Klick besteht aus dem Herunterdrücken und dem Loslassen der Maustaste. Diese beiden Ereignisse lassen sich auch separat bearbeiten. Damit können Sie kontrollieren, ob nicht jemand heimlich die Taste losgelassen hat, ohne sie zuvor zu drücken.
- `void mousePressed(MouseEvent e)`

  Die Methode `mousePressed` wird aufgerufen, wenn die Maustaste gedrückt wird.
- `void mouseReleased(MouseEvent e)`

  Die Methode `mouseReleased` wird gestartet, wenn die Maustaste losgelassen wird. Tatsächlich wird in der Praxis meist `mouseReleased` eingesetzt, wenn ein Mausklick behandelt werden soll.

Das Interface `MouseMotionListener` kümmert sich um die Bewegungen. Sie müssten die folgenden Methoden implementieren:

- `void mouseMoved(MouseEvent ev)`

  Vom Interface `MouseMotionListener` muss auf jeden Fall die Methode `mouseMoved` implementiert werden. Sie wird aufgerufen, wenn die Maus bewegt wird.
- `void mouseDragged(MouseEvent ev)`

  Diese Methode wird aufgerufen, wenn die Maus bei gedrückter Taste bewegt wird.

Alle Mauslauscher-Methoden verwenden als Parameter ein Objekt der Klasse `MouseEvent`, das alle Informationen über den aktuellen Mauszustand enthält. Oft ist das Programm neugierig, wo sich die Maus beim Eintreffen des Ereignisses befand. Die Methode `getX` liefert die X-Koordinate und die Methode `getY` die Y-Koordinate dieser Position.


Der Parameter `MouseEvent` liefert die Informationen, die mit der Auslösung des Mausereignisses zusammenhängen.

Mauseschwänzchen

Diese Materie wird deutlich verständlicher, wenn Sie sie einmal im Programm gesehen haben. Darum sollte jetzt wieder der Compiler etwas zu tun bekommen. Schreiben wir ein Programm, das die Mausereignisse sichtbar macht. Dazu zeichnen wir ein paar Linien.

- Bei jeder Bewegung der Maus soll eine Linie von der letzten zur aktuellen

Position gezogen werden. Diese Information, dass die Maus bewegt wurde, erhalten wir durch Überschreiben der Methode `mouseMoved`.

- ✔ Dort speichert unser Programm diese Positionen in einem Array, das 30 Positionen aufnehmen kann. Ist es voll, wird es wieder von vorn gefüllt. Auf diese Weise werden immer die letzten 30 Ereignisse gespeichert.
- ✔ Durch den Aufruf von `repaint` wird ein Neuzeichnen des Fensters ausgelöst und die Methode `paint` indirekt gestartet. Diese löscht den bisherigen Bildschirm und zeichnet alle Linien, die im Array liegen.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MauseSchwanz extends JFrame
             implements MouseMotionListener {
```

```
// Hilfsklasse für die Speicherung der Linien
class Linie {
    int xAnf=0,yAnf=0;
    int xEnde=0,yEnde=0;
}

private final int MAXPOS = 30; // wie viele Linien?
private int indexPos = 0;      // aktuelle Position
private Linie linie[];         // die letzten Linien
private int letztX, letztY;    // letzte Position

// Konstruktor
public MauseSchwanz() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    // Linienspeicher einrichten
    linie = new Linie[MAXPOS];
    for (int i=0; i<MAXPOS; i++) {
        linie[i] = new Linie();
    }
    // Den Listener aktivieren
    addMouseMotionListener(this);
}

public void mouseDragged(MouseEvent ev) { }

// Die Maus wurde bewegt und ruft diese Methode
public void mouseMoved(MouseEvent ev) {
    // Position speichern
    linie[indexPos].xEnde = letztX;
    linie[indexPos].yEnde = letztY;
    linie[indexPos].xAnf = letztX = ev.getX();
    linie[indexPos].yAnf = letztY = ev.getY();
    // Zeiger im Speicher vorrücken
    indexPos++;
    if (indexPos>=MAXPOS) {
        indexPos = 0;
    }
    repaint(); // Erzwinge Neuzeichnen
}

public void paint(Graphics g) {
    // Bildschirm freimachen
    g.clearRect(0, 0, getWidth(), getHeight());
    // Die älteren Linien zeichnen
```

```
            for (int i=indexPos; i<MAXPOS; i++) {
                g.drawLine(linie[i].xAnf, linie[i].yAnf,
                           linie[i].xEnde, linie[i].yEnde);
            }
            // Die neuesten Linien zeichnen
            for (int i=0; i<indexPos; i++) {
                g.drawLine(linie[i].xAnf, linie[i].yAnf,
                           linie[i].xEnde, linie[i].yEnde);
            }
        }

        public static void main(String[] args) {
            MauseSchwanz fenster = new MauseSchwanz();
            fenster.setTitle("Mausschlängeln mit Java");
            fenster.setSize(400, 300);
            fenster.setVisible(true);
        }
    }
```

Listing 16.20: Eine Linie verfolgt die Maus.

Schauen Sie sich das Ergebnis an! In Abbildung 16.13 ist nicht annähernd nachzuvollziehen, was es für einen Spaß macht, den Mauseschwanz hinter sich herzuziehen.

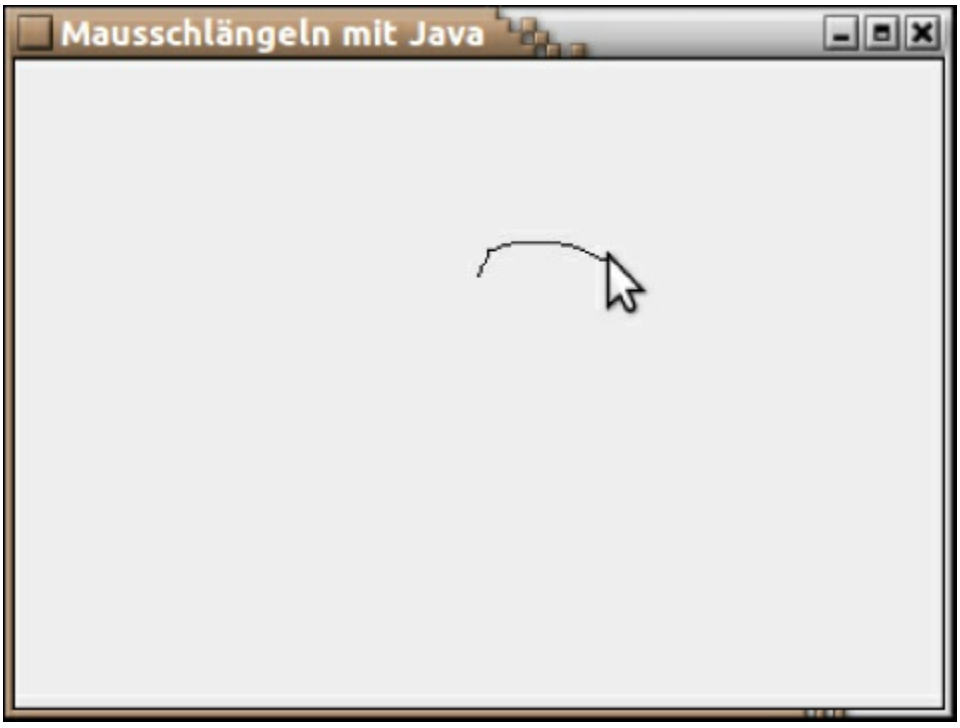


Abbildung 16.13: Mit der Maus unterwegs

Tatsächlich ist kaum zu erkennen, dass es sich um Linien statt um Punkte handelt,

sofern sich die Maus nicht sehr schnell bewegt.

Tastatur

Die Tastatur wird noch deutlich seltener direkt abgefragt als die Maus. Normalerweise arbeiten Programme, die Texteingaben verwenden, mit entsprechenden Kontrollelementen wie Eingabefeldern. Auch die Steuerung der Menüs über Shortcuts wird eben durch die Menüs erledigt und nicht durch die Direktabfrage.

Falls Sie eines Tages dennoch die Tastatur abfragen müssen, gehen Sie ähnlich vor wie beim Fangen der Mausereignisse. Das Interface, das die Tastaturereignisse fängt, heißt `KeyListener`. Durch Aufruf der Methode `addKeyListener` wird das Objekt des `KeyListeners` bei `JFrame` angemeldet.

Wie schon bei der Maus wird die erweiterte Klasse von `JFrame` in der Regel auch `KeyListener` implementieren und so `this` als Parameter beim Aufruf von `addKeyListener` verwenden.

Um einen `KeyListener` zu implementieren, müssen die Methoden `keyTyped`, `keyPressed` und `keyReleased` geschrieben werden. Über diese Methoden wird die Applikation nun informiert, wenn der Anwender eine Taste drückt.

- ✔ `void keyPressed(keyEvent e)`

  Die Taste wurde gedrückt.
- ✔ `void keyReleased(keyEvent e)`

  Die Taste wurde wieder losgelassen.
- ✔ `void keyTyped(keyEvent e)`

  Ein kompletter Tastendruck ist erfolgt, also sowohl `keyPressed` als auch `keyReleased`.

Alle Methoden von `KeyListener` haben einen Parameter der Klasse `KeyEvent`. Über deren Methode `getKeyCode` oder `getKeyChar` kann ermittelt werden, welche Taste gedrückt wurde.

Der `KeyListener` kann mit der Methode `addKeyListener` aktiviert werden.

Der `KeyListener` kann mit der Methode `addKeyListener` aktiviert werden.

```
addKeyListener(new KeyListener() {
    @Override
    public void keyPressed(KeyEvent e) { }
    @Override
    public void keyReleased(KeyEvent e) { }
    @Override
    public void keyTyped(KeyEvent e) { }
});
```

Ich könnte zwar noch ein paar Seiten herausschinden, wenn ich das Einhängen eines Listeners ein weiteres Mal am Beispiel des `KeyListeners` ausführe. Aber dann wird das Buch noch schwerer und kein Mensch will es mehr zur Kasse schleppen.

Aufgaben

Klick dir ein Rechteck

Schreiben Sie ein Programm `KlickRechteck`! Wenn Sie in das Fenster klicken, wird ein Rechteck über dem neuen und dem letzten Klick gespannt. Bei jedem Klick entsteht ein neues Rechteck.

Orientieren Sie sich ruhig an dem Beispiel `MauseSchwanz`, berücksichtigen Sie aber, dass Sie den `MouseListener` und nicht den `MouseMotionListener` implementieren müssen. Dafür sparen Sie sich das Array von Linien.

Eingabe zeichnen

In Kapitel 13 habe ich Ihnen bereits gezeigt, wie man einen Eingabedialog mit `showInputDialog` erstellt. Schreiben Sie ein Programm namens `DrawInput`, das diesen Eingabedialog beim Start aufruft und die Eingabe dann etwa mittig im Fenster anzeigt.

```
String eingabe = JOptionPane.showInputDialog("Eingabe:");
```

Berücksichtigen Sie, dass Sie nach der Eingabe `repaint` aufrufen müssen, damit der Fensterinhalt neu gezeichnet wird.

Das Hufeisen

Erweitern Sie die Wappen-Idee aus dem Beispiel für `Graphics2D`, indem Sie aus dem Wappen ein kleineres ausstanzen und so zu einer Hufeisenform gelangen. Diese Form soll dann einen Farbverlauf von oben Blau nach unten Grün erhalten.

Kapitel 17

Erstellen eines Layouts

IN DIESEM KAPITEL

- Elemente horizontal, vertikal oder grenzwertig angeordnet
- Dialoge per GridBagLayouts oder Kombination aus Layouts erzeugt
- Leichte Benutzung durch zersplitterte Fenster und Schiebebalken

Ist es nicht furchtbar? Wir haben schon wieder ein Henne-Ei-Problem. Ich könnte Ihnen jetzt hier etwas über Kontrollelemente erzählen, zum Beispiel über Buttons, Listboxen und ähnliche Elemente. Über kurz oder lang wird aber das Problem auftauchen, dass diese Elemente völlig unmotiviert und unschön in der Gegend rumhängen. Dann werden Sie sich fragen, was sich der Autor dabei gedacht hat, und ich bin wieder an allem schuld.

Denn die Anordnung dieser Elemente übernimmt das Layout. Aber ganz ohne Kontrollelemente ist ein Layout völliger Unsinn. Nehme ich jetzt plötzlich einen Button hinzu, heißt es, der Willemer drängt dem armen Leser diese gefährlichen Buttons auf und der weiß gar nicht, was das ist.

Also, um es kurz zu machen. Ich haue Ihnen jetzt einfach ein paar Buttons und noch die eine oder andere Listbox um die Ohren und Sie tun so, als ob das gar nicht wehtut, weil Sie schon mal Buttons gesehen haben und mir einfach glauben, dass ich Ihnen später erzählen werde, wie man mit den Dingern richtig umgeht. Geht das in Ordnung? Na ja, Sie haben sowieso keine Wahl. Das heißt: haben Sie doch. Aber wenn Sie sie genutzt hätten, würden Sie das hier gar nicht mehr lesen.

Noch mal auf Anfang

Es geht in diesem Kapitel also um Layouts. Ein Layout beschreibt eine unsichtbare Strategie, um sichtbare Kontrollelemente wie Buttons auf dem Fenster anzuordnen.

Die Strategie kann bedeuten, dass die Elemente untereinander, nebeneinander, im Gitter oder um ein zentrales Element herum gruppiert werden. Wird der Rahmen des Fensters auseinandergezogen, verändern sich einige Elemente und andere bleiben gleich. Für verschiedene Ausrichtungsstrategien gibt es unterschiedliche Layouts in Java, die gegebenenfalls zu kombinieren sind.

Um das Layout eines Fensters zu wechseln, verfügt jedes Fenster und jedes Panel über die Methode `setLayout`. Als Parameter erwartet die Methode ein Objekt des

gewünschten Layouts.

Alle bitte nebeneinander aufstellen: FlowLayout

In einem `FlowLayout` werden die Elemente von links nach rechts angeordnet und anschließend nach unten umbrochen. Prinzipiell funktioniert es also wie beim Schreiben. Erst wird die Zeile gefüllt und dann in der nächsten Zeile fortgesetzt.

Das Minimalprogramm in Listing 17.1 setzt das Layout auf `FlowLayout` und erzeugt zwölf Buttons.

```
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.Container;
import java.awt.FlowLayout;

public class TestFlow extends JFrame {

    TestFlow() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        for (int i=0; i<12; i++) {
            add(new JButton("Button " + i));
        }
        pack();
        this.setVisible(true);
    }

    public static void main(String[] args) {
       new TestFlow();
    }
}
```

Listing 17.1: Demonstration des `FlowLayout`

Wenn das Programm gestartet wird, wird es alle Buttons (Klasse `JButton`) nebeneinanderstellen, sofern der Monitor nur breit genug ist. Erst wenn die Buttons wirklich nicht mehr in eine waagerechte Zeile passen, wird umbrochen. Das werden Sie vielleicht erst feststellen, wenn Sie das Fenster verkleinern.

Zusammenpacken

Dieses Verhalten wird auch durch die Methode `pack` beeinflusst. Sie versucht, vom

System ein Fenster zu ergattern, in dem es alle Elemente unterbringen kann. Und da sich `FlowLayout` ein möglichst breites Fenster wünscht, fragt `pack` danach. Und solange der Monitor noch breit genug ist, bekommt es den Wunsch erfüllt.

Sobald der Benutzer das Fenster verkleinert, ist es mit der Freiheit vorbei. Nun gilt die Größe, die der Anwender diktiert. Diese Festlegung auf die äußere Größe können Sie auch vom Programm aus erzwingen, indem Sie `setSize` aufrufen. Mit `setSize(400,300)` werden eben nur 400 Punkte waagerecht und 300 Punkte senkrecht zur Verfügung gestellt. Damit muss dann `FlowLayout` auskommen. Sie sehen in Abbildung 17.1, wie `FlowLayout` die Buttons umbricht, weil ihm nicht genügend Platz zur Verfügung steht.

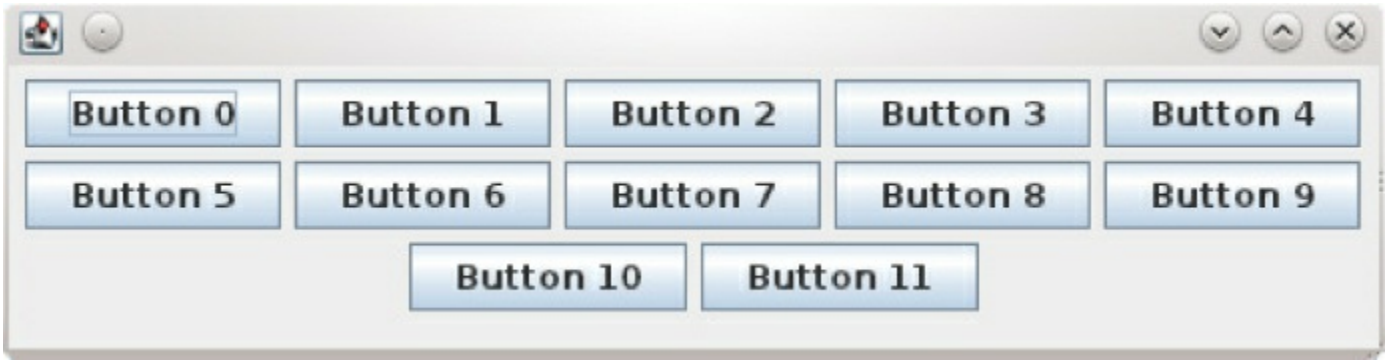


Abbildung 17.1: FlowLayout

Dem Konstruktor von `FlowLayout` können Sie einen Parameter für die Ausrichtung übergeben.

- Der Wert `FlowLayout.LEADING` sorgt dafür, alle Buttons links auszurichten.
- `FlowLayout.CENTER` zentriert die Elemente. Das ist auch die Vorgabe.
- `FlowLayout.TRAILING` sorgt für eine Orientierung am rechten Rand.

Gleichmäßig vergittert: GridLayout

Ein `GridLayout` legt seine Komponenten in einem gleichmäßigen Gitter ab. Der Konstruktor erhält als Parameter die Anzahl der Zeilen und der Spalten, in denen die Kontrollelemente abgelegt werden sollen. Die Spalten und Zeilen bleiben bei Größenänderungen immer jeweils gleich groß.

Die erzeugten Elemente werden erst waagerecht aufgefüllt. Sind alle Zellen in der Waagerechten belegt, wird die nächste Zeile begonnen (siehe Abbildung 17.2).

```
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.Container;
import java.awt.GridLayout; // Hier GridLayout statt FlowLayout!
```

```
public class TestGrid extends JFrame {

    TestGrid() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // Konstruktor bekommt 6 Zeilen und 2 Spalten
        setLayout(new GridLayout(6,2));
        for (int i=0; i<12; i++) { // 12 Buttons erzeugen
            add(new JButton("Button " + i));
        }
        pack();
        this.setVisible(true);
    }

    public static void main(String[] args) {
       new TestGrid();
    }
}
```

Listing 17.2: `GridLayout`



Abbildung 17.2: `GridLayout`

Wenn Sie die Fenstergröße verändern, können Sie sehen, wie die Buttons der Fensterveränderung folgen und ebenso größer und kleiner werden. Spielen Sie noch ein wenig weiter!

- ✔ Was passiert, wenn mehr Kontrollelemente angelegt werden, als im `GridLayout` festgelegt wurden?

- Was passiert, wenn zu wenig Kontrollelemente angelegt werden?
- Wie verhalten sich die Buttons, wenn der Platz viel größer als erforderlich wird? Probieren Sie das auch mal mit dem `FlowLayout`!

Wenn man ein Fenster mit einem `FlowLayout` aufzieht, werden die Buttons nicht größer als nötig. Dagegen ziehen sich die Buttons unter einem `GridLayout` wie Kaugummi auseinander.

Die Mitte wird fett: BorderLayout

Das `BorderLayout` hat sehr unterschiedlich ausgeprägte Bereiche. Es unterteilt seinen Arbeitsbereich folgendermaßen:

- Es gibt einen zentralen Arbeitsbereich, der mit `BorderLayout.CENTER` identifiziert wird. Wird das Fenster vergrößert, wird dieser zentrale Bereich analog mitvergrößert.
- Links und rechts ist je ein Randbereich vorhanden, der jeweils mit `BorderLayout.WEST` und `BorderLayout.EAST` bezeichnet wird. Bei einer Vergrößerung wachsen die Bereiche links und rechts in der Höhe mit dem zentralen Bereich. Ihre horizontale Größe verändert sich nicht.
- Oben und unten liegt je ein Bereich wie ein Deckel und Boden über dem zentralen Bereich und den Randbereichen links und rechts. Auf sie wird als `BorderLayout.NORTH` und `BorderLayout.SOUTH` zugegriffen.

In Abbildung 17.3 können Sie dieses Layout sehr schön sehen.

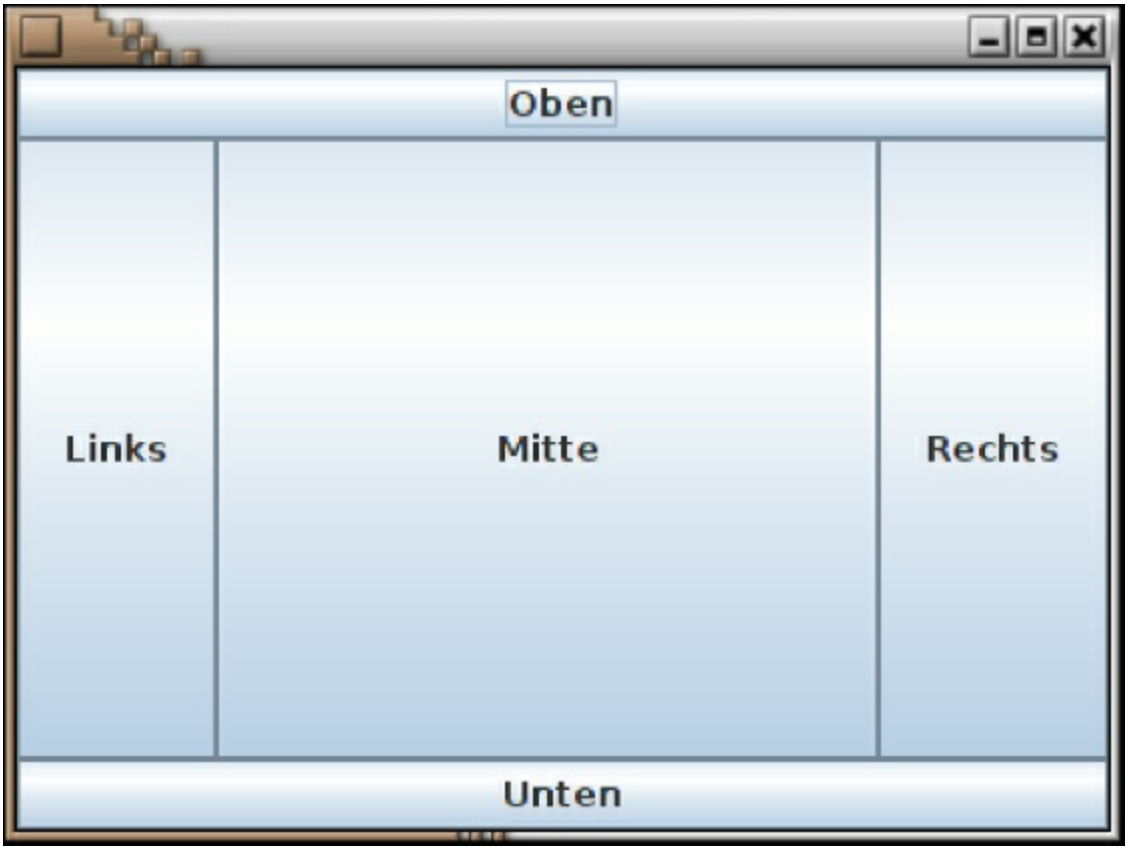


Abbildung 17.3: `BorderLayout`

Vielleicht hilft es Ihnen, wenn Sie sich vorstellen, im Inneren der Buttons könnten Sie waagerecht und senkrecht je ein Seil oder eine Feder spannen. Der Button im Zentrum bekommt dann vom Layout zwei Federn. Wird an dem Button gezogen, geht er auseinander. Die Buttons oben und unten haben waagerecht eine Feder eingebaut und horizontal ein Seil. Bei den Buttons links und rechts ist es umgekehrt.

```
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.Container;
import java.awt.BorderLayout;

public class TestBorder extends JFrame {

    TestBorder() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(400, 300);
        setLayout(new BorderLayout());
        add(BorderLayout.NORTH, new JButton("Oben"));
        add(BorderLayout.SOUTH, new JButton("Unten"));
        add(BorderLayout.WEST, new JButton("Links"));
        add(BorderLayout.EAST, new JButton("Rechts"));
        add(BorderLayout.CENTER, new JButton("Mitte"));
        // pack();
        this.setVisible(true);
    }

    public static void main(String[] args) {
       new TestBorder();
    }
}
```

Listing 17.3: BorderLayout und seine Rer

Es ist nicht zwingend, alle Bereiche zu füllen. Oft ist es praktisch, einzelne Bereiche wegzulassen. Ein Taschenrechner könnte ein Layout haben, das nur aus dem Center-Bereich für das Tastenfeld und dem North-Bereich für das Display besteht.

Kombination der Layouts mit Panels

Die Kunst besteht darin, Elemente gelungen anzuordnen, indem mehrere Layouts ineinander verschachtelt werden. Um ein Layout in ein anderes hineinsetzen zu können, wird statt eines Kontrollelements ein Panel eingebaut. Ein Panel ist unsichtbar, hat aber einen eigenen Layout-Manager und kann über diesen wiederum selbst Kontrollelemente oder auch Panels aufnehmen.

Betrachten wir einen einfachen Dialog, der eine Liste von Fernsehsendern verwalten soll (siehe Abbildung 17.4).

- Die linke Hälfte der Box wird durch die Liste belegt.
- Auf der rechten Seite werden insgesamt drei Buttons angeordnet.
- Zwei davon sind oben für das Einfügen oder Löschen von Sendern zuständig.
- Der dritte Button befindet sich unten rechts und dient dem Schließen der Box.

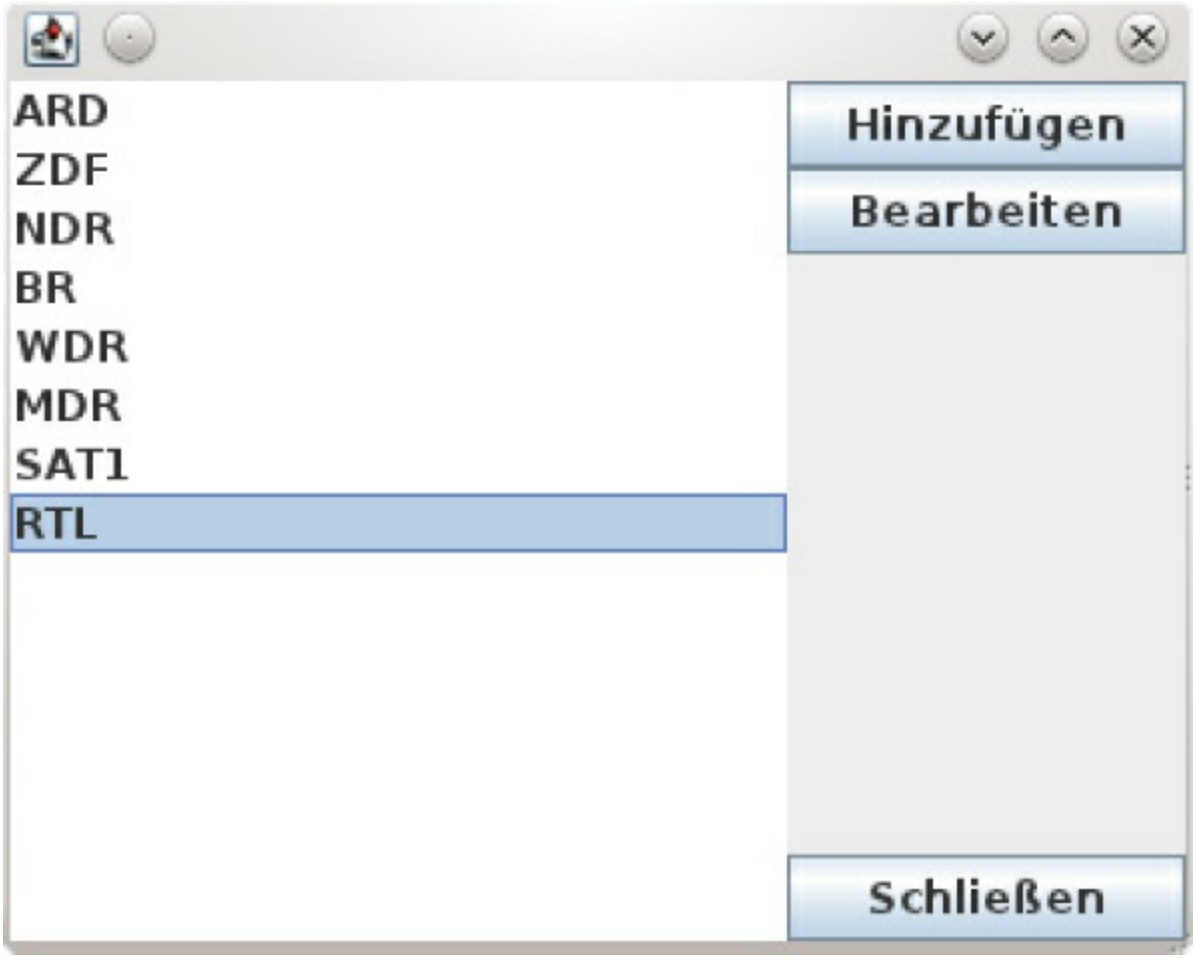


Abbildung 17.4: Ein Beispieldialog

Wie erstellt man solch einen Dialog? Dazu müssen Sie den Dialog gedanklich zerlegen. Zunächst sehen Sie die Zweiteilung.

- Links ist die Liste, die den Hauptanteil des Platzes einnehmen soll. Sie soll sich also in beide Richtungen vergrößern, wenn der Dialog auseinandergezogen wird. Darum schreit sie förmlich nach dem CENTER-Bereich eines `BorderLayouts`.
- Damit kommen die Buttons alle in den EAST-Bereich. Die anderen Zonen des `BorderLayouts` werden nicht besetzt und belegen damit auch keinen Platz.
- Schauen wir nun auf den EAST-Bereich in Abbildung 17.5. Hier setzen wir ein Panel ein und auch diesem geben wir ein `BorderLayout`.
  - Die beiden Buttons oben füllen dabei den NORTH-Bereich.
  - Der Schließbutton belegt den SOUTH-Bereich.

- Diesmal bleibt das CENTER leer, weil dieser leere Raum wachsen soll, wenn die Box vergrößert wird. Man könnte natürlich ein Label mit der Aufschrift »Hier könnte Ihre Werbung stehen!« in den CENTER-Bereich hineinsetzen. EAST und WEST bleiben diesmal leer.



Abbildung 17.5: Dialog-Layout durch Kombination mehrerer Layouts

Aber wir sind noch nicht fertig. Auch die beiden Buttons oben müssen in einem Panel abgelegt werden, weil das `BorderLayout` nur ein Element pro Bereich zulässt. Dieses Panel bekommt nun aber kein `BorderLayout`, sondern ein `GridLayout`, in dem zwei gleichgroße Elemente untereinander angeordnet werden.

Übrigens habe ich in das Zentrum des Dialogs eine `JList` geschummelt, die die Fernsehsender anzeigt. Schließlich ist das netter als ein Button als Platzhalter. Hier nun der Quelltext für den Dialog:

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JList;
import javax.swing.JPanel;
import java.awt.BorderLayout;
import java.awt.GridLayout;

public class DiaBorder extends JFrame {

    DiaBorder() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // Die Liste im Zentrum des Geschehens
        setLayout(new BorderLayout());
        String[] strSender = { "ARD", "ZDF", "NDR", "BR",
                               "WDR", "MDR", "SAT1", "RTL" };
        JList list = new JList(strSender);
        add(list, BorderLayout.CENTER);
        // EAST bekommt ein Panel mit eigenem BorderLayout
        JPanel rechtsPanel = new JPanel();
        rechtsPanel.setLayout(new BorderLayout());
        add(rechtsPanel, BorderLayout.EAST);
        // Der Schließbutten rechts unten
        JButton schliessButton = new JButton("Schließen");
        rechtsPanel.add(schliessButton, BorderLayout.SOUTH);
        // Die anderen Buttons in ein GridLayout oben
        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(2,1));
        rechtsPanel.add(buttonPanel, BorderLayout.NORTH);
        JButton hinzuButton = new JButton("Hinzufügen");
        JButton bearbeitenButton = new JButton("Bearbeiten");
        buttonPanel.add(hinzuButton);
        buttonPanel.add(bearbeitenButton);
        this.setSize(400, 300);
        this.setVisible(true);
    }

    public static void main(String[] args) {
       new DiaBorder();
    }
}
```

Listing 17.4: Die Kombination mehrerer Layouts


Wenn Sie das Beispielprogramm mit einem neueren Java-Compiler übersetzen, erhalten Sie eine Warnung über eine »unchecked« Operation.

Diese Warnung bringt Java seit Version 7, wenn eine `JList` nicht generisch sein sollte. Um die Warnung zu beseitigen, sagen Sie der `JList`, dass sie mit Strings hantiert:

```
JList list = new JList<String>(strSender);
```

Mit den spitzen Klammern legen Sie fest, welchen Typ die Liste verarbeitet. In diesem Fall ist es `String`.

Zerteilung: JSplitPane

Wo wir gerade bei Panels als Containern sind, stelle ich Ihnen das `JSplitPane` vor. Das ist eigentlich kein reiner Layout-Manager, sondern besitzt als Panel sichtbare Kontrollelemente, in diesem Fall einen Schieber, mit dem die beiden Bereiche des Panels getrennt sind. Er ermöglicht es dem Benutzer, die Aufteilung zwischen den beiden Panels selbst zu gestalten.

Wir könnten einen solchen Schieber in dem Dialogbeispiel mit den Fernsehsendern statt des ersten `BorderLayouts` verwenden. Dazu müsste ein `JSplitPane` im CENTER-Bereich des Rahmens angelegt werden.

Links vom Schieber soll die Liste der Sender stehen. Rechts vom Schieber würde wie bisher ein `BorderLayout` verwendet, um die Buttons anzuordnen. Das ist in wenigen Zeilen geschehen. Zunächst wird wie oben die Liste und das rechte Panel erzeugt. Dann wird allerdings das Splitpanel angelegt und die Liste und das rechte Panel dem Konstruktor übergeben. Das Splitpanel wird nun in den CENTER-Bereich des Fensterrahmens gelegt.

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JSplitPane;
import java.awt.BorderLayout;
import java.awt.GridLayout;

public class DiaSplit extends JFrame {

    DiaSplit() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new BorderLayout());

        String[] strSender = { "ARD", "ZDF", "NDR", "BR",
                               "WDR", "MDR", "SAT1", "RTL" };
        JList list = new JList<String>(strSender);
        JPanel rechtsPanel = new JPanel();
        // Der Splitter wird zwischen die Liste und das
        // rechte Panel für die Buttons gesetzt
        JSplitPane splitPane = new JSplitPane(
                JSplitPane.HORIZONTAL_SPLIT,
                list, rechtsPanel);
        splitPane.setOneTouchExpandable(true);
        splitPane.setDividerLocation(250);
        add(splitPane, BorderLayout.CENTER);
        rechtsPanel.setLayout(new BorderLayout());
        JButton schliessButton = new JButton("Schließen");
        rechtsPanel.add(schliessButton, BorderLayout.SOUTH);

        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(2,1));
        rechtsPanel.add(buttonPanel, BorderLayout.NORTH);
        JButton hinzuButton = new JButton("Hinzufügen");
        JButton bearbeitenButton = new JButton("Bearbeiten");
        buttonPanel.add(hinzuButton);
        buttonPanel.add(bearbeitenButton);

        this.setSize(400, 300);
        // pack();
        this.setVisible(true);
    }
```

```
    public static void main(String[] args) {
        new DiaSplit();
    }
}
```

Listing 17.5: Ein Split im Pane

Abbildung 17.6 zeigt das Ergebnis. Der mittlere Balken ist nun verschiebbar.

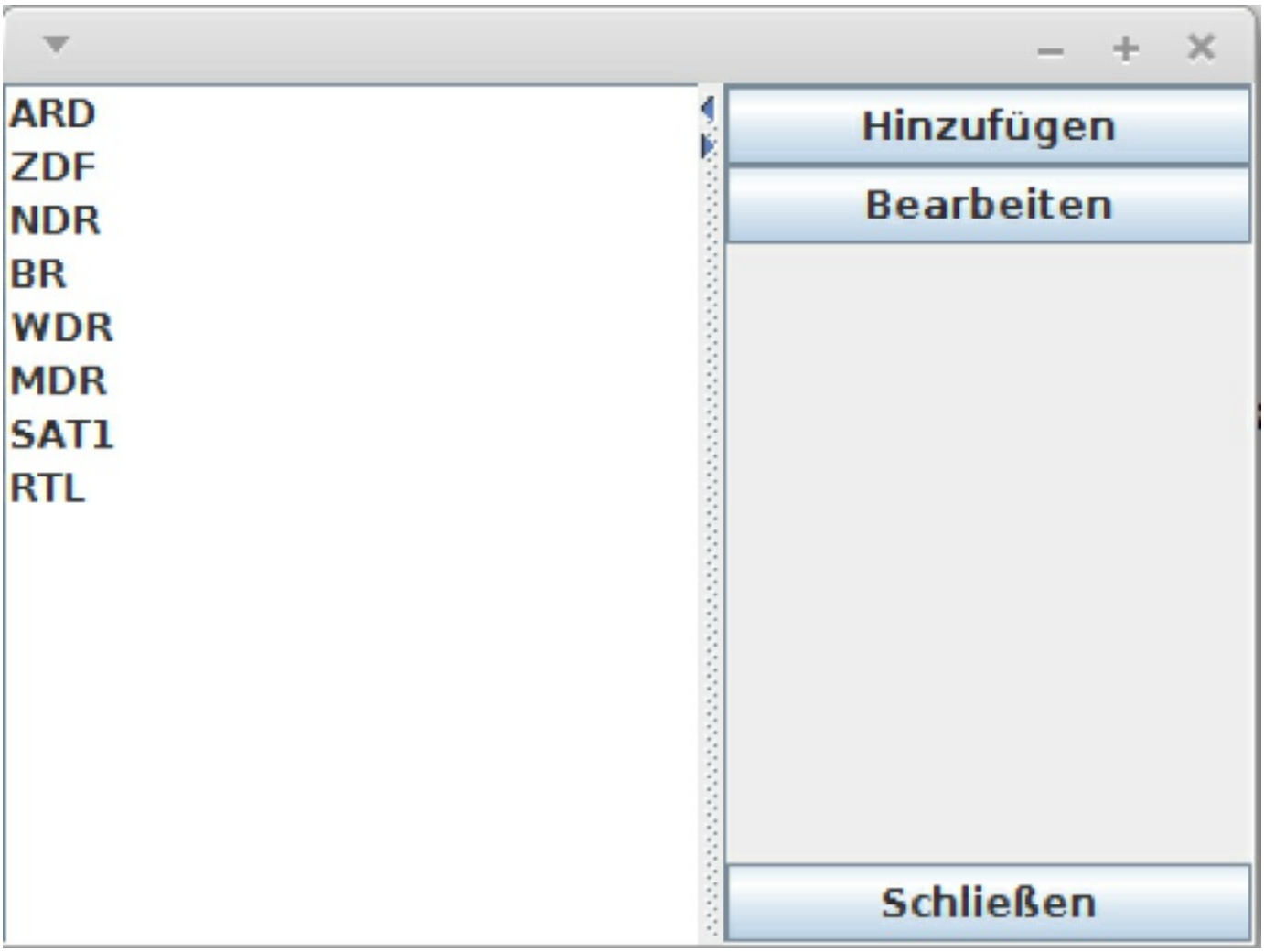


Abbildung 17.6: `JSplit` erzeugt einen verschiebbaren Balken

Schiebebalken inklusive: JScrollPane

Manchmal ist der Inhalt größer als das Fenster. Für solche Fälle haben die grafischen Oberflächen einen Rollbalken, auf Plattdeutsch auch *Scrollbar* genannt. Das tritt vor allem bei Listen auf. Mit dem `JScrollPane` lässt sich sehr einfach eine Liste mit einem oder mehreren Rollbalken versehen. Dabei ist `JScrollPane` nicht zimperlich und arbeitet auch mit anderen Kontrollelementen zusammen.

Im Konstruktor wird das Kontrollelement übergeben, das mit Rollbalken versehen werden soll. Das kann eine Liste oder eine `JTextArea`, also mehrzeilige Texteingabe sein. Die Rollbalken sind so lange nicht vorhanden, wie alles in den vorgegebenen Rahmen passt. Erst wenn die Liste oder der Text größer wird,

erscheinen die Rollbalken und arbeiten ohne weiteres Zutun durch den Programmierer. Und Programmierer finden es auch mal schön, wenn andere die Arbeit übernehmen.

```
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JScrollPane;

public class Scroll extends JFrame {
    public static void main(String[] args) {
        new Scroll();
    }

    Scroll() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        String[] strSender = { "ARD", "ZDF", "BR", "WDR",
                "MDR", "NDR-Das Beste am Norden", "RTL" };
        JList<String> list = new JList<String>(strSender);
        JScrollPane scrollList = new JScrollPane(list);
        add(scrollList);
        pack();
        this.setVisible(true);
    }
}
```

Listing 17.6: Eine Liste mit Scrollbar

Wenn Sie also statt der Listbox ein `JScrollPane` per `add` in das Fenster einbauen und diesem wiederum die Listbox als Parameter übergeben, haben Sie schon die Rollbalken eingebaut.

Die Rollbalken können Sie zunächst nicht sehen, da `pack` dafür sorgt, dass die Liste vollständig dargestellt wird. Sobald Sie das Fenster jedoch verkleinern, erscheinen die Rollbalken, wie in Abbildung 17.7 zu sehen.

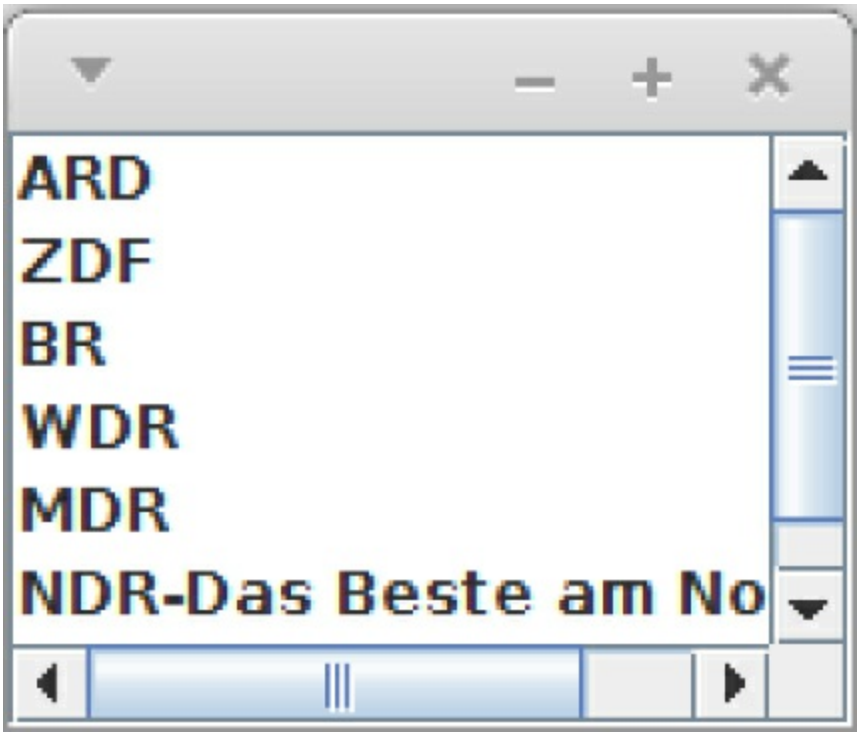


Abbildung 17.7: `JScrollPane` erzeugt einen Rollbalken als Rahmen.

JTabbedPane oder CardLayout

Das `JTabbedPane` organisiert mehrere Dialoge, die es platzsparend hintereinander legt. Um zwischen den Dialogen umzuschalten, stellt es Tabs oder im klassischen Sprachgebrauch »Karteikartenreiter« zur Verfügung. Sie kennen diese sicher von diversen Konfigurationsdialogen.

- Die Komponente `JTabbedPane` bildet den Rahmen.
- Für jede Seite muss ein eigenes Panel erzeugt werden.
- Die Panels werden über die Methode `addTab` eingebunden. Als zweiter Parameter erhält `addTab` den Text, der in dem Tab stehen soll.

Damit hat jede Seite des Dialogs ihr eigenes Panel, in dem die Kontrollelemente über eigene Layouts angeordnet werden.

Für die Darstellung des Textes verwendet das Programm in Listing 17.7 ein `JLabel`. Das ist einfach ein Button, den man nicht drücken kann. Also eigentlich kann `JLabel` gar nichts, außer nett auszusehen. Aber das macht `JLabel` ganz ordentlich.

```
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTabbedPane;
import java.awt.BorderLayout;

public class DiaTab extends JFrame {

    DiaTab() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new BorderLayout());
        JTabbedPane tabbedPane = new JTabbedPane();
        add(tabbedPane, BorderLayout.CENTER);
        JPanel eins = new JPanel(); eins.add(
                new JLabel("Inhalt eins"));
        tabbedPane.addTab("Eins", eins);
        JPanel zwei = new JPanel(); zwei.add(
                new JLabel("Inhalt zwei"));
        tabbedPane.addTab("Zwei", zwei);
        this.setSize(400, 300);
        this.setVisible(true);
    }

    public static void main(String[] args) {
       new DiaTab();
    }
}
```

Listing 17.7: Vom Reiter zum JTabbedPane



Abbildung 17.8: JTabbedPane



`JTabbedPane` verwendet das `CardLayout`, das die Elemente wie bei einem Kartenspiel hintereinander anordnet. Prinzipiell könnten Sie das `CardLayout` auch direkt verwenden, wenn Sie eine Anwendung haben, in der die Tabs

stören.

Das Layout der Dialoge: GridBagLayout

Das `GridBagLayout` ordnet seine Elemente wie das `GridLayout` in einem Raster an. Im Gegensatz zum `GridLayout` erlaubt das `GridBagLayout`, dass die Spalten und Zeilen unterschiedlich groß sind. Damit wird es zum idealen Layout zur Gestaltung von Eingabedialogen. Meist steht links ein kurzer Text und rechts daneben ein deutlich längeres Eingabefeld.

Die Position der Elemente in dem Raster werden durch ein Objekt der Klasse `GridBagConstraints` festgelegt. Dieses Objekt wird beim Hinzufügen eines Elements durch die Methode `add` als zweiter Parameter übergeben. Wo das Element positioniert wird, legen die Attribute `gridx` und `gridy` von `GridBagConstraints` fest. Das folgende Beispiel erzeugt eine Eingabemaske für Name, Adresse und E-Mail einer Person.

```
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

public class GridBag extends JFrame {

    public static void main(String[] args) {
        new GridBag();
    }

    public GridBag() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        GridBagLayout gbLayout = new GridBagLayout();
        GridBagConstraints gbConstraints =
                new GridBagConstraints();
```

```
        setLayout(gbLayout);
        gbConstraints.gridy = 0;
        String[] titel = {"Name", "Adresse", "E-Mail"};
        for (int i=0; i<titel.length; i++) {
            gbConstraints.gridx = 0;
            add(new JLabel(titel[i]), gbConstraints);
            gbConstraints.gridx = 1;
            add(new JTextField(30), gbConstraints);
            gbConstraints.gridy++;
        }
        pack();
        setVisible(true);
    }
}
```

Listing 17.8: Ein einfacher Dialog

Abbildung 17.9 zeigt den entstandenen Dialog. Die Positionen der Elemente werden in den Attributen `gridx` und `gridy` des `GridBagConstraints` festgelegt.

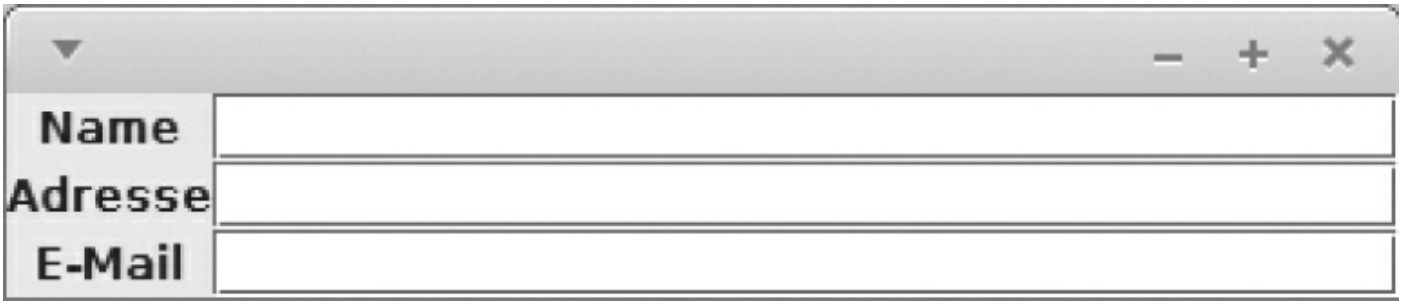


Abbildung 17.9: `GridBagLayout` für eine einfache Eingabemaske

Anordnungsbeschreibung im GridBagConstraints

Ein Objekt des `GridBagConstraints` wird mit Positionsdaten gefüllt und quasi als Begleitbrief dem Kontrollelement beigelegt, wenn es per `add` in das `GridBagLayout` eingebaut wird.

- Die Attribute `gridx` und `gridy` bestimmen die Position des Elements im Raster. Darum ist die Reihenfolge der Bestückung unerheblich.
- `gridwidth` legt fest, wie viele Gitterelemente horizontal, und `gridheight`, wie viele Gitterelemente vertikal belegt werden.

  Erhält das Attribut `gridwidth` oder `gridheight` den Wert `GridBagConstraints.REMAINDER`, werden alle Gitterelemente bis zum äußersten Rand des Containers belegt.
- In Abbildung 17.9 sehen Sie, wie die Beschriftungen in der linken Spalte zentriert ausgerichtet sind. Diese Ausrichtung wird durch das Attribut `anchor`

eingestellt, das standardmäßig auf `GridBagConstraints.CENTER` eingestellt ist. Die Beschriftung wird linksbündig, wenn Sie `GridBagConstraints.WEST` verwenden.

- Mit dem Attribut `fill` wird angegeben, ob das Element den Platz seines Feldes voll nutzen soll, und wenn ja, in welche Richtung.
  - Das Attribut `fill` hat standardmäßig den Wert `GridBagConstraints.NONE`. So wird das Kontrollelement nur den Raum beanspruchen, den es minimal benötigt. Das entspricht also dem Verhalten des `FlowLayouts` oder eben dem Gedanken, dass innerhalb des Elements in beide Richtungen ein Seil gespannt ist, das ein Auseinanderziehen des Elements verhindert.
  - Durch Belegung mit `GridBagConstraints.HORIZONTAL` wird der gesamte Platz, der horizontal zur Verfügung steht, vom Kontrollelement eingenommen. Es gibt also waagerecht eine Feder und senkrecht ein Seil.
  - Analog steht `GridBagConstraints.VERTICAL` für eine vertikale Ausdehnung. Seil und Feder tauschen ihre Plätze.
  - `GridBagConstraints.BOTH` bewirkt eine Ausdehnung in beide Richtungen.
- Die Attribute `weightx` und `weighty` stellen die Gewichtung dar und legen fest, welche Priorität das Element bei der Platzzuweisung gegenüber den anderen Elementen in der horizontalen beziehungsweise vertikalen Reihe haben soll.

  Ist mehr Platz vorhanden, als das Kontrollelement benötigt, wird dieser nach der Gewichtung der nebeneinanderstehenden Elemente in den Attributen `weightx` oder `weighty` verteilt. Eingetragen wird ein Fließkommawert zwischen 0 und 1.
- `insets` nehmen den Abstand zum Rand auf. Für das Setzen von `insets` muss ein Objekt der Klasse `Insets` angelegt werden. Der Konstruktor erhält die Abstände als Parameter.

  ```
  gbConstraints.insets = new Insets(2,2,2,2);
  ```

  Diese Anweisung sorgt dafür, dass um das Kontrollelement in jeder Richtung ein Abstand von 2 Pixeln eingehalten wird.
- `ipadx` und `ipady` bestimmen die Zusatzpunkte zur minimalen Elementbreite beziehungsweise-höhe.

Das BoxLayout und die Box

Während die bisher vorgestellten Layouts Erbstücke aus den AWT-Zeiten sind,

wurde das `BoxLayout` für Swing gemacht. Darum wird es auch nicht aus der AWT-Bibliothek importiert, sondern von Swing.

Man könnte das `BoxLayout` als die Swing-Version von `FlowLayout` ansprechen. Allerdings kann man die Elemente nicht nur horizontal, sondern auch vertikal anordnen. Dieser kleine Unterschied ist schon sehr hilfreich. So kann man das `BoxLayout` geschickt verschachteln und so bereits viele Anordnungen realisieren, ohne gleich auf das kompliziertere `GridBagLayout` zugreifen zu müssen.



Abbildung 17.10: `BoxLayout`

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.BoxLayout;

public class Boxer extends JFrame {

    Boxer() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new BoxLayout(getContentPane(),
                BoxLayout.Y_AXIS));
        add(new JButton("Anton"));
        add(new JButton("Berta"));
        add(new JButton("Caesar"));
        pack();
        setVisible(true);
    }

    public static void main(String[] args) {
       new Boxer();
    }
}
```

Listing 17.9: BoxLayout

Wie Sie sehen, erwartet der Konstruktor von `BoxLayout` als ersten Parameter das Panel, das er verwaltet. Der zweite Parameter gibt an, ob die Ausrichtung in Richtung der X- oder der Y-Achse erfolgt. Die möglichen Konstanten lauten `BoxLayout.X_AXIS` und `BoxLayout.Y_AXIS`.

Box

Das `BoxLayout` wird durch den Container `Box` verwendet, der noch ein paar Möglichkeiten der Ausrichtung hinzufügt.

Dazu werden zwischen die Komponenten Pseudo-Elemente eingebaut, die die Aufgabe haben, Abstand zum Nachbarn zu halten.

- ✔ `Box.createVerticalGlue` und `Box.createHorizontalGlue` erzeugen einen Kleber, der sich ausdehnt, wenn man ihn zieht.
- ✔ `Box.createRigidArea` erzeugt einen Klotz, der zwischen die Elemente geschoben wird und der beim Auseinanderziehen konstant bleibt.

Der `Filler` wird generiert und erhält drei Dimensionen über den Konstruktor.

- ✔ Die erste Dimension ist der Minimalabstand.

- ✔ Die zweite Dimension ist der bevorzugte Abstand.
- ✔ Die dritte Dimension ist der maximale Abstand. Soll dieser beliebig groß werden, gibt man `Short.MAX_VALUE` an.

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.Box;
import java.awt.Dimension;

public class BoxTest extends JFrame {
    BoxTest() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Box box = Box.createVerticalBox();

        box.add(new JButton("Anton"));
        box.add(Box.createRigidArea(new Dimension(5,5)));

        box.add(new JButton("Berta"));
        box.add(Box.createVerticalGlue());
        box.add(new JButton("Caesar"));
        // Filler wird vorbereitet
        Dimension min = new Dimension(30, 30);
        Dimension pref = new Dimension(120, 120);
        Dimension max = new Dimension(250, 250);
        box.add(new Box.Filler(min, pref, max));
        box.add(new JButton("Dorian"));
        // die Box wird ins JFrame eingebaut
        add(box);

        pack();
        setVisible(true);
    }

    public static void main(String[] args) {
       new BoxTest();
    }
}
```

Listing 17.10: Box



Abbildung 17.11: Box

Null-Diät für Layout-Manager

Sie mögen keine Layout-Manager? Kein Problem: Es gibt auch die Möglichkeit, Kontrollelemente ohne jegliches Layout direkt auf dem Container zu positionieren. Dazu wird die Methode `setLayout` mit dem Parameter `null` aufgerufen.

```
panel.setLayout(null);
```

Positionieren mit setBounds

Nun werden die Kontrollelemente einfach hinzugefügt. Da sie nicht angeordnet werden können, überdecken sie sich gegenseitig. Anschließend positionieren Sie die Kontrollelemente über deren Methode `setBounds` pixelgenau nebeneinander. Als Parameter erwartet `setBounds` vier Integerwerte:

- ✔ Die X-Position
- ✔ Die Y-Position
- ✔ Die Breite des Elements
- ✔ Die Höhe des Elements

Im folgenden Beispiel wird ein `JButton` namens `button` positioniert.

```
button.setBounds(xPos, yPos, breite, hoehe);
```

Der Vorteil dieser Vorgehensweise liegt klar auf der Hand. Der Programmierer

kann völlig frei bestimmen, wo die Elemente liegen. Der Haken an dieser Methode ist, dass sich die Absolutpositionen nicht nach dem Platzbedarf der Kontrollelemente richten. Verwendet die Umgebung des Programms große Schriften, ist vielleicht die Beschriftung des Buttons unlesbar.

Sie sollten dieses Vorgehen also nicht nur wählen, weil Sie Manager nicht leiden können. Sie sollten schon einen wichtigen Grund haben. Ansonsten müssen Sie den Programmieraufwand selbst betreiben, der Ihnen der Layout-Manager vermutlich besser, schneller und fehlerfreier abnimmt.

Aufgaben

Der Umgang mit den Layouts fällt dem Programmierer nicht in den Schoß. Sie müssen die Layouts geschickt kombinieren. Außerdem gibt es unterschiedliche Lösungen, ein Layout zu gestalten. Lassen Sie sich nicht entmutigen, wenn Ihre ersten Dialoge etwas vermurkst aussehen. Nehmen Sie es als Motivation zum Üben. Hier sind ein paar Anregungen:

Taschenrechner

Schreiben Sie ein Programm `Taschenrechner`, das genauso aussieht, wie Sie sich den idealen Taschenrechner vorstellen. Falls Sie überhaupt keine Idee dazu haben, unterbreite ich Ihnen ein paar Vorschläge. Oben gibt es eine Anzeige, die Sie mit einem `JTextField` am effektivsten darstellen. Darunter befinden sich jede Menge Tasten, die in der Regel alle gleich groß sind und den Raum optimal füllen.

Eingabedialog

Was benötigt ein einfacher Eingabedialog? Zunächst benötigt er natürlich eine Eingabezeile, die Sie mit `JTextField` erzeugen können. Es muss aber auch eine Beschriftung vorhanden sein, die links oder über der Eingabe steht. Dazu werden Buttons benötigt, die das Okay oder den Abbruch des Anwenders entgegennehmen. Das kleine Symbol könnten Sie durch ein `JLabel` simulieren. Füllen Sie es entweder mit einem großen X oder spicken Sie mal im nächsten Kapitel, wie man mit einem `JLabel` Bilder darstellt.

Ordnen Sie die Elemente zunächst nach Ihrem persönlichen Geschmack an. Schauen Sie dann, wie der Originaldialog aussieht. Überlegen Sie, welcher Ansatz welche Vorteile hat.

Kapitel 18

Kontrollelemente im Swing

IN DIESEM KAPITEL

Von Buttons, Eingabefeldern und Comboboxen

Ein Programm zur Berechnung des Body-Mass-Index

Fokus, Multiliner und andere

Wir werden nun in die wunderbare Welt der grafischen Kontrollelemente eintauchen. Da gibt es Buttons, Listen und Eingabefelder. Falls Sie sich erinnern, haben wir damit beim Thema Layout schon einmal gespielt. Nun werden wir schauen, wie wir sie in die Programme einbauen, ihre Aktionen empfangen und mit ihnen Daten austauschen.

Hochgradiger Simpel: Das Label

Das einfachste Kontrollelement ist ein Label. Das ist ganz lieb und tut nichts. Es steht einfach nur da und zeigt einen Text oder ein Symbol an.

Text im Label

Für den Umgang mit Text-Labels gibt es folgende Möglichkeiten:

- Der Text wird dem Konstruktor als Parameter übergeben.
- `JLabel` bietet die Methode `setText` an, um nachträglich einen Text zu setzen oder zu ändern.
- Über die Methode `getText` können Sie den Text wieder auslesen, falls Sie ihn vergessen haben sollten.

Bilder ins Label

Für die Darstellung eines Bildes in einem `JLabel` akzeptiert dieses als Konstruktorparameter ein `Icon`. Dieses wiederum können Sie leicht erzeugen, indem Sie ein `ImageIcon` erzeugen und den Pfad als Parameter übergeben.

Wenn Sie Ihre Programme mit Bildern versehen, werden Sie die Bilder nicht vom Dateisystem des Rechners lesen, sondern dem Programm beipacken wollen. Dazu gehen Sie in folgenden Schritten vor:

1. Im Verzeichnis *src* legen Sie ein Package für Ihre Bilder und Symbole an, beispielsweise *images*.
2. Nun holen Sie sich die URL des gewünschten Bildes.
   ```
   URL url = getClass().getResource("/images/schiff.png");
   ```
3. Diese URL verwenden Sie zur Erzeugung eines Icons.
   ```
   Icon icon = new ImageIcon(url);
   ```
4. Dieses Icon kann schließlich dem `JLabel` im Konstruktor übergeben werden.
   ```
   JLabel bildlabel = new JLabel(icon);
   ```

Das Programm in Listing 18.1 zeigt zwei `JLabels` direkt nebeneinander. Das erste stellt ein Icon, das zweite einen Text dar.

```
import java.awt.FlowLayout;
import java.net.URL;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MeinLabel extends JFrame {

    public static void main(String[] args) {
        new MeinLabel();
    }

    public MeinLabel() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // Setze die JLabel per FlowLayout nebeneinander
        setLayout(new FlowLayout());
        // Ein JLabel mit Bild
        URL url = getClass().getResource("/images/schiff.png");
        Icon icon = new ImageIcon(url);
        JLabel bildlabel = new JLabel(icon);
        add(bildlabel);
        // Ein JLabel mit Text
        JLabel textlabel = new JLabel("Gleich weg!");
        add(textlabel);
        textlabel.setText("Neuer Text");
        String str = textlabel.getText();
        System.out.println(str);
```

```
            pack();
            setVisible(true);
        }
}
```

Listing 18.1: `JLabel` zeigt ein Bild.

Nach dem Start des Programms sollte das Fenster aus Abbildung 18.1 auf Ihrem Bildschirm erscheinen.



Abbildung 18.1: Zweimal `JLabel`. Ein Bild und ein Text.

Der Aufruf `pack` tut das, was der Name vermuten lässt: Er packt das Fenster auf seine kleinstmögliche Fläche zusammen. Sollte also das Bild nur ein Pixel groß sein, könnte es schwer werden, das Programm auf dem Bildschirm wiederzufinden. Alternativ können Sie mit dem Aufruf `setSize(400,300)` die Größe des Fensters vorgeben.

Der Konstruktor ist mehrfach überladen. Sie können – wie gesehen – einen String oder ein Symbol übergeben. Ein Label lässt sich auch mit beidem füllen. Als ersten Parameter geben Sie dann den Text an, als zweiten das Symbol.

Der dritte Parameter gibt dann an, wie die beiden ausgerichtet werden. Er wird mit einer der Konstanten `SwingConstants.LEFT`, `SwingConstants.CENTER` oder `SwingConstants.RIGHT` gefüllt.

```
JLabel(String text)
JLabel(Icon icon)
JLabel(String text, Icon icon, int ausrichtung)
```

Das Auffangen der Ereignisse

Labels dienen ja nun meist nur der Dekoration. Aber die meisten anderen Kontrollelemente informieren das Programm über die Aktionen des Benutzers. Allerdings nur dann, wenn sich das Programm dafür interessiert. Und da kommt der vertraute Lauscher wieder ins Spiel.

Wenn Ihr Programm erfahren soll, dass ein Button geklickt, ein Schieber verschoben oder ein Textfeld gewechselt wurde, muss das umfassende Fenster das Interface `ActionListener` oder einen seiner Kumpel implementieren.

Beim Auftreten von Ereignissen der Kontrollelemente ruft die Schnittstelle die

Methode `actionPerformed` der Schnittstelle `ActionListener` auf. Also muss Ihre Klasse `ActionListener` und insbesondere die Methode `actionPerformed` implementieren.

```
public class ButtonAction extends JFrame
                          implements ActionListener
```

Das Programm wird aber nur über die Ereignisse informiert, für die es sich auch angemeldet hat. Dies tut es, indem die Methode `addActionListener` des zu überwachenden Kontrollelements aufgerufen und der `ActionListener` als Parameter übergeben wird. Da ja die Klasse den `ActionListener` implementiert, kann hier `this` angegeben werden. Wenn Sie jetzt noch nicht verwirrt sind, gelingt mir das vielleicht mit Abbildung 18.2.

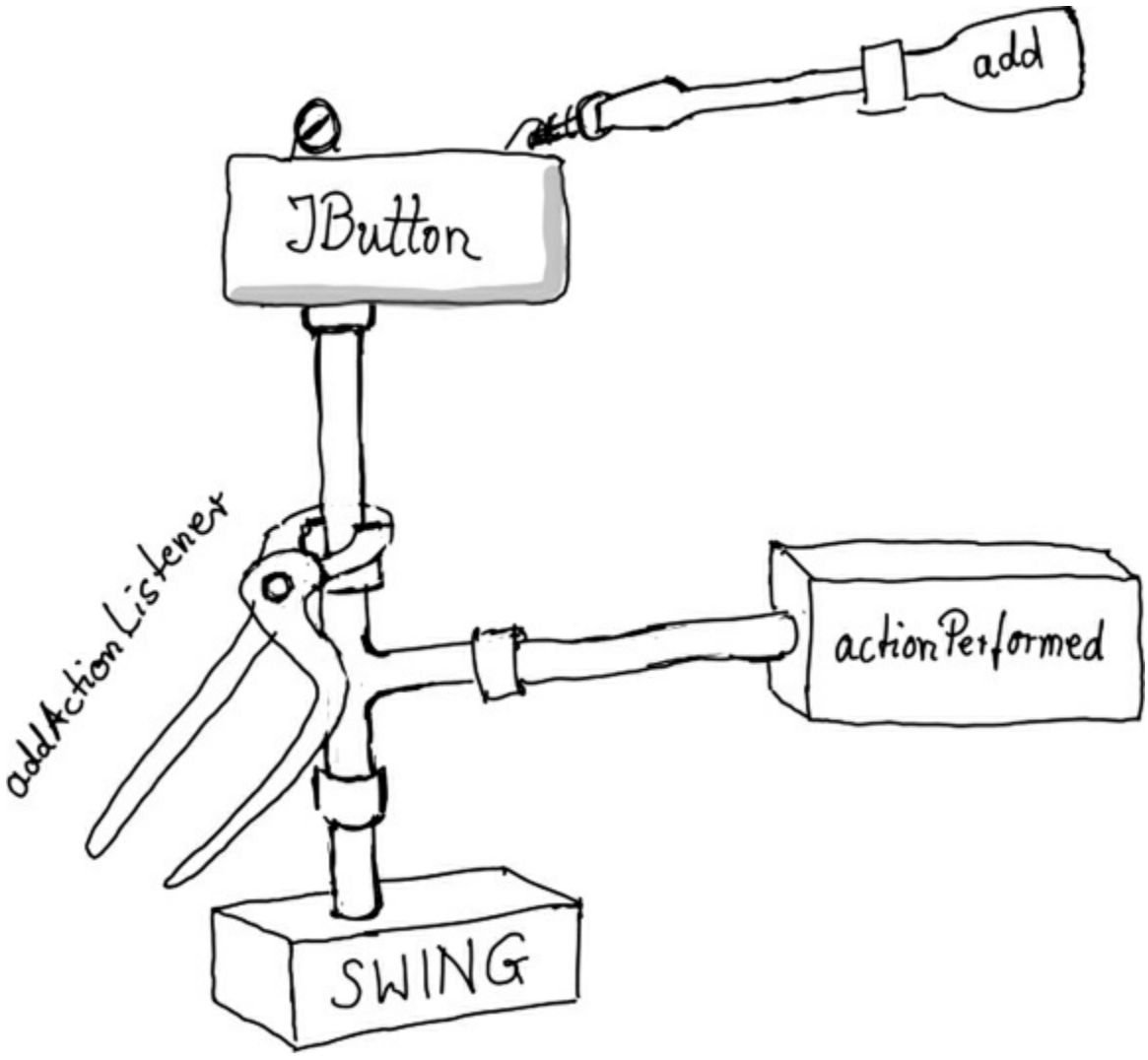


Abbildung 18.2: Einbau eines `ActionListeners`

Zusammengefasst führen drei Schritte zu der Aktion des Benutzers:

1. Die Methode `add` setzt das Element dort in das Layout, wo es später erscheinen soll.
2. Soll das Programm über die Ereignisse des Elements informiert werden, ruft es

`addActionListener(this)` auf.

3. Die Methode `actionPerformed(event)` wird beim Eintreffen von Ereignissen der angemeldeten Elemente von Swing aufgerufen. Wenn Sie die Methode überschrieben haben, erfahren Sie davon in Ihrem Programm.

Die Methode `actionPerformed` ist der Tummelplatz aller angemeldeten Ereignisse. Allerdings müssen Sie ja auch nicht alle Ereignisse bearbeiten. Denken Sie an eine Dialogbox. Da lassen Sie den Anwender so lange herumbasteln, bis er meint, dass er fertig ist. Dann klickt er auf den OK-Button und dieses Ereignis interessiert dann das Programm. Erst danach werten Sie die Eingaben des Benutzers aus.

Das ActionEvent liefert die Hintergründe

Die Methode `actionPerformed` erfährt, welches Element was getan hat, über ihren Parameter des Typs `ActionEvent`. Diesen Parameter können Sie befragen, was eigentlich passiert ist.

```
@Override
public void actionPerformed(ActionEvent ev) {
```

Der Parameter `ev` liefert folgende Methoden, um Näheres über das eingetretene Ereignis zu erfahren.

- Die Methode `getSource` liefert das Objekt, das das Ereignis auslöste. Sie erfahren also, welches Kontrollelement das Ereignis auslöste. Einfache Elemente wie Buttons können nicht mehr als klicken. Eine weitere Untersuchung dürfte sich dann erledigen. Eine Liste könnte dagegen einen Selektionswechsel oder einen Doppelklick erlebt haben.
- Die Methode `getActionCommand` liefert einen String, der das Ereignis beschreibt. Bei einem Button erscheint hier die Beschriftung.
- Die Methode `getModifiers` liefert die Kontrolltasten, die während des Ereignisses gedrückt waren. Ein Button könnte beispielsweise `button1` lauten, wenn die linke Maustaste gedrückt wurde.
- Die Methode `getWhen` liefert einen Zeitstempel.
- Die Methode `paramString` liefert einen String, in dem alle Informationen zusammengefasst sind.

Druckvolle Buttons

Die Action kann beginnen. Wir wählen den Button als ersten Auslöser von Ereignissen.

1. Der Konstruktor des Fensterrahmens erzeugt einen Button, der unter Swing den

Namen `JButton` trägt.

2. Mit der `JFrame`-Methode `add` bauen Sie den Button an passender Stelle in das Fenster ein.
3. Sie melden das eigene Objekt per `addActionListener` als `ActionListener` an, weil Sie die Klasse bereits als Implementierung des Interface `ActionListener` definiert hatten.
4. Darum mussten Sie die Methode `actionPerformed` implementieren, in der Sie erfahren, dass etwas passiert ist.

```
import javax.swing.*;
import java.awt.event.*;

public class ButtonAction extends JFrame
                          implements ActionListener {
    public static void main(String[] args) {
        new ButtonAction();
    }

    JButton click;  // Für alle zugreifbar

    public ButtonAction() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        click = new JButton("Drueck mich!");
        click.addActionListener(this);
        add(click);
        pack();
        setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent ev) {
        if (ev.getSource()==click) {
            JOptionPane.showMessageDialog(null,
                    "Sehr druckvoll!");
        }
    }
}
```

Listing 18.2: JButton möchte gern gedrückt werden.

Mit dem Aufruf der Methode `getSource` ermitteln Sie, ob wirklich der `JButton` `click` das Ereignis ausgelöst hat. Alternativ könnte man zur Bestimmung des

Buttons auch `getActionCommand` aufrufen. Der zurückgegebene String enthält die Beschriftung des Buttons.

Die Methoden, die Sie bei `JLabel` kennengelernt haben, funktionieren auch bei `JButton`. So lassen sich Buttons auch mit Symbolen versehen. Sie können den Text nachträglich mit `setText` ändern und mit `getText` auslesen.

Starten Sie das Programm. Nach dem Anklicken müsste eine Messagebox erscheinen, die die Qualität des Anklickens beschreibt. Abbildung 18.3 zeigt das Ergebnis.

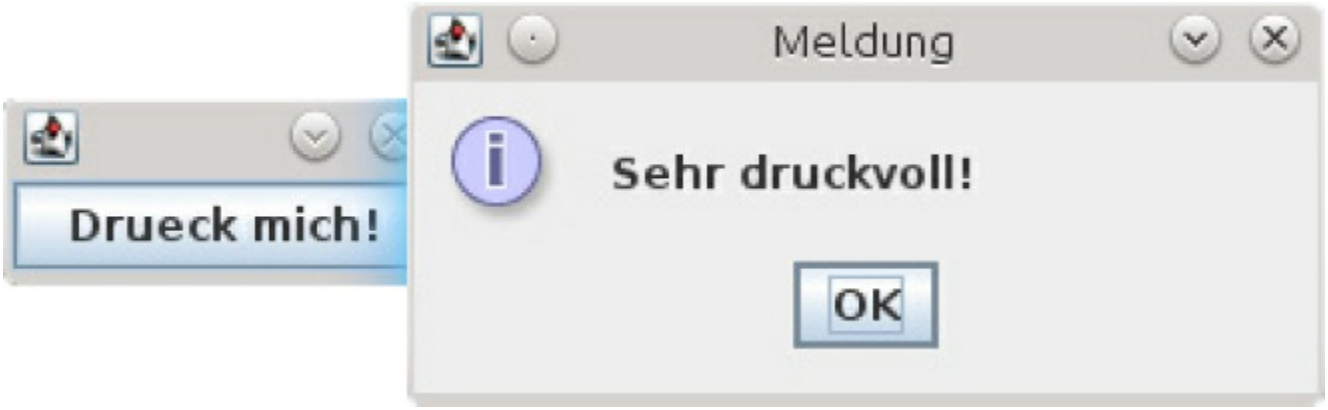


Abbildung 18.3: Wenn man Druck auf einen Button ausübt

Ein BMI-Programm als Spielwiese

Damit nicht alles so trocken ist, sollen die Kontrollelemente an einem langsam wachsenden Programm demonstriert werden. Langsam wachsend? Da fällt mir mein Bauch ein. Und was ist da naheliegender, als den BMI zu berechnen? BMI steht für Body-Mass-Index und stellt fest, ob ein Mensch für sein Gewicht groß genug ist.

Wie der BMI funktioniert und wie er berechnet wird, finden Sie etwas ausführlicher auf der Seite `www.bmi-rechner.net`. Es ist eine der frustrierendsten Seiten des Internets, vor allem, wenn man wie der Autor etwas untergroß ist und die Aussicht auf Wachstum nach oben gering ist.

Um den BMI zu berechnen, muss man das Gewicht durch das Quadrat der Größe teilen. Je nach Alter und Geschlecht gibt es noch den einen oder anderen Zu- oder Abschlag. Der errechnete Index wird dann mit ein paar Grenzwerten verglichen, die entscheiden, ob man fettleibig, normalgewichtig oder ein Fall für die künstliche Ernährung ist.

Die Eingabefelder

Der Button ist als aktives Element natürlich wichtig, weil er gebraucht wird, um weitere Verarbeitungen anzustoßen. Für Daten wie Namen, Telefonnummern, Gewicht oder Geldbeträge wird ein Eingabefeld benötigt, das unter Swing

`JTextField` heißt.

```
JTextField eingabeFeld = new JTextField();
```

Der Konstruktor akzeptiert Parameter:

- ✔ Wird eine ganze Zahl übergeben, wird damit die Anzahl der Zeichen festgelegt, die das Eingabefeld akzeptieren soll.
- ✔ Sie können auch einen String übergeben. Der wird dann als Vorgabe in das Eingabefeld gesetzt.

Den Inhalt des Eingabefelds können Sie mit der Methode `getText` wieder auslesen. Sie müssen sich also keinen neuen Namen merken. Und Sie ahnen schon, dass Sie mit `setText` einen String in das Eingabefeld hineinstellen können.

Das Eingabefeld arbeitet nur mit Strings. Benötigen Sie einen Zahlenwert, so müssen Sie den String in eine Zahlenvariable umwandeln.

```
eingabeFeld.setText("1.4142135");
String inhalt = eingabeFeld.getText();
double wert = Double.parseDouble(inhalt);
```

Berechnung und Bewertung des BMI

Das Beispielprogramm in Listing 18.3 ermittelt den Body-Mass-Index aus den Eingaben des Anwenders zu Größe und Gewicht und bewertet, ob ihm diese Kombination gefällt (siehe Abbildung 18.4).

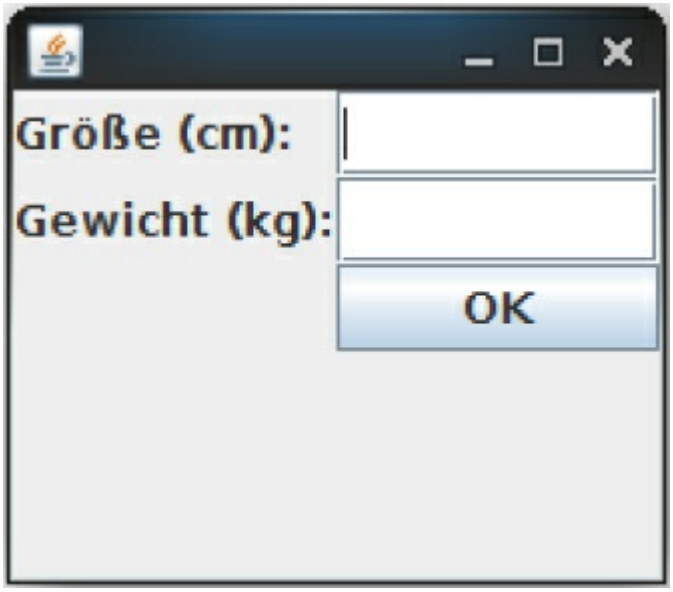


Abbildung 18.4: Body-Mass-Index-Rechner

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

public class BMI extends JFrame implements ActionListener {
    public static void main(String[] args) {
        new BMI();
    }

    JTextField txtGroesse, txtGewicht;
    JLabel labMeldung, labBmi;

    public BMI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new GridLayout(4, 2));
        JLabel labGroesse = new JLabel("Größe (cm):");
        JLabel labGewicht = new JLabel("Gewicht (kg):");
        labMeldung = new JLabel(" ");
        labBmi = new JLabel(" ");
        txtGroesse = new JTextField();
        txtGewicht = new JTextField();
        JButton btnStart = new JButton("OK");
        btnStart.addActionListener(this); // Aktivieren!
        // Zusammenbau aller Elemente
        add(labGroesse);
        add(txtGroesse);
        add(labGewicht);
        add(txtGewicht);
        add(labBmi);
        add(btnStart);
        add(labMeldung);
        pack();
        setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent ev) {
        if (txtGroesse.getText().length() > 0
                && txtGewicht.getText().length() > 0) {
            try {
                // Der Eingabe-String wird zur Zahl gewandelt
                int groesse =
                        Integer.parseInt(txtGroesse.getText());
                int gewicht =
                        Integer.parseInt(txtGewicht.getText());
                double bmi =
```

```
                    gewicht / (groesse*groesse/10000.0);
                // Eine Nachkommastelle reicht beim BMI
                bmi = Math.round(bmi * 10.0) / 10.0;
                labBmi.setText("BMI = " + bmi);
                if (bmi < 20.0) {
                    labMeldung.setText("Untergewicht");
                } else if (bmi <= 25.0) {
                    labMeldung.setText("Normalgewicht");
                } else if (bmi <= 30) {
                    labMeldung.setText("übergewicht");
                } else if (bmi <= 40) {
                    labMeldung.setText("Adipositas");
                } else {
                    labMeldung.setText("Starke Adipositas");
                }
            } catch (NumberFormatException ex) {
                labMeldung.setText("Fehlerhafte Eingabe");
            }
        } else {
            labMeldung.setText("Fehlende Eingaben");
        }
    }
}
```

Listing 18.3: Gewichtsinterpretation durch BMI

Der Druck auf den Button bewirkt den Start der Methode `actionPerformed`. Dort wird zunächst geprüft, ob die Eingabe korrekt ist. Die Exception für die Umwandlung des Strings in eine Zahl wird gefangen. Und dann wird in einer gestaffelten `if`-Anweisung geprüft, in welche BMI-Kategorie das Verhältnis aus Gewicht und Größe fällt. Das Ergebnis wird dann per `setText` im Label angezeigt.

Der Aufruf von `repaint` ist nicht erforderlich, da Kontrollelemente die Anzeige sofort automatisch durchführen.

Abarten des Eingabefelds

Eine Abwandlung von `JTextField` ist `JPasswordField`, mit dem Sie Passwörter einlesen können. Die Eingabe wird dabei verdeckt.

Das `JTextArea` stellt ein mehrzeiliges Eingabefeld dar. Damit wird ein kleiner Editor in die Anwendung mit eingebaut. Das ist auch dann nützlich, wenn Sie innerhalb einer Anwendung einen längeren Text darstellen wollen. Soll der Text wirklich nur gelesen werden, kann er durch den Aufruf der Methode `setEditable(false)` vor einer Veränderung durch den Benutzer geschützt werden.

Fallunterscheidung mit Radiobuttons

Radiobuttons haben ihren Namen von den Stationstasten am Radio. Es kann immer nur ein Sender gleichzeitig laufen, egal wie viele Stationstasten am Radio vorhanden sind. Wenn Sie solche Radiobuttons auch in Ihrem Programm nutzen wollen, gehen Sie in folgenden Schritten vor:

1. Jede Stationstaste wird als Objekt der Klasse `JRadioButton` angelegt. Tatsächlich sind ja die Radiobuttons Sonderfälle eines Buttons.
2. Die Stationstasten müssen zu einer `ButtonGroup` zusammengefasst werden. Sie sorgt dafür, dass bestimmt nicht mehr als ein Button gleichzeitig gedrückt ist.
3. Wie jedes Kontrollelement muss auch jeder Radiobutton mit der Fenstermethode `add` hinzugefügt werden.
4. Falls erforderlich, wird die Methode `addActionListener` aufgerufen, um Wechsel-er-eig-nisse zu fangen.

Auch Radionbuttons erzeugen Events, wenn sie umgeschaltet werden. Sind die Radiobuttons Bestandteil eines größeren Dialogs, ist dies oft nicht erforderlich, weil Sie erst nach Drücken des OK-Buttons die Dialogelemente auslesen.

Wenn Sie dagegen bei jedem Wechsel eine Aktivität auslösen wollen, muss `ActionListener` definiert werden und mit der Methode `addActionListener` an den Button gebunden werden. Das ist nützlich, wenn beim Wechsel der Auswahl beispielsweise ein wechselnder Hinweistext erscheinen soll. Listing 18.4 führt dies am Beispiel eines Wahlzettels vor, der die Möglichkeiten »Ja«, »Nein« und »Enthaltung« zulässt.

```java
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.ButtonGroup;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JRadioButton;

public class RadioButton extends JFrame
        implements ActionListener {
    public static void main(String[] args) {
        new RadioButton();
    }

    JLabel beschriftung = new JLabel("");

    public RadioButton() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        JRadioButton radioJa = new JRadioButton("Ja");
        JRadioButton radioNein = new JRadioButton("Nein");
        JRadioButton radioEnthaltung =
                new JRadioButton("Enthaltung");
        // Zu einer logischen Einheit zusammenfassen
        ButtonGroup grpGeschlecht = new ButtonGroup();
        grpGeschlecht.add(radioJa);
        grpGeschlecht.add(radioNein);
        grpGeschlecht.add(radioEnthaltung);
        // Einen Radiobutton anwählen
        radioEnthaltung.setSelected(true);
        // Ins Fenster einbauen
        setLayout(new GridLayout(4,1));
        this.add(radioJa);
        this.add(radioNein);
        this.add(radioEnthaltung);
        this.add(beschriftung);
        // Muss nicht sein: Aktivität aktivieren
        radioJa.addActionListener(this);
        radioNein.addActionListener(this);
        radioEnthaltung.addActionListener(this);
        pack();             // Zusammensetzen
        setVisible(true);  // Sichtbar machen
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        JRadioButton obj = (JRadioButton) e.getSource();
        beschriftung.setText(obj.getText());
```

```
    }
}
```

Listing 18.4: Elektronischer Stimmzettel

Sie können einen Radiobutton mit der Methode `isSelected` auslesen. Liefert die Methode `true` zurück, ist dieser Button angewählt.

BMI

Bei der Berechnung des BMI gilt der Grundsatz, dass Männer ungestraft fetter werden dürfen. Das erzürnt sicher die eine oder andere Gleichstellungsbeauftragte, sorgt aber für Kasse bei den Diätanbietern. Es ist einfach ungerecht, dass Frauen größer sein müssen, um bei gleichem Gewicht den gleichen BMI zu bekommen.

Unser BMI-Programm soll allen Gleichstellungsideen zum Trotz zwischen Männern und Frauen unterscheiden. Das sind übersichtlich viele Möglichkeiten und ein klarer Fall für Radiobuttons.

```
JRadioButton radioMaennlich = new JRadioButton("männlich");
JRadioButton radioWeiblich = new JRadioButton("weiblich");
```

Sie können in das Fenster eingesetzt und angeklickt werden. Allerdings kann man zunächst noch sowohl männlich als auch weiblich sein. Um die typische Radio-Funktion zu erreichen, müssen die Buttons einer `ButtonGroup` zugeordnet werden. Die kümmert sich dann darum, dass nur einer der Buttons gedrückt ist.

```
ButtonGroup grpGeschlecht = new ButtonGroup();
grpGeschlecht.add(radioMaennlich);
grpGeschlecht.add(radioWeiblich);
radioMaennlich.setSelected(true);
```

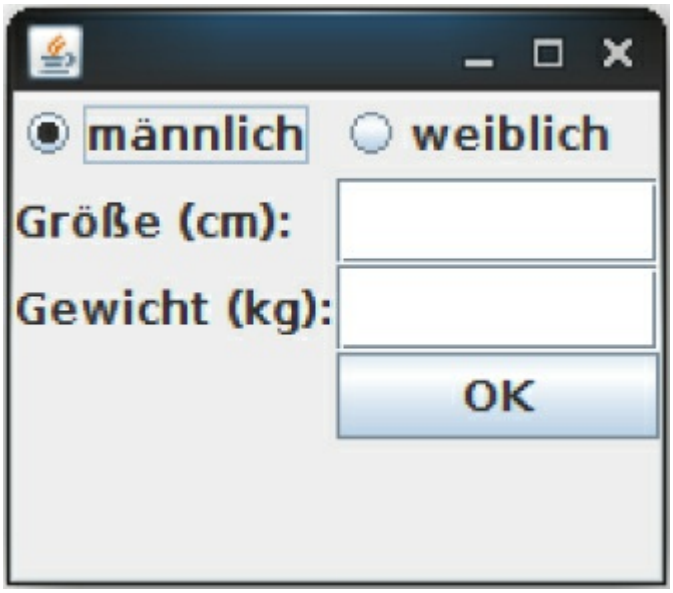


Abbildung 18.5: Body-Mass-Index-Rechner mit Radiobutton

Wir bauen in die oberste Zeile des BMI-Rechners einen Radiobutton für das

Geschlecht ein. Bei Frauen ist der Grenzwert zur nächsten Stufe bereits einen Punkt früher erreicht als bei Männern.

Ob ein Radiobutton gedrückt ist, kann über den Aufruf der Methode `isSelected` ermittelt werden.

Radiobuttons im Eigenbau

Versuchen Sie selbst einmal, die Radiobuttons einzubauen. Mit den angegebenen Informationen ist das möglich. Sie müssen die Auswertung des BMI-Programms noch so ergänzen, dass Frauen einen Punkt höher gesetzt werden. Und weil ich heute noch Fahrrad fahren will, um meinen BMI wenigstens außerhalb der übelsten Peinlichkeit zu bekommen, gibt es keine Musterlösung. Ist bestimmt auch gar nicht nötig! Das schaffen Sie schon!

Bevor Sie aber in Aufregung über den faulen Autor geraten – was übrigens auch den Blutdruck steigen lässt und gar nicht gut für Sie ist – können Sie sich das Listing im folgenden Abschnitt anschauen. Dort finden Sie eine Musterlösung für die Radiobuttons, die allerdings auch schon die Combobox des nächsten Abschnitts enthält.

Combobox: Auswahl aus einer Liste

Eine Combobox ist eine Liste, die ausklappt, wenn man auf den Button auf der rechten Seite klickt. Damit lassen sich Listen kompakt darstellen. Sicher genauso häufig werden sie als Ersatz für Radiobuttons verwendet.

Sie haben den fantasiebegabten Mensakoch bereits im Zusammenhang mit der Polymorphie kennengelernt. Er möchte nun seine Speisen über eine Combobox auswählen. Darüber hinaus möchte er auch neue Speisen eintippen können. Hier nutzt er die Möglichkeit, die Combobox als Texteingabefeld zu verwenden.

Er kann die Combobox mit der Methode `setEditable` dazu bringen, dass ihr Auswahlfeld editierbar ist. Übergeben Sie der Methode `true`, dann machen Sie aus der Combobox ein `JTextField` mit eingebauter Auswahl.

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class Combobox extends JFrame
                      implements ActionListener {
    public static void main(String[] args) {
        new Combobox();
    }

    JLabel beschriftung = new JLabel("");
    JComboBox<String> waehler;
    int index = 0;

    public Combobox() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        String[] fuellung = {"Tomatensuppe",
                "Kartoffelschnee", "Gulaschsuppe",
                "Puree de pommes de terre", "Erbsensuppe"};
        waehler = new JComboBox<String>(fuellung);
        waehler.setEditable(true);
        setLayout(new GridLayout(2, 1));
        this.add(waehler);
        this.add(beschriftung);
        waehler.addActionListener(this);
        pack(); // Zusammensetzen
        setVisible(true); // Sichtbar machen
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource()==waehler) {
            if (e.getActionCommand().equals(
                    "comboBoxEdited")) {
                // Combobox wurde editiert
                String str = (String)
                        waehler.getSelectedItem();
                if (str!=null && !str.isEmpty()) {
                    // Neuen Eintrag in die Liste
                    waehler.addItem(str);
                }
            } else {
                index = waehler.getSelectedIndex();
                if (index>=0) {
```

```
                    System.out.println(index);
                    beschriftung.setText(
                            waehler.getItemAt(index));
                }
            }
        }
    }
}
```

Listing 18.5: Speisenauswahl per Combobox

Sie sehen hier, wie die Ereignisse der Combobox gefangen werden. Über die Methode `getSource` kann festgestellt werden, welches Element die Aktion ausgelöst hat. An sich ist das im Beispiel überflüssig, da es nur ein Kontrollelement gibt.

Im nächsten Schritt wird mit der Methode `getActionCommand` festgestellt, was für ein Ereignis aufgetreten ist. An der Kennung »comboBoxEdited« wird erkannt, dass der Anwender den Text in der Combobox verändert und mit der Taste Return abgeschlossen hat. Der Text kann dann per `getSelectedItem` ausgelesen werden.

Eines der eher typischen Combobox-Ereignisse ist das Auswählen eines anderen Elements aus der Liste. In diesem Fall kann mit `getSelectedIndex` festgestellt werden, welches der Listenelemente jetzt angewählt ist.

BMI

Bei gleichem Gewicht und gleicher Größe wird der BMI besser, je älter man ist. Statt Fahrrad zu fahren oder zu joggen, setzt man sich einfach hin und altert. Auch eine Methode, um schlank zu werden.

Die BMI-Tabelle besagt, dass es Grenzen im Alter von 25, 35, 45, 55 und 65 gibt. Beim Überschreiten jeder dieser Grenzen darf man sich einen BMI-Punkt mehr gönnen oder, anders ausgedrückt, ziehen wir je einen Punkt ab.

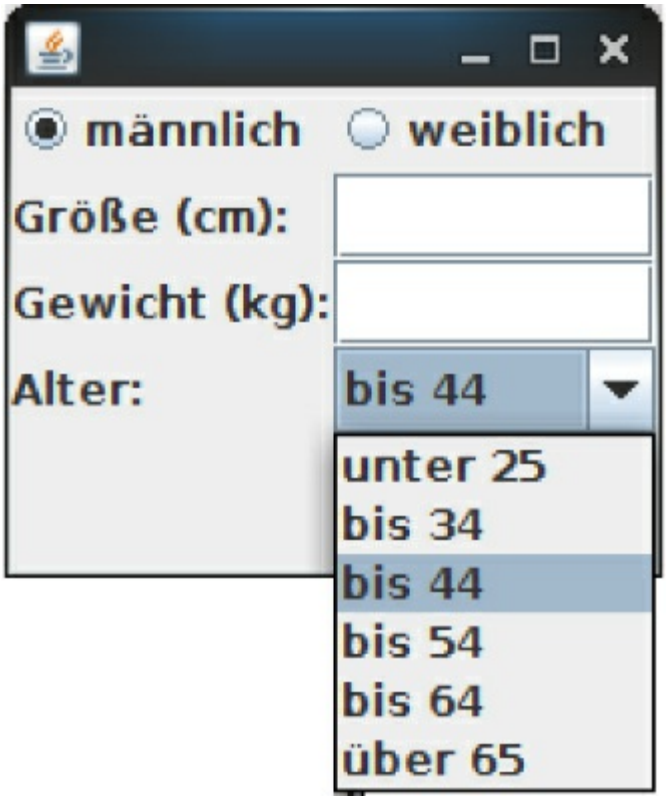


Abbildung 18.6: Body-Mass-Index-Rechner mit Combobox

Das könnten wir wieder mit Radiobuttons lösen. Das gäbe dann aber sechs Radiobuttons mit Beschriftung, was viel zu viel Platz einnimmt. Schicker ist eine Combobox. Eine Combobox besteht aus einem Eintrag mit einem Button. Klickt man auf den Button, wird eine Liste ausgeklappt, aus der man einen Eintrag auswählen kann.

Vor dem Erstellen einer Combobox legen Sie ein String-Array an, in dem die Einträge der Liste abgelegt sind. Dieses String-Array wird dem Konstruktor von `JComboBox` als Parameter übergeben und schon ist die Combobox fertig. Statt des Arrays kann auch eine `ArrayList` verwendet werden.

```
String[] strAlterWahl = { "unter 25", "bis 34", "bis 44",
                          "bis 54", "bis 64", "über 65" };
JComboBox wahlAlter = new JComboBox(strAlterWahl);
```



Java 7 gibt hier eine Warnung aus. Der Compiler klagt, dass der Typ der Combobox nicht geprüft werden kann. Seit Java 7 sieht der korrekte Code an dieser Stelle folgendermaßen aus:

```
String[] strAlterWahl = { "unter 25", "bis 34", "bis 44",
                          "bis 54", "bis 64", "über 65" };
JComboBox wahlAlter = new JComboBox<String>(strAlterWahl);
```

Der Compiler kann nun prüfen, ob das übergebene Array vom Typ `String` ist. Die spitzen Klammern legen dies fest. Hintergrund ist, dass Sie in einer Combobox nicht nur aus Strings, sondern auch aus komplexeren Typen auswählen.

Wollen Sie allerdings die Kompatibilität zu älteren Java-Laufzeitsystemen erhalten, dann müssten Sie den bisherigen Code belassen. Sie könnten die Warnungsmeldung des Compilers unterdrücken, indem Sie der Methode eine Annotation `@SuppressWarnings` voranstellen, womit Sie dem Compiler sagen: »Ja, ich weiß, dass es hier eine Warnung gibt. Ich habe das geprüft und halte dieses Vorgehen für ungefährlich.«

```
@SuppressWarnings({"unchecked"})
public BMIAlter() {
```

Auswahl ermitteln

Für unser Programm ist interessant, welcher der Einträge vom Benutzer ausgewählt wurde. Dazu wird die Methode `getSelectedIndex` aufgerufen, die die Nummer des Eintrags angibt. Wurde kein Element ausgewählt, liefert die Methode -1.

```
int alter = wahlAlter.getSelectedIndex();
if (alter > 0) bmi -= alter;
```

Alternativ kann man mit der Methode `getSelectedItem` auch den String holen, der ausgewählt worden ist.

Hier sehen Sie nun die komplette BMI-Anwendung.

```
import javax.swing.*;
import java.awt.GridLayout;
import java.awt.BorderLayout;
import java.awt.event.*;

public class BMIAlter extends JFrame
                      implements ActionListener {
  JTextField txtGroesse, txtGewicht;
  JLabel labMeldung, labBmi;
  JRadioButton radioMaennlich, radioWeiblich;
  JComboBox<String> wahlAlter;

  public BMIAlter() {
    JLabel labGroesse, labGewicht;
    JButton btnStart;

    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // der Hauptcontainer bekommt ein Borderlayout
    setLayout(new BorderLayout());
    // hinein wird ein kleiner JPanel gesetzt
    JPanel innen = new JPanel();
    add(innen, BorderLayout.CENTER);
    innen.setLayout(new GridLayout(5, 2));
    // erzeuge die Kontrollelemente
    // Radio für M/W
    radioMaennlich = new JRadioButton("männlich");
    radioWeiblich = new JRadioButton("weiblich");
    ButtonGroup grpGeschlecht = new ButtonGroup();
    grpGeschlecht.add(radioMaennlich);
    grpGeschlecht.add(radioWeiblich);
    radioMaennlich.setSelected(true);
    // Die Labels
    labGroesse = new JLabel("Größe (cm):");
    labGewicht = new JLabel("Gewicht (kg):");
    JLabel labAlter = new JLabel("Alter:");
    labMeldung = new JLabel(" ");
    labBmi = new JLabel(" ");
    // Die Eingabefelder
    txtGroesse = new JTextField();
    txtGewicht = new JTextField();
    // Die Combobox für das Alter
    String[] strAlterWahl = { "unter 25", "bis 34", "bis 44",
                              "bis 54", "bis 64", "über 65" };
    wahlAlter = new JComboBox<String>(strAlterWahl);
    // Der Startbutton
    btnStart = new JButton("OK");
    // Nur der Button ist aktiv
```

```
  btnStart.addActionListener(this);
  // Zusammenbau aller Elemente
  innen.add(radioMaennlich);
  innen.add(radioWeiblich);
  innen.add(labGroesse);
  innen.add(txtGroesse);
  innen.add(labGewicht);
  innen.add(txtGewicht);
  innen.add(labAlter);
  innen.add(wahlAlter);
  innen.add(labBmi);
  innen.add(btnStart);
  // So kann die Meldung über die gesamte Breite gehen
  add(labMeldung, BorderLayout.SOUTH);
  // Zusammenpacken und darstellen
  pack();
  setVisible(true);
}

@Override public void actionPerformed(ActionEvent ev)
{
  // Das Ereignis kann nur vom Button kommen.
  if (txtGroesse.getText().length() > 0
  && txtGewicht.getText().length() > 0) {
    try {
      int groesse = Integer.parseInt(txtGroesse.getText());
      int gewicht = Integer.parseInt(txtGewicht.getText());
      double bmi = gewicht / (groesse*groesse / 10000.0);
      // Eine Nachkommastelle reicht beim BMI
      bmi = Math.round(bmi * 10.0) /10.0;
      labBmi.setText("BMI = " + bmi);
      // Korrigiere BMI bei Frauen
      if (radioWeiblich.isSelected()) {
         bmi++;
      }
      // Alterszuschlag
      int alter = wahlAlter.getSelectedIndex();
      if (alter > 0) bmi -= alter;
      if (bmi < 20.0) {
        labMeldung.setText("Untergewicht");
      } else if (bmi <= 25.0) {
        labMeldung.setText("Normalgewicht");
      } else if (bmi <= 30) {
        labMeldung.setText("Übergewicht");
      } else if (bmi <= 40) {
        labMeldung.setText("Adipositas");
      } else {
```

```
            labMeldung.setText("Starke Adipositas");
          }
        } catch (NumberFormatException ex) {
          labMeldung.setText("Fehlerhafte Eingabe");
        }
      } else {
        labMeldung.setText("Fehlende Eingaben");
      }
    }

    public static void main(String[] args) {
      new BMIAlter();
    }
}
```

Listing 18.6: BMI kann nun auch Alter und Geschlecht berücksichtigen.

Eine doch nicht so einfache Liste

Ein naher Verwandter der `JComboBox` ist `JList`. Die Liste klappt nur nicht aus, sondern zeigt die komplette Liste an. Sie braucht zwar mehr Raum, ist aber übersichtlicher. Vor allem, wenn mehrere Elemente selektiert werden sollen, ist eine `JList` vorzuziehen.

Eine `JList` kommt fast nie allein. Sie benötigen fast immer Kollegen:

- Da sind die Schiebebalken, die eine größere Liste sichtbar machen. Freundlicherweise übernimmt diese Rolle `JScrollPane`. Sie brauchen ihn nur anzulegen, eine `Jlist` hineinzupacken und diesen statt der Liste in das Elternpanel einzubauen. `JScrollPane` erledigt dann den Rest.
- Die Liste speichert ihre Daten in einem separaten Datenmodell. Dies läuft unter dem Namen `DefaultListModel` und verwaltet die Daten, wie das Löschen oder Hinzufügen. `JList`übernimmt den repräsentativen Teil.

Das Beispiel soll dem Benutzer eine Liste anzeigen, aus der er Elemente mit einem Button in eine andere Liste hinüberschaffen kann, aber auch wieder zurück. So etwas benötigt man immer wieder, wenn ein Benutzer eine Auswahlliste selbst erstellen soll.

Für die Liste auf der linken Seite wird zunächst das `DefaultListModel` erstellt. Hier wird der Typparameter `String` verwendet, damit die Typsicherheit gewährleistet ist. Dieses Modell wird mit einer Liste von Köstlichkeiten aus der Mensa über die Methode `addElement` gefüllt. Dieses Modell erhält die Liste als

Konstruktorparameter. Damit sind Modell und Liste miteinander verbunden.

```
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.List;
import javax.swing.*;

public class Listen extends JFrame
                    implements ActionListener {

    JButton btLks = new JButton("<-");
    JButton btRts = new JButton("->");
    DefaultListModel<String> modellLks =
            new DefaultListModel<String>();
    DefaultListModel<String> modellRts =
            new DefaultListModel<String>();
    JList<String> listeLks;
    JList<String> listeRts;

    public static void main(String[] args) {
        new Listen();
    }

    public Listen() {
        // Baue den Inhalt der linken Liste
        String[] fuellunglks = { "Tomatensuppe",
                "Kartoffelschnee", "Gulaschsuppe",
                "Puree de pommes de terre", "Erbsensuppe" };
        for (String element : fuellunglks) {
            modellLks.addElement(element);
        }
        listeLks = new JList<String>(modellLks);
        listeRts = new JList<String>(modellRts);
        // Eine Liste braucht fast immer eine Scrollbar
        JScrollPane scrollLks = new JScrollPane(listeLks);
        JScrollPane scrollRts = new JScrollPane(listeRts);
        // mach die Buttons scharf
        btRts.addActionListener(this);
        btLks.addActionListener(this);
        // --- Baue das Layout
        // Zwei Buttons untereinander in Grid
        JPanel btPanel = new JPanel();
        btPanel.setLayout(new GridLayout(2, 1));
        JPanel panRts = new JPanel();
        // Verhindere Aufblähen durch FlowLayout
        panRts.setLayout(new FlowLayout());
        panRts.add(btRts);
```

```
        btPanel.add(panRts);
        JPanel panLks = new JPanel();
        panLks.setLayout(new FlowLayout());
        panLks.add(btLks);
        btPanel.add(panLks);
        // 3 Spalten: Liste, Buttons, Liste
        setLayout(new GridLayout(1, 3));
        this.add(scrollLks);
        this.add(btPanel);
        this.add(scrollRts);
        // Rahmen fertigstellen
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack(); // Zusammensetzen
        setVisible(true); // sichtbar machen
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == btLks) {
            List<String> elements =
                    listeRts.getSelectedValuesList();
            for (String element : elements) {
                modellLks.addElement(element);
                modellRts.removeElement(element);
            }
        } else if (e.getSource() == btRts) {
            List<String> elements =
                    listeLks.getSelectedValuesList();
            for (String element : elements) {
                modellRts.addElement(element);
                modellLks.removeElement(element);
            }
        }
    }
}
```

Listing 18.7: Von der linken Liste in die rechte

Wenn einer der Buttons gedrückt wird, werden über die Methode `getSelectedValuesList` die selektierten Elemente der Liste abgefragt. Hier ist die Liste gefragt, weil sie für die Benutzeraktionen zuständig ist.

Anschließend werden die selektierten Elemente in der jeweils anderen Liste mit

`addElement` eingehängt und aus der alten Liste mit `removeElement` ausgetragen. Dafür muss das Modell angesprochen werden, da dieses die Daten verwaltet. Die optische Rückmeldung in Form der sichtbaren Liste übernimmt das Modell automatisch.

Ein kurzer Blick auf das Layout

Das Layout wird aus einer Verschachtelung zweier `GridLayouts` erzeugt. In dem äußeren Grid werden linke Liste, Buttonleiste und rechte Liste nebeneinander angeordnet.

Die Buttonleiste ist wieder ein `GridLayout`, das dafür sorgt, dass die Buttons übereinander stehen. Würde man es dabei belassen, würden die Buttons allen zur Verfügung stehenden Raum einnehmen, was nicht schön aussieht. Also erhält jeder Button ein eigenes `FlowLayout`, was ihn auf die wirklich notwendige Größe reduziert.

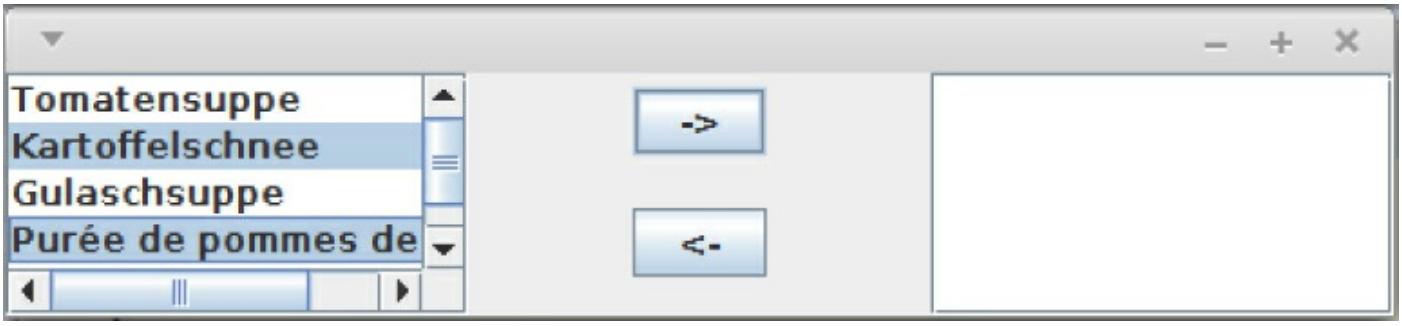


Abbildung 18.7: Eine Auswahlliste

Der JTable und sein Modell

Ein `JTable` ist ein Swing-Kontrollelement und stellt eine Tabelle zur Verfügung, wie man sie von Tabellenkalkulationen kennt.

Erstellen einer Tabelle

Das Programm in Listing 18.8 erstellt eine einfache Tabelle einer Hitparade. Die Tabelle wird bei der Erstellung durch String-Arrays befüllt.

```
import javax.swing.JFrame;
import javax.swing.JTable;
import javax.swing.JScrollPane;

public class JTable1Simpel extends JFrame {
    public static void main(String[] args) {
        new JTable1Simpel();
    }

    JTable1Simpel() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        String [][] inhalt = { {"Beatles","Help"},
                               {"Beatwatt","Is That All"},
                               {"ABBA","Waterloo"} };
        String[] titel = {"Interpret", "Titel"};
        JTable table = new JTable(inhalt, titel);
        add(new JScrollPane(table));
        setSize(400, 300);
        setVisible(true);
    }
}
```

Listing 18.8: JTable-Hitparade

Mit diesen wenigen Zeilen lässt sich bereits eine recht hübsche Tabelle zaubern, wie sie in Abbildung 18.8 zu sehen ist. Ein Rollbalken ist sowieso sinnvoll. Aber interessanterweise fehlt der Tabelle die Überschrift, wenn man den Rollbalken weglässt.

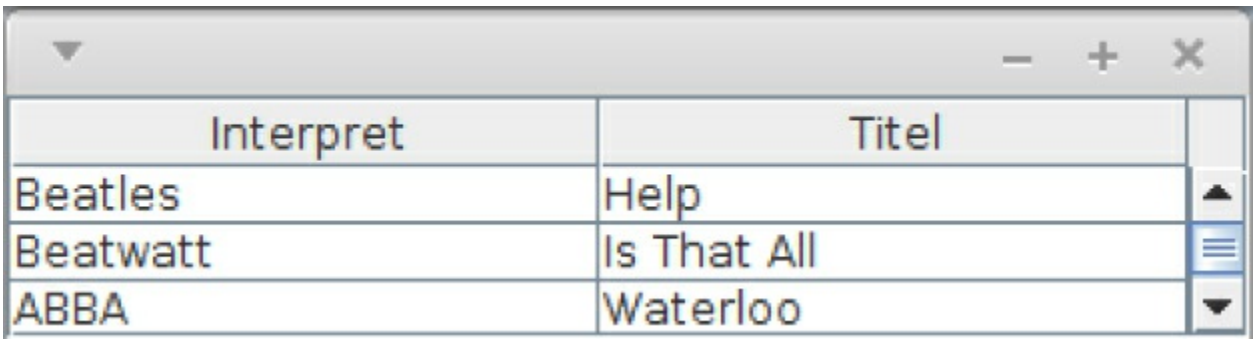


Abbildung 18.8: Eine einfache Hitparade mit einem `JTable`

Das Modell füllt die Tabelle

Das Auffüllen mit Strings über den Konstruktor ist nur interessant, wenn sich die Tabelle niemals ändert. Änderungen in der Tabelle durchzuführen, ist nicht vorgesehen. Hier trennt Swing Ansicht (View) und Daten (Model).

Eine `JTable` kann mit einem `TableModel` verknüpft werden, das für die Tabelle die Daten hält. Dieses füttert die `JTable` mit Daten.

Zunächst aber sehen Sie in Listing 18.9 das Hauptprogramm, das sich vom bisherigen nur geringfügig unterscheidet.

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;

public class JTable2Main extends JFrame {
    public static void main(String[] args) {
        new JTable2Main();
    }

    public JTable2Main() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JTable2Modell data = new JTable2Modell();
        JTable table = new JTable(data);
        add(new JScrollPane(table));
        setSize(400, 300);
        setVisible(true);
    }
}
```

Listing 18.9: Hauptprogramm für den `JTable` mit Datenmodellierer

Erweiterung von AbstractTableModel

Die Daten werden in einer Erweiterung der abstrakten Klasse `AbstractTableModel` verwaltet. Diese fordert die Implementierung folgender Methoden ein:

- ✔ `getColumnCount`: Die Methode liefert die Anzahl der Spalten.
- ✔ `getRowCount`: Die Methode liefert die Anzahl der Zeilen.
- ✔ `getValueAt`: Die Methode liefert den Dateninhalt an der Position, die als X- und Y-Koordinate übergeben wird.

Darüber hinaus kann es sich lohnen, folgende Methoden zu überschreiben.

- ✔ `getColumnName`: Optional kann diese Methode überschrieben werden, die die Spaltenüberschrift zurückgibt. Als Parameter wird die Spaltennummer übergeben.
- ✔ `setValueAt`: So können Sie einzelne Zellen mit Werten füllen. Als Parameter erwartet die Methode den Feldinhalt und die Koordinaten.

Somit ist die Modellklasse dafür zuständig, die Daten für die Tabelle zu beschaffen, wenn sie benötigt werden. Wenn Sie zu Listing 18.9 noch die Klasse in Listing 18.10 hinzufügen, gelingt es Ihnen, äußerlich das gleiche Programm zu schreiben, das Sie schon aus Abbildung 18.8 kennen. Immerhin haben Sie ein längeres Listing produziert.

```
import java.util.ArrayList;

import javax.swing.table.AbstractTableModel;

public class JTable2Modell extends AbstractTableModel {
    String [][] inhalt = { {"Beatles","Help"},
            {"Beatwatt","Is That All"},
            {"ABBA","Waterloo"} };
    String[] titel = {"Interpret", "Titel"};
    ArrayList<String[]> data = new ArrayList<String[]>();

    public JTable2Modell() {
        for (String[]hit : inhalt) {
            data.add(hit);
        }
    }

    public void neuerHit(String interpret, String titel) {
        String[] hit = new String[2];
        hit[0] = interpret;
        hit[1] = titel;
        data.add(hit);
    }

    @Override
    public int getRowCount() {
        return data.size();
    }

    @Override
    public int getColumnCount() {
        return 2;
    }

    @Override
    public String getColumnName(int spalte) {
        return titel[spalte];
    }

    @Override
    public Object getValueAt(int zeile, int spalte) {
        return data.get(zeile)[spalte];
    }

    @Override
    public void setValueAt(Object value, int zeile, int spalte) {
        data.get(zeile)[spalte] = (String)value;
```

```
    }
}
```

Listing 18.10: Das Datenmodell für die `JTable`-Hitparade

Mit der Methode `neuerHit` können Sie einfach einen neuen Titel an die Hitparade anhängen. Sobald Sie dies tun, wird automatisch die Sicht aktualisiert. Beispielsweise könnten Sie Ihr Hauptprogramm um die folgende Zeile ergänzen:

```
data.neuerHit("Queen", "Bohemian Rapsody");
```



Anstatt die Klasse `AbstractTableModel` zu erweitern, könnte das Datenmodell auch das Interface `TableModel` implementieren. Zunächst sind die Unterschiede gering. In den ersten Zeilen muss es heißen:

```
import javax.swing.event.TableModelListener;
import javax.swing.table.TableModel;

public class MeinTableModell implements TableModel {
```

Der Compiler wird sich melden und darauf bestehen, dass es da unimplementierte Methoden gibt. Wenn man Eclipse oder Netbeans verwendet, werden diese mit einem Mausklick mit Default-Werten erzeugt. Vor allem sollten Sie bei `getColumnClass` den Rückgabewert nicht bei `null` belassen, sonst ist die Tabelle leer.

```
@Override
public Class getColumnClass(int arg0) {
    return String.class;
}
```

JTable-Methoden

Bei einer Tabelle können Sie durch Anklicken eine Zeile der Tabelle selektieren. Durch Zuhilfenahme der Strg- oder ⇧-Taste können Sie sogar mehrere Zeilen selektieren.

Zum Umgang mit selektierten Zeilen stellt `JTable` einige Methoden zur Verfügung:

- ✔ Der Aufruf von `getSelectedRowCount` ermittelt, wie viele Zeilen selekiert sind.
- ✔ Die Methode `getSelectedRow` liefert den Index der angewählten Zeile.
- ✔ Es ist nun keine Überraschung, dass `getSelectedColumn` den Index der angewählten Spalte liefert.
- ✔ Durch den Aufruf von `clearSelection` werden alle Selektionen aufgelöst.

Die folgenden Zeilen holen den Index der aktuell selektierten Zeile. In der ersten

Zeile wird ermittelt, ob wirklich genau eine Zeile selektiert ist. Dann wird die Zeilennummer ermittelt.

```
if (meinJTable.getSelectedRowCount()==1) {
    int zeile = meinJTable.getSelectedRow();
```



Änderungen an den Daten wie das Löschen oder Einfügen von Zeilen erfolgt direkt am Datenmodell und nicht über die Tabelle. Die Tabelle wird aber automatisch aktualisiert.

Editierer und Renderer

In Tabellen stehen üblicherweise in den Spalten immer gleichartige Daten. So stehen Namen, Adressen, Gehälter oder Geburtsdaten jeweils brav geordnet untereinander. `JTable` nimmt das zum Anlass, Spalten speziell auf einen Datentyp anpassen zu lassen. Man spricht dabei vom *Rendern*.

- Die spezielle Anpassung an den Datentyp erreichen Sie, indem Sie im Datenmodell die Methode `getColumnClass` überschreiben und den Typ der Spalte zurückgeben.
- Sie können sogar Zellen editierbar machen, indem Sie die Methode `isCellEditable` überschreiben. Als Parameter erhalten Sie die X- und Y-Koordinate und können dann `true` oder `false` zurückgeben, je nachdem, ob der Anwender das Feld direkt bearbeiten können soll oder nicht.

  Der dann aktivierte Editor prüft beispielsweise Zahlen, ob sie zulässig sind, und verhindert ein Verlassen des Feldes, bis die Eingabe korrekt ist.

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;

public class JTable3EditRender extends JFrame {
    public static void main(String[] args) {
        new JTable3EditRender();
    }

    public JTable3EditRender() {
        setTitle("EditRender");
        JTable3Model model = new JTable3Model();
        JTable table = new JTable(model);
        JScrollPane scrollPane = new JScrollPane(table);
        add(scrollPane);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }
}
```

Listing 18.11: Hauptprogramm für den editierbaren und gerenderten `JTable`

Der Unterschied liegt im Datenmodell. Für die Aufnahme der Daten wird ein `Object`-Array für vier Spalten aufgesetzt. Sie demonstrieren die Eingabe von Strings, Zahlenwerten und Booleans. Letzerer wird durch einen Haken repräsentiert. Als Datensammler wurde eine `ArrayList` bemüht. Sie ist flexibel. Noch flexibler wäre höchstens noch eine Datenbank.

```
import java.util.ArrayList;
import javax.swing.table.AbstractTableModel;

public class JTable3Model extends AbstractTableModel {
    final String[] spaltenBeschriftung =
        { "String", "Double", "Integer", "Boole" };
    ArrayList<Object[]> dat = new ArrayList<Object[]>();

    public JTable3Model() {
        for (int i = 0; i < 5; i++) {
            Object[] obj = new Object[4];
            obj[0] = "";
            obj[1] = new Double(0.0);
            obj[2] = new Integer(0);
            obj[3] = new Boolean(false);
            dat.add(obj);
        }
    }

    public int getColumnCount() {
        return spaltenBeschriftung.length;
    }

    public int getRowCount() {
        return dat.size();
    }

    public String getColumnName(int col) {
        return spaltenBeschriftung[col];
    }

    public Object getValueAt(int row, int col) {
        return dat.get(row)[col];
    }

    public Class getColumnClass(int c) {
        return getValueAt(0, c).getClass();
    }

    public boolean isCellEditable(int row, int col) {
        return true;
    }

    public void setValueAt(Object value, int row, int col) {
        dat.get(row)[col] = value;
    }
```

```
}
```

Listing 18.12: Das Datenmodell der `JTable`

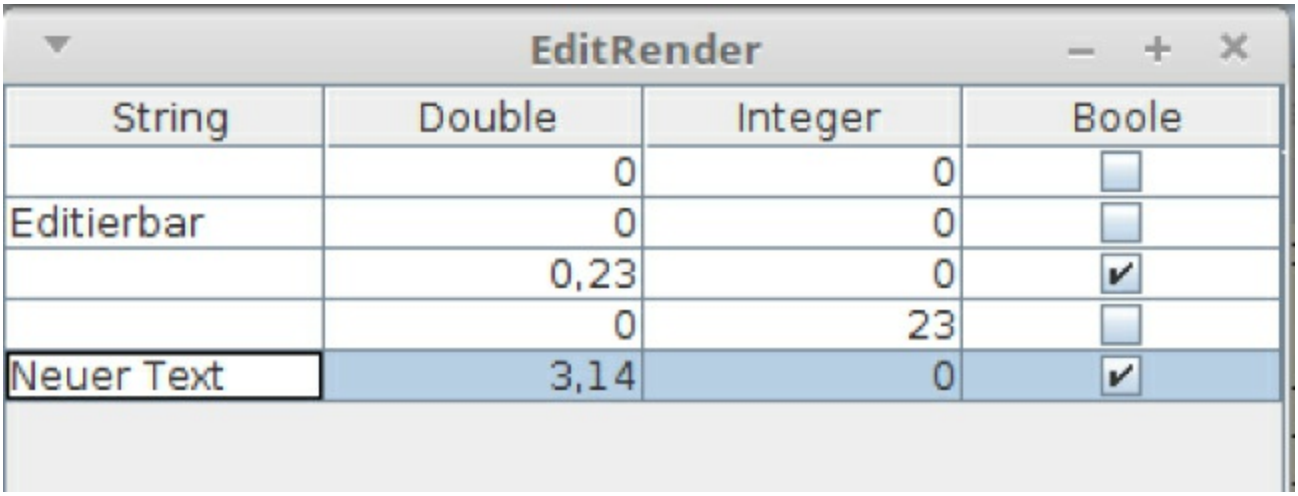


Abbildung 18.9: `JTable` berücksichtigt Typen und ist editierbar.

Wie erfährt die Anwendung von den Aktionen des Benutzers? Alle Änderungen wandern durch die Methode `setValueAt` und können dort abgegriffen werden.

Bäume pflanzen mit JTree

Viele Daten haben eine Baumstruktur. Aus einer Wurzel wachsen Äste, die jeweils wieder Blätter tragen. Für solche Strukturen stellt Swing `JTree` zur Verfügung.

Einen Baum wachsen lassen

1. Ein Baum beginnt damit, einen Knoten zu bilden. Ein Knoten ist ein Objekt der Klasse `DefaultMutableTreeNode`. Der erste Knoten ist die Wurzel.
2. Dem Konstruktor von `DefaultMutableTreeNode` werden Daten als Parameter übergeben.
3. Sie können an einen Knoten über die Methode `add` einen neuen Knoten anhängen. Dadurch wird der Knoten zum Ast.
4. Knoten ohne angehängte Knoten bezeichnet man als Blatt.
5. Der Wurzelknoten wird dem Konstruktor von `JTree` als Parameter für dessen Konstruktor übergeben.

Ein Beispiel für eine solche Datenstruktur zu finden, die nicht völlig trivial ist, aber dennoch nicht so kompliziert wird, dass sie schließlich den Umgang mit `JTree` verdeckt, scheint für den Buchautor zum Dilemma zu werden. Aber nein, die einfachste hierarchische Struktur ist das Dateisystem. Also liest das Programm in Listing 18.13 die Verzeichnisstruktur des Computers.

Eine Baumstruktur wird von Informatikern als rekursive Struktur bezeichnet. Und so ist es auch naheliegend, dass die Methode `durchsuche`, die den Verzeichnisbaum ausliest, rekursiv arbeitet. Damit werden Sie sich anfreunden müssen. Sollten Sie keine rekursive Struktur haben, werden Sie vermutlich auch keinen `JTree` benötigen.

```
import java.io.File;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import javax.swing.tree.DefaultMutableTreeNode;

public class VerzeichnisJTree extends JFrame {
    public static void main(String[] args) {
        new VerzeichnisJTree();
    }

    // Startverzeichnis: Hier das Benutzerverzeichnis
```

```
    final static String wurzel =
            System.getProperty("user.home");

    public VerzeichnisJTree() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        DefaultMutableTreeNode knoten =
                new DefaultMutableTreeNode(wurzel);
        durchsuche(new File(wurzel), knoten);
        JTree baum = new JTree(knoten);
        this.add(new JScrollPane(baum));
        this.setVisible(true);
        this.setSize(400, 300);
    }

    // Durchsuche rekursiv das Verzeichnis
    public void durchsuche(File datei,
            DefaultMutableTreeNode knoten) {
        if (datei.isDirectory()) {
            File[] eintraege = datei.listFiles();
            if (eintraege != null) {
                for (File eintrag : eintraege) {
                    // Ein Knoten für den Eintrag
                    DefaultMutableTreeNode neuKnoten =
                            new DefaultMutableTreeNode(
                                    eintrag.getName());
                    knoten.add(neuKnoten);
                    // Verzeichnis? Auch durchsuchen!
                    if (eintrag.isDirectory()) {
                        durchsuche(eintrag, neuKnoten);
                    }
                }
            }
        }
    }
}
```

Listing 18.13: Das Programm erfasst das Benutzerverzeichnis im JTree.

Je nachdem, wie viele Dateien in Ihrem Benutzerverzeichnis vorliegen, verzögert sich der Start des Programms. Immerhin muss es alle Datei- und Verzeichnisnamen auslesen. Es entsteht ein Fenster, das ähnlich aussieht wie in Abbildung 18.10.

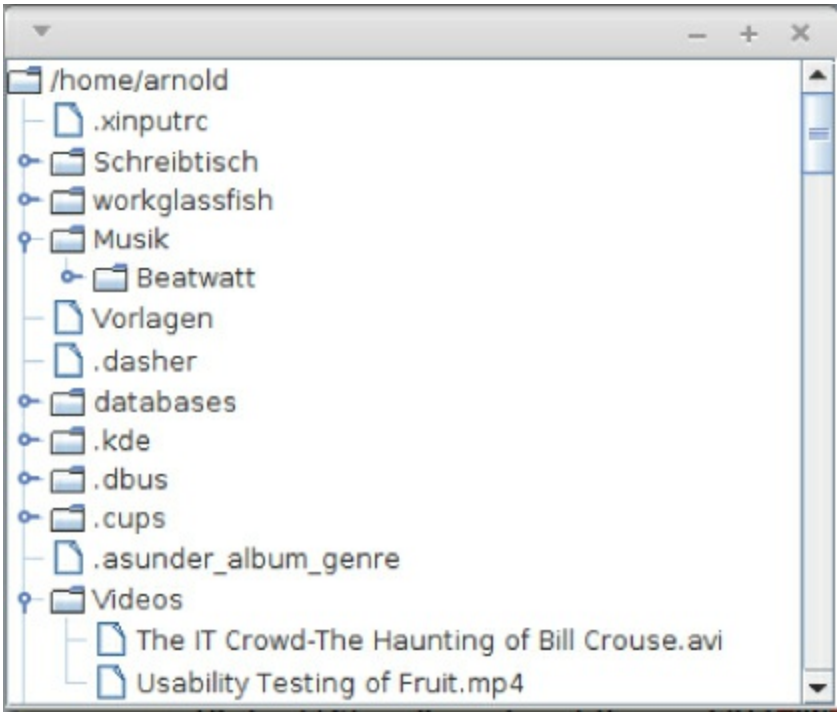


Abbildung 18.10: `JTree` stellt das Benutzerverzeichnis dar.

Sobald Sie auf einen Ordner doppelt klicken, öffnet sich der Ast, der direkt dahinter liegt. Klicken Sie noch einmal doppelt, schließt sich der Ast wieder. Dasselbe können Sie erreichen, indem Sie auf das Symbol vor dem Ordner klicken.



Falls Sie das Wurzelelement nicht jedes Mal anzeigen wollen, verstecken Sie es doch einfach:

```
tree.setRootVisible(false);
```

Ereignisse im Baum

Für manchen Baum ist es ja schon schön, dass er da ist und gut aussieht. Für einen Weihnachtsbaum ist das die einzige Existenzberechtigung. In den meisten Fällen soll das Programm allerdings etwas tun, wenn ein Element des Baums angeklickt wird.

Wenn in den Baum geklickt wird, möchten Sie natürlich wissen, welche Daten angeklickt wurden. Das Ereignis dazu liefert Ihnen ein `TreeSelectionListener`.

1. Zunächst melden Sie den Listener an Ihrem `JTree` an.

   ```
   JTree baum = new JTree(wurzel);
   baum.getSelectionModel().addTreeSelectionListener(this);
   ```

2. Als Parameter erhält die Methode `addTreeSelectionListener` das Objekt der Klasse, das die Selektionsereignisse fängt. Die Klasse muss dazu das Interface `TreeSelectionListener` implementieren.

   ```
   public class VerzeichnisJTree extends JFrame
                implements TreeSelectionListener {
   ```

3. Das Interface `TreeSelectionListener` fordert, dass die Methode `valueChanged` implementiert wird.

```
@Override
public void valueChanged(TreeSelectionEvent e) {
```

Die Event-Listener-Methode `valueChanged` ermittelt aus dem Event-Objekt den Knoten, der angeklickt worden ist. Da das Event auch ausgelöst wird, wenn beispielsweise ein Symbol aufgefächert wurde, kann es sein, dass es keinen solchen Knoten gibt. Er muss also zunächst auf `null` abgefragt werden, bevor über die Methode `getUserObject` das hinterlegte Objekt hervorgezaubert werden kann. In unserem Baum wurden die Dateinamen, also nur Strings hinterlegt.

```
@Override
public void valueChanged(TreeSelectionEvent e) {
    TreePath path = e.getNewLeadSelectionPath();
    DefaultMutableTreeNode node = (DefaultMutableTreeNode)
            path.getLastPathComponent();
    if (node != null) {
        String str = (String) node.getUserObject();
        System.out.println(str);
    }
}
```

Listing 18.14: Auswerten des angeklickten Knotens

Check den Button

Es gibt da noch eine Kategorie von Buttons, die den Haken haben, dass sie einen Haken haben. Darum nennt man sie auch `JCheckBox`. Durch Anklicken werden sie markiert oder abgewählt. Sie kennen diese Haken aus dem Internet, wo Sie angeben müssen, ob Sie die Allgemeinen Geschäftsbedingungen der Firma akzeptieren und erlauben, dass man Ihnen beliebige Summen von Ihrem Konto abbuchen, Ihren Namen, Ihre Adresse und Ihre Bankverbindung weitergeben und verkaufen darf.

Zusätzlich zu ihrer Beschriftung hat die `JCheckBox` also einen Status. Sie kann selektiert sein, also einen Haken tragen oder eben nicht.

- ✔ Dieser Haken wird entweder vom Benutzer gesetzt oder aber von der Applikation durch Aufruf der Methode `setSelected`. Als Parameter erwartet die Methode einen booleschen Wert, der angibt, ob der Haken gesetzt ist oder nicht.
- ✔ Der Zustand des Hakens wird durch die Methode `isSelected` erfragt.

Die Anwendung finden Sie in Listing 18.15. Damit könnten Sie die Einladung zur nächsten Dinner-Party mit einem selbst geschriebenen Java-Programm unterstützen. Sie können anklicken, wer kommen soll, und automatisch eine Einladungsliste erstellen.

In dem Listing wird auch berücksichtigt, dass manche Gäste sich nicht vertragen. Laden Sie eine Person ein, kann es passieren, dass der Haken bei einer anderen Person automatisch verschwindet.

```
import javax.swing.*;
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.*;

public class DinnerParty extends JFrame implements ActionListener {
    public static void main(String[] args) {
        new DinnerParty();
    }

    JTextArea anzeige;
    JCheckBox chkSchoen, chkMeier, chkFischer;
    JButton zeig;
    String[] gaeste = {"Fischer", "Meier", "Schmidt", "Müller"};
    JCheckBox[] check;

    public DinnerParty() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new BorderLayout());
        JPanel butPanel = new JPanel();
        butPanel.setLayout(new GridLayout(0,1));
        anzeige = new JTextArea();
        anzeige.setEditable(false);
        // Schiebebalken um Anzeige
        JScrollPane scrollRahmen = new JScrollPane(anzeige);
        add(scrollRahmen, BorderLayout.CENTER);
        check = new JCheckBox[gaeste.length];
        for (int i=0; i<gaeste.length; i++) {
            check[i] = new JCheckBox(gaeste[i]);
            check[i].addActionListener(this);
            butPanel.add(check[i]);
        }
        zeig = new JButton("Zeig' mal!");
        zeig.addActionListener(this);
        butPanel.add(zeig);
        add(butPanel, BorderLayout.LINE_START);
        setSize(600, 300);
        setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == check[0]
                && check[0].isSelected()) {
            check[2].setSelected(false);
        } else  if (e.getSource()==check[1]
```

```
                && check[1].isSelected()) {
            check[3].setSelected(false);
        } else  if (e.getSource()==check[2]
                && check[2].isSelected()) {
            check[0].setSelected(false);
        } else  if (e.getSource()==check[3]
                && check[3].isSelected()) {
            check[1].setSelected(false);
        } else  if (e.getSource()==zeig) {
            String str = "";
            for (int i=0; i<gaeste.length; i++) {
                if (check[i].isSelected()) {
                    str += gaeste[i] + "\n";
                }
            }
            anzeige.setText(str);
        }
    }
}
```

Listing 18.15: Die Auswahl der Gäste per Checkbox

In der Methode `actionPerformed` erzeugt der Button `zeig` eine Liste in einem Eingabefeld. Daneben werden alle Checkbuttons gefangen. Wurde beispielsweise der Button 0 angehakt, wird Button 2 enthakt. Und damit die Reihenfolge nicht relevant ist, wird bei Button 2 eben Button 0 enthakt. Sie können noch beliebige Animositäten einbauen. Allerdings reduziert das die Anzahl Ihrer Gäste nicht unerheblich.

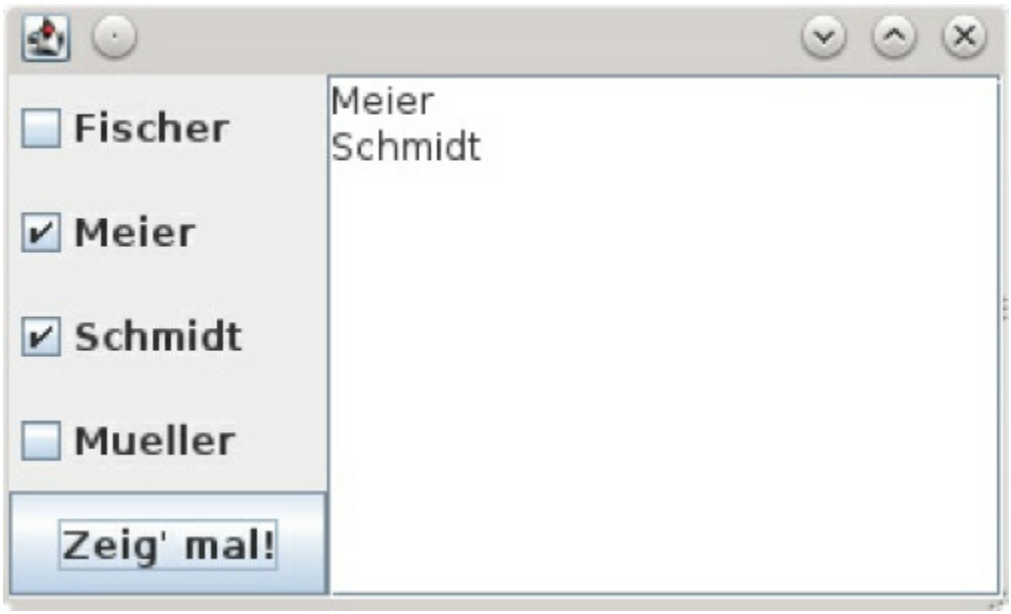


Abbildung 18.11: Eine computergesteuerte Dinner-Party

Menüs und Dateidialoge

Der große Erfolg der grafischen Oberflächen basiert darauf, dass ein Anwender selbst unbekannte Programme schnell bedienen kann, weil alle Optionen übersichtlich dargestellt werden. Dazu tragen Menüleisten natürlich erheblich bei, auch wenn sie von Designern wohl in letzter Zeit nicht so geschätzt werden.

- Die Swing-Klasse, die das Menü aufnimmt, heißt `JMenuBar`. Ein Objekt dieser Klasse wird durch die Methode `setMenuBar` in einen `JFrame` gehängt.
- In dieser Leiste werden dann mehrere Menüs eingetragen. Das erste ist typischerweise das Dateimenü und das letzte das Hilfemenü. Für diese Menüs verwendet Swing die Klasse `JMenu`.
- Jedes der Menüs enthält Menüpunkte, die durch die Swing-Klasse `JMenuItem` abgebildet werden. Ein `JMenuItem` unterscheidet sich nur geringfügig von einem normalen Button.

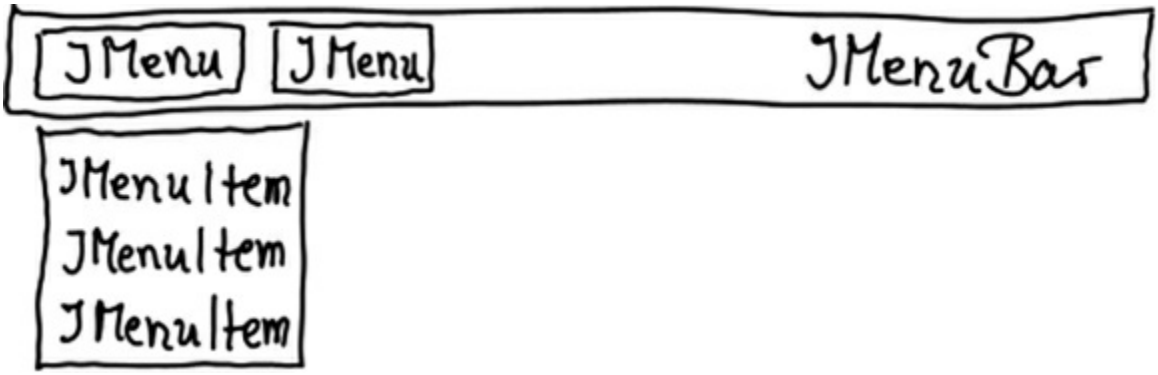


Abbildung 18.12: Aufbau einer Menüleiste in Swing

Das folgende Programm erstellt eine Menüleiste mit Datei- und Hilfemenü. Das Dateimenü verfügt über die typischen Punkte wie Datei Sichern und Ende. Das Hilfemenü enthält nur den Menüpunkt ÜBer. Hier gibt es keine großen Überraschungen.

```
import javax.swing.JFrame;
import javax.swing.*;
import java.awt.GridLayout;
import java.awt.event.*;

public class Menue extends JFrame implements ActionListener {
    public static void main(String[] args) {
        new Menue();
    }

    JMenuItem dateiOeffnen, dateiSpeichern,
              dateiSpeichernUnter, dateiQuit;
    JMenuItem hilfeUeber;
    JLabel labName;

    public Menue() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
    setLayout(new GridLayout(2, 1));
    add(new JLabel("Hier könnte Ihre Werbung stehen!"));
    labName = new JLabel("---nichts---");
    add(labName);

    // Das Menü wird erstellt
    JMenuBar leiste = new JMenuBar();

    JMenu datei = new JMenu("Datei");
    leiste.add(datei); // Dateimenü in die JMenuBar
    dateiOeffnen = new JMenuItem("Öffnen...");
    datei.add(dateiOeffnen); // Menüpunkt einbauen
    // Listener setzen
    dateiOeffnen.addActionListener(this);
    // Jetzt die restlichen Menüpunkte analog ...
    dateiSpeichern = new JMenuItem("Speichern");
    datei.add(dateiSpeichern);
    dateiSpeichern.addActionListener(this);
    dateiSpeichernUnter
            = new JMenuItem("Speichern unter...");
    datei.add(dateiSpeichernUnter);
    dateiSpeichernUnter.addActionListener(this);
    dateiQuit = new JMenuItem("Ende");
    datei.add(dateiQuit);
    dateiQuit.addActionListener(this);

    JMenu hilfe = new JMenu("Hilfe");
    leiste.add(hilfe);
    hilfeUeber = new JMenuItem("Über");
    hilfe.add(hilfeUeber);
    hilfeUeber.addActionListener(this);

    setJMenuBar(leiste); // Menü in den Rahmen hängen
    pack();
    setVisible(true);
}

@Override
public void actionPerformed(ActionEvent ev) {
    // Auf Menüpunkte abfragen
    if (ev.getSource()==dateiOeffnen) {
        // Datei öffnen! Starte die Auswahlbox!
        JFileChooser auswahl = new JFileChooser();
        if (auswahl.showOpenDialog(this)
                ==JFileChooser.APPROVE_OPTION) {
            labName.setText(auswahl.getSelectedFile()
                                    .getName());
```

```
            }
        } else if (ev.getSource()==dateiSpeichern) {
        } else if (ev.getSource()==dateiSpeichernUnter) {
            JFileChooser auswahl = new JFileChooser();
            if (auswahl.showSaveDialog(this)
               ==JFileChooser.APPROVE_OPTION) {
                labName.setText(auswahl.getSelectedFile()
                                       .getName());
            }
        } else if (ev.getSource()==dateiQuit) {
            // Der Anwender will nach Hause
            System.exit(0);
        } else if (ev.getSource()==hilfeUeber) {
            // Reklame!
            JOptionPane.showMessageDialog(null,"Ein Menü!");
        } else {
            // Das ist eigentlich nicht vorgesehen ...
            JOptionPane.showMessageDialog(null,"Was ist das?");
        }
    }
}
```

Listing 18.16: Swing-Applikation mit Menü

Dateiauswahldialog

In der Methode `actionPerformed` wird für die Menüpunkte Datei öffnen und Datei Speichern Unter jeweils ein Dateiauswahldialog gestartet, wie er bei den meisten Programmen verwendet wird. Die Klasse für die Dateiauswahldialoge heißt `JFileChooser`.

Mit den Methoden `showOpenDialog` und `showSaveDialog` kann ein Dialog zum Lesen und zum Schreiben einer Datei geöffnet werden. Als Parameter dient das Elternfenster, das hier der aktuelle `JFrame` ist.

Ist der Rückgabewert der Methode `JFileChooser.APPROVE_OPTION`, so hat der Benutzer eine Datei ausgewählt. Über die Methode `getSelectedFile` erhält das Programm eine Referenz auf `File`. Dieses Objekt kann wiederum über die Methode `getName` nach ihrem Namen befragt werden. Häufiger werden Sie allerdings den Dateinamen mit dem kompletten Pfad benötigen. Diese Information liefert Ihnen`getAbsolutePath`. Sie können `File` allerdings auch beispielsweise direkt dem Konstruktor von `FileReader` übergeben, um die Datei anschließend auszulesen. Details dazu werden wir in Kapitel 23 behandeln.

Der Fokus und der Multiliner

Der Fokus steht nicht nur für eine Zeitschrift und ein Auto, sondern bezeichnet auch die Eigenschaft einer Komponente, dass sie für Tastaturereignisse empfänglich ist. Wenn der Fokus von einer Komponente auf eine andere wechselt, erzeugt Swing ein Ereignis, das von einem `FocusListener` gefangen werden kann.

Dieses Ereigniss ermöglicht Ihnen, ein Eingabefeld gleich überprüfen zu können, noch bevor der Benutzer den abschließenden Button angeklickt hat. Sie können auch schon einmal Zwischenergebnisse berechnen, wenn in einem Feld eine Zahl steht, die in eine Berechnung einfließt. Eingabefehler können so direkt nach der Eingabe bemängelt werden und nicht erst, wenn der Anwender längst vergessen hat, was er sich dabei gedacht hat.

Um das Fokusereignis zu fangen, implementiert das Programm die Schnittstelle `FocusListener`. Das bedeutet, dass es die Methoden `focusGained` und `focusLost` überschreiben muss.

Das folgende Beispiel ist ein Programm mit zwei Eingabefeldern, deren Fokus beobachtet wird. Zwischen den beiden Eingabefeldern wird ein Eingabefeld benutzt, um die Ereignisse anzuzeigen. Bei der Gelegenheit wird noch vorgeführt, wie ein `JTextArea`-Element in ein `JScrollPane` gesteckt wird, das dann die Verwaltung der Rollbalken übernimmt, wenn der Textinhalt größer wird als der Programmausschnitt anzeigt.

```
import javax.swing.*;
import java.awt.BorderLayout;
import java.awt.event.*;

public class TestFocus extends JFrame implements FocusListener {
    public static void main(String[] args) {
        new TestFocus();
    }

    JTextArea anzeige;
    JTextField txtOben, txtUnten;

    public TestFocus() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new BorderLayout());
        anzeige = new JTextArea();
        anzeige.setEditable(false);
        JScrollPane scrollRahmen = new JScrollPane(anzeige);
        txtOben = new JTextField();
        txtOben.addFocusListener(this);
        txtUnten = new JTextField();
        txtUnten.addFocusListener(this);
        add(scrollRahmen, BorderLayout.CENTER);
        add(txtOben, BorderLayout.NORTH);
        add(txtUnten, BorderLayout.SOUTH);
        setSize(400, 300);
        setVisible(true);
    }
```

```
    @Override
    public void focusGained(FocusEvent e) {
        if (e.getSource() == txtOben) {
            anzeige.append("Focus erhalten: txtOben\n");
        } else if (e.getSource() == txtUnten) {
            anzeige.append("Focus erhalten: txtUnten\n");
        }
    }

    @Override
    public void focusLost(FocusEvent e) {
        if (e.getSource() == txtOben) {
            anzeige.append("Focus verloren: txtOben\n");
        } else if (e.getSource() == txtUnten) {
            anzeige.append("Focus verloren: txtUnten\n");
        }
    }
}
```

Listing 18.17: Fokuswechsel fangen

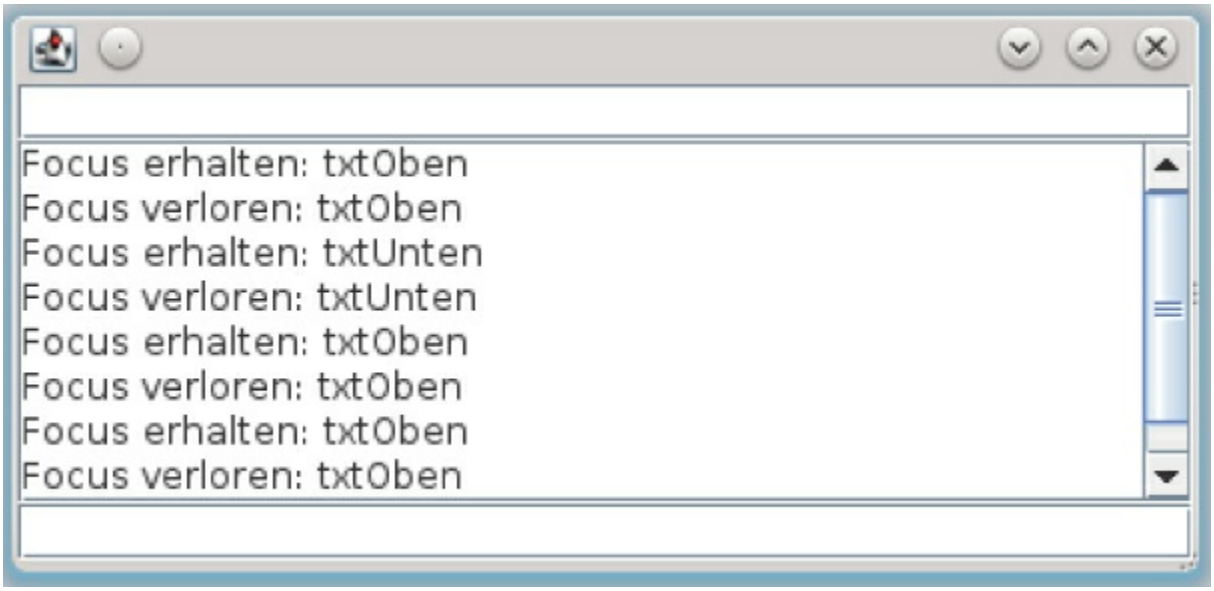


Abbildung 18.13: Fokuswechsel

Aufgaben

Damit Sie den Umgang mit den Kontrollelementen üben können, tische ich Ihnen ein paar Programmideen auf.

Der Spannungsabfall der LEDs

LEDs verkraften nur eine gewisse Stromstärke, gemessen in mA. Damit LEDs

nicht durchbrennen, benötigen sie einen Vorwiderstand. Für die Berechnung dieses Widerstands ist es wichtig zu wissen, wie viel Spannung in der LED selbst abfällt. Dieser hängt von der Farbe der LED ab. Schreiben Sie ein Programm `LEDFarbe`, das dem Elektroniker hilft, indem es für jede LED-Farbe einen Radiobutton zur Verfügung stellt. Hier wählt der Anwender seine Farbe. Klickt er danach einen Button mit der Beschriftung Anzeigen an, dann erhält er den entsprechenden Spannungsabfall. Den Spannungsabfall müssen Sie übrigens in den Sondermüll geben, sofern es sich nicht um Bio-Strom aus artgerechter Haltung handelt.

Farbe	Spannungsabfall
Weiß	4.0
Rot	1.6
Gelb	2.2
Grün	2.1
Blau	2.9

Tabelle 18.1: Spannungsabfall an LEDs

Sollte Ihnen das Thema gefallen, könnten Sie auch ein Programm schreiben, das aus der Batteriespannung und der LED-Farbe errechnet, welcher Vorwiderstand notwendig ist. Der muss nun so groß sein, dass die Stromstärke nicht über 18 bis 20 mA steigt. Sonst brauchen Sie eine neue LED.

Die Widerstandsformel aus der Physik lautet R = U/I. Daraus ergibt sich die Formal für den Widerstand:

```
r = (uBatterie - uLED) / (iLED);
```

Der Wert iLED ist in der Regel 20 mA, also 0,02 A. Einige LEDs haben eine andere Stromstärke, also möchten Sie vielleicht ein Eingabefeld dafür spendieren. Eine Musterlösung gibt es unter dem Namen `LEDVorwiderstand`. Aber Ihre eigene Lösung ist bestimmt besser!

Noch ein Taschenrechner

Ich hatte Ihnen ja in Kapitel 17 bereits vorgeschlagen, ein Layout für einen Taschenrechner zu erstellen. Nun könnten Sie ihn mit Leben füllen. Fangen Sie die Ereignisse der Buttons.



Die Auswertung der Buttons wird einfacher, wenn Sie in der Methode `actionPerformed` die Beschriftung mit der Methode `e.getActionCommand` auslesen.

In der einfachsten Version sollte bei jedem Buttondruck die Beschriftung in der Anzeige erscheinen.

Wenn Sie Lust auf Knobeln haben, können Sie die Zahlen so erscheinen lassen wie auf einem normalen Taschenrechner. Beim Klicken auf ein Rechensymbol wird die Anzeige wieder gelöscht.

Am besten wäre es natürlich, wenn Sie dem Taschenrechner das Rechnen beibringen könnten. Eine Musterlösung finden Sie unter dem originellen Namen `Taschenrechner`. Nehmen Sie sich Zeit! Es ist nicht ganz trivial.

Und falls Sie Informatikstudent sind, bauen Sie bitte noch Klammern in den Taschenrechner ein.

Ein JTabbedPane mit Buttons und Eingabefeld

Zum Abschluss machen wir ein kleines Gedächtnistraining: Erinnern Sie sich noch an das `JTabbedPane`? Ja, genau: Diese Fenster mit den Laschen hatten Sie im Zusammenhang mit Layouts kennengelernt.

Sie kennen diese Reiter oder Laschen sicher von vielen Einstelldialogen. Und ich bin sicher, Sie wollten auch schon immer einmal einen solchen Einstelldialog mit Reitern selbst schreiben. Sie müssen ja keinen kompletten Einstelldialog erstellen. Es reicht, wenn Sie unter einer Lasche drei Radiobuttons anordnen. Sie könnten dem Anwender damit ermöglichen, zwischen Links, Mitte und Rechts zu wählen, damit er bei der nächsten Wahl schon ein bisschen Übung hat. Im Programm darf man kostenlos und immer wieder wählen, bei der nächsten Wahl nur einmal und man bezahlt die Zeche die nächsten vier Jahre.

Der andere Tab führt zu einem Texteingabefeld. Da kann dann der Anwender einen Kommentar hineinschreiben, den das Programm aus Datenschutzgründen garantiert nicht auswertet. Meine Lösung finden Sie unter `TabbedCards` in den Musterlösungen.

Sollten Sie die Aufgabe zu leicht finden, bauen Sie doch ein zweites Eingabefeld ein und verfolgen Sie den Fokus. Diesmal ohne Netz oder doppelte Musterlösung.

Kapitel 19

Bermuda im Swing

IN DIESEM KAPITEL

Das Bermudaspiel erhält seine zentrale Suchfunktion.

Mit einer Mausbedienung ist es leicht zu bedienen.

Mit Images wird es dann noch schick gemacht.

Das Spiel Bermuda habe ich Ihnen in Kapitel 8 vorgestellt. Im Kapitel 12 wurde Bermunda erwachsen, also objektorientiert. Davon hatte der Anwender wenig, denn immer noch lief das Programm auf einer Textkonsole, und der Anwender musste die Koordinaten für die Suche nach den Schiffen eintippen. Ich fürchte, mit dieser Benutzerführung werden nur Ihre engsten Angehörigen lobende Worte für Ihr erstes selbst geschriebenes Spiel finden. In diesem Kapitel soll Bermuda im Fenster laufen und mit der Maus sprechen. Swing wird uns die Möglichkeiten dazu eröffnen.



Zur Wiederholung noch einmal kurz die Spielregeln: In einem Spielfeld von neun auf sieben Positionen versteckt das Spiel vier Schiffe, die der Spieler finden soll. Bei jedem Versuch erhält der Spieler eine Information, in wie viele Richtungen Schiffe gepeilt wurden. Acht Richtungen sind möglich, neben den vier Himmelsrichtungen auch die Diagonalen. Aus diesen Zahlen kann der Spieler irgendwann schließen, wo die Schiffe verborgen sind. Sind alle Schiffe gefunden, ist das Spiel beendet.

Klassenbildung

Die Klassen für das Spielfeld, die Position, die Flotte und die Schiffe können Sie unverändert aus Kapitel 12 »Bermuda-Klassen« übernehmen. Das Hauptprogramm und die Klasse `Darstellung` werden durch ein `JFrame` und ein `JPanel` ersetzt.

Die anderen Klassen können einfach aus Kapitel 12 übernommen werden.

- Die Klasse `Schiff` finden Sie in Listing 12.4.
- Die Klasse `Position` finden Sie in Listing 12.3.
- Die Klasse `Flotte` finden Sie in Listing 12.7.

- Die Klasse `Spielfeld` finden Sie in Listing 12.8.

Grafik aus Images

Ob Ihr Bermuda gut aussieht, wird mehr von Ihrem Zeichentalent als von Ihren Java-Kenntnissen abhängen. Vielleicht motivieren Sie einen Bekannten oder Verwandten, der so etwas gern macht. Erzeugen Sie mit Paint oder GIMP ein paar PNG-Dateien. Ich habe Grafiken im Format 32 x 32 Pixel selbst erstellt. Und wie Sie in Abbildung 19.1 sehen können, gibt mein Talent nicht besonders viel her.

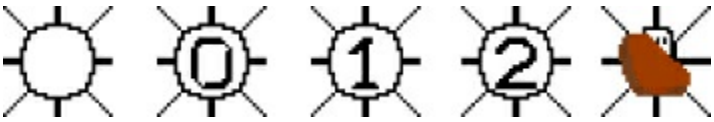


Abbildung 19.1: Die Bermuda-Images

Jede dieser Grafiken endet in allen Richtungen mit einer Senkrechten, einer Waagerechten und zwei Diagonalen. So entsteht beim Kacheln mit diesen Images automatisch ein Raster, das Sie nicht extra auf dem Bildschirm zeichnen müssen. In der Mitte befindet sich ein Kreis. Dieser bleibt für leere Felder leer und bekommt den Dateinamen *leer.png*. Für die Richtungsbeschriftung gibt es Images mit den Ziffern von 0 bis 8, die unter den Dateinamen *mark0.png* bis *mark8.png* gespeichert werden. Zu guter Letzt wird auch ein Schiff benötigt, das in der Datei *schiff.png* abgelegt wird.

Sie finden meine Bilder bei den Quelltexten für dieses Buch zum Download unter `http://www.willemer.de/java4d`. Die Qualität sollte Sie motivieren, dass Sie die Bilder besser hinbekommen.

Speicherung der Bilder im Projekt

Die Bilder gehören zum Programm und sollen später in der JAR-Datei enthalten sein. Dazu müssen sie in das Projekt kopiert werden. Dazu legen Sie einen Ordner im Projekt an, indem Sie mit der rechten Maustaste in das Package Ihrer Java-Quelltexte klicken und im Kontextmenü New|Folder auswählen. Als Namen geben Sie beispielsweise *image* an.

Ihre Bilddateien können Sie in Ihrem Dateimanager auswählen und direkt in den angelegten Ordner im Projekt ziehen.

Im Programm benötigen Sie ein Objekt der Klasse `Image`. Dieses kann durch die Methode `drawImage` dargestellt werden. Sie können mit einer etwas verschachtelten Anweisung aus Ihrer PNG-Datei ein `Image`-Objekt erzeugen. Das folgende Beispiel erstellt ein Image für ein Schiff.

```
schiff = new ImageIcon(
        getClass().getResource("image/schiff.png")
        ).getImage();
```

Das `Image` können Sie dann in der Zeichen-Methode `paint` mit der Methode `drawImage` zeichnen, wie in den folgenden Zeilen zu sehen:

```
@Override public void paint(Graphics g) {
    // ...
    g.drawImage(schiff, x, y, this);
```

Ein Fensterrahmen für Bermuda

Nun wollen wir dem Spiel eine hübsche Oberfläche verpassen. Dazu schreiben wir eine Klasse, die `JFrame` erweitert. Dieses organisiert in einem `BorderLayout` die Hauptelemente:

- Ein Label (`lblMeldung`): Hier werden Meldungen an den Spieler abgesetzt. Zurzeit ist dies ausschließlich die Meldung über den Sieg. Ansonsten zeigt es, dass der Spieler bitte vier Schiffe suchen soll.
- Ein Button (`btnNeuesSpiel`): Damit kann ein neues Spiel gestartet werden. Ob es aus Verzweiflung geschieht oder weil das Spiel gewonnen wurde, ist dem Button egal. Die Verwaltung des Buttons wird an das Panel weitergereicht.
- Das Panel (`panel`): Hier wird das Spielfeld angezeigt und die Aktionen des Benutzers aufgefangen.

```
import java.awt.BorderLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class BermudaFrame extends JFrame {
    public static void main(String[] args) {
        new BermudaFrame();
    }

    private BermudaPanel panel;
    public JLabel lblMeldung = new JLabel("Suche nach vier Schiffen");
    JButton btnNeuesSpiel = new JButton("Neues Spiel");

    public BermudaFrame() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLayout(new BorderLayout());
        add(lblMeldung, BorderLayout.NORTH);
        panel = new BermudaPanel(this);
        add(panel, BorderLayout.CENTER);
        add(btnNeuesSpiel, BorderLayout.SOUTH);
        btnNeuesSpiel.addActionListener(panel);
        setTitle("Bermuda");
        setVisible(true);
        pack();
    }
}
```

Listing 19.1: `BermudaSwing` beginnt wie jede Swing-Anwendung.

Die eigentliche Grafik- und Maussteuerung verlagern wir in das `BermudaPanel`. Selbst den `ActionListener` für den Button haben wir dem Panel aufgedrückt.

Das Bermuda-Panel

Das Panel stellt das Spielfeld dar, behandelt die Ereignisse und übernimmt auf diesem Weg auch die Spielleitung.

- ✔ Für die Darstellung des Spielfelds wird die Methode `paint` überschrieben. Das Spielfeld wird einfach durch das Nebeneinandersetzen der Images erzeugt. Welches Image gesetzt wird, ist davon abhängig, welches Symbol im Spielfeld gespeichert ist.
- ✔ Um die Mausklicks zu fangen und die Suchfunktion des Spielfelds auszulösen, implementiert das `BermudaPanel` den `MouseListener` und überschreibt die Methode `mouseClicked`.
- ✔ Für die Reaktionen auf den Button, der ein neues Spiel anfordert, implementiert

`BermudaPanel` das Interface `ActionListener` und überschreibt dazu die Methode `actionPerformed`.

Reaktionen auf die Maus

Die Methode `mouseClicked` empfängt im `MouseEvent` die Mausposition während des Klicks. Diese Mauskoordinate muss in eine Feldkoordinate umgerechnet werden. Da die Images dicht gekachelt wurden, muss dazu einfach nur durch die Distanz zwischen den Kacheln geteilt werden. Das Ergebnis kann dann der Methode `suche` als Parameter übergeben werden.

Am Ende sorgt der Aufruf von `repaint` dafür, dass die Änderungen auf dem Spielfeld sofort auf dem Fenster erscheinen.

Das Interface `MouseListener` erwartet, dass das Panel ein paar Methoden mehr implementiert, die es gar nicht benötigt. Die leere Methoden finden Sie am Schluss des Listings.

Reaktionen auf den Button

Wird der Button für ein neues Spiel angeklickt, wird einfach ein neuer Spielstand erzeugt. Der alte Stand wird irgendwann von der automatischen Garbage Collection weggeräumt.

```
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

import javax.swing.ImageIcon;
import javax.swing.JPanel;

public class BermudaPanel extends JPanel implements MouseListener,
 ActionListener {

    Spielfeld spiel = new Spielfeld();
    Image leer, schiff, ziffer[];
    boolean spielEnde = false;
    String statusStr = "                ";
    int distanz = 32;
    BermudaFrame frame;

    public BermudaPanel(BermudaFrame frame) {
        this.frame = frame;
        addMouseListener(this); // Fange Mausereignisse
        leer = new ImageIcon(getClass().getResource(
                "image/leer.png")).getImage();
        schiff = new ImageIcon(getClass().getResource(
                "image/schiff.png")).getImage();
        ziffer = new Image[9];
        for (int i = 0; i < 9; i++) {
            ziffer[i] = new ImageIcon(getClass().
                    getResource("image/mark"+i+".png")).
                    getImage();
        }
        distanz = new ImageIcon(getClass().getResource(
                "image/schiff.png")).getIconWidth();
    }

    @Override
    public void paint(Graphics g) {
        for (int i = 0; i < Spielfeld.MAXX; i++) {
            for (int j = 0; j < Spielfeld.MAXY; j++) {
                Position pos = new Position(i, j);
                char c = spiel.get(pos);
                Image img = null;
                if (c >= '0' && c <= '8') {
                    img = ziffer[c - '0'];
```

```
                } else if (c == Flotte.SCHIFF) {
                    img = schiff;
                } else {
                    img = leer;
                }
                g.drawImage(img, i * distanz,
                                 j * distanz, this);
            }
        }
    }

    // Das Panel meldet seine Platzbedürfnisse an
    @Override
    public Dimension getPreferredSize() {
        return new Dimension((Spielfeld.MAXX) * distanz,
                (Spielfeld.MAXY) * distanz);
    }

    // ---------- Methoden des MouseListeners -----------
    @Override
    public void mouseClicked(MouseEvent me) {
        Position pos = new Position(me.getX()/distanz,
                                    me.getY()/distanz);
        if (spiel.istImSpielfeld(pos)) {
            char ergebnis = spiel.suche(pos);
            spiel.set(pos, ergebnis);
            if (spiel.alleSchiffeGefunden()) {
                frame.lblMeldung.setText("Spiel gewonnen!");
            }
        }
        repaint();
    }

    public void mouseReleased(MouseEvent me) {}
    public void mousePressed(MouseEvent me) {}
    public void mouseEntered(MouseEvent me) {}
    public void mouseExited(MouseEvent me) {}

    // --- Methode des ActionListeners (Button) ---
    @Override
    public void actionPerformed(ActionEvent e) {
        spiel = new Spielfeld();
        frame.lblMeldung.setText("Suche nach Schiffen");
        repaint();
    }
}
```

Listing 19.2: Das `BermudaPanel`, das die ganze Arbeit macht

Nach dem Start sollte das Programm nach einigen Spielzügen ähnlich wie in Abbildung 19.2 aussehen. Das hat doch was. Ja, ich weiß, die Zeichnungen könnten besser sein. Man muss schon wissen, dass das Schiff ein Schiff sein soll.

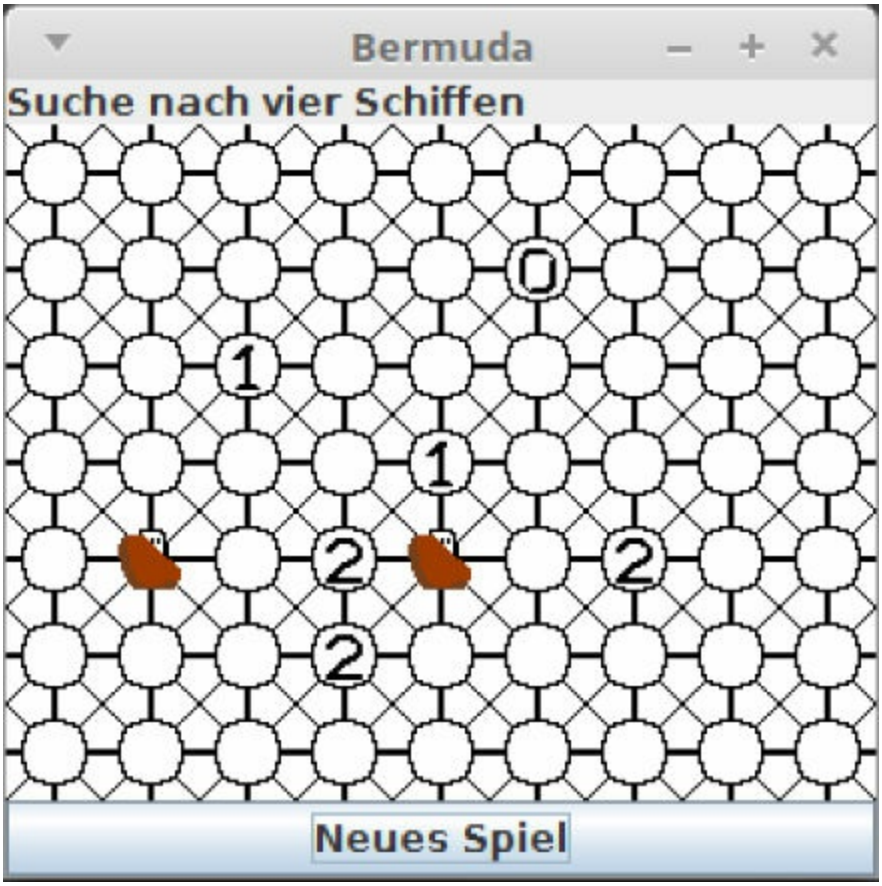


Abbildung 19.2: Mit Swing sieht Bermuda schon besser aus.

Erstellen einer Bermuda-JAR-Datei

Wenn Sie das fertige Bermuda-Spiel woanders spielen wollen als auf Ihrem Entwicklungsrechner, müssen Sie neben den Class-Dateien nun auch noch die Bilddateien mitschleppen und in der richtigen Position ablegen. Nein, das sieht nicht schick aus. Aber zum Glück gibt es da ja noch die Möglichkeit, eine JAR-Datei zu erstellen, und dort können Sie auch die Bilder unterbringen.

JAR-Datei generieren

Erinnern Sie sich noch, wie ich Ihnen erzählt habe, wie man eine JAR-Datei erzeugt? – Kein Problem! Ich erzähle es Ihnen gerne noch einmal: Sie klicken das Projekt mit der rechten Maustaste an, und klicken auf Export.... Es erscheint eine Dialogbox, in der Sie unter dem Stichwort Java den Punkt Runnable JAR File finden. Das ist Ihr Traumpartner. Sie kommen über den Button Next weiter.

Im nächsten Dialog spezifizieren Sie, wie Ihre JAR-Datei aussehen soll. Unter Launch Configuration können Sie nochmals das Projekt auswählen. Unter Export Destination können Sie bestimmen, wo die JAR-Datei hinkommen und wie sie

heißen soll. Über den Button Browse geht das am einfachsten. Mit Finish schließen Sie alles ab.

Kapitel 20

Java FX, das neue Swing?

IN DIESEM KAPITEL

- Kontrollelemente auf der Bühne ihres Fensters
- Kontrollelemente von Event zu Event
- Schickmachen mit CSS
- FXML für SceneBuilder und andere Automaten

Manchmal werden Versionsnummern übersprungen. Windows sprang beispielsweise von Version 8.1 gleich zu Version 10, weil Microsoft offenbar mit Windows 8 nicht mehr in Verbindung gebracht werden wollte. Java zeigt, dass es noch krasser geht, und sprang von Java FX 2 gleich zu Java FX 8. Allerdings kann man zugeben, dass es damit zusammenhing, dass Java FX eben mit der Java-Version 8 wiederbelebt worden war. Aber dass man sich an Java FX 2 nicht mehr gern erinnerte, mag dennoch ein Grund gewesen sein.



Java FX 8 wurde mit Java 8 eingeführt. Darum wird Java 8 vorausgesetzt. Darüber hinaus müssen Sie für Eclipse über den Menüpunkt Info|Eclipse Marketplace das Plugin `e(fx)clipse` installieren.

AWT, Swing und Java FX

Java FX ist nicht die erste grafische Oberfläche für Java. Zuerst wurde AWT verwendet, das heutzutage nur noch begrenzt Freude auslöst. Sehr weit verbreitet ist immer noch Swing. Es lieh sich beispielsweise das Layout und das Eventhandling vom alten AWT, sodass dies immer noch sehr präsent ist. Klassen von Swing erkennt man leicht daran, dass meist ein `J` vor den Klassennamen gesetzt wurde, also `JButton` für einen Button.

Java FX basiert weder auf AWT noch auf Swing. Es verwendet aber oftmals Namen, die auch bei AWT noch üblich waren. So nennen beide einen Button einfach Button. Das kann zu Missstimmungen führen, wenn der Programmierer die falsche Bibliothek importiert.



Insbesondere bei Eclipse ist es ein beliebtes Vorgehen, einfach den Klassennamen in ein Listing zu tippen und darauf zu warten, dass die IDE den Fehler rot unterkringelt. Wenn Sie nun den Mauszeiger über den Fehler

schieben, schlägt Eclipse verschiedene Lösungen vor, unter anderem, welche Bibliothek importiert werden könnte, um das Problem zu beheben. Hier sollten Sie aufpassen, dass auch wirklich eine Java-FX-Bibliothek, die mit `javafx` beginnt, und nicht etwa eine AWT-Bibliothek, die mit `java.awt` beginnt, ausgewählt wird.

Mit Eclipse zur FX-Anwendung

Um unter Eclipse mit Java FX spielen zu dürfen, müssen Sie zunächst eine Erweiterung installieren. Jedenfalls ist das zur Zeit der Drucklegung noch so.

- ✔ Wählen Sie im Menü Help|Eclipse Marketplace.
- ✔ Im Eingabefeld des Dialogs tippen Sie »fx« ein.
- ✔ Es erscheint `e(fx)clipse` in der Liste. Klicken Sie auf Install.
- ✔ Sie müssen die Lizenzbedingungen akzeptieren und können dann mit Finish abschließen.

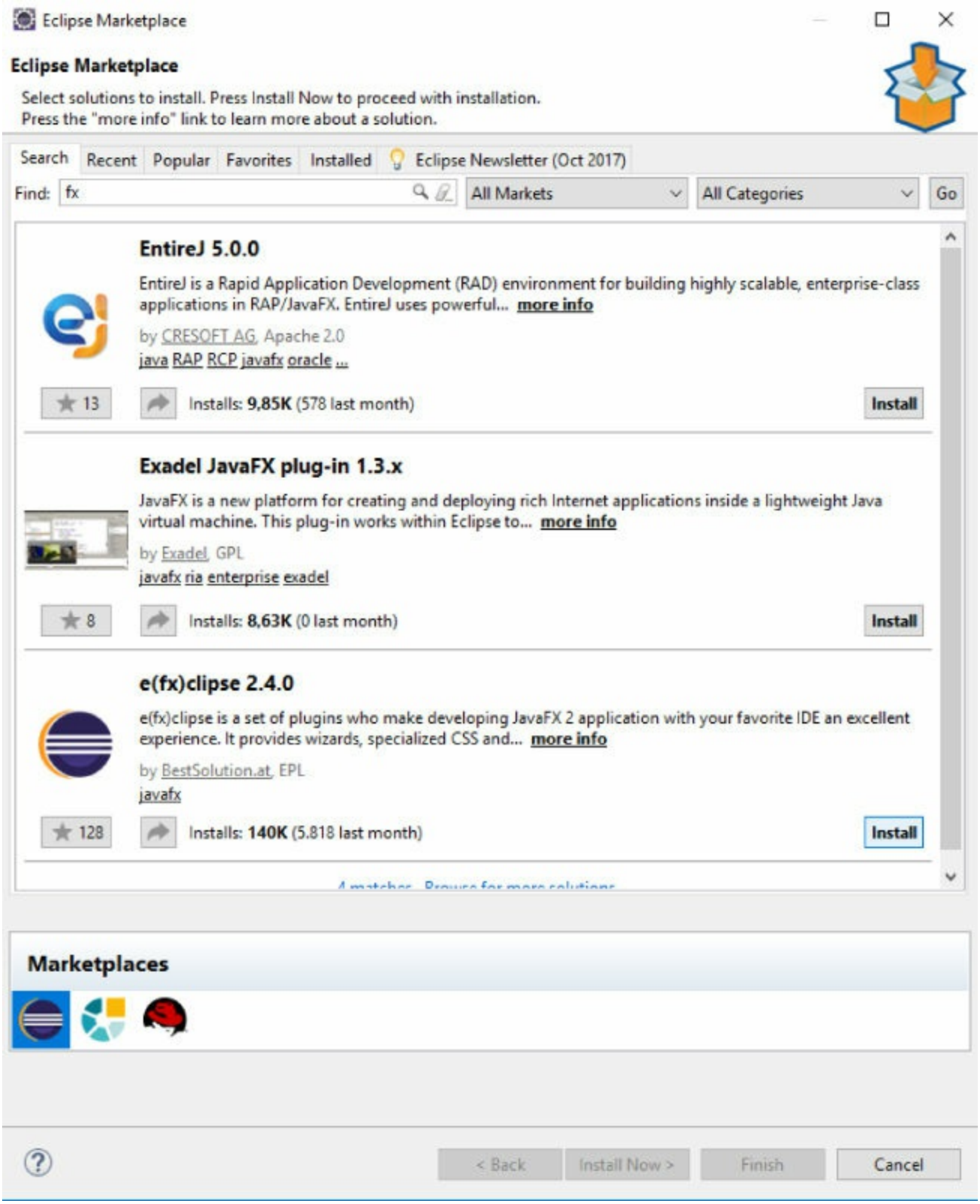


Abbildung 20.1: Installation von `e(fx)clipse`

Wenn Sie unter Eclipse `e(fx)clipse` installiert haben, steht Ihnen dort auch ein speziell vorbereitetes Projekt für Java-FX-Anwendungen zur Verfügung.

- ✔ Klicken Sie dazu im Menü File|New|Project an.
- ✔ Es erscheint der inzwischen wohlvertraute Dialog zur Bestimmung des Projekt-Typs. In der darin befindlichen Liste wählen Sie unter dem Eintrag JavaFX den Punkt JavaFX-Project aus.

- ✔ Nun müssen Sie nur noch den Namen angeben und den Dialog bestätigen.

Daraufhin erstellt Eclipse einen Source-Rahmen, in den Sie Ihre Anwendung bauen können.

```
package application;

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;

public class Main extends Application {
    @Override
    public void start(Stage stage) {
        try {
            BorderPane borderpane = new BorderPane();
            Scene scene = new Scene(borderpane,400,400);
            scene.getStylesheets().add(getClass()
                    .getResource("application.css")
                    .toExternalForm());
            stage.setScene(scene);
            stage.show();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 20.1: Der Rahmen einer FX-Anwendung von Eclipse

Ich habe mir erlaubt, eine Zeile zu umbrechen, damit das Buch nicht im Querformat erscheinen muss.

Bühnendekoration

Das von Eclipse erzeugte Listing 20.1 zeigt, wie unter Java FX ein Fenster und seine Elemente erzeugt werden.

Von der Application zur Stage

Eine Java-Anwendung wird zur Java FX-Anwendung, indem sie `Application` erweitert und die Methode `start` überschreibt.

Die Methode `start` wird automatisch aufgerufen, sobald die Anwendung in ihrer Hauptmethode `main` durch den Aufruf von `launch` den Startschuss gibt.

Die Methode `start` erhält als Parameter ein Objekt der Klasse `Stage`. Wenn darüber die Methode `show` aufgerufen wird, erscheint ein leeres Fenster.

```
public class FirstFX extends Application {
    public static void main(String[] args) {
        launch(args); // sorgt indirekt für den Aufruf von start
    }

    @Override
    public void start(Stage hauptStage) {
        // ...
        hauptStage.show();
    }
}
```

Listing 20.2: Rahmen einer Java FX-Anwendung

Der Parameter der Methode `start` ist eine Referenz auf ein Objekt der Klasse `Stage` (zu Deutsch Bühne). Diese Bühne entspricht dem Rahmen, den das Fenster einnimmt. Hier spielt die Musik und Ihre Anwendung.

Diese Musik können Sie als Szene auf der Bühne betrachten. Darum wird in der Methode `start` zuerst eine `Scene` angelegt. Diese wird über die Methode `setScene` mit der `Stage` verknüpft. Damit die Bühne nicht leer ist, habe ich hier einen Button erzeugt und beschriftet.

Zuvor wurde aber bereits ein `BorderPane` erzeugt, auf dem die `Scene` aufbaut. Das `BorderPane` ist eine Spezialisierung von `Pane`. Dabei handelt es sich um ein Brett, auf dem Elemente angeordnet werden können. Davon gibt es verschiedene Spezialisierungen. `BorderPane` beispielsweise zeigt ein großes Element in der Mitte und erlaubt die Positionierung anderer Elemente und Bretter links und rechts, aber auch oben und unten, also an den Rändern. Ich komme noch darauf zurück.

Zusammenfassung

Um ein Java-FX-Programm zu schreiben, müssen Sie zunächst ein FX-Projekt anlegen. In dem Programm müssen folgende Schritte erfolgen:

- ✔ Das Programm erweitert `Application`.
- ✔ Es überschreibt die Methode `start`.
- ✔ In der Hauptmethode `main` wird `launch` aufgerufen.

- In der Methode `start` wird über den Parameter der Klasse `Stage` die Methode `show` aufgerufen.

Einbindung von CSS

In Listing 20.1 können Sie sehen, dass es möglich ist, CSS-Dateien in FX einzubetten. CSS kennen Sie vielleicht aus dem Webdesign. Hier wie dort dient CSS dazu, den Stil für das Design anzupassen.

```
scene.getStylesheets().add(getClass()
        .getResource("application.css")
        .toExternalForm());
```

Listing 20.3: Einbinden einer CSS-Datei

Hier wird der Inhalt der Datei *application.css* in der Stylesheet-Liste der `Scene` eingebunden. Als Beispiel habe ich ein kleines CSS definiert, das `awdesign` definiert, die einen hellblauen Hintergrund festlegt.

```
.awdesign {
    -fx-background-color:lightblue;
}
```

Listing 20.4: Ein kleines Design

Natürlich könnnen Sie noch weitere Stile definieren. Um den Stil `awdesign` auf Elemente anzuwenden, müssen Sie die Methode `getStyleClass` verwenden. Erhält ein Element diese Stilklasse, werden sich alle Kindelemente danach richten und bekommen einen hellblauen Hintergrund.

```
scene.getStyleClass().add("awdesign");
```

Mit NetBeans zur Java-FX-Anwendung

NetBeans ist auf Java FX etwas besser vorbereitet. Hier müssen Sie nichts installieren, sondern können sofort ein FX-Projekt erstellen.

- Rufen Sie aus dem Menü den Punkt Datei|Neues Projekt auf. Es erscheint der Dialog für das Projekt.
- Aus der Liste Kategorien wählen Sie JavaFX und aus der Liste Projekte wählen Sie JavaFX Application (siehe Abbildung 20.2). Mit Weiter kommen Sie weiter.
- Geben Sie im Folgedialog den Namen der Hauptklasse ein, beispielsweise `Main`, und klicken Sie dann auf Fertigstellen.

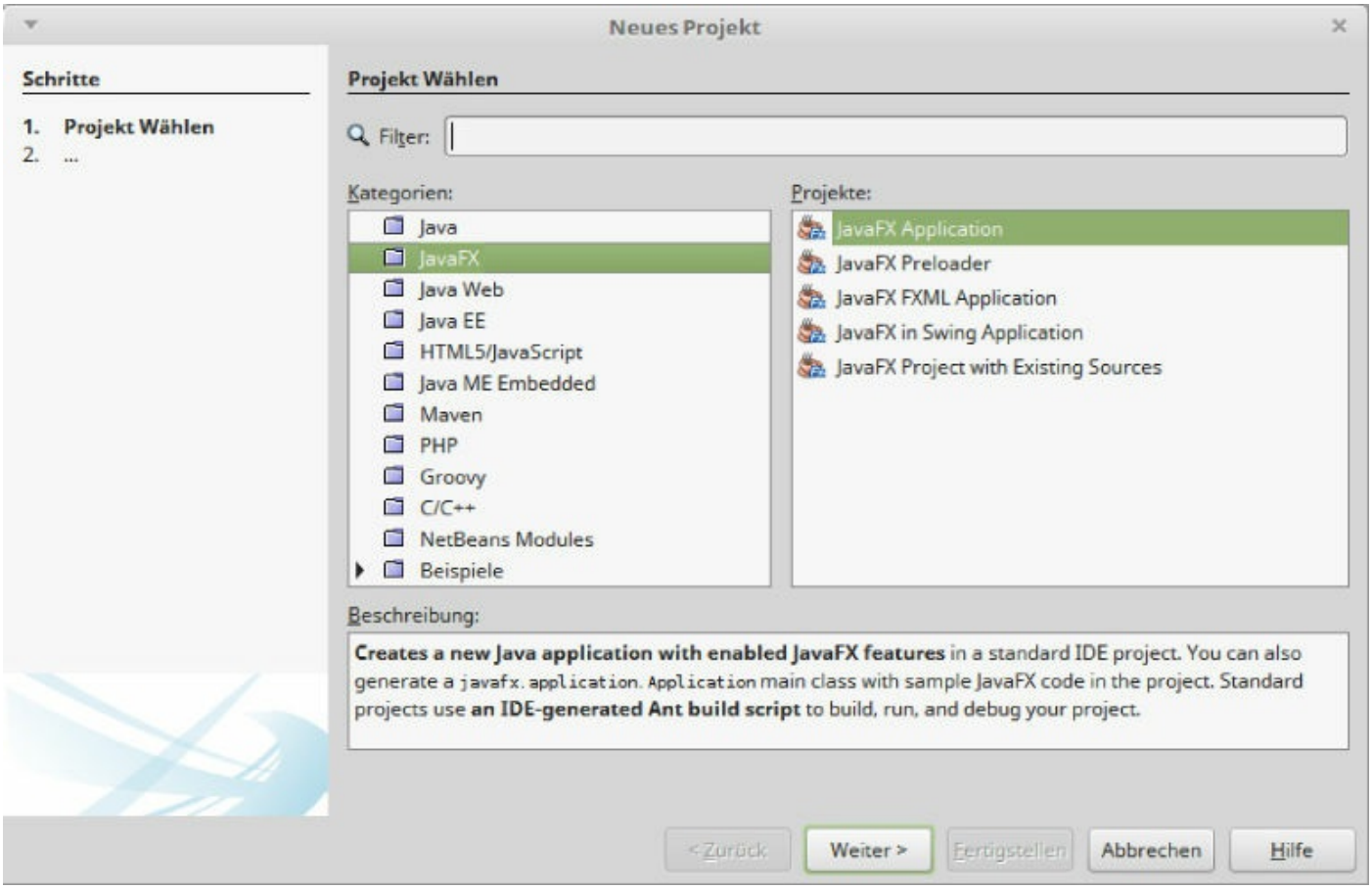


Abbildung 20.2: NetBeans besitzt bereits Projekte für FX.

Anschließend erstellt NetBeans ein Rahmenprogramm für Ihre FX-Applikation. Im Gegensatz zu Eclipse verwendet NetBeans nicht das `BorderPane`, sondern das langweilige `StackPane`, entschädigt aber dafür mit einem `Button`, der sogar auf Knopfdruck reagiert, indem er einen `EventHandler` aufsetzt.

```
package main;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class Main extends Application {

    @Override
    public void start(Stage primaryStage) {
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(btn);

        Scene scene = new Scene(root, 300, 250);

        primaryStage.setTitle("Hello World!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 20.5: Das Rahmenprogramm von NetBeans fängt Ereignisse.

Auch hier werden die Schritte durchgeführt, die zum Aufbau der FX-Anwendung benötigt werden, die bereits im vorigen Abschnitt beschrieben wurden:

- ✔ Das Programm erweitert `Application`.
- ✔ Es überschreibt die Methode `start`.
- ✔ In der Hauptmethode `main` wird `launch` aufgerufen.
- ✔ In der Methode `start` wird über den Parameter der Klasse `Stage` die Methode `show` aufgerufen.

Ein Button als Kontrollelement

NetBeans zaubert in sein FX-Rahmenprogramm in Listing 20.5 nicht nur die Erzeugung eines Buttons, sondern fängt die Aktion des Buttons und reagiert darauf.

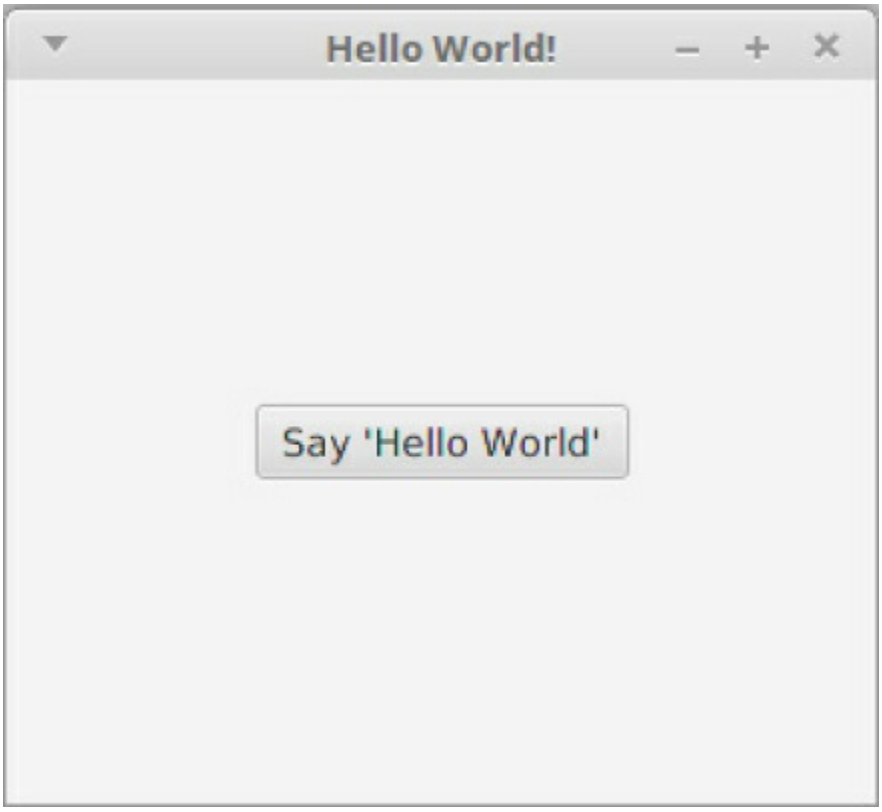


Abbildung 20.3: Der Button nach Start des NetBeans-Rahmenprogramms

Der Button wird per `new` erzeugt und kann mit der Methode `setText` beschriftet werden. Er wird sichtbar, indem er den Kindern des Panes hinzugefügt wird, das an der `Scene` angemeldet wird.

```
Button btn = new Button();
btn.setText("Say 'Hello World'");
// ...
StackPane root = new StackPane();
root.getChildren().add(btn);
Scene scene = new Scene(root, 300, 250);
```

Das im NetBeans-Rahmen verwendete `StackPane` legt seine Kinder übereinander, sodass nur das letzte Element zu sehen ist. Da hier der `Button` das einzige Element ist, ist davon wenig zu spüren.

Aktivitäten eines einsamen Buttons

Ein Button erhält vor allem dadurch seinen Charme, dass er eine Aktion auslöst. Das Programm sollte dieses Ereignis fangen und darauf reagieren. Genau dies tut das Beispielprogramm von NetBeans in Listing 20.5. Zugegebenermaßen sind die entsprechenden Zeilen sehr kompakt und so etwas schwer zu lesen:

```
btn.setOnAction(new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");
    }
});
```

Wenn Sie die Zeilen auffächern, kommen dabei folgende Schritte heraus:

- Die Methode `setOnAction` meldet einen `EventHandler` für den Button `btn` an.
- Dieser `EventHandler` wird direkt innerhalb des Parameters erzeugt.
- Dabei wird eine Erweiterung erzeugt, deren Methode `handle` überschrieben wird. Diese reagiert später auf den Druck des Buttons.

Diese sehr kurze Fassung finden Sie häufig in Listings, wenn es nur darum geht, ein Einzelereignis gleich an Ort und Stelle zu behandeln. Häufiger wird stattdessen die Klasse, in der der Button angelegt wird, zum `EventHandler` gemacht, indem er das Interface `EventHandler` implementiert. Da es aber ganz viele unterschiedliche Events gibt, wird dem `EventHandler` als Typparameter der Eventtyp mitgegeben. Für Buttons ist das ein `ActionEvent`.

Implementiert eine Klasse den `EventHandler` für das `ActionEvent`, muss sie auch die Methode `handle` implementieren. Freundlicherweise hilft Eclipse, wenn man auf die Fehlermeldung klickt. Damit man etwas sieht, habe ich kurzerhand einen kurzen Text auf der Konsole ausgegeben.

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;

public class FirstFX1 extends Application
            implements EventHandler<ActionEvent> {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage hauptStage) {
        Button button = new Button();
        button.setText("Drück mich!");
        button.setOnAction(this);
        Scene scene = new Scene(button, 400, 300);
        hauptStage.setScene(scene);
        hauptStage.show();
    }

    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hab' dich auch lieb!");
    }
}
```

Listing 20.6: Die Klasse wird zum `EventHandler`

Mit dem Pane um Fassung ringen

Eine `Scene` kann nur ein Element aufnehmen. Sofern Sie nicht für sehr spartanische Oberflächen schwärmen, wird Ihnen das vermutlich nicht reichen. Ein `Pane` dagegen nimmt nicht nur mehrere Kontrollelemente auf, es stellt je nach Typ auch eine Strategie zu deren Anordnung zur Verfügung.

Darum werden Sie also typischerweise zunächst ein `Pane` anlegen, das die Kontrollelemente und gegebenenfalls auch weitere Panes aufnimmt, um dieses Wurzel-Pane der `Scene` als Konstruktorparameter zu übergeben.

Das Pane und seine Kinder

Ein Pane dient als Brett, auf dem der Programmierer seine Kontrollelemente festnagelt. Java FX bietet unterschiedliche Bretter an, die sich darin unterscheiden, wie sie die Anordnung der Elemente organisieren. Eigentlich ist »festnageln« nicht passend. Tatsächlich passen sich die Elemente in Position und Größe ihrem Eltern-Pane an. Das Ausrichteverhalten des Panes und seiner Kinder wird besonders dann deutlich, wenn der Anwender ein Fenster vergrößert.

HBox und VBox: Neben- oder untereinander

Wenn Sie mehrere Elemente nebeneinandersetzen sollen, hilft `HBox` weiter. Finden die Elemente keinen Raum, weil das Außenfenster zu klein ist, werden sie zunächst auf ein Minimum zusammengestaucht und, wenn das nicht reicht, abgeschnitten. Das ist grausam, sorgt aber dafür, dass in einer `HBox` verlässlich immer nur eine Zeile belegt ist.


Wenn die Elemente in die nächste Zeile fließen sollen, müssten Sie zu `FlowPane` greifen.

Sollen alle Elemente untereinander stehen, verwenden Sie `VBox`, die exakt das Gleiche tut, nur eben vertikal.

Ihr Kinderlein kommet!

Listing 20.7 demonstriert, wie Sie Elemente in die Boxen bringen:

- Sie ermitteln über die Methode `getChildren` die Liste aller Kindknoten. Darin befinden sich alle Elemente und Panes, die vom Pane verwaltet werden.
- Dieser Liste fügen Sie über die Methode `add` ein neues Element hinzu. Mit der Methode `addAll` können mehrere Elemente gleichzeitig eingefügt werden, die dazu kommasepariert als Parameter übergeben werden.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class FxHbox extends Application {

    @Override
    public void start(Stage hauptStage) throws Exception {
        HBox pane = new HBox();
        String str = "";
        for (int i = 0; i < 5; i++) {
            str += i;
            Button b = new Button(str);
            pane.getChildren().add(b);
        }
        Scene scene = new Scene(pane, 400, 300);
        hauptStage.setScene(scene);
        hauptStage.show();

    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 20.7: Der Programmausschnitt demonstriert `HBox`.

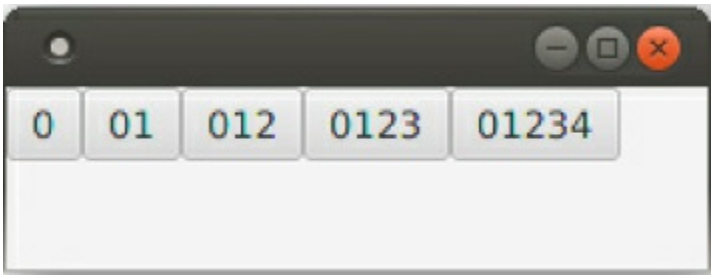


Abbildung 20.4: Buttons in einer `HBox`

Falls Sie den Abstand der Elemente zueinander als zu kuschelig empfinden, können Sie dem Konstruktor zwei Zahlen als Parameter übergeben. Der erste Parameter gibt den horizontalen, der zweite den vertikalen Abstand in Pixeln vor.

FlowPane: Umbrechende Elemente

Im `FlowPane` werden die enthaltenen Elemente nebeneinander angeordnet, bis der Rand zu einem Umbruch zwingt. Sie können in Listing 20.7 jedes Auftreten von `HBox` einfach durch `FlowPane` ersetzen und ein paar Buttons mehr erstellen. Nun können Sie den Unterschied zwischen beiden erkennen, weil nicht mehr alle Elemente in eine Reihe passen.

```
import javafx.application.Application;
import javafx.geometry.Orientation;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;

public class FxFlowPane extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage hauptStage) throws Exception {
        FlowPane pane = new FlowPane(
                Orientation.HORIZONTAL, 5, 3);
        String str = "";
        for (int i = 0; i < 15; i++) {
            str += i;
            Button b = new Button(str);
            pane.getChildren().add(b);
        }
        Scene scene = new Scene(pane, 400, 200);
        hauptStage.setScene(scene);
        hauptStage.show();

    }
}
```

Listing 20.8: `FlowPane` umbricht, wenn die Zeile voll ist.

Ein `FlowPane` kann Elemente nicht nur nebeneinander, sondern auch untereinander anordnen. Um das zu erreichen, geben Sie dem Konstruktor als Parameter den Wert `Orientation.VERTICAL`.

```
FlowPane flowpane = new FlowPane(Orientation.HORIZONTAL, 5, 3);
```

Hier habe ich Ihnen noch zwei Parameter hinzugeschummelt. Diese sind optional.

Sie schaffen damit ein wenig Abstand zwischen die Kinder. Die erste Zahl bestimmt den Abstand nebeneinander und die zweite untereinander.

GridPane: Elemente im Raster

Das `GridPane` stellt ein Raster zur Verfügung, an dem sich die Elemente festhalten können. Der Einsatz dieses Rasters ist ganz typisch für Eingabemasken. Dort befinden sich meist in der linken Spalte die Beschriftungen und in der rechten Spalte die dazu gehörenden Eingabefelder.

Elemente ins Raster einfügen

Das `GridPane` verfügt über die Methode `add`, um Kinder aufzunehmen. Die Parameter sorgen für eine freundliche Aufnahme.

- Der erste Parameter ist das Element, das eingefügt wird. Im Beispiel werden `Label` zur Beschriftung und `TextField` für die Eingabe von Textzeilen verwendet. Dies dürfte auch die typische Anwendung sein.
- Es folgt der Index der Spalte und der Zeile der Rasterposition für das Element. Die Rasterbreite und -höhe bestimmt sich aus dem breitesten beziehungsweise dem höchsten Element.
- Soll sich das Element über mehrere Spalten oder Reihen erstrecken, geben Sie in zwei weiteren Parametern die Anzahl der Spalten und der Reihen an. In Listing 20.9 sehen Sie, wie das erste Label als Überschrift über zwei Spalten geht.

Ohne Parameter setzt `add` alle Elemente in das gleiche Feld an der Position 0, 0. Das dürfte nur im Ausnahmefall von irgendwelchem Nutzen sein, könnte aber die Erklärung sein, falls in Ihrem Programm Elemente auf wundersame Weise nicht auftauchen wollen.

Listing 20.9 zeigt eine typische Eingabemaske, in der die Beschriftungen links stehen und die Eingabefelder rechts daneben.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class FxGridPane extends Application {

    @Override
    public void start(Stage hauptStage) throws Exception {
        GridPane pane = new GridPane();
        TextField editName = new TextField();
        TextField editStr = new TextField();
        TextField editOrt = new TextField();
        pane.add(new Label("Ihre Anschrift"), 0, 0, 2, 1);
        pane.add(new Label("Name:"), 0, 1);
        pane.add(editName, 1, 1);
        pane.add(new Label("Straße:"), 0, 2);
        pane.add(editStr, 1, 2);
        pane.add(new Label("Ort:"), 0, 3);
        pane.add(editOrt, 1, 3);
        Scene scene = new Scene(pane, 400, 300);
        hauptStage.setScene(scene);
        hauptStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 20.9: Eine einfache Adresseingabemaske

Das Ergebnis des Listings 20.9 sehen Sie in Abbildung 20.5.

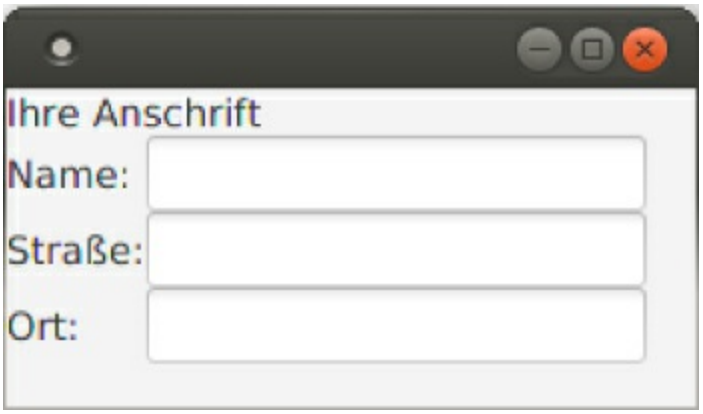


Abbildung 20.5: Eingabemaske mit `GridPane`

Spaltenbreite mit Constraints manipulieren

Das `GridPane` wird die Größe der Spalten und Zeilen automatisch bestimmen. Manchmal möchten Sie die Größe aber beeinflussen. Dazu verwenden Sie Constraints. Die Constraint-Liste des `GridPane` erhalten Sie über die Methoden `getColumnConstraints` für die Spalten und `getRowConstraints` für die Zeilen. In den allermeisten Fällen werden Sie sich aber wohl für die Spaltenbreite interessieren.

Den Constraints können Sie über die Methoden `add` beziehungsweise `addAll` Constraints hinzufügen. Das erste Constraint, das Sie hinzufügen, betrifft die erste Spalte, das zweite Constraint die zweite Spalte und so weiter.

- Pixelweise justieren:

  Dem Konstruktor der `ColumnConstraints` wird die Breite übergeben. Bei einem Parameter ist es eine feste Breite. Werden drei Parameter übergeben, sind das die Mindest-, die Lieblings- und die Maximalbreite.

```
pane.getColumnConstraints().add(new ColumnConstraints(10));
pane.getColumnConstraints().add(
        new ColumnConstraints(10, 100, Double.MAX_VALUE));
```

  Die erste Zeile im Listing wird die Breite der ersten Spalte auf 10 Pixel festlegen, was in den meisten Fällen zu wenig ist. Die zweite Zeile ist realitätsnäher. Sie legt fest, dass die Spalte mindestens 10 Pixel breit ist, idealerweise 100 Pixel bekommt und maximal den größtmöglichen Wert nicht ablehnen wird.

- Relativ justieren:

  Sie können auch das Verhältnis der Spalten relativ zueinander festlegen. Dazu rufen Sie für die `ColumnConstraints` die Methode `setPercentWidth` auf. Das Listing demonstriert, wie man es erreicht, dass die erste Spalte 30 % und die zweite Spalte 70 % der Breite einnimmt.

```
ColumnConstraints col0 = new ColumnConstraints();
col0.setPercentWidth(30);
ColumnConstraints col1 = new ColumnConstraints();
col1.setPercentWidth(70);
pane.getColumnConstraints().addAll(col0, col1);
```

- Raum optimal nutzen:

  Die Maske wird vermutlich am besten aussehen, wenn die Beschriftungsspalte die einmal ermittelte Breite beibehält und die zweite Spalte den kompletten Rest des Platzes einnimmt. Dies erreichen Sie über die Methode `setHgrow` einer `ColumnContraints` wie im Listing demonstriert.

```
ColumnConstraints col0 = new ColumnConstraints();
col0.setHgrow(Priority.NEVER);
ColumnConstraints col1 = new ColumnConstraints();
col1.setHgrow(Priority.ALWAYS);
pane.getColumnConstraints().addAll(col0, col1);
```

Ausrichtung der Elemente im Raster

Sie können festlegen, wie die Elemente innerhalb Ihres Raster-Rechtecks ausgerichtet werden. Dazu stellt `GridPane` einige Methoden zur Verfügung, deren erster Parameter jeweils das entsprechende Element ist.

- `setHalignment` gibt die horizontale Ausrichtung der Elemente innerhalb eines Rasterfelds an. Der zweite Parameter ist eine Konstante aus `HPos`, die LEFT, CENTER oder RIGHT sein kann.

```
pane.setHalignment(lblOrt, HPos.RIGHT);
```

- `setValignment` gibt die vertikale Ausrichtung der Elemente innerhalb eines Rasterfelds an. Der zweite Parameter ist eine Konstante aus `VPos`, die TOP, CENTER oder BOTTOM sein kann.

```
pane.setValignment(lblOrt, VPos.BASELINE);
```

- `setHgrow` und `setVgrow` bestimmen das Vergrößerungsverhältnis. Als zweiter Parameter dient ein Wert aus `Priority`. Dieser ist NEVER oder ALWAYS. Wenn kein Element in der Zeile ALWAYS ist, können sich andere Elemente mit SOMETIMES den Platz teilen.

```
pane.setHgrow(editOrt, Priority.NEVER);
```

- `setMargin` setzt einen Rand um das Kontrollelement. Als zweiter Parameter werden `Insets` verwendet. Dessen Konstruktorparameter beginnen oben und folgen dann im Uhrzeigersinn, also oben, rechts, unten und links.

```
pane.setMargin(lblOrt, new Insets(7,7,7,7));
```

Abstände

Das `GridPane` ist sehr sozial und gibt allen verfügbaren Platz an seine Kinder weiter. Da diese wiederum alles nehmen, was sie kriegen können, sieht das Ergebnis manchmal etwas gedrängt aus. Sie können das `GridPane` dazu bringen, etwas Platz zwischen die Spalten und Zeilen zu schieben. Dazu werden die Methoden `setHgap` und `setVgap` eingesetzt.

```
pane.setHgap(horizonalAbstand);
pane.setVgap(vertikalAbstand);
pane.setGridLinesVisible(true);
```

Für die Entwicklung eines `GridPane` kann es sehr hilfreich sein, die Linien zu sehen, an denen es seine Kinder ausrichtet. Die Methode `setGridLinesVisible` zeigt die Linien der Raster an.

BorderPane: Zentrales Element mit Rändern

Das `BorderPane` betont das Element in der Mitte und gibt ihm den meisten Raum. Um diesen Bereich ordnet das `BorderPane` Regionen an, die für Steuerungselemente verwendet werden können. Es besitzt Methoden, um Elemente oder Panes an diesen Positionen anzuordnen. Jede dieser Methoden verwendet ein Pane oder ein Kontrollelement als Parameter.

- ✔ `setCenter`

  Das übergebene Element wird im Zentralbereich des `BorderPane` positioniert. Dieser dehnt sich beim Vergrößern des Fensters aus. Hier werden oft Bilder, Eingabemasken oder Texte dargestellt.
- ✔ `setTop`

  Im oberen Bereich wird meist eine Menüleiste oder eine Symbolleiste positioniert. Der Bereich wird auf die benötigte Höhe gesetzt. Die Höhe ändert sich nicht, wenn die Fenstergröße verändert wird.
- ✔ `setBottom`:

  Ganz unten befindet sich oft eine Statusleiste, in der angezeigt wird, was die Anwendung gerade so treibt und ob es ihr gut geht. Bei Dialogen platzieren Sie hier die Buttons. Auch diese Leiste verändert sich in der Höhe typischerweise nicht.
- ✔ `setLeft`:

  Manche Anwendung stellt links neben dem Hauptinhalt einen Navigationsbereich dar.
- ✔ `setRight`

  Auf der rechten Seite werden gern Detailsichten angezeigt.

 Sie müssen nicht alle Bereiche belegen. Beispielsweise könnten Sie auch für

einen Taschenrechner ein solches `BorderPane` verwenden. In diesem Fall würde das Display idealerweise in den Top-Bereich gehören. Die Tasten würden Sie dann in die Mitte legen. Die Bereiche unten, links und rechts benötigen Sie nicht.

Listing 20.10 demonstriert den Aufbau eines `BorderPanes`. Im Zentrum wird eine Liste abgelegt. Der Anwender wird erwarten, dass die Liste größer wird, wenn er das Fenster vergrößert. Und genau das tut das `BorderPane` auch. Die Liste enthält die Dinge, die besser werden müssen, wenn man einen zufriedenen und in sich ruhenden Menschen fragt. Ein Wutbürger benötigt da etwas mehr. Damit muss er trotz aller Wut noch warten, bis wir zur `ListView` kommen.

Die Überschrift wurde als `Label` über der Liste platziert. Links und rechts steht ein `Button` bereit, um Elemente in die Liste einzufügen oder zu entfernen. Bislang sind die beiden genauso ohne jede Funktion wie der OK-Button am Fuße des Fensters.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ListView;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class Border extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage hauptStage) {
        BorderPane layoutContainer = new BorderPane();
        layoutContainer.setTop(new Label(
                "Was alles besser werden muss"));
        layoutContainer.setCenter(new ListView<String>());
        layoutContainer.setBottom(new Button("OK"));
        layoutContainer.setLeft(new Button("Löschen"));
        layoutContainer.setRight(new Button("Hinzufügen"));
        Scene scene = new Scene(layoutContainer, 400, 200);
        hauptStage.setScene(scene);
        hauptStage.show();
    }
}
```

Listing 20.10: BorderPane mit Liste und Buttons

Abbildung 20.6 zeigt, wie das Programm aussehen wird, wenn Sie es starten.

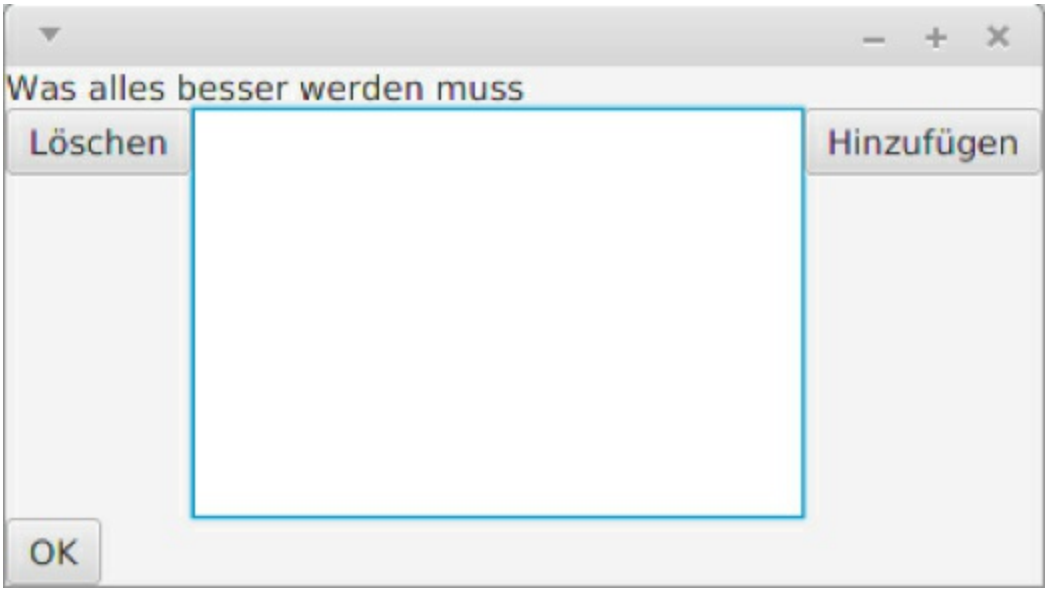


Abbildung 20.6: `BorderPane` mit Liste und Rändern

Ausrichtung

Die Elemente werden linksbündig in ihren Bereichen ausgerichtet. Nur im zentralen Bereich wird das Element horizontal wie vertikal zentriert. Mit der statischen Methode `BorderPane.setAlignment` können Elemente innerhalb ihres Bereichs ausgerichtet werden.

```
import javafx.geometry.Pos;
...
Button okButton = new Button("OK");
layoutContainer.setBottom(okButton);
BorderPane.setAlignment(okButton, Pos.CENTER);
```

Listing 20.11: Zentrierte Ausrichtung des OK-Buttons

Sie sehen, dass die Methode `setAlignment` das Button-Objekt benötigt. Darum musste der OK-Button explizit angelegt werden. In einer realen Umgebung ist dies aber eh erforderlich, da Sie dem Button ja vermutlich auch irgendeine Funktion zuordnen wollen.

StackPane

Das `StackPane` dient dazu, dass alle Kinder übereinandergestapelt werden. Dadurch ist nur das jeweils obere sichtbar. Elemente werden mit dem Aufruf `getChildren().add(button)` hinzugefügt. Seine besondere Berühmtheit erlangt er wohl durch sein Auftauchen in diversen Hello-World-Programmen für FX sowie im Rahmenprogramm von NetBeans. Es steht allerdings zu vermuten, dass `StackPane` in diesem Zuge den Titel des am häufigsten ersetzten Panes der FX-Umgebung erlangen wird.

Kombination von Panes

Um vernünftige Dialoge zu bauen, wird es notwendig sein, dass Sie mehrere Arten

von Panes kombinieren. Das Beispiel in Listing 20.12 zeigt ein `BorderPane`, in dessen Zentrum ein `GridPane` dafür sorgt, dass die Maske sauber ausgerichtet ist. Im Bottom-Bereich des `BorderPane` ist wiederum ein `FlowPane` aktiv, das die Buttons nebeneinander anordnet.

```
import javafx.application.Application;
import javafx.geometry.NodeOrientation;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.ColumnConstraints;
import javafx.scene.layout.FlowPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Priority;
import javafx.stage.Stage;

public class FxBorderGrid extends Application {

    @Override
    public void start(Stage hauptStage) throws Exception {
        BorderPane border = new BorderPane();
        GridPane grid = new GridPane();
        ColumnConstraints col0 = new ColumnConstraints();
        col0.setHgrow(Priority.NEVER);
        ColumnConstraints col1 = new ColumnConstraints();
        col1.setHgrow(Priority.ALWAYS);
        grid.getColumnConstraints().addAll(col0, col1);
        grid.setHgap(4);
        grid.setVgap(4);
        TextField editName = new TextField();
        TextField editStr = new TextField();
        TextField editOrt = new TextField();
        grid.add(new Label("Ihre Anschrift"), 0, 0, 2, 1);
        grid.add(new Label("Name:"), 0, 1);
        grid.add(editName, 1, 1);
        grid.add(new Label("Straße:"), 0, 2);
        grid.add(editStr, 1, 2);
        grid.add(new Label("Ort:"), 0, 3);
        grid.add(editOrt, 1, 3);
        border.setCenter(grid);
        FlowPane flow = new FlowPane(4, 4);
        flow.setNodeOrientation(NodeOrientation.RIGHT_TO_LEFT);
        flow.getChildren().add(new Button("OK"));
        flow.getChildren().add(new Button("Abbrechen"));
        border.setBottom(flow);
        Scene scene = new Scene(border);
        hauptStage.setScene(scene);
        hauptStage.show();
    }
```

```
    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 20.12: Kombinierte Panes

In diesem Beispiel wurde auf die Größenvorgabe im Konstruktor von `Scene` verzichtet. Das Fenster richtet sich also ausschließlich nach den Bedürfnissen seiner Kinder.



Abbildung 20.7: `GridPane`-Eingabemaske im `BorderPane`

Sie sehen also, dass die Kunst des Layouts von grafischen Oberflächen darin besteht, die verschiedenen Panes geschickt ineinander zu verschachteln, bis zuletzt Ihre Kontrollelemente sauber dahin kommen, wo sie hingehören.

Reiter und TabPane

Einige Dialoge sind so umfangreich, dass sie nicht in ein Fenster passen. Dann verwendet man gern Laschen oder Reiter, wie man sie von Hängeregistern kennt. Dieses Modell empfindet ein `TabPane` nach. Aus diesem Grund ist ein `TabPane` eine Mischung aus Kontrollelement und Layout. Wird ein Reiter angeklickt, funktioniert er ähnlich wie ein Button. Als Reaktion wird eine neue Dialogseite aufgeschlagen.

Erstellen der Reiter und ihrer Seiten

Das `TabPane` nimmt Objekte der Klasse `Tab` auf. Ein `Tab` erhält als Konstruktorparameter den Namen, der im Reiter angezeigt wird, und dient gleichzeitig als Rahmen für ein Pane. `TabPane` sammelt die Tabs in einer Liste, die über die Methode `getTabs` erreicht wird. Ein neues `Tab` wird über die Methode `add` dieser Liste hinzugefügt:

```
TabPane tabPane = new TabPane();
Tab tabGast = new Tab("Gast");
Tab tabWohnung = new Tab("Zimmer");
Tab tabBelegung = new Tab("Belegung");
tabPane.getTabs().add(tabGast);
tabPane.getTabs().add(tabWohnung);
tabPane.getTabs().add(tabBelegung);
```

Listing 20.13: Ein TabPane nimmt Tabs auf.



Abbildung 20.8: Ein TabPane mit drei Reitern

Dialoge hinter die Tabs setzen

Sie können die Reiter zwar anklicken, aber es erscheint jeweils eine leere Fläche, da die Tabs noch keinen Inhalt haben. Um diesen bedauerlichen Zustand zu ändern, wird die Methode `setContent` aufgerufen. Als Parameter erwartet sie ein Element oder sinnvoller ein Pane, in dem mehrere Elemente angeordnet werden können.

```
GridPane paneGast = new GridPane();
// Hier wird der Dialog für den Gast aufgebaut
tabGast.setContent(paneGast);
```

Schließkreuz

Die Reiter sind im Normalzustand mit einem Schließkreuz versehen, die es dem Anwender ermöglichen, ein Tab zu schließen. In den meisten Fällen ist das nicht erwünscht. Mit der folgenden Zeile lässt sich das verhindern:

```
tabPane.setTabClosingPolicy(TabClosingPolicy.UNAVAILABLE);
```

Umschalten

Sie können zwischen den Reitern auch aus dem Programm umschalten. Das kann sinnvoll sein, wenn Sie gleich eine bestimmte Seite als Standard aufschlagen wollen. Die folgende Zeile schaltet automatisch auf die Belegungsseite:

```
tabPane.getSelectionModel().select(tabBelegung);
```

Der Parameter der Methode `select` kann entweder das Tab selbst oder seine Nummer sein.

Reaktion auf das Umschalten

In einigen Fällen möchten Sie, dass das `TabPane` dem Programm meldet, dass jemand einen anderen Reiter anklickt. Das kann beispielsweise interessant sein,

wenn Sie dann erst die Eingabefelder des Dialogs prüfen möchten. Die folgenden Zeilen erzeugen einen `ChangeListener`, der gleich vor Ort die Methode `changed` implementiert, um auf die Änderung zu reagieren.

```
tabPane.getSelectionModel().selectedItemProperty()
        .addListener(new ChangeListener<Tab>() {
    @Override
    public void changed(
            ObservableValue<? extends Tab> observable,
            Tab oldValue, Tab newValue) {
        if (newValue == tabGast) {
            // ...
        } else if (newValue == tabBelegung) {
            // ...
        }
    }
});
```

Alles zusammengesetzt

Das Programm in Listing 20.14 baut einen fast vollständigen Dialog für die Zimmerbelegung auf. Der Reiter für den Gast enthält den Dialog, der die Adresse des Gastes erfasst. Der Reiter für die Belegung zeigt eine `ListView`, die zunächst leer ist. Den Reiter für das Zimmer habe ich Ihnen überlassen, damit Sie auch etwas zu tun haben.

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.geometry.NodeOrientation;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ListView;
import javafx.scene.control.Tab;
import javafx.scene.control.TabPane;
import javafx.scene.control.TabPane.TabClosingPolicy;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.FlowPane;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class FxTabPane  extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage stage) throws Exception {
        TabPane tabPane = new TabPane();
        tabPane.setTabClosingPolicy(TabClosingPolicy.UNAVAILABLE);
        Tab tabGast = new Tab("Gast");
        Tab tabWohnung = new Tab("Zimmer");
        Tab tabBelegung = new Tab("Belegung");
        tabPane.getTabs().add(tabGast);
        tabPane.getTabs().add(tabWohnung);
        tabPane.getTabs().add(tabBelegung);
        tabPane.getSelectionModel().select(tabBelegung);
        // Tabs mit Dialogen füllen
        GridPane gastDialog = new GridPane();
        gastDialog.add(new Label("Name:"), 0, 1);
        gastDialog.add(new TextField(), 1, 1);
        gastDialog.add(new Label("Straße:"), 0, 2);
        gastDialog.add(new TextField(), 1, 2);
        gastDialog.add(new Label("Ort:"), 0, 3);
        gastDialog.add(new TextField(), 1, 3);
        tabGast.setContent(gastDialog);
        tabBelegung.setContent(new ListView<String>());
        // Reaktionen
        tabPane.getSelectionModel().selectedItemProperty()
                .addListener(new ChangeListener<Tab>() {
```

```
            @Override
            public void changed(
                    ObservableValue<? extends Tab> observable,
                    Tab oldValue, Tab newValue) {
                if (newValue == tabGast) {
                    System.out.println("Gast");
                    // ...
                } else if (newValue == tabBelegung) {
                    System.out.println("Belegung");
                    // ...
                }
            }
        });
        // TabPane in einen Border mit Schließ button
        BorderPane border = new BorderPane(tabPane);
        FlowPane flow = new FlowPane (4, 4);
        flow.setNodeOrientation(
                NodeOrientation.RIGHT_TO_LEFT);
        flow.getChildren().add(new Button("Schließen"));
        border.setBottom(flow);
        Scene scene = new Scene(border, 400, 200);
        stage.setScene(scene);
        stage.show();
    }
}
```

Listing 20.14: Ein TabPane mit mehreren Dialogen

Das `TabPane` selbst wurde in das Zentrum eines `BorderPane` gesteckt. Auf diese Weise könnte in den Bottom noch ein Schließbutton eingebaut werden, der von allen Seiten des `TabPane` erreichbar ist.

Es kommt Leben in die Bude: EventHandler

Bisher haben wir die Elemente nur zusammengebaut und geschaut, dass sie auch nett aussehen. Aber wirklich interessant wird es erst, wenn Daten eingegeben und vom Programm übernommen werden sollen. Java FX arbeitet ereignisorientiert. Das bedeutet, sobald etwas passiert, wird Ihr Programm darüber informiert.

Allerdings wollen Sie nicht alles wissen, was auf der grafischen Oberfläche passiert. Das meiste macht so ein FX-Programm wunderbar ohne Ihre Hilfe. Darum müssen Sie FX erzählen, was Sie wissen wollen, also für welches Event Ihr

Programm angemeldet werden soll. Dann benötigen Sie auch einen Event-Handler, also ein Stück Programm, das die Ereignisse auswertet. An einem Button wollen wir das einmal durchspielen:

1. Damit Sie Ereignisse fangen können, implementieren Sie das Interface `EventHandler`.
2. Das Interface ist ein Generics, das in seinen spitzen Klammern den Typparamter des Ereignisses festlegt, nämlich das Event. Im Beispiel des Buttons ist es das `ActionEvent`.
3. Die Implementierung beschränkt sich auf die Methode `handle`. Darin erfolgen Ihre Reaktionen auf Buttonereignisse. Sie hat als einzigen Parameter die Art des Events, das bei dem Generics angegeben wird.
4. Damit Java FX von Ihrem `EventHandler`-Objekt erfährt, müssen Sie dieses über die Button-Methode `setOnAction` anmelden.

In seiner fast kürzestmöglichen Variante erfolgt die Anmeldung für das Auffangen eines Klicks auf einen Button mit den folgenden Zeilen.

```
Button button = new Button("Drück mich!");
button.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hab' dich auch lieb!");
    }
});
```

Listing 20.15: Anmeldung und Implementierung des Event-Handlers

Schauen wir noch einmal kurz, was diese Zeilen genau tun:

- ✔ Die Methode `setOnAction` meldet einen `EventHandler` für den Button `button` an.
- ✔ Dieser `EventHandler` wird direkt innerhalb des Parameters erzeugt.
- ✔ Dabei wird eine Erweiterung erzeugt, deren Methode `handle` überschrieben wird. Diese reagiert später auf den Druck des Buttons.

Natürlich können Sie das auch ordentlich in einzelnen Schritten machen, indem Sie die Klasse `EventHandler` und ein Objekt Ihrer Ableitung als Parameter an `setOnAction` übergeben. Das lohnt sich vor allem dann, wenn Sie das Ereignis mehrerer Auslöser überwachen wollen.

Das Programm in Listing 20.16 enthält einen `Button` und ein `Label`. Sobald Sie den `Button` drücken, erhält das `Label` eine neue Beschriftung.

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class FxDirektEvent extends Application {

    @Override
    public void start(Stage hauptStage) {
        Label label = new Label("Hallo Benutzer");
        Button button = new Button();
        button.setText("Drück mich!");
        button.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                label.setText("Hab' dich auch lieb!");
            }
        });
        BorderPane layoutContainer = new BorderPane();
        layoutContainer.setBottom(button);
        layoutContainer.setCenter(label);
        Scene scene = new Scene(layoutContainer, 400, 300);
        hauptStage.setScene(scene);
        hauptStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 20.16: Direkte Behandlung des Events

Alles in Lambda

Freunde der Lambda-Programmierung können das sogar noch kürzer. Lambda steht seit Java 8 zur Verfügung. Damit können anonyme Klassen mit genau einer Methode kurzgefasst werden.

```
button.setOnAction(
        event ->
        System.out.println("Hab' dich auch lieb!"));
```

Die Konstruktion, im Parameter von `setOnAction` ein anonymes Objekt zu erzeugen, das eine Handle-Funktion implementiert, kann dank Lambda noch einmal verkürzt werden. Mit Lambda können Sie direkt den Methodenrumpf einer anonymen Klasse füllen, die nur eine Methode enthält. Und das ist bei den Event-Handlern der Fall.

Die Klasse als EventHandler-Implementierung

Etwas eleganter wirkt es, wenn die Klasse, die sich um den Aufbau der GUI kümmert, auch gleich den `EventHandler` implementiert. Dazu muss sie nur angeben, dass sie `EventHandler` mit dem entsprechenden Event implementiert. Das folgende Beispiel kümmert sich um das `ActionEvent` eines Buttons.

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class FxMainEvent extends Application
        implements EventHandler<ActionEvent> {

    private Label label = new Label("Hallo Benutzer");

    @Override
    public void start(Stage hauptStage) {
        Button button = new Button();
        button.setText("Drück mich!");
        button.setOnAction(this);
        BorderPane layoutContainer = new BorderPane();
        layoutContainer.setBottom(button);
        layoutContainer.setCenter(label);
        Scene scene = new Scene(layoutContainer, 400, 300);
        hauptStage.setScene(scene);
        hauptStage.show();
    }
```

```
    @Override
    public void handle(ActionEvent event) {
        label.setText("Hab' dich auch lieb!");
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 20.17: Die `Application`-Klasse fängt das Event.

Nun sind der Aufbau und die Ereignisbehandlung getrennt. Allerdings müssen Sie so das Label globalisieren, damit es von der Ereignisbehandlung erreichbar ist.

Wer war's?

Die Methode `handle` erfährt durch ihren Parameter die Randbedingungen eines Ereignisses. So kann die Methode feststellen, welcher Button angeklickt wurde. Solange es nur einen Button gibt, ist diese Information nur mäßig spannend. Wird aber eine zentrale Ereignismethode für mehrere Kontrollelemente verwendet, will man einfach wissen, wer der Täter ist. Diese Frage beantwortet die Methode `getSource`. Die folgende Methode `handle` unterscheidet, ob der Benutzer auf den OK-Button oder den Abbruch-Button geklickt hat.

```
@Override
public void handle(ActionEvent event) {
    if (event.getSource()==okButton) {
        System.out.println("Action!");
    } else if (event.getSource()==cancelButton) {
        System.out.println("War alles nur Spa ß !");
    }
}
```

Ein paar Kontrollelemente

Für einen Dialog benötigt man mindestens ein paar Eingabefelder, passende Beschriftungen und den einen oder anderen Button. Darüber hinaus gibt es aber noch einige Möglichkeiten, um die Interaktion mit dem Benutzer zu suchen.

Das passive Label

Ein Label ist das faulste Kontrollelement. Es zeigt keine Aktivität und löst keine Ereignisse aus. Das Label ist also ein richtiger Langweiler, wird aber dennoch immer wieder eingeladen. Es ist nämlich dekorativ und kann von anderen Elementen beschriftet werden, damit die zeigen können, was für tolle Hechte sie

sind.

```
import javafx.scene.control.Label;
// ...
Label beschriftung = new Label("Erhellender Text");
```

Die Beschriftung können Sie mit der Methode `setText` auch nach der Initialisierung vom Programm aus ändern. Das wird dann wichtig, wenn Sie das Label verwenden, um Meldungen auszugeben.

Mit der Methode `getText` können Sie die Beschriftung eines Labels vom Programm aus auslesen.

Darüber hinaus können Sie mit der Methode `setFont` den Zeichensatz des Labels ändern. Als Parameter wird ein `Font` erwartet. Das folgende Beispiel verwendet einen Serif-Font mit normalem Gewicht und einer Größe von 20 Punkt. Um eine fette Schrift zu verwenden, würden Sie `BOLD` verwenden. Mit Blick auf mein Übergewicht verzichte ich darauf aber lieber.

```
label.setFont(Font.font("Serif", FontWeight.NORMAL, 20));
```

Der actiongeladene Button

Der `Button` ist ein `Label` mit Drückfunktion. Sie können ihn anklicken und damit löst er ein `ActionEvent` aus. Genau dieses Ereignis interessiert Sie, denn ansonsten würden Sie ja keinen Button in Ihr Fenster setzen.

EventHandler

Um das Ereignis aufzufangen, benötigen Sie einen `EventHandler`. Sobald Sie diesen haben, müssen Sie dem Button mitteilen, dass es da jemanden gibt, der sich auf seine Ereignisse freut. Dieses Interesse melden Sie mit der Methode `setOnAction` beim `Button` an.

All das wird Ihnen verdächtig bekannt vorkommen. Und tatsächlich habe ich Sie damit bereits zuvor unterhalten. In Listing 20.16 können Sie schauen, wie Sie Ereignisse angemessen behandeln.

Button-Methoden

Der Beschriftungstext eines Buttons kann dem Konstruktor gleich übergeben werden. Der `Button` verfügt aber auch über alle Methoden, die das `Label` kennt, schließlich hat er all dies geerbt. Und darum kann mit der Methode `setText` die Beschriftung nachträglich geändert werden. Und mit `getText` können Sie die Beschriftung auslesen. Als Besonderheit können Sie innerhalb des Textes einen Unterstrich angeben. Dadurch wird das folgende Zeichen – typischerweise in Kombination mit der Alt-Taste – als Auslöser per Tastatur definiert.

Sie können einen Button entschärfen, indem Sie der Methode `setDisable` als Parameter den Wert `true` übergeben. Er wird daraufhin leicht grau. Eine Methode

`setEnable` gibt es nicht. Stattdessen muss als Parameter `false` übergeben werden.

Die Methode `isDisable` gibt einen Wert vom Typ `boolean` zurück, der angibt, ob der Button deaktiviert ist.

Die Methode `setOnAction` setzt den `ActionHandler` für den `Button`. Das entsprechende Objekt muss als Parameter übergeben werden. Aber ich vermute, das haben Sie schon einmal irgendwo gehört.

Eingabefeld TextField

In fast jedem Dialog finden Sie ein Eingabefeld, in dem der Anwender etwas eintippen kann. Diese Elemente heißen bei Java FX `TextField`.

Befüllen und Auslesen des Eingabefelds

In der Regel werden diese Felder beim Start eines Dialogs gesetzt und beim Beenden ausgelesen. Dazu stellt `TextField` zwei Methoden zur Verfügung:

- Die Methode `getText` benötigt keinen Parameter und gibt den Inhalt der Eingabe als `String` zurück. Diese Methode wird typischerweise aufgerufen, wenn ein Dialog mit OK bestätigt wird.
  ```
  String strEingabe = eingabe.getText();
  ```
- Mit der Methode `setText` kann das Programm einen Vorgabewert in das Eingabefeld setzen.
  ```
  TextField eingabe = new TextField();
  eingabe.setText("Vorgabeeingabe");
  ```

Ausrichtung

Sie können den Text in einem `TextField` durch den Aufruf der Methode `setAlignment` ausrichten. Als Parameter wählen Sie:

- `Pos.CENTER_RIGHT`: für eine mittige vertikale, rechtsbündige Ausrichtung
- `Pos.CENTER`: für eine komplett mittige Ausrichtung
- `Pos.CENTER_LEFT`: für eine mittige vertikale, linksbündige Ausrichtung

Sie können die vertikale Ausrichtung ändern, indem Sie `CENTER` durch `TOP` für oben oder `BOTTOM` für unten ersetzen.

Fokus

Ein Eingabeelement hat den Fokus, wenn es bereit ist, die Eingaben des Anwenders entgegenzunehmen. Normalerweise muss sich ein Programm nicht um den Fokus kümmern. Wenn Sie allerdings noch vor dem Beenden des Dialogs die Korrektheit der Eingaben prüfen wollen, ist es hilfreich zu wissen, wann der Benutzer von einem Feld zum nächsten wechselt. Das ist genau der Zeitpunkt, wenn ein Feld den

Fokus verliert und ein anderes den Fokus erhält.

In solchen Fällen muss das Programm auf das Fokuswechselereignis reagieren. Dazu setzt es einen Listener auf die Fokus-Eigenschaften. Das Programm in Listing 20.18 zeigt das Vorgehen. Das `ActionEvent` des OK-Buttons zeigt das Auslesen des Eingabefelds.

```
import javafx.application.Application;
import javafx.beans.InvalidationListener;
import javafx.beans.Observable;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxTextField extends Application {
    @Override
    public void start(Stage hauptStage) {
        Label label = new Label();
        VBox pane = new VBox();
        TextField eingabe = new TextField();
        eingabe.focusedProperty().addListener(new Invalidation
          Listener() {
            @Override
            public void invalidated(Observable observable) {
                if (eingabe.isFocused()) {
                    System.out.println("Fokus bekommen");
                } else {
                    System.out.println("Fokus verloren");
                }
            }
        });
        Button ok = new Button("OK");
        ok.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                label.setText(eingabe.getText());
            }

        });
        pane.getChildren().add(label);
        pane.getChildren().add(eingabe);
        pane.getChildren().add(ok);
        Scene scene = new Scene(pane, 250, 150);
        hauptStage.setScene(scene);
        hauptStage.show();
    }
    public static void main(String[] args) {
        launch(args);
```

```
    }
}
```

Listing 20.18: Fokuswechsel abfangen

Die Fokuswechsel werden einfach auf der Konsole ausgegeben.

```
Fokus verloren
Fokus bekommen
Fokus verloren
Fokus bekommen
Fokus verloren
Fokus bekommen
Fokus verloren
Fokus bekommen
```

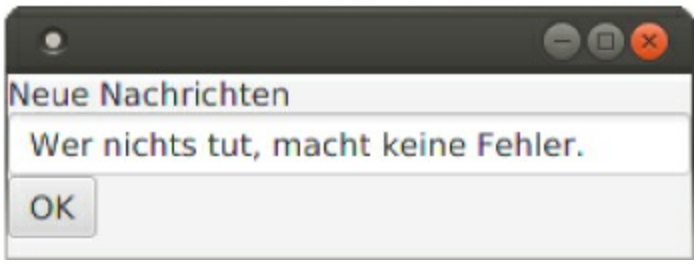


Abbildung 20.9: Fokuswechselanzeige

Beschränkung der Eingabe

Anwender sind manchmal lästig. Sie kommen auf die Idee, absoluten Blödsinn an Stellen einzutippen, wo er nichts verloren hat. Das Programm muss dann wie Aschenputtel die schlechte Eingaben aussortieren. Dabei können die Programme leider weder auf die Hilfe von Tauben noch auf die von Hörenden hoffen.

Als Beispiel soll ein Textfeld dienen, das nur numerische Werte zulässt. Wie in der objektorientierten Programmierung üblich, erweitert die Klasse `FxTextFieldNum` in Listing 20.19 dazu die Klasse `TextField`.

```
class NumTextField extends TextField {
    @Override
    public void replaceText(int start, int end, String text) {
        if (text.isEmpty() || "0123456789.,-".contains(text)) {
            super.replaceText(start, end, text);
        }
    }
}
```

Listing 20.19: Numerischer Filter in `TextField`

Die Methode `replaceText` wird überschrieben. Sie wird intern bei jeder Zeicheneingabe aufgerufen. Die neue Methode überprüft, ob die Eingabe genehm ist. Dann ruft sie die Methode der Basisklasse auf.

Wenn Sie ein Eingabefeld nun von der Klasse `FxTextFieldNum` erzeugen, wird der Anwender nur noch Ziffern, Komma, Punkt und Minuszeichen eingeben können.

Abgehakt: CheckBox oder ToggleButton

Eine `CheckBox` und ein `ToggleButton` unterscheiden sich funktional wenig. Beide visualisieren einen booleschen Zustand. Sie unterscheiden sich aber in ihrem Erscheinungsbild:

- Eine `CheckBox` ist eine Sonderform eines Buttons, der einen Haken haben kann oder eben nicht. Wie der Name schon andeutet, würde er prima in eine Checkliste passen. Der Punkt ist eben abgehakt oder nicht.
- Ein `ToggleButton` verhält sich genauso, sieht aber aus wie ein `Button`. Wird er angeklickt, bleibt er in seiner Vertiefung. Bei nochmaligem Anklicken erhebt er sich wieder und sieht dann wieder so aus wie ein normaler `Button`.

Beide Varianten sind in Abbildung 20.10 zu sehen. Die beiden Buttons sind im angewählten Zustand. Das Bild wurde durch das Programm in Listing 20.20 erzeugt. Bei jedem Klick wird ein Event ausgelöst, der über den EventHandler gefangen werden kann, sofern er per `setOnAction` angemeldet wurde.



Abbildung 20.10: `CheckBox` und `ToggleButton`

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.CheckBox;
import javafx.scene.control.Label;
import javafx.scene.control.ToggleButton;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxToggleCheck extends Application
        implements EventHandler<ActionEvent> {

    public static void main(String[] args) {
        launch(args);
    }

    private Label label
            = new Label("CheckBox und ToggleButton");
    CheckBox check = new CheckBox("Check");
    ToggleButton toggle = new ToggleButton("Toggle");
```

```
    @Override
    public void start(Stage hauptStage) {
        check.setOnAction(this);
        toggle.setOnAction(this);
        VBox pane = new VBox();
        pane.getChildren().add(check);
        pane.getChildren().add(toggle);
        pane.getChildren().add(label);
        Scene scene = new Scene(pane, 250, 80);
        hauptStage.setScene(scene);
        hauptStage.show();
    }

    @Override
    public void handle(ActionEvent event) {
        String str = "CheckBox: " + check.isSelected()
                + " ToggleButton: " + toggle.isSelected();
        label.setText(str);
        if (event.getSource()==check) {
        CheckBox chk = (CheckBox) event.getSource();
        chk.setSelected(false);
        }
    }
}
```

Listing 20.20: CheckBox und ToggleButton

Die Elemente bringen folgende Methoden mit:

- `setSelected`: Das Element wird gesetzt oder entsetzt, je nachdem, ob der Parameter `true` oder `false` ist. Damit entspricht das Element einem Politiker, der auch entsetzt ist, wenn er nicht gewählt wurde.
- `isSelected`: Das Element liefert einen booleschen Wert, der angibt, ob das Element angewählt ist oder nicht.
- `setText`: Damit können Sie die Beschriftung des Elements nachträglich ändern.

Es kann nur einen geben: RadioButton

Ein `RadioButton` unterscheidet sich von einer `CheckBox` insofern, als dass der `RadioButton` immer im Rudel auftritt. Der Name stammt daher, dass man bei einem Radio sinnvollerweise immer nur einen Sender auswählen kann. Wird also auf einen `RadioButton` geklickt, springt der bisher angewählte heraus.

Damit das klappt, müssen die `RadioButtons` in einer Gruppe organisiert werden. Diese Gruppe ist ein Objekt der Klasse `ToggleGroup`. Ein `RadioButton` erlangt die Mitgliedschaft in einer solchen Gruppe, indem es von dieser über die Methode `setToggleGroup` eingeladen wird.

Das Programm in Listing 20.21 verwendet ein String-Array, um die Namen von Fernsehsendern zu speichern. Es erzeugt dann in einer Schleife jeweils einen `RadioButton`, den es in einem globalen Array namens `rbSender` speichert. Jedem `RadioButton` wird in der Schleife ein Platz im `pane` zugewiesen, eine Mitgliedschaft in der `ToggleGroup` angeboten und erhält auch einen `EventHandler` per `setOnAction`.



Abbildung 20.11: `RadioButtons` für das Fernsehprogramm

Der `EventHandler` muss für `RadioButtons` eigentlich nur dann realisiert werden, wenn das Programm beim Wechseln der `RadioButtons` sofort reagieren will. Oftmals werden `RadioButtons` allerdings erst ausgelesen, wenn ein OK-Button gedrückt wird. Dann reicht es natürlich, nur den OK-Button zu behandeln und von dort mit `isSelected` zu prüfen, welcher `RadioButton` gewählt ist.

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxRadioButton extends Application
        implements EventHandler<ActionEvent> {

    public static void main(String[] args) {
        launch(args);
    }

    private Label label = new Label("RadioButtons");
    RadioButton []rbSender = null;

    @Override
    public void start(Stage hauptStage) {
        VBox pane = new VBox();
        ToggleGroup senderGroup = new ToggleGroup();
        String strSender[] = {"ARD", "ZDF", "NDR", "arte"};
        rbSender = new RadioButton[strSender.length];
        for (int i=0; i<strSender.length; i++) {
            rbSender[i] = new RadioButton(strSender[i]);
            rbSender[i].setOnAction(this);
            rbSender[i].setToggleGroup(senderGroup);
            pane.getChildren().add(rbSender[i]);
        }
        pane.getChildren().add(label);
        Scene scene = new Scene(pane, 250, 150);
        hauptStage.setScene(scene);
        hauptStage.show();
    }

    @Override
    public void handle(ActionEvent event) {
        String str = "";
        label.setText(str);
        for (RadioButton rb : rbSender) {
            if (event.getSource() == rb) {
                System.out.println("array");
                label.setText(rb.getText());
            }
```

```
        }
        // Ohne Speichern der RadioButtons:
        if (event.getSource().getClass()
                == RadioButton.class) {
            System.out.println("class");
            RadioButton rb = (RadioButton) event.getSource();
            str = rb.getText();
            label.setText(str);
        }
    }
}
```

Listing 20.21: RadioButtons zur Senderwahl

In der Methode `handle` wird der auslösende `RadioButton` über die Methode `getSource` bestimmt. Dieser wird dann per Schleife aus dem Array der `RadioButtons` ermittelt.

Da das Programm die Beschriftung der `RadioButtons` ausliest, ist eine Speicherung des Arrays nicht zwingend erforderlich. Die Methode `handle` könnte genauso gut ermitteln, ob es sich bei dem Objekt um einen `RadioButton` handelt, und dann einfach den Text auslesen. Voraussetzung ist natürlich, dass der `EventHandler` nur einen Satz `RadioButtons` verwaltet.

```
if (event.getSource().getClass() == RadioButton.class) {
    RadioButton rb = (RadioButton) event.getSource();
    str = rb.getText();
    label.setText(str);
}
```

Die schnelle Wahl: ComboBox

Die `ComboBox` ermöglicht, ganz ähnlich wie `RadioButtons` die Auswahl aus einer Liste von Vorgaben. Allerdings belegt sie weniger Platz, weil nicht alle Möglichkeiten ständig sichtbar sind, sondern hinter einer Klappe verschwinden. Die `ComboBox` zeigt immer nur den aktuell gewählten Eintrag. Beim Anklicken erscheint eine Liste, die nach oben und unten über das Element hinausragt. Das gewählte Element erhält einen Haken.



Abbildung 20.12: Eine `ComboBox`

Beim folgenden Listing wird eine kleine `ComboBox` für die Fernsehsender erstellt. Sie sollten schon einmal einen Blick auf die `ObservableList` werfen. Die wird noch besonders spannend.

```
public class FxComboBox extends Application {

    @Override
    public void start(Stage hauptStage) {
        ObservableList<String> datenliste
            = FXCollections.observableArrayList("ARD", "ZDF", "N3");
        ComboBox<String> combo = new ComboBox<String>(datenliste);
        VBox pane = new VBox();
        pane.getChildren().add(combo);
        Scene scene = new Scene(pane, 250, 80);
        hauptStage.setScene(scene);
        hauptStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 20.22: Auswahl des Fernsehsenders per `ComboBox`

Über die Methode `getValue` stellt die `ComboBox` den aktuell ausgewählten String zur Verfügung. Wurde bislang nichts ausgewählt, ist der Rückgabewert `null`.

ObservableList und Wechselereignisse

Als Karton für die Senderliste wurde eine `ObservableList` verwendet. Eine `ObservableList` kann nicht nur Strings verwalten, wie die Generics-Definition schon vermuten lässt. Darum soll das Beispiel etwas komplizierter werden. In der Liste wird eine eigene Klasse abgelegt.

Statt der sehr simplen Klasse `MeineKlasse` kann natürlich eine beliebig umfangreiche Klasse verwendet werden. Das Spannende ist, dass Sie bei der

Auswahl aus der `ComboBox` ein komplettes Objekt in die Finger bekommen.

```
public class MeineKlasse {
    private final int key;
    private final String wert;

    public MeineKlasse(int key, String wert) {
        this.key = key;
        this.wert = wert;
    }

    public int getKey() {
        return key;
    }

    public String toString() {
        return wert;
    }
}
```

Listing 20.23: Die Klasse `MeineKlasse` für die `ObservableList`

Die Methode `toString` sollte die Repräsentation eines Objekts zurückgeben, so wie sie in der Liste der Einträge erscheinen soll.



Für den Umgang mit den Daten der Listen wird Wissen aus Kapitel 15 verwendet. Dort finden Sie auch Informationen über Generics.

Die Elemente können in einer `ObservableList` vom Typ `MeineKlasse` abgelegt werden, die in der Lage ist, beliebige Typen aufzunehmen. Anschließend wird sie Schritt für Schritt gefüllt. Die Liste wird dem Konstruktor der `ComboBox` übergeben.



Änderungen an der `ObservableList` während des Programms werden sofort auf die `ComboBox` durchschlagen. Die Liste ist also das Datenmodell für die `ComboBox`.

```
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxComboBox extends Application {

    @Override
    public void start(Stage hauptStage) {
        ObservableList<MeineKlasse> meineDaten
                = FXCollections.observableArrayList();
        meineDaten.add(new MeineKlasse(10, "Aktiv"));
        meineDaten.add(new MeineKlasse(20, "Faul"));
        meineDaten.add(new MeineKlasse(30, "Tot"));
        ComboBox<MeineKlasse> comboBox
                = new ComboBox<MeineKlasse>(meineDaten);
        comboBox.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                ComboBox<MeineKlasse> cb
                        = (ComboBox<MeineKlasse>) event.getSource();
                MeineKlasse mk = cb.getValue();
                int index = cb.getSelectionModel().getSelectedIndex();
                System.out.println("index: "+index+" Wert: "+mk);
            }
        });
        VBox pane = new VBox();
        pane.getChildren().add(comboBox);
        Scene scene = new Scene(pane, 250, 80);
        hauptStage.setScene(scene);
        hauptStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 20.24: ComboBox mit selbst gebauten Klassen

Die Vorgehensweise ermöglicht es, über die `ComboBox` direkt auf die Daten zuzugreifen, ohne dass es notwendig ist, über die Beschriftung der Einträge in einer weiteren Datenstruktur nach den eigentlichen Daten zu suchen.

Lang und länger: ListView

Die `ListView` zeigt offen ihre Elemente auf dem Bildschirm. Damit benötigt sie natürlich mehr Platz, sorgt aber gerade dadurch für eine gute Übersicht. Enthält sie mehr Elemente, als Platz auf dem Bildschirm ist, erzeugt sie automatisch einen Schiebebalken.

Wie schon bei der `ComboBox` verwendet auch die `ListView` für die Datenverwaltung eine `ObservableList`. Wir halten uns gar nicht erst mit String-Arrays auf, sondern verwenden gleich eine eigene Klasse für die Liste und greifen auf die bereits bekannte `MeineKlasse` aus Listing 20.23 zurück.

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxListView extends Application {

    @Override
    public void start(Stage hauptStage) {
        ObservableList<MeineKlasse> meineDaten
                = FXCollections.observableArrayList();
        meineDaten.add(new MeineKlasse(10, "Aktiv"));
        meineDaten.add(new MeineKlasse(20, "Faul"));
        meineDaten.add(new MeineKlasse(30, "Tot"));
        ListView<MeineKlasse> listView
                = new ListView<MeineKlasse>(meineDaten);
        listView.getSelectionModel().selectedItemProperty()
                .addListener(
                new ChangeListener<MeineKlasse>() {

            @Override
            public void changed(
                    ObservableValue<? extends MeineKlasse>
                            observable,
                    MeineKlasse alt, MeineKlasse neu) {
                System.out.println("Alt: " + alt
                                + " Neu: " + neu);
            }
        });
        VBox pane = new VBox();
        pane.getChildren().add(listView);
        Scene scene = new Scene(pane, 250, 80);
        hauptStage.setScene(scene);
        hauptStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 20.25: Eine `ListView` überwacht die Wechselereignisse.

Ereignisse der Liste

Die Events einer Liste betreffen das Anklicken der Elemente. Wenn Sie diese Änderungen in der Selektion überwachen wollen, setzen Sie einen `ChangeListener` ein, der je nach Typ den Index oder das Objekt liefert.

- Der Index des ausgewählten Elements wird über den Aufruf von `getSelectionModel().getSelectedIndex()` ermittelt.
- Das ausgewählte Objekt bestimmt der Aufruf von `getSelectionModel().getSelectedItem()`.

In Listing 20.25 wird das Objekt herangezogen. Entsprechend wird es in der Methode `changed` als Parameter übergeben. Die zweite Parametervariable `neu` enthält das neu gewählte Objekt, die dritte Parametervariable `alt` das Objekt, das zuvor gewählt war.

Wollen Sie nicht auf das Objekt zugreifen, sondern lieber auf den Index, also die Zeile, in der das Objekt steht, verwenden Sie die Variante in Listing 20.26.

```
listView.getSelectionModel().selectedIndexProperty()
        .addListener(new ChangeListener<Number>() {

    @Override
    public void changed(ObservableValue<? extends Number>
            observable, Number altIndex, Number neuIndex) {
        int alt = altIndex.intValue();
        int neu = neuIndex.intValue();
        System.out.println("Alt: " + alt + " Neu: " + neu);
    }
});
```

Listing 20.26: Wechsel meldet den Index.

Multiselektion

Eine `ListView` erlaubt es, mehrere Elemente gleichzeitig zu selektieren. Das Beispielprogramm in Listing 20.27 ermöglicht die Zusammenstellung besonders leckerer Speiseeiskombinationen.

Dazu wird das Selektionsmodell ermittelt und in den Modus MULTIPLE umgestellt. Anschließend wird ein `ChangeListener` dem Selektionsmodell hinzugefügt. Dessen Methode `changed` wird bei jeder Änderung der Auswahl aufgerufen. Im Beispiel ermittelt es jedes Mal die gewählten Elemente und fügt sie in der zweiten `ListView` ein.

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.control.SelectionMode;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class FxListViewMulti extends Application {

    @Override
    public void start(Stage stage) {
        ObservableList<String> eissorten =
                FXCollections.observableArrayList(
                    "Vanille", "Nuss", "Lakritz", "Zwiebel",
                    "Erdbeer", "Zitrone", "Anis", "Camembert");
        ListView<String> angebot = new ListView<>(eissorten);
        ListView<String> meinEis = new ListView<>();
        HBox pane = new HBox(angebot, meinEis);
        Scene scene = new Scene(pane);
        stage.setScene(scene);
        stage.show();
        // Ich darf mehrere Kugeln aussuchen
        angebot.getSelectionModel().setSelectionMode(
                SelectionMode.MULTIPLE);
        angebot.getSelectionModel().selectedIndexProperty()
                .addListener(new ChangeListener<Number>() {
            @Override
            public void changed(
                    ObservableValue<? extends Number> ob,
                    Number alt, Number neu) {
                // Setze alle ausgewählten in mein Eis
                meinEis.setItems(angebot.getSelectionModel()
                        .getSelectedItems());
            }
        });
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 20.27: Mehrfache Eisauswahl

Die Übernahme der Sorten zur Wunscheisliste erfolgt über den `ChangeListener`. Das Ergebnis können Sie in Abbildung 20.13 sehen.



Abbildung 20.13: Eisauswahl mit zwei `ListViews`

Lang und breit: TableView

Eine Liste besteht in den meisten Fällen aus mehreren Spalten. So ist die Adressliste mancher Fanclubs breiter als lang. Das Ergebnis sieht dann aus wie eine Tabellenkalkulation.

Eine `TableView` ist also eine `ListView` mit mehreren Spalten. Und natürlich führen diese zu mehr Aufwand. Eine Tabelle werden Sie einsetzen, wenn Sie eine Liste von einem Datenobjekt mit mehreren Attributen darstellen wollen. Wenn Sie Badman und Sabberman zum Dartspielen eingeladen haben, brauchen sie eine Highscore-Tabelle und die können Sie mit einem Java-Programm verwalten.

Zunächst definieren Sie die Klasse für die Spieler.

```
package fxtable;

public class Spieler {
    private String name = "";
    private int punkte = 0;

    public Spieler(String name, int punkte) {
        this.name = name;
        this.punkte = punkte;
    }

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getPunkte() {
        return this.punkte;
    }

    public void setPunkte(int punkte) {
        this.punkte = punkte;
    }
}
```

Listing 20.28: Die Datenklasse `Spieler` für die Highscore-Tabelle

TableView und ObservableList

Wie schon bei `ListView` und `ComboBox` setzen Sie eine `ObservableList` zur Verwaltung Ihrer Daten ein. Die Liste wird durch die Methode `setItems` zur Basis der `TableView` erkoren.

```
TableView<Spieler> table = new TableView<Spieler>();
ObservableList<Spieler> data
        = FXCollections.observableArrayList(
    new Spieler("Badman", 0),
    new Spieler("Sabberman", 0),
    new Spieler("Spinnerman", 0),
    new Spieler("Captain Allergica", 100),
    new Spieler("Booman", 0));
table.setItems(data);
```

Spaltendefinition

Die Highscore-Tabelle besteht aus zwei Spalten, dem Spielernamen und der Punktzahl. Eine Spalte wird durch die Klasse `TableColumn` realisiert.

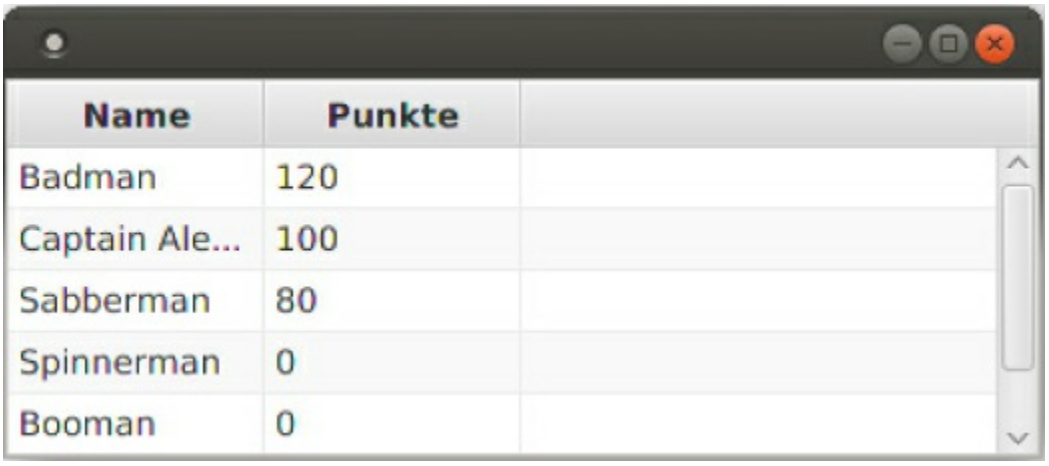


Abbildung 20.14: Die `TableView` zeigt den Highscore.

Die Spalten erhalten als Generics einmal die Klasse `Spieler`, aber zusätzlich den Typ der aufgezeichneten Spalte. Beim Namen ist dies `String`, bei den Punkten ist das `Integer`, weil `int` in den Generics nichts verloren hat.

```
TableColumn<Spieler, String> spalteName
        = new TableColumn<Spieler, String>("Name");
spalteName.setMinWidth(100); // Mindestbreite setzen
spalteName.setCellValueFactory(
        new PropertyValueFactory<Spieler, String>("name"));
// Die Spalte wird in die Tabelle gehängt.
table.getColumns().add(spalteName);
```

Mit der `CellValueFactory` wird dafür gesorgt, dass der Inhalt der Spalte auch dargestellt wird. Der Parameter des Konstruktors der Klasse `PropertyValueFactory` muss mit dem Namen der Getter-Methode übereinstimmen. Bei `name` muss es also eine Methode `getName` zum Ermitteln des Spalteninhalts geben.

Schließlich wird die Spalte durch den Aufruf von `getColumns().add` in die Tabelle

eingefügt.

Sortierter Highscore

Eine Highscore-Tabelle muss natürlich sortiert sein. Zum Glück kennt die `ObservableList` auch die Methode `sort`. Allerdings will sie dafür einen `Comparator` haben, der ihr erzählt, wie die Elemente der Klasse `Spieler` sortiert werden sollen. Das funktioniert also wie bei allen anderen Java-Collections, die Sie in Kapitel 15 kennengelernt haben.

```
Comparator<Spieler> scoreVergleicher
        = new Comparator<Spieler>() {

    public int compare(Spieler orig, Spieler vgl) {
        int oPkte = orig.getPunkte();
        int vPkte = vgl.getPunkte();
        if (oPkte < vPkte)
            return 1;
        if (oPkte > vPkte)
            return -1;
        return 0;
    }
};
// Nun kann sort mit scoreVergleicher aufgerufen werden:
data.sort(scoreVergleicher);
```

Listing 20.29: `Comparator` für die Spieler der `TableView`

Änderungen an der `ObservableList` schlagen sich sofort auf die `TableView` durch, weil die List das Datenmodell für die `TableView` darstellt.

Editierbare Tabelle

Sabberman ist etwas ungehalten, weil er meint, sein Name wäre falsch geschrieben. Badman ist sich nicht ganz sicher. Daraufhin kommen auch Ihnen Zweifel, ob Sie nicht bereits eine andere Schreibweise gesehen haben. Jedenfalls wäre es praktisch, wenn Sie in der Tabelle auf die Schnelle die Namen korrigieren könnten.

Die Inhalte einer `TableView` sind für den Anwender durch Doppelklick änderbar, wenn Sie folgende Schritte einleiten:

- ✔ Die Tabelle muss sich für editierbar erklären.
- ✔ Jede Spalte muss eine `CellFactory` anbieten.
- ✔ Die Spalte muss das `CellEditEvent` per `setOnEditCommit` fangen.
- ✔ Die Methode `handle` von `CellEditEvent` muss überschrieben werden.

```
spalteName.setCellFactory(
        TextFieldTableCell.forTableColumn());
spalteName.setOnEditCommit(
    new EventHandler
            <CellEditEvent<Spieler, String>>() {
        @Override
        public void handle(
                CellEditEvent<Spieler, String> t) {

            ((Spieler) t.getTableView().getItems().get(
                    t.getTablePosition().getRow())
                    ).setName(t.getNewValue());
        }
    }
);
// Die Tabelle muss zum Editieren freigeschaltet werden.
table.setEditable(true);
```

Listing 20.30: Einbau einer direkten Namensänderung

Etwas anders läuft es, wenn Sie die Punkte ändern wollen. Das war ein besonderer Wunsch von Badman. Ich kann mir allerdings nicht vorstellen, dass Badman schummeln wollte.

Was auch immer das Motiv war, bei Zahlen ist es erforderlich, explizit den Typ `Integer` anzugeben, damit die Tabelle von vornherein die Eingabe falscher Zeichen verhindert.

```
spaltePunkte.setCellFactory(
        TextFieldTableCell.<Spieler, Integer>
            forTableColumn(new IntegerStringConverter()));
spaltePunkte.setOnEditCommit(
    new EventHandler<CellEditEvent<Spieler, Integer> >() {
        @Override
        public void handle(
                CellEditEvent<Spieler, Integer> evt) {

            Integer neuWert = evt.getNewValue();
            evt.getTableView().getItems().get(
                evt.getTablePosition().getRow())
                .setPunkte(neuWert);
            // Punktzahl wurde geändert: neu sortieren!
            data.sort(scoreVergleicher);
        }
    }
);
```

Listing 20.31: Änderung von ganzzahligen Werten in der Tabelle

Alles in einem ..

Nun habe ich Ihnen lauter kleine Programmschnipsel vorgelegt. Spannend ist natürlich, wie das alles zusammenpasst. Damit Sie es nicht allein zusammenpuzzeln müssen, habe ich Ihnen das komplette Listing zusammengesetzt.

```
package fxtable;

import java.util.Comparator;

import javafx.application.Application;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableColumn.CellDataFeatures;
import javafx.scene.control.TableColumn.CellEditEvent;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.control.cell.TextFieldTableCell;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.FlowPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
import javafx.util.Callback;
import javafx.util.converter.IntegerStringConverter;

public class FxTable extends Application {

    @Override
    public void start(Stage stage) {
        TableView<Spieler> table = new TableView<Spieler>();
        ObservableList<Spieler> data =
                FXCollections.observableArrayList(
                new Spieler("Badman", 120),
                new Spieler("Sabberman", 80),
                new Spieler("Spinnerman", 0),
                new Spieler("Captain Alergica", 100),
                new Spieler("Booman", 0));

        table.setItems(data);
        table.setEditable(true);

        TableColumn<Spieler, String> spalteName
                = new TableColumn<Spieler, String>("Name");
        spalteName.setMinWidth(100);
        // Der String im Konstruktor von
        // PropertyValueFactory muss mit der get-Methode
```

```
// von Spieler korrespondieren
spalteName.setCellValueFactory(
        new PropertyValueFactory
                <Spieler, String>("name"));
spalteName.setCellFactory(
        TextFieldTableCell.forTableColumn());
// Spalte soll änderbar sein
spalteName.setOnEditCommit(new EventHandler
        <CellEditEvent<Spieler, String>>() {

    @Override
    public void handle(
            CellEditEvent<Spieler, String> t) {

        ((Spieler) t.getTableView().getItems().get(
                    t.getTablePosition().getRow())
                    ).setName(t.getNewValue());
    }
});

// Sortieren nach der Highscore-Liste
Comparator<Spieler> scoreVergleicher
        = new Comparator<Spieler>() {
    public int compare(Spieler orig, Spieler vgl) {
        int oPkte = orig.getPunkte();
        int vPkte = vgl.getPunkte();
        if (oPkte < vPkte)
            return 1;
        if (oPkte > vPkte)
            return -1;
        return 0;
    }
};
data.sort(scoreVergleicher);

TableColumn<Spieler, Integer> spaltePunkte
        = new TableColumn<>("Punkte");
spaltePunkte.setMinWidth(100);
spaltePunkte.setCellValueFactory(new Callback
        <CellDataFeatures<Spieler, Integer>,
        ObservableValue<Integer>>() {

    @Override
    public ObservableValue<Integer> call(
            CellDataFeatures<Spieler, Integer> c) {
        return new SimpleIntegerProperty(
            c.getValue().getPunkte()).asObject();
```

```
            }
        });
        spaltePunkte.setCellFactory(
                TextFieldTableCell.<Spieler, Integer>
                forTableColumn(
                        new IntegerStringConverter()));
        spaltePunkte.setOnEditCommit(
                new EventHandler
                    <CellEditEvent<Spieler, Integer> >() {

            @Override
            public void handle(
                    CellEditEvent<Spieler, Integer> evt) {

                Integer neuWert = evt.getNewValue();
                evt.getTableView().getItems().get(
                        evt.getTablePosition().getRow())
                        .setPunkte(neuWert);
                // Punktzahl geändert: neu sortieren!
                data.sort(scoreVergleicher);
            }
        });

        table.getColumns().add(spalteName);
        table.getColumns().add(spaltePunkte);

        Scene scene = new Scene(table);
        stage.setScene(scene);
        stage.show();
        stage.setWidth(450);
        stage.setHeight(550);
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 20.32: Highscore in der `TableView`

Grafische Programme brauchen Menüs

Das Menü ist eines der Elemente, die den grafischen Oberflächen zum Durchbruch verholfen haben. Jeder Anfänger findet im Menü ohne Einarbeitung die wichtigsten Funktionen, ohne sich durch herstellerspezifische Einstellmöglichkeiten zu wühlen. Und ein kurzer Spaziergang durch die Menüs verrät, was das Programm leistet. Wer auf ein Menü verzichtet, lässt den Anwender im Regen stehen.

Ein Menü wird aus drei Elementen zusammengesetzt:

- Die `MenuBar` ist die Querleiste, an der die Menüs angenagelt werden.
- Das `Menu` ist die Leiste, die beim Anklicken nach unten fällt, beispielsweise das Datei-Menü.
- Beim `MenuItem` handelt es sich eigentlich um einen Button, der an einem Menü klebt. Beispielsweise klebt der Menüpunkt für das Drucken standardmäßig am Datei-Menü. Dieser Menüpunkt löst das Ereignis aus, auf das das Programm reagiert.

Im ersten Schritt wird die Menüleiste erstellt und die Menüs werden angeklebt.

```
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
...
MenuBar menuBar = new MenuBar();
Menu menuDatei = new Menu("Datei");
Menu menuEdit = new Menu("Bearbeiten");
Menu menuView = new Menu("Ansicht");
menuBar.getMenus().addAll(menuDatei, menuEdit, menuView);
```

Listing 20.33: Erstellung der Menüleiste mit den Klappmenüs

Die Menüleiste muss in das Fenster eingebaut werden. Der Rahmen des Fensters ist meist sinnvollerweise ein `BorderPane`, in dessen Top-Bereich ein Menü geradezu ideal hineinpasst.

```
borderpane.setTop(menuBar);
```

Ohne Menüpunkte ist eine Menüleiste reichlich sinnlos. Sie werden erzeugt und anschließend in das passende `Menu` einzeln über die Methode `add` eingehängt. Alternativ können Sie mit der Methode `addAll` mehrere Menüpunkte auf einmal einbauen.

```
MenuItem mnOpen = new MenuItem("Öffnen");
MenuItem mnSave = new MenuItem("Speichern");
MenuItem mnEnde = new MenuItem("Ende");
menuDatei.getItems().addAll(mnOpen, mnSave, mnEnde);
```

Listing 20.34: Einbau der Menüpunkte

Zu guter Letzt soll natürlich auch irgendetwas passieren, wenn der Benutzer auf die Menüpunkte klickt, sonst hätten Sie sich den ganzen Zirkus ja auch sparen können. In dieser Hinsicht verhalten sich Menüpunkte wie Buttons und binden mit der Methode `setOnAction` einen Eventhandler an ihre Aktion.

```
mnEnde.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.exit(0);
    }
});
```

Listing 20.35: `EventHandler` aktivieren

Im nächsten Abschnitt werden Sie noch ein paar Standarddialoge kennenlernen, die wir dann in einem Beispielprogramm über das Menü aufrufen.

Standarddialoge

Java FX unterstützt den Programmierer mit einer Reihe von Standarddialogen. Einerseits ist deren Verwendung deutlich einfacher, als sich selbst einen Dialog zu bauen. Andererseits wird durch die Verwendung des Standarddialogs eine Einheitlichkeit zwischen den Anwendungen gewährleistet, von der der Benutzer profitiert.

Ein Nachrichtendialog: Alert

Wenn Sie eine kurze Meldung für den Anwender absetzen wollen, stellt Ihnen FX einen `Alert` zur Verfügung. Dieser Standarddialog enthält ein Feld für eine Nachricht und einen Button für die Bestätigung.

```
Alert alert = new Alert(AlertType.INFORMATION);
alert.setTitle("Titelzeile");
alert.setHeaderText("Dies ist eine Information!");
alert.setContentText("Ich habe da etwas zum Drücken!");
alert.showAndWait().ifPresent(rs -> {
    if (rs == ButtonType.OK) {
        System.out.println("Das war gut!");
    }
});
```

Listing 20.36: Eine Informationsbox

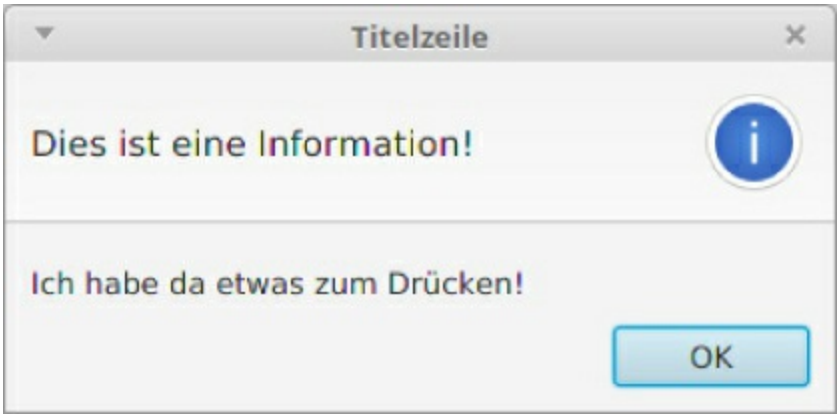


Abbildung 20.15: Nachrichtendienst

Als Parameter erhält `Alert` einen Typ, der besonders im Symbol des Dialogs sichtbar wird. Dazu gehören:

- ✔ `AlertType.INFORMATION`: Informations-i
- ✔ `AlertType.WARNING`: Warnschild
- ✔ `AlertType.ERROR`: Rotes Warnsymbol
- ✔ `AlertType.CONFIRMATION`: Fragezeichen

Weitere Methoden stehen Ihnen für die Gestaltung zur Verfügung.

- ✔ Mit `setTitle` bestimmen Sie die Titelzeile des Dialogs.
- ✔ Die Methode `setHeaderText` erlaubt einen Kopftext für den Dialog. Wenn Sie als Parameter `null` übergeben, wird kein Kopfteil angezeigt.
- ✔ Der Methode `setContentText` übergeben Sie den Inhalt der Nachricht, wegen der Sie den Dialog überhaupt ins Leben gerufen haben.
- ✔ Die Methode `showAndWait` startet den Dialog und wartet auf dessen Ende.

Eingabedialog

Für die Eingabe einer Zeile gibt es einen eigenen `TextInputDialog`, der ganz ähnlich bedient wird wie ein `Alert`. Dieser Dialog ist praktisch, wenn Sie auf die Schnelle eine Eingabe des Benutzers benötigen.

```
TextInputDialog dialog = new TextInputDialog("Vorgabe");
dialog.setTitle("Eingabedialog");
dialog.setHeaderText("Bitte eingeben!");
dialog.setContentText("Wie hei ß en Sie?");
Optional<String> result = dialog.showAndWait();
if (result.isPresent()) {
    System.out.println("Dein Name: " + result.get());
}
```

Listing 20.37: Einlesen einer Zeile

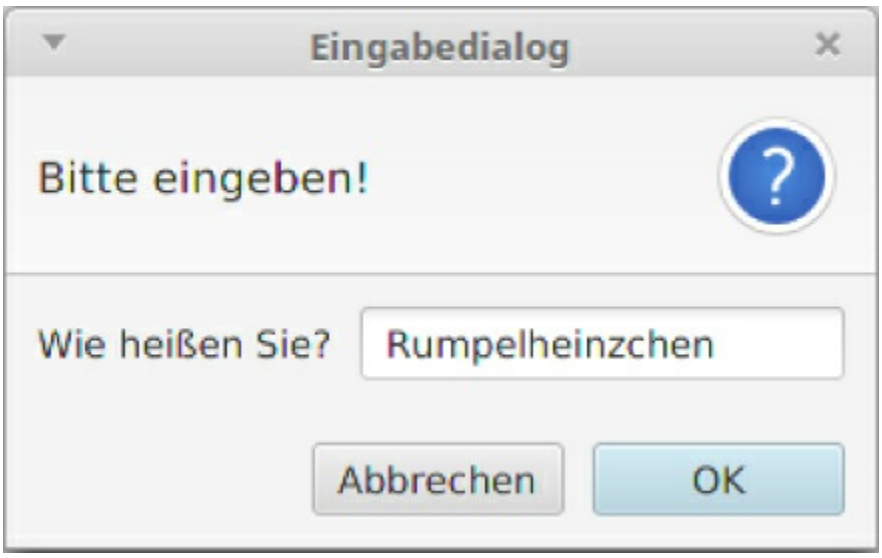


Abbildung 20.16: Eingabeaufforderung

Mehrere Buttons

Im folgenden Beispiel wird ein `Alert` so angepasst, dass er mehrere Buttons erhält. Das ist vor allem für Ja-Nein-Abfragen sehr praktisch.

```
Alert alert = new Alert(AlertType.CONFIRMATION);
alert.setTitle("Bestätigung");
alert.setHeaderText("Frei definierte Buttons");
alert.setContentText("War das klug?");
ButtonType btJa = new ButtonType("Ja");
ButtonType btNein = new ButtonType("Nein");
ButtonType btEgal = new ButtonType("Egal");
ButtonType btAbbruch = new ButtonType("Abbrechen",
        ButtonData.CANCEL_CLOSE);
alert.getButtonTypes().setAll(btJa, btNein, btEgal, btAbbruch);
Optional<ButtonType> result = alert.showAndWait();
if (result.get() == btJa) {
    System.out.print("Ja");
} else if (result.get() == btNein) {
    System.out.print("Nein");
} else if (result.get() == btEgal) {
    System.out.print("Egal");
} else {
    System.out.print("Weg hier!");
}
System.out.println(result.get().getText());
```

Listing 20.38: Mehrere Buttons

Sie sehen in Abbildung 20.17, dass `Alert` versucht, die Buttons gleich groß zu halten und so die breitere Beschriftung des Abbrechen-Buttons abschneidet. Dabei ist besonders lästig, dass die Altert-Dialoge selbst unter Linux nicht vom Benutzer vergrößerbar sind.

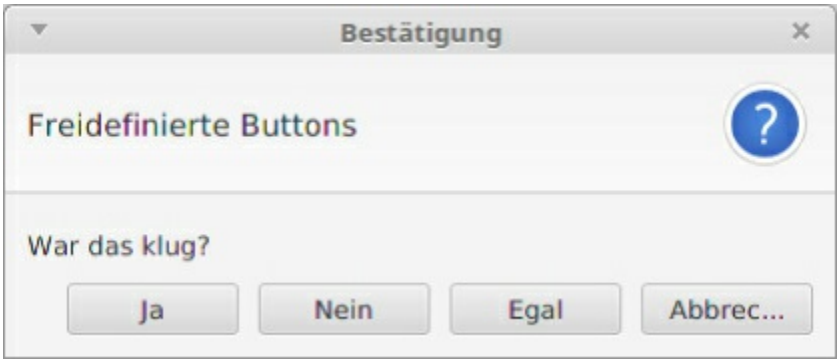


Abbildung 20.17: Mehrere Buttons für Ja-Nein-Fragen

Weitere Details über den Umgang mit Dialogen finden Sie unter der URL:
http://code.makery.ch/blog/javafx-dialogs-official

Die Dateiauswahlbox

Das Öffnen von Dateien ist insbesondere deshalb lästig, weil die verschiedenen Betriebssysteme sehr unterschiedliche Ansprüche an ihre Dateinamen haben. Da ist

das Trennzeichen mal ein Schrägstrich und mal ein Backslash, der immerhin die Fingerfertigkeit des Benutzers trainiert.

Idealerweise will man als Programmierer damit nichts zu tun haben. Und darum verwenden Sie am besten einen Standarddialog zum Öffnen der Datei. Der Benutzer freut sich, weil er sich im vertrauten Dialog durch die Verzeichnisse wühlen kann, und der Programmierer freut sich, weil der Dialog ein Objekt der Klasse `File` bekommt, die er einfach öffnen und lesen kann.

```
FileChooser fileChooser = new FileChooser();
File file = fileChooser.showOpenDialog(stage);
if (file != null) {
    System.out.println(file.getPath());
}
```

Listing 20.39: Dialog zum Öffnen einer Datei

Als Ergebnis zeigt das Programm den Dateinamen inklusive Pfad auf der Konsole an.

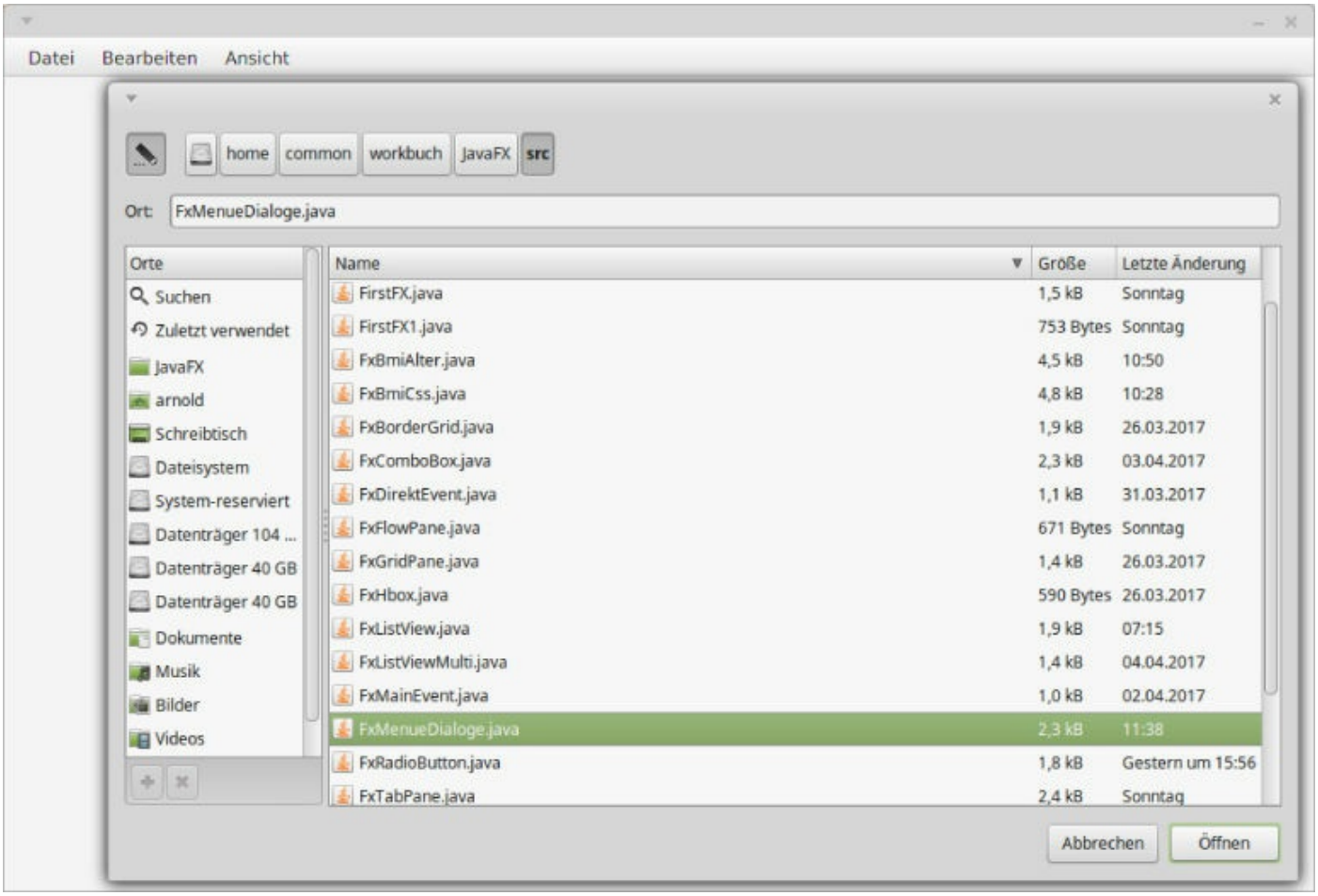


Abbildung 20.18: Dateiauswahldialog

Schreiben einer Datei

Wenn der Benutzer in eine Datei schreiben soll, ist der Dialog geringfügig anders, da die Datei ja nicht existieren muss, die er schreiben will.

```
FileChooser fileChooser = new FileChooser();
fileChooser.setTitle("Speichern der Datei");
File file = fileChooser.showSaveDialog(stage);
if (file != null) {
    System.out.println(file.getPath());
}
```

Listing 20.40: Dialog zum Schreiben einer Datei

Verzeichnis auswählen

Eine besondere Variante der Dateidialoge ist die Auswahl eines Verzeichnisses. Dafür gibt es dann auch eine eigene Klasse `DirectoryChooser`.

```
DirectoryChooser directoryChooser = new DirectoryChooser();
File selectedDirectory = directoryChooser.showDialog(stage);
if (selectedDirectory != null) {
    System.out.println(file.getPath());
}
```

Beispiel für ein Menü mit Standarddialogen

Wie versprochen, sehen Sie hier ein Programm mit Menü und einigen Standarddialogen am Stück. Da Menüs eine ganze Reihe von Menüpunkten enthalten, ist es sinnvoller, die ganze Klasse den `EventHandler` implementieren zu lassen, als diesen bei jedem Punkt einzeln abzufeiern.

```
import java.io.File;
import java.util.Optional;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.ButtonBar.ButtonData;
import javafx.scene.control.ButtonType;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
import javafx.scene.control.MenuItem;
import javafx.scene.control.TextInputDialog;
import javafx.scene.layout.BorderPane;
import javafx.stage.DirectoryChooser;
import javafx.stage.FileChooser;
import javafx.stage.Stage;

public class FxMenueDialoge extends Application
                implements EventHandler<ActionEvent> {
    public static void main(String[] args) {
        launch(args);
    }

    MenuItem mnOpen = new MenuItem("Öffnen");
    MenuItem mnSave = new MenuItem("Speichern");
    MenuItem mnDir = new MenuItem("Verzeichnis");
    MenuItem mnEnde = new MenuItem("Ende");
    MenuItem mnNote = new MenuItem("Meldung");
    MenuItem mnInput = new MenuItem("Eingabe");
    MenuItem mnJaNein = new MenuItem("Ja/Nein");
    Stage stage = null;
```

```
@Override
public void start(Stage primaryStage) throws Exception {
    stage = primaryStage;
    MenuBar menuBar = new MenuBar();
    Menu menuDatei = new Menu("Datei");
    Menu menuEdit = new Menu("Bearbeiten");
    Menu menuView = new Menu("Ansicht");
    menuBar.getMenus().addAll(
            menuDatei, menuEdit, menuView);
    BorderPane borderpane = new BorderPane();
    borderpane.setTop(menuBar);
    menuDatei.getItems().addAll(
            mnOpen, mnSave, mnDir, mnEnde);
    mnOpen.setOnAction(this);
    mnSave.setOnAction(this);
    mnDir.setOnAction(this);
    mnEnde.setOnAction(this);
    menuView.getItems().addAll(
            mnNote, mnInput, mnJaNein);
    mnJaNein.setOnAction(this);
    mnNote.setOnAction(this);
    mnInput.setOnAction(this);
    Scene scene = new Scene(borderpane, 400, 300);
    stage.setScene(scene);
    stage.show();
}

@Override
public void handle(ActionEvent event) {
    if (event.getSource() == mnEnde) {
        System.exit(0);
    } else if (event.getSource() == mnOpen) {
        FileChooser fileChooser = new FileChooser();
        File file = fileChooser.showOpenDialog(stage);
        if (file != null) {
            System.out.println(file.getPath());
        }
    } else if (event.getSource() == mnSave) {
        FileChooser fileChooser = new FileChooser();
        fileChooser.setTitle("Speichern der Datei");
        File file = fileChooser.showSaveDialog(stage);
        if (file != null) {
            System.out.println(file.getPath());
        }
    } else if (event.getSource() == mnDir) {
        DirectoryChooser dirChooser
                = new DirectoryChooser();
```

```
        File selDir = dirChooser.showDialog(stage);
        if (selDir != null) {
            System.out.println(selDir.getPath());
        }
    } else if (event.getSource() == mnNote) {
        Alert alert = new Alert(AlertType.INFORMATION);
        alert.setTitle("Titelzeile");
        alert.setHeaderText(
                "Dies ist eine Information!");
        alert.setContentText(
                "Ich habe da etwas zum Drücken!");
        alert.showAndWait().ifPresent(rs -> {
            if (rs == ButtonType.OK) {
                System.out.println("Pressed OK.");
            }
        });
    } else if (event.getSource() == mnInput) {
        TextInputDialog dialog
                = new TextInputDialog("Vorgabe");
        dialog.setTitle("Eingabedialog");
        dialog.setHeaderText("Bitte eingeben!");
        dialog.setContentText("Wie hei ß en Sie?");
        Optional<String> result = dialog.showAndWait();
        if (result.isPresent()) {
            System.out.println("Dein Name: "
                + result.get());
        }
    } else if (event.getSource() == mnJaNein) {
        Alert alert = new Alert(AlertType.CONFIRMATION);
        alert.setTitle("Bestätigung");
        alert.setHeaderText("Freidefinierte Buttons");
        alert.setContentText("War das klug?");
        ButtonType btJa = new ButtonType("Ja");
        ButtonType btNein = new ButtonType("Nein");
        ButtonType btEgal = new ButtonType("Egal");
        ButtonType btAbbruch = new ButtonType(
                "Abbrechen", ButtonData.CANCEL_CLOSE);
        alert.getButtonTypes().setAll(
                btJa, btNein, btEgal, btAbbruch);
        Optional<ButtonType> result
                = alert.showAndWait();
        if (result.get() == btJa) {
            System.out.print("Ja");
        } else if (result.get() == btNein) {
            System.out.print("Nein");
        } else if (result.get() == btEgal) {
            System.out.print("Egal");
```

```
            } else {
                System.out.print("Weg hier!");
            }
            System.out.println(result.get().getText());
        }
    }
}
```

Listing 20.41: Ein Menü für den Aufruf verschiedener Standarddialoge

Schickmachen mit CSS

Java FX unterstützt CSS (Cascading Style Sheets). CSS kennen Sie vielleicht aus dem Bereich der Webseitengestaltung. Sie ermöglichen es, die Farben nach der Corporate Identity der Firma durchgehend zu gestalten. So richtig praktisch sind die CSS, wenn die Corporate Identity der Firma durch eine feindliche Übernahme geändert wird. Dann muss nur an einer Stelle geändert werden und schon stehen die Webseiten in den Farben des neuen Besitzers da.

Mit CSS ist es nun auch möglich, das Aussehen der Software in dieser Weise zu gestalten. So können alle Buttons einer Anwendung einen neongrünen Hintergrund bekommen, aber nur der Notfall-Button mit Rot hinterlegt werden. Die Formate bauen aufeinander auf. Ein allgemeines Format wird durch ein spezielleres überschrieben.

Ein Style kann einem Element direkt mit der Methode `setStyle` zugewiesen werden. Der folgende Button bekommt einen gelben Hintergrund.

```
button.setStyle("-fx-background-color: yellow;");
```

CSS-Datei anlegen und einbinden

Sehr viel sinnvoller ist es allerdings, die Styles in einer Datei zu sammeln. Diese Datei legen Sie sinnvollerweise im Projekt ab. Sie erzeugen eine CSS-Datei mit den folgenden Schritten:

1. Den Ordner *src* des Projekts mit der rechten Maustaste anklicken.
2. Aus dem Kontextmenü New|Others wählen.
3. Im Dialog unter Web den Punkt CSS File wählen. Mit Next geht es weiter.
4. Den Dateinamen, beispielsweise *styles.css* angeben. Finish.

Diese Datei wird auf die folgende Weise in die `Scene` der Applikation eingebunden. Als Parameter wird der Dateiname angegeben, der gegebenenfalls um einen Pfad ergänzt werden muss.

```
scene.getStylesheets().add("styles.css");
```

Alternativ können Sie die CSS-Datei aus den Ressourcen des Projekts laden. Der Zugriff darauf ist ein wenig umständlicher.

```
scene.getStylesheets().add(getClass()
        .getResource("styles.css").toExternalForm());
```

Im einfachen Fall, dass die CSS-Datei so erzeugt wurde, wie oben gezeigt, verhalten sich beide Aufrufe identisch.

Stile definieren

In der Datei können mehrere Stile für unterschiedliche Ziele festgelegt werden. Sie können die FX-Kontrollelemente jeweils anders gestalten. Dazu verwenden Sie den Namen der Kontrollelemente wie `button` oder `label`.

Das folgende Beispiel sorgt dafür, dass alle Buttons eine grüne Schrift und einen roten Hintergrund erhalten. Labels erhalten dafür einen violetten Hintergrund. Ob Sie ein solches Programm noch verkaufen können, wage ich allerdings zu bezweifeln.

```
.button {
    -fx-text-fill: green;
    -fx-background-color:red;
}
.label {
    -fx-background-color: #FF00FF;
}
```

Falls Ihr Button nun doch einen gelben Hintergrund hat, liegt das vermutlich daran, dass die direkte Einfärbung des Buttons noch aktiv ist. Wenn Sie einzelne Objekte mit einem Stil versehen, überschreibt das den Standard, den Sie in der Datei angeben.

Anstatt die Kontrollelemente pauschal zu färben, können Sie auch Namen vergeben, die dann von verschiedenen Elementen direkt verwendet werden. Die folgende Definition legt den Stil `grausam` fest, der anschließend noch mit dem Kontrollelement in Verbindung gebracht werden muss.

```
#grausam {
    -fx-background-color: gray;
}
```

Dazu versehen Sie Elemente mit einer ID. In der letzten Definition wird für die Kennung `.grausam` ein grauer Hintergrund spezifiziert. Die folgende Anweisung sorgt dafür, dass dem Element `textfeld` dieser Stil zugewiesen wird.

```
textfeld.setId("grausam");
```

In der folgenden Übersicht werden die verbreitetsten Style Sheets von Java FX mit typischen Inhalten aufgeführt:

- ✔ Schriften:
  - `-fx-font-size`: 20px;
  - `-fx-font-family`: “Arial”;
  - `-fx-font-weight`: bold;
  - `-fx-text-fill`: orange;

    Schriften erhalten auf diese Weise Farbe.
- ✔ Hintergrund
  - `-fx-background-color`: lightblue;

    Der Hintergrund erhält eine babyblaue Farbe. Natürlich ist auch rosa möglich.
  - `-fx-background-color`: linear-gradient(gray, darkgray);

    Die Farbe geht linear von Grau nach Dunkelgrau über.
  - `-fx-background-radius`: 15 15 15 15;

    Der Hintergrund wird in den Ecken abgerundet. Links oben beginnend werden im Uhrzeigersinn die Eckradien festgelegt.
  - `-fx-background-insets`: 2 2 2 2;

    Sie können mit einem Rahmen ein wenig Luft schaffen. Oben beginnend geben die Werte im Uhrzeigersinn an, wie viel Platz zum Rand gelassen werden soll.
  - `-fx-background-image`: url(“file:/home/ich/Bilder/bild.png”);

    Sie können den Hintergrund auch mit Bildern schmücken.
  - `-fx-background-size`: 270, 160;

    Große Bilder können Sie mit dieser Anweisung auf die gewünschte Größe anpassen.
  - `-fx-background-repeat`: no-repeat;

    Ist das Bild zu klein, wird der Hintergrund mit dem Bild gekachelt. Diese Anweisung verhindert eine solche Wiederholung.
  - `-fx-effect`: dropshadow(two-pass-box , darkblue, 10, 0.0 , 4, 5);

    Nun wird ein Schatten angelegt, der natürlich nur sichtbar wird, wenn am Rand etwas Platz ist.
- ✔ Randerscheinungen
  - `-fx-padding`: 15;

Dieser Wert hält die Kinder auf Abstand. Das funktioniert leider nur beim Programmieren.

- `-fx-spacing`: 10;

  Definiert einen Rand innerhalb des Rahmens.

Weiterführende Links

Unter dieser URL zählt Oracle alle denkbaren Styles auf:

`docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html`

Diese URL führt zu einer schönen Einführung in das Thema CSS unter Java FX:

`www.guigarage.com/2016/02/javafx-and-css`

Komponieren in FXML im SceneBuilder

Statt die Oberfläche klassisch in einem Programm zusammenzubauen, können Sie diese auch mit der XML-Abart FXML beschreiben und dann per Annotationen nutzen. Aber warum sollten Sie, nachdem Sie Java so prima beherrschen, auch noch FXML lernen?

Die Antwort lautet, dass Sie XML gar nicht lernen sollen. Das Interessante an diesem Ansatz ist, dass XML sehr viel einfacher von Programmgeneratoren erzeugt werden kann als Java-Code. Nicht Sie sollen FXML erstellen, sondern eine Anwendung, mit der Sie Ihre Oberfläche grafisch zusammenklicken.



Bevor Sie einen bösen Brief an den Verlag aufsetzen, warum dieser Tunichtgut von Autor Sie erst durch die Panes und dann durch die Kontrollelemente jagt, um Ihnen zum Schluss zu erzählen, dass es dafür Generatoren gibt, lesen Sie erst das Kapitel zu Ende. Sie werden feststellen, dass auch beim Einsatz des Generators Kenntnisse im Umgang mit Panes und Elementen extrem hilfreich sind.

Der SceneBuilder

Für Java FX stellt Oracle den SceneBuilder zur Verfügung. Mit diesem Programm lassen sich Eingabedialoge aus mehreren Elementen zusammenklicken. Dieses Programm läuft in einem separaten Fenster zur Entwicklungsumgebung, kann aber FXML-Dateien direkt in einem Projekt bearbeiten.

Installation und Einrichtung unter Eclipse

Der SceneBuilder wird von Oracle selbst lediglich als Source zur Verfügung gestellt. Die folgende Website bietet aber auch ein fertiges Installationspaket für

alle gängigen Plattformen an:

http://gluonhq.com/products/scene-builder

Windows

Für Windows erhalten Sie das Angebot, einen Installer für 32- oder 64-Bit-Systeme herunterzuladen. Es handelt sich um eine EXE-Datei, die Sie nach dem Download ausführen können. Die Installation erfolgt in diesen Schritten:

- ✔ Bestätigen Sie die Lizenz.
- ✔ Standardmäßig findet die Installation im Verzeichnis *AppData/Local/SceneBuilder* des Benutzers statt. Das Verzeichnis *AppData* ist tückischerweise standardmäßig unsichtbar. Am besten notieren Sie sich den Installationspfad. Sie werden ihn brauchen.
- ✔ Das Installationsprogramm fragt nach einer E-Mail-Adresse und bietet an, up to date zu sein. Den Haken enfernen Sie, sofern Sie nicht gern Werbung bekommen.
- ✔ Das Programm startet sofort. Es soll aber später aus Eclipse heraus gestartet werden. Also schließen Sie den SceneBuilder.

Linux

Für Linux finden Sie auf der Seite sowohl ein *rpm*- als auch ein *deb*-Paket, womit die wichtigsten Linux-Systeme abgedeckt sind. Für Ubuntu laden Sie das *deb*-Paket und installieren es mit dem folgenden Befehl:

```
sudo dpkg -i scenebuilder-8.3.0.deb
```

Leider installiert das Paket den SceneBuilder im Anwendungsmenü unter der Rubrik Sonstiges anstatt unter Entwicklung. Das kann man aber schnell über den Menüeditor der jeweiligen Oberfläche geradeziehen.

Viel wichtiger ist aber, dass Eclipse den Ort der Installation findet. Das Paket wählt standardmäßig den Pfad */opt/SceneBuilder*. Dort finden Sie die ausführbare Datei unter dem Namen *SceneBuilder*.

Integration des SceneBuilders in Eclipse

Der SceneBuilder wird erst dann wirklich nützlich, wenn er in die IDE eingebunden ist. Eclipse ist in der Lage, die FXML-Dateien mit dem SceneBuilder zu bearbeiten. Allerdings muss Eclipse dazu wissen, wo sich der SceneBuilder befindet.

1. Dazu klicken Sie zunächst in das Menü Window|Preferences.
2. Im darauf erscheinenden Dialog klicken Sie in der linken Spalte den Punkt JavaFX an.
3. Auf der rechten Seite können Sie nun das SceneBuilder Executable über den Button Browse auswählen.

Eine FXML-Datei erzeugen

Bevor Sie den SceneBuilder einsetzen können, müssen Sie natürlich zunächst eine FXML-Datei haben. Eclipse hilft Ihnen dabei, eine solche FXML-Datei anzulegen.

1. Zunächst legen Sie ein Java-FX-Projekt an. Aus dem Menü wählen Sie File|New|Project. Im Dialog wählen Sie JavaFX|JavaFX Project.
2. Klicken Sie auf Next. Nun können Sie den Projektnamen eingeben und mit Finish das Projekt erzeugen.
3. Sie klicken das Projekt mit der rechten Maustaste an und wählen aus dem Kontextmenü File|New|Other.
4. Es erscheint ein Dialog. Darin wählen Sie JavaFX und darunter New FXML Document aus.
5. Geben Sie einen Dateinamen ein. Eclipse wird ihn um die Endung *.fxml* ergänzen. Mit Finish schließen Sie die Erzeugung ab.

Es entsteht der Rahmen einer FXML-Datei.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.AnchorPane?>

<AnchorPane xmlns:fx="http://javafx.com/fxml/1">
    <!-- TODO Add Nodes -->
</AnchorPane>
```

Daraus können Sie nun Ihren Dialog erzeugen. Dazu benötigen Sie gute FXML-Kenntnisse oder den SceneBuilder.

FXML-Datei mit dem SceneBuilder verarbeiten

Wenn Sie nach diesen Vorbereitungen eine FXML-Datei im Projekt mit der rechten Maustaste anklicken, können Sie über den Punkt Open With SceneBuilder den SceneBuilder aufrufen.

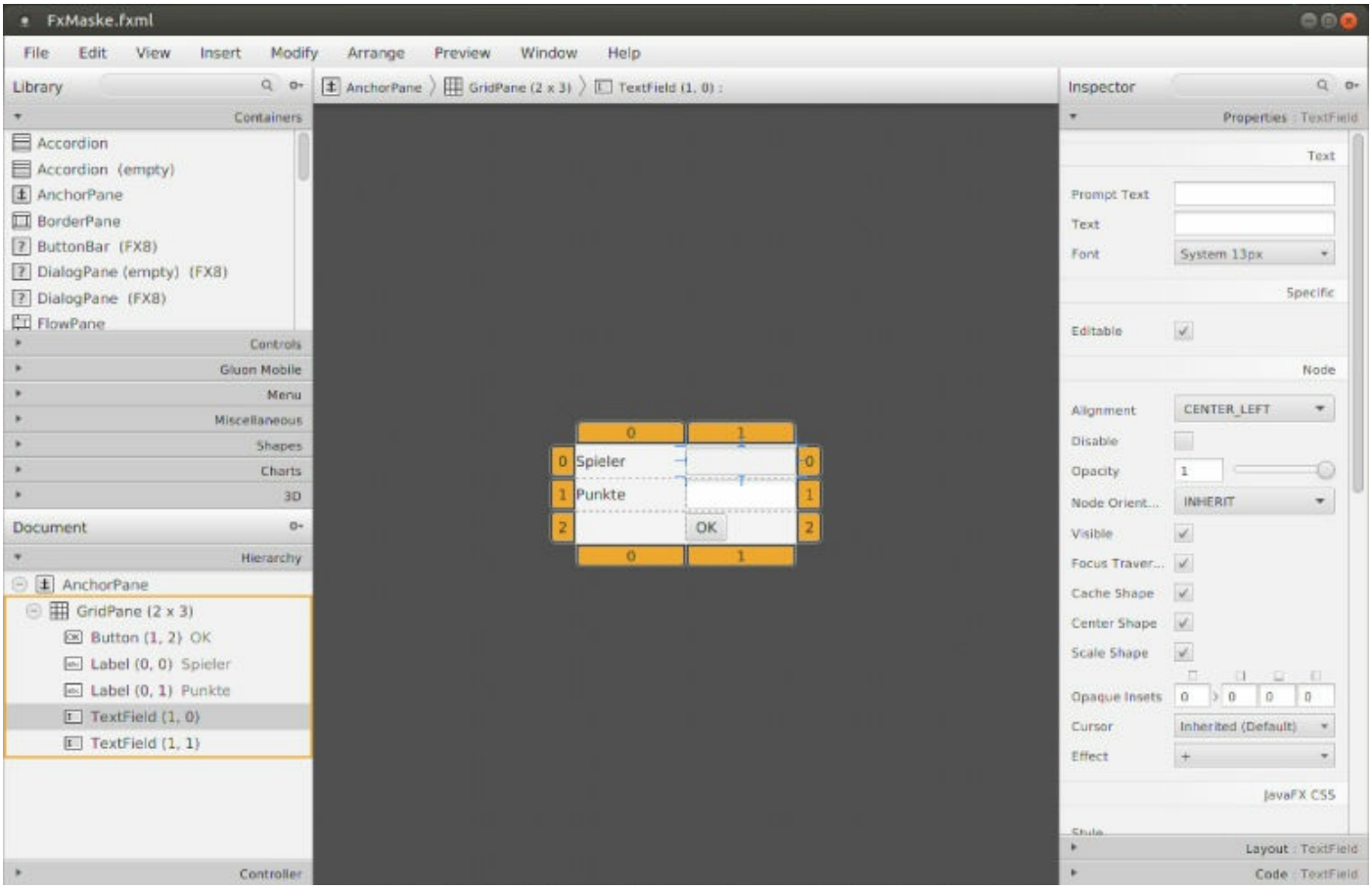


Abbildung 20.19: Der SceneBuilder mit einem Dialogfenster

Er übernimmt die FXML-Datei aus dem Eclipse-Projekt und stellt sie grafisch dar.

- In der linken Spalte zeigt der SceneBuilder Ressourcen wie Container oder Kontrollelemente, aus denen Sie durch Mausziehen auswählen und im Mittelfeld Ihr Fenster zusammenklicken können.
- In der rechten Spalte erscheint die jeweils angewählte Ressource. Sie können dort deren Attribute einstellen. Übrigens können Sie hier auch festlegen, welche CSS-Datei verwendet werden soll.

Nach dem Sichern und Verlassen des SceneBuilders sieht Ihre FXML-Datei deutlich anders aus. Das Beispiel in Listing 20.42 habe ich ein wenig zurechtgestutzt, damit es besser ins Buch passt. Ihre FXML-Datei sieht vermutlich eh anders aus.

Bei der Definition des `GridPanes` sehen Sie, dass `layoutX` und `layoutY` negativ sind. Dadurch befindet sich der Rahmen außerhalb des Fensters. Am besten, man löscht beide Variablen. Auch ohne die genaue Syntax von FXML zu kennen, erkennen Sie, dass hier in einem `GridPane` zwei `TextField`- und ein `Button`-Element angeordnet sind, die durch zwei `Label` dekoriert werden. Die Einträge `fx:controller`, `fx:id` und `onAction` sind nicht durch den SceneBuilder entstanden, sondern von Hand eingefügt.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.ColumnConstraints?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.layout.RowConstraints?>

<AnchorPane xmlns:fx="http://javafx.com/fxml/1"
          xmlns="http://javafx.com/javafx/8.0.111"
   fx:controller="application.SpielerController">
   <children>
      <GridPane layoutX="-88.0" layoutY="-106.0">
        <columnConstraints>
          <ColumnConstraints hgrow="SOMETIMES" prefWidth="100.0" />
          <ColumnConstraints hgrow="SOMETIMES" prefWidth="100.0" />
        </columnConstraints>
        <rowConstraints>
          <RowConstraints prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints prefHeight="30.0" vgrow="SOMETIMES" />
        </rowConstraints>
         <children>
            <Button text="OK" GridPane.columnIndex="1"
                 GridPane.rowIndex="2"
                 fx:id="btOk" onAction="#klickOk"/>

<label text="Spieler" />
<label text="Punkte" GridPane.rowIndex="1" />
            <TextField fx:id="tfSpieler" GridPane.columnIndex="1" />
            <TextField fx:id="tfPunkte" GridPane.columnIndex="1"
                 GridPane.rowIndex="1" />
         </children>
      </GridPane>
   </children>
</AnchorPane>
```

Listing 20.42: FXML von SceneBuilder erzeugt

Einbinden der FXML-Datei in Java

Die FXML-Datei startet nicht von allein. Dazu benötigen Sie eine Hauptklasse unter Java.

1. Klicken Sie mit der rechten Maustaste auf das Projekt.
2. Wählen Sie New|Other aus.

3. In dem folgenden Dialog suchen Sie JavaFX. Dort befindet sich ein Unterpunkt namens Classes.
4. Darin gibt es einen Punkt JavaFX Main Class.
5. Wenn Sie diesen anklicken, erscheint ein Dialog, weil diese Klasse gern einen Namen haben möchte. `Main` wäre nicht schlecht.

Sie erhalten ein Gerippe für ein Java-FX-Programm und füllen die Methode `start` mit den Zeilen in Listing 20.43.

```
package application;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class Main extends Application {
    @Override
    public void start(Stage primaryStage) {
        try {
            BorderPane root = new BorderPane();
            Parent p = FXMLLoader.load(getClass().getResource(
                    "/FxMaske.fxml"));
            Scene scene = new Scene(p);

            scene.getStylesheets().add(getClass().getResource(
                    "application.css").toExternalForm());
            primaryStage.setScene(scene);
            primaryStage.show();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 20.43: Hauptprogramm für ein FXML-Fenster

Wenn Sie das Programm als Java Application starten, sollte das Fenster korrekt

dargestellt werden, verhält sich aber ausgesprochen passiv. Man wünscht sich ein paar Aktivitäten. Beispielsweise könnte eine Methode aufgerufen werden, wenn auf den Button geklickt wird. Diese könnte dann die Eingabefelder auslesen. Dazu müssen in der FXML-Datei ein paar Kleinigkeiten eingetragen werden.

- Im äußeren Pane wird hinterlegt, wie die Klasse heißt, die das Fenster steuert, also der Controller.

```
fx:controller="application.SpielerController"
```

- Die Elemente müssen mit Variablennamen im Java-Programm in Verbindung gebracht werden. Dazu dienen `fx:id`-Einträge.

```
<TextField fx:id="tfSpieler" GridPane.columnIndex="1"...
<TextField fx:id="tfPunkte" GridPane.columnIndex="1" ...
```

- Der Button hat zusätzlich mit `onAction` die Möglichkeit, auf die Eventmethode zu verweisen.

```
<Button text="OK" GridPane.columnIndex="1"
    GridPane.rowIndex="2"
    fx:id="btOk" onAction="#klickOk"/>
```

Die Aktivitäten werden von der Controller-Klasse übernommen. Sie wird als normale Klasse für das Projekt erzeugt und heißt im Beispiel `SpielerController`. Sie sehen, wie die Annotation `@FXML` ein Verweis auf die `fx:id`-Elemente in der FXML-Datei darstellt.

```
package application;

import java.net.URL;
import java.util.ResourceBundle;

import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.TextField;
```

```
public class SpielerController implements Initializable {

    @FXML
    private TextField tfSpieler;

    @FXML
    private TextField tfPunkte;

    private SpielerModel model;

    @Override
    public void initialize(URL location, ResourceBundle resources) {
        model = new SpielerModel();
    }

    public void klickOk() {
        model.setName(tfSpieler.getText());
    }
}
```

Die eigentlichen Daten ermittelt die Controller-Klasse aus der Modell-Klasse. Sie heißt hier `SpielerModel`. Diese Klasse verwaltet die Daten für den Dialog.

```
package application;

public class SpielerModel {
    private String spieler = "";
    private int punkte = 0;

    public void setName(String name) {
        System.out.println("Name="+name);
        this.spieler = name;
    }
}
```

Vereinfachend ist hier nur die Methode `setName` ausgeführt. Sie übernimmt den String in das lokale Attribut und sorgt für eine Ausgabe des Spielernamens auf der Konsole, damit Sie sehen können, dass die Methode auch aufgerufen wird. Wenn Sie das Programm starten, einen Spieler eintragen und auf den Button OK klicken, sollte die Ausgabe zu sehen sein.

Falls es nicht klappt, haben Sie eine schwere Zeit vor sich, nämlich die der Fehlersuche. Leider kann Ihnen Eclipse dabei insbesondere in den FXML-Dateien nicht so gut helfen wie im Java-Quellcode. Hinzu kommt, dass viele Verbindungen zwischen View und Controller nur auf Bezeichnern in String-Literalen beruhen, die vom Compiler nicht geprüft werden.

Aufgaben

Der Wahlzettel zur Gemeinderatswahl

Programmieren Sie doch einen virtuellen Gemeinderatswahlzettel auf der Basis der `CheckBox`. Erfinden Sie zwölf Kandidatennamen, von denen aber nur maximal vier gewählt werden dürfen. Wird ein fünfter gewählt, muss dieser neue Haken wieder zurückgesetzt werden. Welcher Haken zuletzt gesetzt wurde, erfahren Sie aus der Methode `event.getSource`. Anschließend können Sie diesen mit `setSelected` zurücksetzen.

```
CheckBox chk = (CheckBox) event.getSource();
chk.setSelected(false);
```

Die Wahl des Landtags

Nachdem Sie die Gemeinderatswahl schon so erfolgreich mit virtuellen Wahlzetteln ausgestattet haben, können Sie sich nun an Landtagswahlen heranmachen. Der Unterschied liegt darin, dass Sie nur eine Partei wählen können, und damit ist der `RadioButton` die Idealbesetzung.

Kapitel 21

Bermuda im FX-Kleid

IN DIESEM KAPITEL

Bermuda in der FX-Version

Szenenaufzug mit Canvas

Das aus Kapitel 8 bekannte Spiel macht unter grafischen Oberflächen erst richtig Spaß. In Kapitel 12 wurden bereits die Klassen definiert, die Sie hier übernehmen können. Das gilt insbesondere für die folgenden Klassen

- Die Klasse `Schiff` finden Sie in Listing 12.4.
- Die Klasse `Position` finden Sie in Listing 12.3.
- Die Klasse `Flotte` finden Sie in Listing 12.7.
- Die Klasse `Spielfeld` finden Sie in Listing 12.8.

Für die FX-Version von Bermuda benötigen Sie natürlich ein FX-Projekt, dessen Erstellung in Kapitel 20 bereits beschrieben wurde.

Für die Realisierung als FX-Version benötigen wir zwei neue Klassen.

- `Main` ist das Hauptprogramm und baut die `Scene` auf.
- `BermudaCanvas` realisiert das grafische Spielfeld, nimmt die Maustastenklicks entgegen und reagiert darauf.

Das Hauptprogramm

Das Rahmenprogramm bietet keine großen Überraschungen. Die überschriebene Methode `handle` baut ein `BorderPane` auf, dessen Zentrum durch das Spielfeld im `BermudaCanvas` realisiert wird. Im Top-Bereich steht ein `Label`, das den Spielzustand darstellen soll. Da es von `BermudaCanvas` gefüllt wird, wird es als Parameter dem Konstruktor übergeben.

Es gibt noch einen Button im Bottom-Bereich, mit dem der Spieler jederzeit ein neues Spiel starten kann. Die Verarbeitung des Button-Klicks erfolgt auf der Stelle in der Klasse `Main`, löst aber die Methode `neuesSpiel` aus der Klasse `BermudaCanvas` aus.

```
package application;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class Main extends Application {

    @Override
    public void start(Stage primaryStage) {
        Label lblMeldung = new Label("Bermuda");
        BermudaCanvas pane = new BermudaCanvas(lblMeldung);
        Button btNeuSpiel = new Button("Neues Spiel");
        btNeuSpiel.setOnAction(
                    new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                pane.neuesSpiel();
            }
        });
        try {
            BorderPane root = new BorderPane();
            // pane = new BermudaCanvas(lblMeldung);
            root.setCenter(pane);
            root.setTop(lblMeldung);
            root.setBottom(btNeuSpiel);
            Scene scene = new Scene(root);
            primaryStage.setScene(scene);
            primaryStage.show();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 21.1: Das Hauptprogramm `Main`

Das Canvas für die Grafikdarstellung

Für die grafische Darstellung wird ein `Canvas` erweitert. Ein `Canvas` hat gegenüber anderen Nodes den Vorteil, dass Sie darauf auf Pixelebene zeichnen können. Das ist bei Spielen wie Bermuda sehr hilfreich.

Innerhalb eines `Canvas` kann mit der Methode `getGraphicsContext2D` der zugehörige Grafikkontext geholt werden und schon kann die Malerei beginnen.

Bermuda verwendet für jedes Symbol ein `Image`, das jeweils aus einer Bilddatei geladen wird. Ich habe meine Bilder mit 32 Pixeln Höhe und Breite gezeichnet und räume ein, dass sie sicher nicht jedermanns Geschmack darstellen. Aber Sie können ja selbst den virtuellen Pinsel schwingen. Das Programm lädt nacheinander die Images ein und bestimmt die maximale Höhe und Breite und kachelt nach dem Maximum das `Canvas`. Aus der Größe ermittelt es auch die benötigte Größe des Canvas. Die Darstellung übernimmt die Methode `drawImage`.

In der Methode `handle` werden die Mausereignisse gefangen. Die Mausposition wird durch die Kachelgröße geteilt und schon haben Sie eine Koordinate, mit der Sie die Klasse `Spielfeld` belästigen können.

```
package application;

import javafx.event.EventHandler;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.Label;
import javafx.scene.image.Image;
import javafx.scene.input.MouseEvent;

public class BermudaCanvas extends Canvas implements
 EventHandler<MouseEvent>

    private Spielfeld spiel;
    private Image leer, ziffer[];
    private Image schiff;
    int distanz;
    private Label lblMeldung;

    private void zeigePos(char c, Position pos) {
        GraphicsContext gc = this.getGraphicsContext2D();
        Image img = null;
        if (c >= '0' && c <= '8') {
            img = ziffer[c - '0'];
        } else if (c == Flotte.SCHIFF) {
            img = schiff;
        } else {
            img = leer;
        }
        gc.drawImage(img, pos.getX() * distanz, pos.getY() * distanz);
    }

    private void zeigeFeld() {
        for (int i = 0; i < Spielfeld.MAXX; i++) {
            for (int j = 0; j < Spielfeld.MAXY; j++) {
                Position pos = new Position(i, j);
                char c = spiel.get(pos);
                zeigePos(c, pos);
            }
        }
    }

    private void ermittleDistanz(Image img) {
        int distImg = (int) img.getWidth();
        if (distImg>distanz) {
            distanz = distImg;
        }
    }
```

```
public BermudaCanvas(Label frameLabel) {
    lblMeldung = frameLabel;
    this.spiel = new Spielfeld();
    this.setOnMousePressed(this);
    leer = new Image("image/leer.png");
    schiff = new Image("image/schiff.png");
    ermittleDistanz(leer);
    ermittleDistanz(schiff);
    ziffer = new Image[9];
    for (int i = 0; i < 9; i++) {
        ziffer[i] = new Image("image/mark" + i + ".png");
        ermittleDistanz(ziffer[i]);
    }
    this.setWidth((Spielfeld.MAXX) * distanz);
    this.setHeight((Spielfeld.MAXY) * distanz);
    zeigeFeld();
}

@Override
public void handle(MouseEvent event) {
    int x = (int) event.getX();
    int y = (int) event.getY();
    Position pos = new Position(x/distanz, y/distanz);
    if (spiel.istImSpielfeld(pos)) {
        char ergebnis = spiel.suche(pos);

            zeigePos(ergebnis, pos);
            spiel.set(pos, ergebnis);
            if (spiel.alleSchiffeGefunden()) {
                lblMeldung.setText("Spiel gewonnen!");
            }
        }
    }

    public void neuesSpiel() {
        lblMeldung.setText("Bermuda");
        spiel = new Spielfeld();
        zeigeFeld();
    }
}
```

Listing 21.2: BermudaCanvas: Die grafische Umsetzung

Wenn alles fertig ist, sollte Ihr Programm wie in Abbildung 21.1 aussehen.

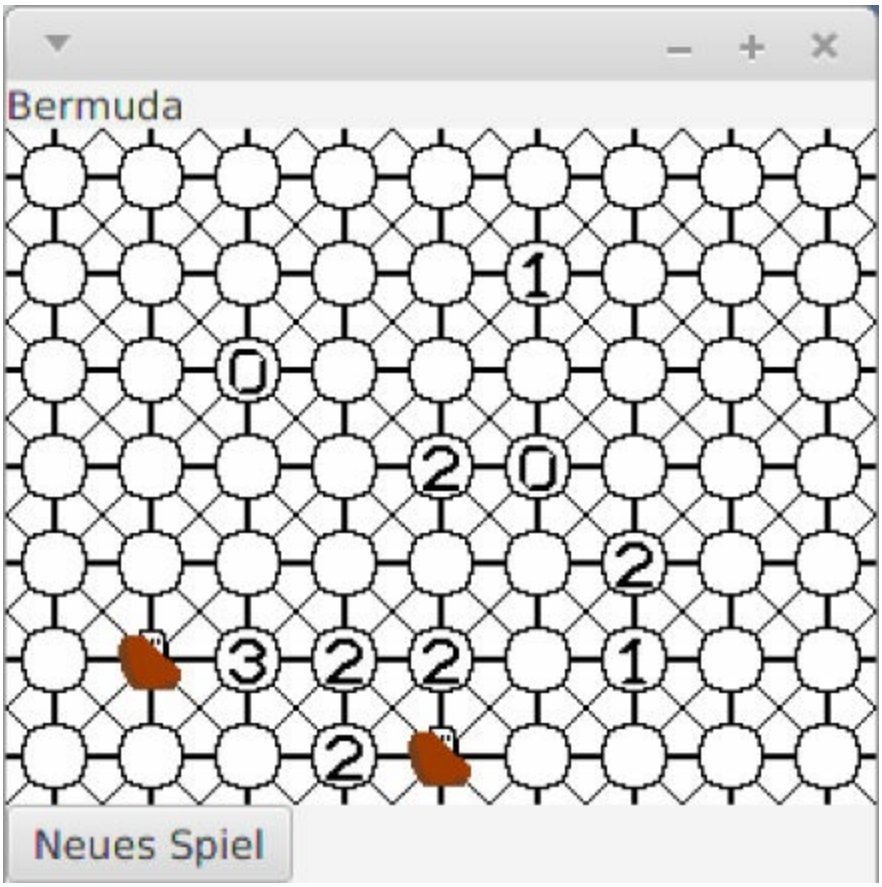


Abbildung 21.1: Bermuda mit Java FX

Kapitel 22

Android

IN DIESEM KAPITEL

Mit Android Studio Apps entwickeln

Android-Entwicklung in Java, nur etwas anders

Bermuda auf dem Smartphone

Würden die Menschen ihren Ehepartner genauso oft und liebevoll streicheln wie ihre Smartphones, würde die Scheidungsrate vermutlich rapide sinken. Jedenfalls sind Android-Handys hipp und die Programmierung erfolgt in Java. Grund genug, einmal kurz hinzusehen.

Alles ist ein wenig anders

Das Programmieren für diese putzigen Geräte läuft etwas anders ab als die Entwicklung von Java-Programmen für die großen Computer. Die schlechte Nachricht: Es ist nicht praktikabel, auf dem Handy die Quelltexte zu editieren und dort den Compiler zu starten. Das wird also auf einem PC passieren müssen. Die gute Nachricht: Aus genau diesem Grund müssen Sie Ihren Computer nicht wegwerfen.

Dalvik Virtual Machine

Allerdings laufen die generierten Programme nicht auf dem PC. Wieso das? Ich hatte doch behauptet, dass Java – dank der virtuellen Maschine – einmal geschrieben überall läuft. Das ist ja prinzipiell richtig und auch so geplant gewesen. Aber dann kam diese doofe Realität und warf alles durcheinander. Android leistet sich den Luxus einer anderen virtuellen Maschine namens Dalvik Virtual Machine (DVM). Dalvik ist ein Ort in Island, der seine Berühmtheit vor allem der Tatsache verdankt, dass die Dalvik Virtual Machine nach ihm benannt wurde. Gerüchte besagen, der Entwickler der DVM habe Verwandte dort gehabt. Wäre ich also mit ihm verwandt, würde die DVM vielleicht heute Norgaardholz Virtual Machine heißen.

Übrigens ist auch Oracle von dieser Inkompatibilität nicht begeistert und streitet sich mit Google deswegen vor Gericht. Ich weiß gar nicht, ob es schon zu einer Entscheidung gekommen ist. Es ist auch eigentlich egal, wie es ausgehen wird. Es

läuft vermutlich darauf hinaus, dass eine der beiden Firmen reicher wird und ansonsten alles bleibt wie bisher.

Aber der Bytecode der Java Virtual Machine (JVM) kann mittels eines kleinen Tools namens dx in den DVM-Code übersetzt werden. Dateien mit diesem Code werden mit der Endung *.dex* gekennzeichnet. Dieses Tool wiederum kann in die Entwicklungsumgebung integriert werden. Auf diese Weise steht dem Java-Programmierer ein Großteil der Möglichkeiten der Java-Umgebung zur Verfügung. Allerdings gibt es Unterschiede bei der Programmierung der grafischen Oberfläche und in einigen anderen Bereichen.



Android verwendet eine andere virtuelle Maschine als Java, nämlich die Dalvik Virtual Machine (DVM). Ein Großteil der Java-Bibliotheken funktioniert auch unter Android.

Der DVM-Code ist ohne weiteres Zutun nicht auf Ihrem Entwicklungsrechner lauffähig. Darum können Sie ihn dort nicht testen. Dazu verwendet die Entwicklungsumgebung einen *Emulator*, der ein Android-Gerät nachempfindet. Sie können alternativ ein echtes Android-Gerät über USB verbinden und Ihren Code darauf testen.

Arbeitsumgebung schaffen

Die Arbeitsumgebung zur Erstellung von Android-Apps benötigt eine IDE. Lange Zeit wurde Eclipse in Kombination mit einem Plug-in dafür eingesetzt. Inzwischen wird die IDE Android Studio dafür verwendet. Sie können sie über die URL `http://developer.android.com` beziehen, die als zentrale Anlaufstelle für Android-Entwickler gilt. Das Android-Software-Development-Kit (SDK) ist ein wichtiger Bestandteil der Entwicklungsumgebung und wird automatisch mit dem Android Studio installiert. Das SDK enthält auch die API (Application Programming Interface). Eine API beschreibt die Bibliotheksaufrufe. Bei neueren Android-Versionen kommen immer wieder neue Funktionen hinzu.

Android Studio einrichten

Unter der URL `http://developer.android.com/studio` finden Sie die Möglichkeit, Android Studio für Ihre Betriebssystemversion herunterzuladen. Je nach dem Betriebssystem, mit dem Sie arbeiten, gibt es da ein paar Unterschiede.

Windows

Wenn Sie die URL `http://developer.android.com/studio` mit einem Windows-PC öffnen, bietet sie den Download einer ausführbaren Version. Für diejenigen unter uns, die nicht lesen können, gibt es noch ein Video, das im Grunde nur zeigt, wie man die heruntergeladene Datei startet und den Anweisungen des Assistenten

folgt. Dessen Anweisungen sind nicht weiter schwierig. Im Grunde müssen alle Seiten bestätigt werden und durch den Button Next kommt man jeweils eine Seite weiter.

Das Android Studio wird unter Programme einsortiert, wohingegen das SDK im Heimatverzeichnis des Benutzers unter dem Verzeichnis *AppData/Local/Android/sdk* abgelegt wird. Schließlich lädt das Installationsprogramm noch Teile des SDKs herunter und dekomprimiert sie.

Wenn das Android Studio startet, schlägt es vor, das SDK zu validieren. Auch wenn Sie nun hofften, erste Schritte probieren zu dürfen, werden Sie sich darauf wohl einlassen müssen. Nach mehreren Next-Buttons und einem Finish beginnt der Download und das Entpacken.

Nach Abschluss der erfolgreichen Aktualisierung Ihres SDKs wissen Sie, was Sie von der Bandbreite Ihrer Internetanbindung zu halten haben. Wenn Sie das nächste Mal Finish bestätigen, öffnet sich Android Studio.

Mac

Wenn Sie die URL `http://developer.android.com/studio` mit einem Mac öffnen, wird Ihnen eine DMG-Datei angeboten. Nach dem Download befindet sich die Datei wie eine Festplatte auf dem Desktop. Das zugehörige Fenster öffnet sich, wie in Abbildung 22.1 zu sehen.



Abbildung 22.1: Nach dem Download

Intuitiv tun Sie nun die folgenden Schritte:

1. Sie verschieben das Android-Studio-Symbol auf den mit Applications beschrifteten Ordner.
2. Dann klicken Sie auf das Symbol Android Studio doppelt.
3. Im nächsten Schritt weist Sie der Mac auf die Gefahren von Programmen hin, die aus dem Internet geladen wurden. Wenn Sie sich diesen Gefahren tapfer stellen wollen, können Sie das Programm nun starten.
4. Das Programm fragt, ob Sie die Einstellungen eines alten Android Studios übernehmen wollen oder ob dies eine erstmalige Installation ist.
5. Der Assistent zeigt, was er zu tun gedenkt. Next.
6. Sie können statt der Standard-Installation eine Custom-Variante wählen, wenn Sie dafür besondere Gründe haben. Next.
7. Einen Überblick über das, was nun installiert wird, beenden Sie mit Finish.
8. Nun lädt und installiert Android Studio haufenweise Dateien aus dem Internet, bis die Leitung glüht. Dann erfragt Android Studio das Administratorkennwort. Next.
9. Nach der Installation beenden Sie den Assistenten mit Finish. Automatisch startet Android Studio und will von Ihnen wissen, ob Sie ein neues Projekt anlegen oder ein existierendes übernehmen wollen. Es verhält sich also ab jetzt genauso wie auf den anderen Plattformen.

Linux

Wenn Sie die URL `http://developer.android.com/studio` mit einem Linux-Rechner öffnen, wird Ihnen eine ZIP-Datei angeboten, die Sie nach dem Download bitte selbst entpacken mögen. In der Beschreibung steht, wie Sie das Android Studio einrichten.

1. Suchen Sie sich ein Terminal (unter Ubuntu mit Strg + Alt + T). Sie finden in der Regel mindestens eines in Ihrem Anwendungsmenü.
2. Werden Sie Administrator. Je nach System:
   - `su -`
   - `sudo -s`
3. `cd /opt`

   Wechseln Sie in das Zielverzeichnis.
4. `unzip /home/user/Downloads/android-studio-ide*.zip`

   Ersetzen Sie »user« in dem Pfad durch Ihren Benutzernamen. Falls Sie mehrere Versionen von Android Studio heruntergeladen haben, müssen Sie statt des

Sterns die genaue Versionsnummer angeben.

5. Öffnen Sie ein weiteres Terminal als normaler Anwender.
6. Geben Sie dort den Befehl für den ersten Start von Android Studio ein:
   `/opt/android-studio/bin/studio.sh`

   Sie können sich für diesen Befehl auch einen Starter auf dem Desktop oder im Anwendungsmenü bauen. Der Vorgang unterscheidet sich danach, welchen Desktop Sie unter Linux benutzen.

Der erste Start von Android Studio

Android Studio sieht auf allen Plattformen ähnlich aus. Beim ersten Start erscheint das kleine Fenster in Abbildung 22.2, um Ihnen die Wahl zu lassen, ob Sie ein neues oder ein altes Projekt öffnen wollen.

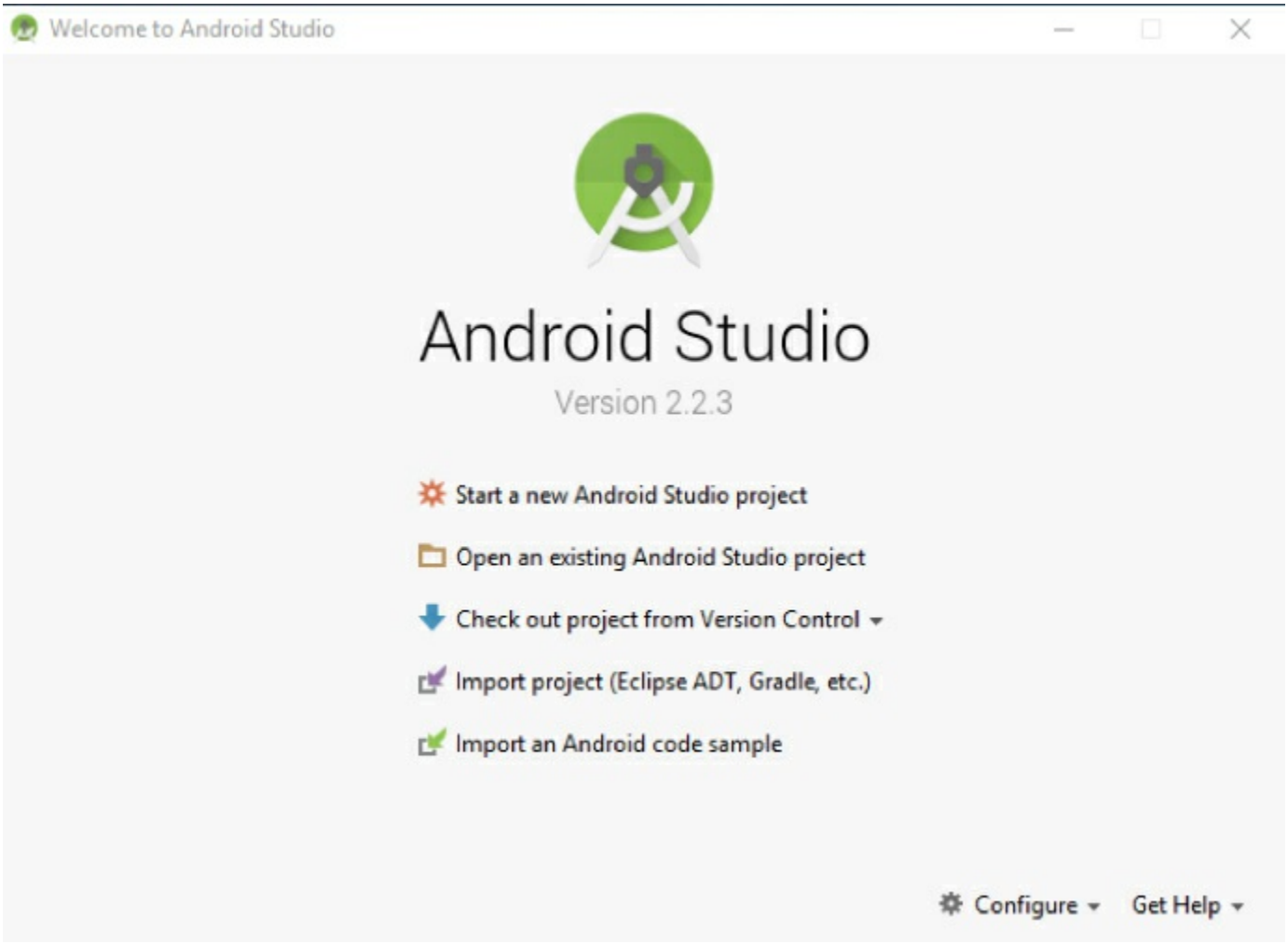


Abbildung 22.2: Android Studio nach der Installation

Die Entscheidung fällt nicht schwer, wenn Sie zum ersten Mal Android Studio öffnen. Woher sollten Sie auch ein existierendes Projekt haben?

Das erste eigene Projekt

Erstellen Sie Ihre erste eigene App. »Hello World« kann jeder! Warum also kleckern? Nein, ich schaue in den Spiegel und schon ist es klar: Wir brauchen ein

Programm, um die Üppigkeit zu zügeln. Gegen die Fettleibigkeit ist zwar keine App gewachsen, aber immerhin kann die App mit dem BMI (Body-Mass-Index) auf das eigene Versagen hinweisen. Und das ist ja auch schön. Wohl wissend, was der Plan ist, können wir nun weiterfahren:

Verify Settings

Unter dem Titel Verify Settings stellt Android Studio klar, dass es gedenkt, noch reichlich Dateien herunterzuladen. Da es sich um deutlich mehr als 1 GByte handelt, haben Sie nun Zeit, einen Kaffee aufzusetzen. Falls Sie in einer von Politikern unbewohnten Umgebung leben, dauert es vermutlich über Nacht oder über das Wochenende.

Start a new Android Studio project

Aus einem Dialog können Sie nun ein neues Projekt starten oder ein existierendes Projekt öffnen. Vermutlich haben Sie noch kein Projekt, also starten Sie neu. Sobald Sie das neue Projekt anwählen, erscheint ein Dialog, der einiges wissen will:

- Den Namen des Projekts: Es schlägt `My Application` vor. Sie könnten mit `Bmi` einen besseren Namen vorschlagen.
- Den Domainnamen der Firma: Hier wird der Benutzername mit `example.com` vorgeschlagen. Traditionsgemäß verwendet man den eigenen Domänennamen, da der weltweit eindeutig ist. Meine Apps könnten beispielsweise im Package `de.willemer` stehen.
- Beides zusammen ergibt einen Pfad, der in der unteren Zeile zu sehen ist.

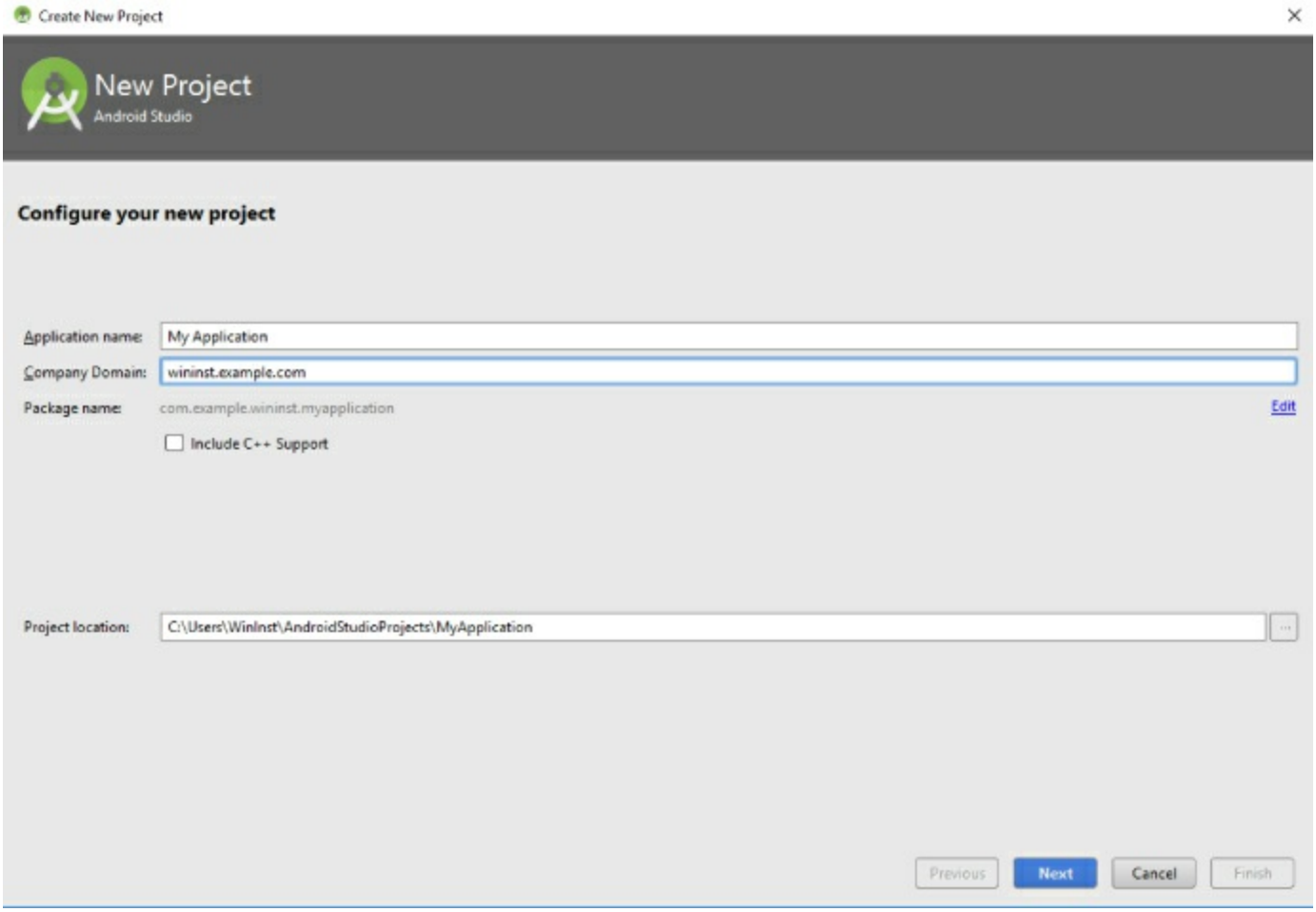


Abbildung 22.3: Neues Projekt im Android Studio

Target Android Devices

Nach dem nächsten Next will das Android Studio mehr dazu erfahren, auf welchen Zielgeräten die App laufen soll. Sie sehen in Abbildung 22.4, dass die APIs durchnummeriert sind. Die API 15 bezieht sich auf die Android-Version 4.0.3. Je höher Sie das minimale SDK setzen, desto mehr Elemente neuerer Versionen können Sie einsetzen. Allerdings verlieren Sie damit auch Kunden, die ein etwas älteres Modell verwenden.

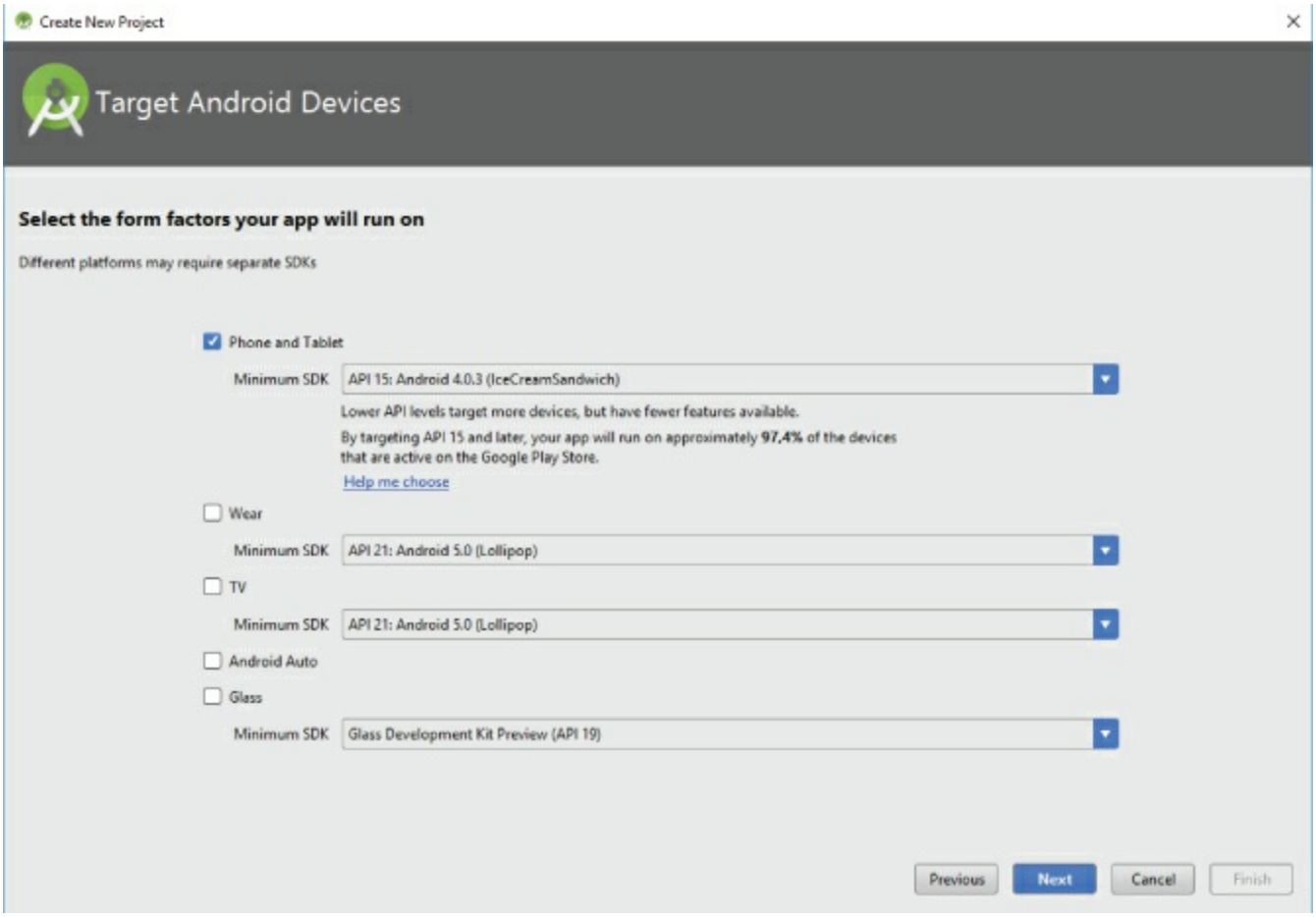


Abbildung 22.4: Zielgeräte für die App auswählen

Sie können noch einige zusätzliche Android-Versionen wie Wear, TV oder Things anwählen, sofern Sie für solche Geräte entwickeln wollen. Mit dem Button Next erreichen Sie den nächsten Dialog.

Activity

Der Assistent bittet nun um die Auswahl der Activity. Diese Activity beschreibt den ersten Bildschirm, mit dem sich Ihre App dem Benutzer präsentiert. Das Thema Activity wird uns noch beschäftigen. Zunächst ist die Vorgabe keine schlechte Wahl.

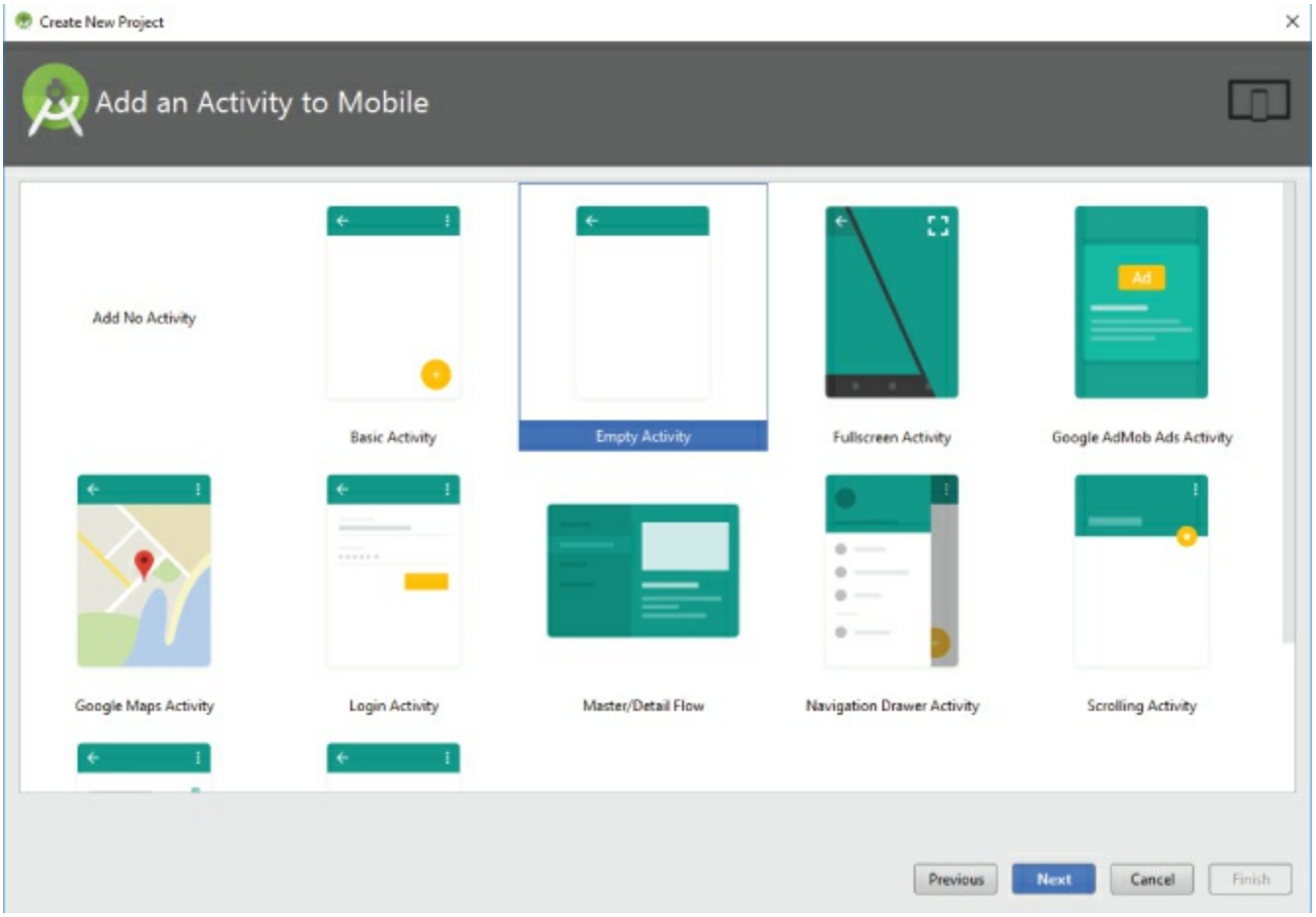


Abbildung 22.5: Activity für die App auswählen

Configure

Im letzten Bildschirm will das Android Studio noch den Namen der ersten Activity wissen und schlägt `MainActivity` vor. Auch die anderen Vorgaben können Sie übernehmen, sofern Sie nicht einen anderen Namen bevorzugen. Mit Next geht es weiter.

Auf dem nächsten Fenster werden einige Komponenten installiert und der Button Finish freigegeben.

Android Studio feiert Eröffnung

Nun erarbeitet sich das Android Studio ein Projekt und bemüht bei Windows noch die Firewall. Diese muss die Kommunikation zulassen. Android Studio startet immer mit einem Tipp, den Sie mit Close verlassen können. Immer noch arbeitet Android Studio im Hintergrund. Sie sehen das an der Statuszeile unten im Fenster. Warten Sie in Ruhe ab, bis das »Indexing« und die anderen Meldungen verschwunden sind. Erst dann sehen Sie das neue Projekt in der Darstellung wie in Abbildung 22.6.

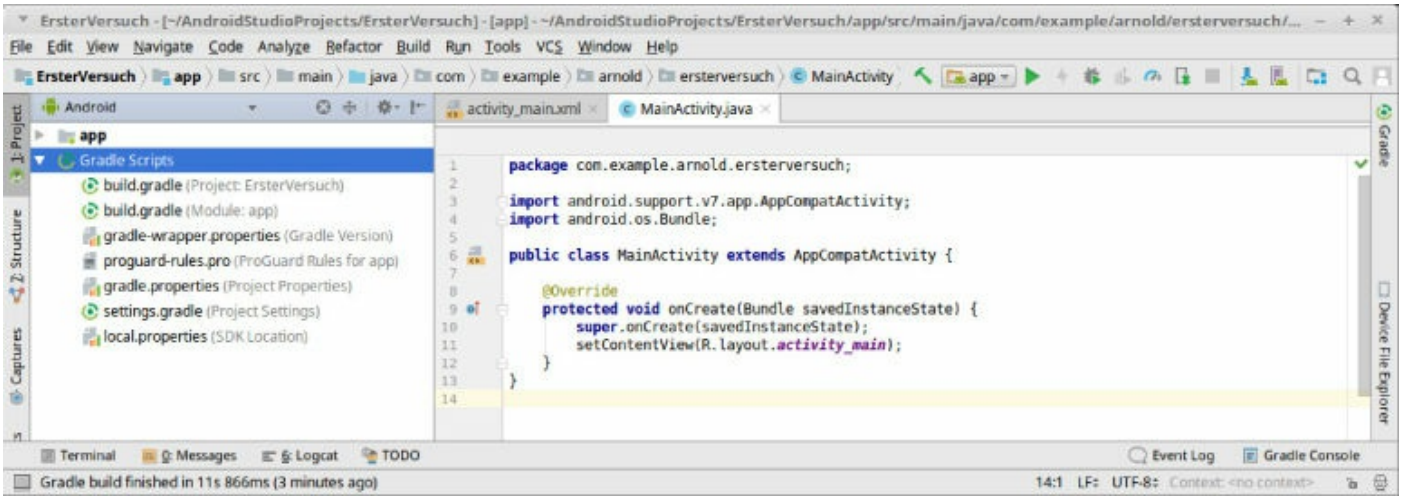


Abbildung 22.6: Android Studio mit dem ersten Projekt

Es kann sein, dass Ihre Anwendung meldet, dass einige Plattformen fehlen und diese erst heruntergeladen werden müssen. Spätestens jetzt werden Sie erkennen, dass für die Android-Entwicklung eine gute Internet-Anbindung große Vorteile bietet.

Eine BMI-Activity als Beispiel

Das Android Studio hat eine fertige Hello-World-Anwendung geschaffen, die Sie natürlich auch ausprobieren können. Sie können aber auch gleich in die Vollen gehen und eine kleine App schreiben. Wenn Sie mich sehen könnten, würde es Sie sofort dazu drängen, eine Body-Mass-Index-Anwendung zu schreiben. Die würde mir die Schamesröte ins Gesicht treiben, damit wenigstens mein Teint sehenswert ist.

Unter der URL `www.bmi-rechner.net` finden Sie den theoretischen Hintergrund für die BMI-Berechnung. Nehmen Sie das Thema übrigens nicht allzu ernst. Es gibt viele kritische Stimmen, die den BMI für Unsinn halten. Ob zu den kritischen Stimmen ausschließlich massige Körper mit hohem BMI gehören, kann ich allerdings auch nicht sagen. Da der BMI aber reichlich Eingabeparameter bietet, eignet er sich ideal, um das Programmieren mit grafischen Oberflächen zu üben. Um für unser BMI-Projekt eine schicke Eingabe zu gestalten, benötigen wir ein paar Kontrollelemente und deren Parameter:

- Die Größe in cm: Dazu eignet sich ein einfaches Eingabefeld. Sie können Android sogar beschwatzen, ein spezielles Zahleneingabefeld zu verwenden. Dann kann der Benutzer nur Zahlen eintippen.
- Das Gewicht in kg: Auch hier eignet sich das numerische Eingabefeld.
- Ob Sie Männchen oder Weibchen sind, können Sie mit einem Radiobutton am besten realisieren.
- Auch das Alter können Sie in 10-Jahres-Stufen bestimmen. Anders ausgedrückt, müssen Sie nicht zwingend abnehmen, um Ihr Idealgewicht zu

bekommen. Es reicht auch, wenn Sie nur alt genug werden. Ab dem Rentenalter hilft es aber nichts mehr. Dann müssen Sie vielleicht weniger essen oder zum Äußersten greifen und Sport treiben.

- ✔ Sie möchten vielleicht mit einem Button die Berechnung auslösen.
- ✔ Natürlich wird noch eine Ausgabefläche für das frustrierende Ergebnis benötigt.
- ✔ Ein paar Beschriftungen hier und da erhöhen vielleicht die Benutzerfreundlichkeit.

All diese Elemente können Sie über einen interaktiven Designer mit der Maus zusammenstellen.

1. Öffnen Sie das Verzeichnis *app*, indem Sie auf das Dreieck klicken.
2. Darunter finden Sie das Verzeichnis *res*.
3. Darin liegt *layout*.
4. Und zu guter Letzt finden Sie die Datei *activity_main.xml*.
5. Wenn Sie diese doppelt anklicken, stellt sich Ihr Android Studio vermutlich ganz ähnlich wie in Abbildung 22.7 dar.

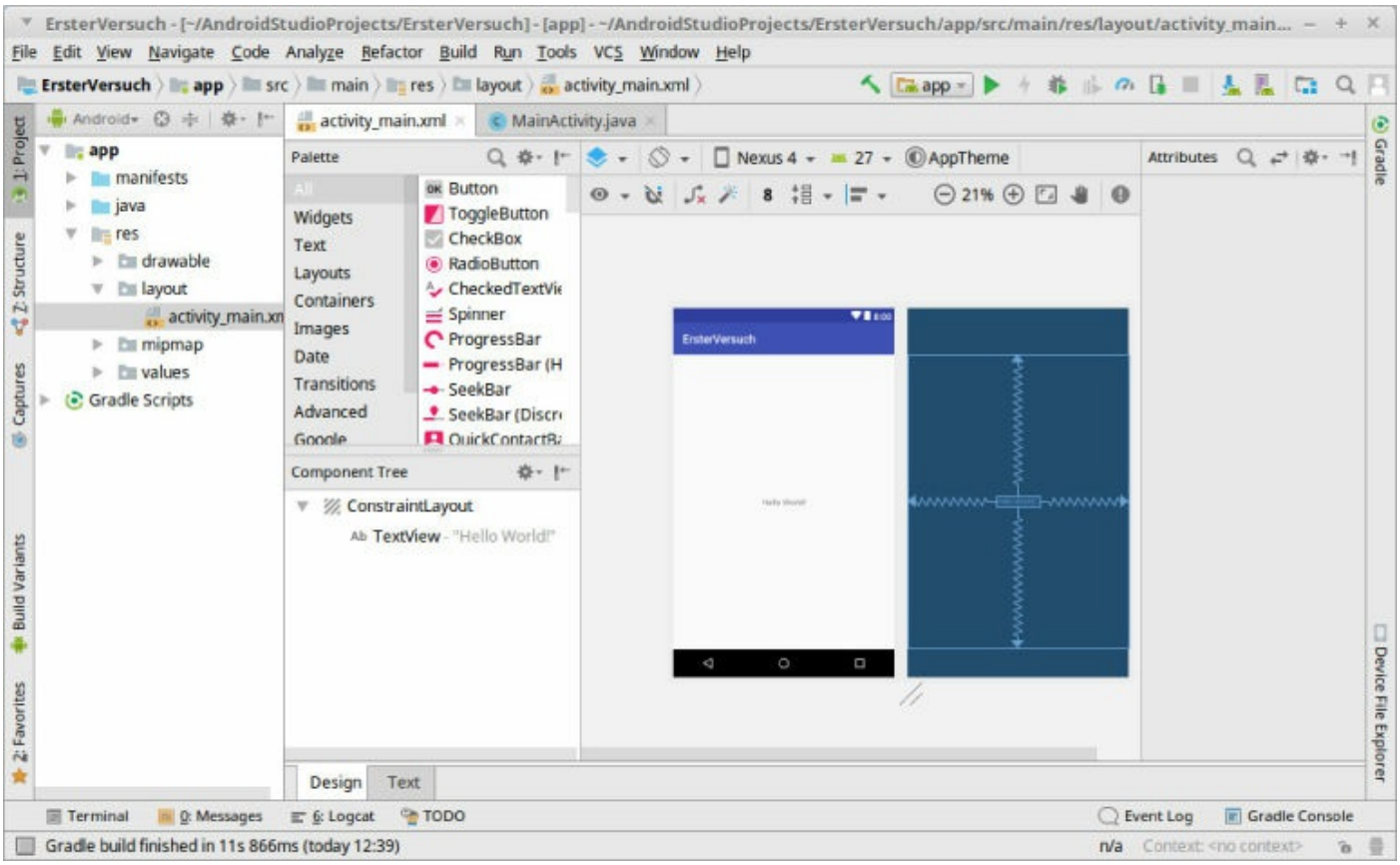


Abbildung 22.7: Gestaltung der Activity

Die Aufteilung ist mehrspaltig.

- ✔ Projekt-Explorer: Auf der linken Seite sehen Sie das Projekt. Im Gegensatz zu

Eclipse zeigt Ihnen Android Studio aber immer nur das aktuelle Projekt und nicht alle Projekte eines Workspace. Wenn Sie ein anderes Projekt bearbeiten wollen, schließen Sie das aktuelle über File|Close Project und kommen so zum Startdialog, wo Ihnen Ihre bisherigen Projekte angeboten werden oder Sie ein neues anlegen können.

- Palette: In dieser Spalte finden Sie Kontrollelemente, Container und andere Dinge, mit denen Sie Ihre Activity gestalten können.
- Rechts daneben sehen Sie die Gesamtsicht des Displays.
- Ganz rechts außen sehen Sie die Attribute der ausgewählten Activity-Elemente.

Unter dem rechten Fensterbereich sehen Sie die Buttons Design und Text. Damit können Sie zwischen der grafischen Gestaltung und der dabei erzeugten XML-Struktur umschalten. Damit werden wir uns beizeiten noch amüsieren. Erst einmal wird geklickt:

- Aus der Palette unter der Rubrik Text können Sie eine TextView finden.
- Ziehen Sie das Element mit der Maus an seine Zielposition auf dem Bildschirm. Wenn nicht alles so gerade ist, nehmen Sie es nicht so krumm. Daran basteln wir noch.
- Nun benötigen Sie noch je ein Eingabefeld für Größe und Gewicht. Dazu verwenden Sie ebenfalls unter der Rubrik Text je ein Element Plain Text. Dieser ist universell einsetzbar für alle Texteingaben. Weiter unten finden Sie auch Spezialisten für die Eingabe von Zahlen unter der Bezeichnung Number. Diese Elemente verhindern von vornherein die Eingabe irgendwelcher Zeichen, die nicht zu Zahlen passen. Auch diese Elemente ziehen Sie auf das Display.
- Unter der Rubrik Widgets finden Sie RadioButtons. Sie benötigen zwei davon für das Geschlecht.
- Ebenfalls unter Widgets gibt es einen Spinner. Dabei handelt es sich um eine Klappbox, in der der Benutzer sein Alter auswählen kann.
- Aktiv wird die Activity zunächst durch einen Button, den Sie auch unter der Rubrik Widget finden.

Nun geht es an die Attribute. Alle Elemente, auf die Sie später vom Programm aus zugreifen wollen, müssen eine ID bekommen. Das erreichen Sie, wenn Sie auf ein Kontrollelement klicken. Auf der rechten Seite öffnet sich dann ein Eingabefeld für die Attribute. Gleich das erste Feld ist mit ID beschriftet.

- Klicken Sie die `EditText`-Elemente an und geben Sie ihnen die Namen `edGroesse` und `edGewicht`.
- Die `TextView`, in der das Gewichtsurteil landen soll, soll `tvErgebnis` heißen.

- ✔ Die `RadioButtons` erhalten die Namen `rbMaennlich` und `rbWeiblich`.
- ✔ Der Alters-`Spinner` könnte `spAlter` heißen.
- ✔ Den `Button` könnten Sie `btCalc` nennen.
- ✔ Die Beschriftungen benötigen zwar keine Namen, aber wenn Sie gerade so schön in Geberlaune sind, könnten Sie auch diese taufen.

Im Component Tree unterhalb der Palette tauchen die Elemente untereinander auf. Eventuell finden Sie dort Warnungen, die besagen, dass die Beschriftungen »hard coded« sind. Das stimmt auch. Aber darum kümmern wir uns später.

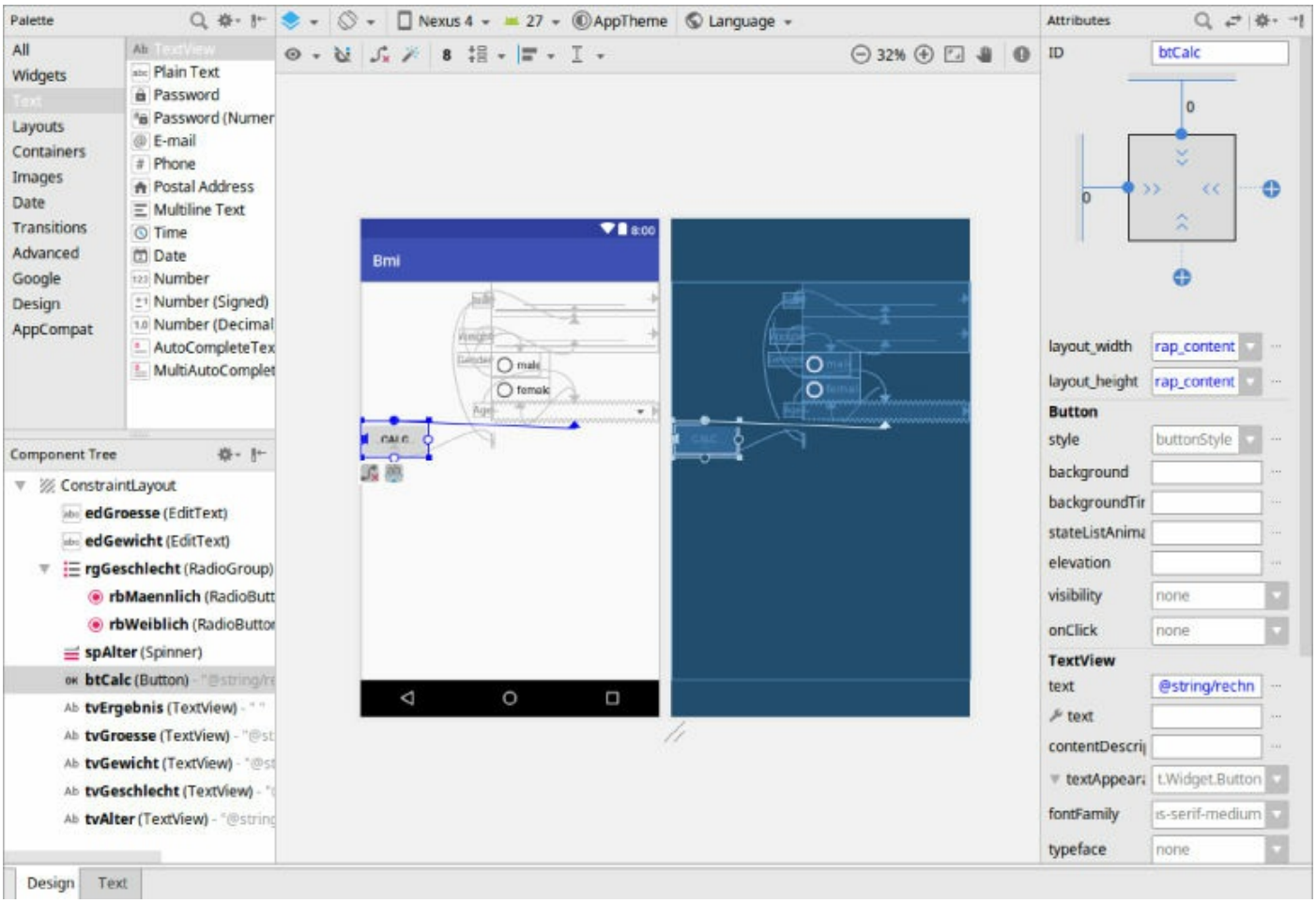


Abbildung 22.8: Die Kontrollelemente auf dem Display

Die App startet

Nun haben wir eine Activity zusammengehauen. Feinschliff kommt später. Jetzt soll die App erst einmal starten. Dazu wählen Sie im Menü Run den Menüpunkt Run 'App' aus oder klicken das zentral liegende nach rechts weisende Dreieck an und diese App wird gestartet.

Der Start kann in einer Android-Emulation oder auf einem per USB angeschlossenen Smartphone erfolgen. Android Studio fragt Sie in einem Dialog, welchen Emulator Sie verwenden wollen. Sollte ein Android-Gerät per USB angeschlossen sein, wird dieses angezeigt.

Android-Emulator

Noch haben Sie keinen Emulator. Sie müssen also einen erstellen. Genau dazu gibt es den Button Create New Virtual Device. Wenn Sie diesen anklicken, erscheint ein Assistent für die Einrichtung des Emulators.

1. Im ersten Fenster wählen Sie die Hardware aus. Auf diese Weise können Sie sowohl niedrig auflösende wie auch Geräte mit Riesendisplay austesten, ohne sie gleich kaufen zu müssen. Mit Next kommen Sie weiter.
2. Im nächsten Dialog wählen Sie das System aus, also das System Image. Es spricht viel dafür, dass es die noch nicht gibt, also klicken Sie auf Download und Ihre Internet-Leitung muss noch einmal rauchen.
3. Der nächste Dialog zeigt die Zusammenfassung. Sie können dem virtuellen Device einen Namen geben und mit Finish die Definition abschließen.

Sie landen wieder im Auswahldialog und können das neu erzeugte virtuelle Device anwählen und mit OK starten.

Der Emulator benötigt einige Zeit, da er erst einmal das komplette Android-System starten muss. Das kann dauern. Suchen Sie sich einen Kaffee. Irgendwann ist Android endlich gestartet und zeigt Ihnen ein Schloss. Das müssen Sie erst zur Seite ziehen. Dann startet endlich Ihre Anwendung.

Abbildung 22.9 zeigt die Android-Emulation mit der BMI-App. Vielleicht sieht es bei Ihnen noch etwas schief aus. Machen Sie sich nichts daraus, Schönheit wächst erst mit dem Alter.

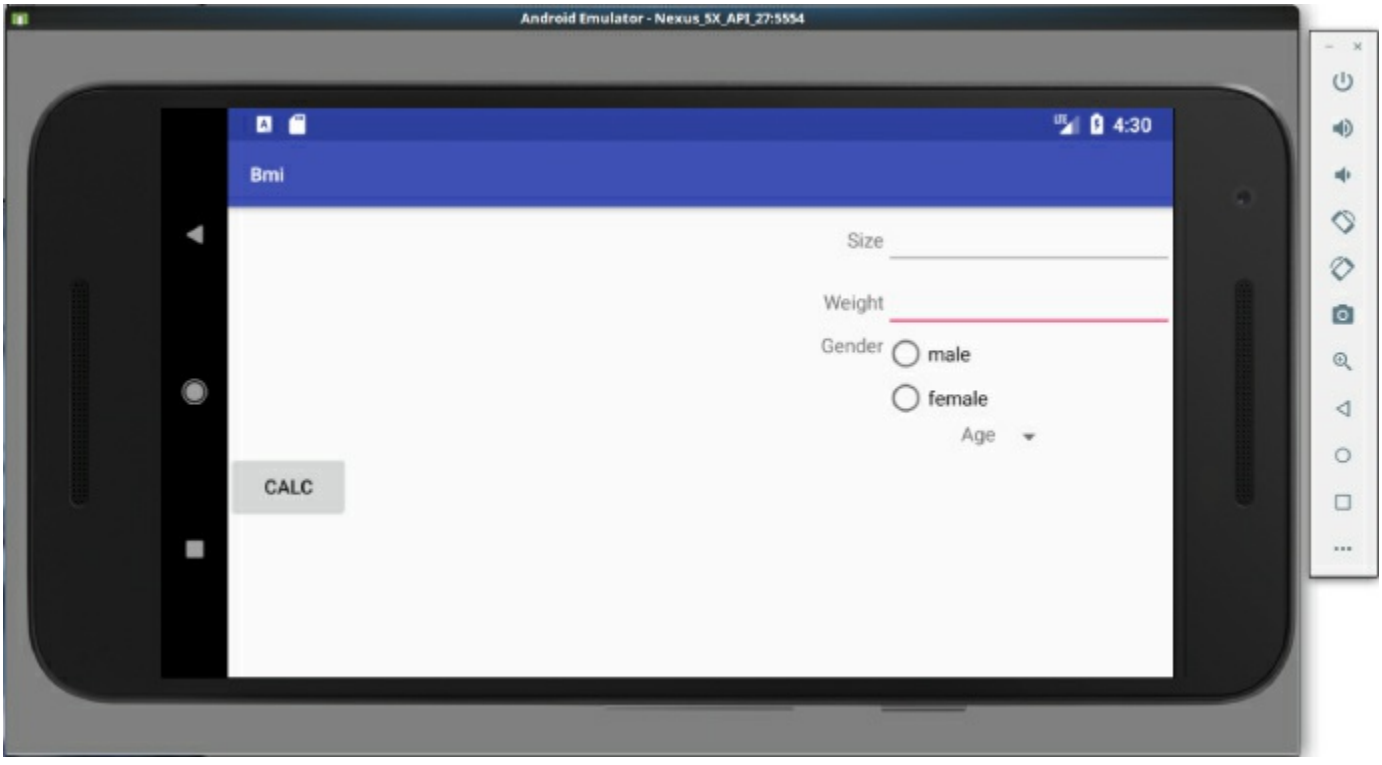


Abbildung 22.9: Die Android-Emulation mit der BMI-App

Ich habe mir erlaubt, den Emulator mit einem der Buttons zu drehen. Damit steht die App im Querformat. Die Darstellung auf dem Display muss also nicht nur flexibel mit den verschiedenen Display-Größen umgehen können, sondern muss auch im Querformat gut aussehen. Sie sehen, unsere App spielt bereits halbwegs mit.

In der Emulation können Sie die Applikation bedienen und ausprobieren. Rechts neben dem übergroßen Smartphone finden Sie Elemente, um das Display zu drehen, Bildschirmfotos zu machen oder andere Einstellungen durchzuführen.

Wenn Sie genug haben, finden Sie rechts neben dem Fenster einen Ausschalter. Anstatt die ganze Emulation jedes Mal zu stoppen und zu starten, reicht es, in der IDE neben dem grünen Dreieck ein rotes Quadrat zu suchen. Damit können Sie die App einfach stoppen.

Android-Gerät

Wenn Sie ein eigenes Android-Gerät über USB anschließen, können Sie es über die Android Debugging Bridge (ADB) steuern und damit Ihre Anwendung auf echter Hardware testen. Dazu schiebt das Android-SDK die App auf das Gerät, installiert und startet sie dort. Wie der Name der ADB schon vermuten lässt, ist auf diesem Weg auch ein Debugging möglich.

Android-Smartphones sind allerdings von Haus aus dagegen geschützt, dass wildfremde Apps auf dem Gerät installiert werden können. Standardmäßig starten nur Apps aus Googles Spielzeugladen. Ihre neue Anwendung gibt es dort aber noch nicht. Darum suchen Sie auf dem Gerät in den Einstellungen nach den Entwicklereinstellungen:

1. Starten Sie auf dem Smartphone Einstellungen.
2. Wechseln Sie zu Sicherheit.
3. Erlauben Sie fremde Quellen unter Unbekannte Herkunft.
4. Bei Smartphones ab Android 4 müssen die Entwickleroptionen erst freigeschaltet werden. Dazu wechselt man aus den Einstellungen zu ÜBer... in den Systemeinstellungen und tippt wiederholt auf den Eintrag Build Number. Irgendwann meldet das Smartphone, dass die Entwickleroptionen freigeschaltet wurden.
5. Wechseln Sie aus den Einstellungen zu den Entwickleroptionen.
6. Schalten Sie das USB-Debugging frei.
7. Sie finden hier auch einen Schalter, der dafür sorgt, dass das Display beim Laden nie in den Ruhestand versetzt wird. Bei einigen Geräten heißt diese Option Aktiv Lassen.

Aus Sicherheitsgründen und um Google nicht das Vertriebsmonopol zu nehmen,

schalten Sie nach Ihren Tests diese Einstellungen wieder zurück. Den Entwicklermodus können Sie allerdings bei den meisten Geräten nicht zurücknehmen.

Zunächst wollen Sie aber Ihre Programme ausprobieren.

1. Verbinden Sie das Gerät mit Ihrem Computer. Sofort erscheint ein Dialog, der Sie fragt, ob man diesem Computer vertrauen kann. Falls Sie Ihrem Computer vertrauen, bestätigen Sie dies gegenüber dem Android-Gerät. Sie können es anweisen, diesem Computer immer zu vertrauen, indem Sie einen entsprechenden Haken setzen.
2. Sie starten im Android Studio Run|Run.
3. Es startet ein Dialog, der alle virtuellen und realen Android-Geräte auflistet.
4. Sie wählen das gewünschte Gerät aus.
5. Nun muss typischerweise die Plattform für das Smartphone geladen werden. Dazu klicken Sie auf Install And Continue.
6. Das Programm startet auf dem Smartphone und kann dort wie jede andere App bedient werden.

Gestaltung einer App

Der grafische Editor zur Gestaltung einer Activity erzeugt eine XML-Datei. Das Aussehen und die Gestaltung der App wird üblicherweise nicht direkt in Java programmiert, sondern in XML-Dateien festgelegt. Erst in der Reaktion auf Benutzeraktionen kommt Java ins Spiel und kontrolliert die Programmlogik.

Neben der Anordnung der Elemente wird auch die Beschriftung durch XML-Dateien erledigt. Das ist vor allem deshalb interessant, weil der Programmierer seine Apps über Googles Play-Store weltweit anbietet. So kann man sagen, dass eine englische Oberfläche Pflicht ist. Mit jeder weiteren Sprache gewinnen Sie die Herzen und vielleicht auch die Lizenzen einer weiteren Nation.

- *res/layout/activity_main.xml:* Diese Datei enthält die XML-Beschreibung des Bildschirms, die in der grafischen Bearbeitung entstanden ist. Diese haben Sie bereits durch das Verschieben der Kontrollelemente verändert.

  Für jede weitere Activity wird eine neue XML-Datei im Ordner *res/layout* abgelegt.
- *res/values/strings.xml:* Hier gehören die Texte der Oberfläche hin. So ist es leichter, fremdsprachliche Versionen zu schaffen. Für jede Sprache gibt es eine eigene Datei.

Das Activity-Design in XML

Das Designer-Tool erzeugt aus Ihren Schiebereien mit armen Kontrollelementen eine XML-Struktur, die Sie nachbearbeiten können, ja in den meisten Fällen auch nachbearbeiten müssen. Denn was auf einem kleinen Display noch ordentlich aussieht, kann auf einem großen Tablet chaotisch wirken. Aber vor allem muss Ihre App immer noch gut aussehen, wenn der Anwender das Display wendet. Und dazu reicht es nicht, ein paar Punkte auf dem Display festzulegen. Sie müssen festlegen, wie die Eingabeelemente zusammenhängen.

Sie erreichen den XML-Code, indem Sie unter der Palette einen Reiter Text anklicken, der sich neben dem Reiter Design befindet. Nun sehen Sie Ihr Design als XML in einem einfachen Editor.

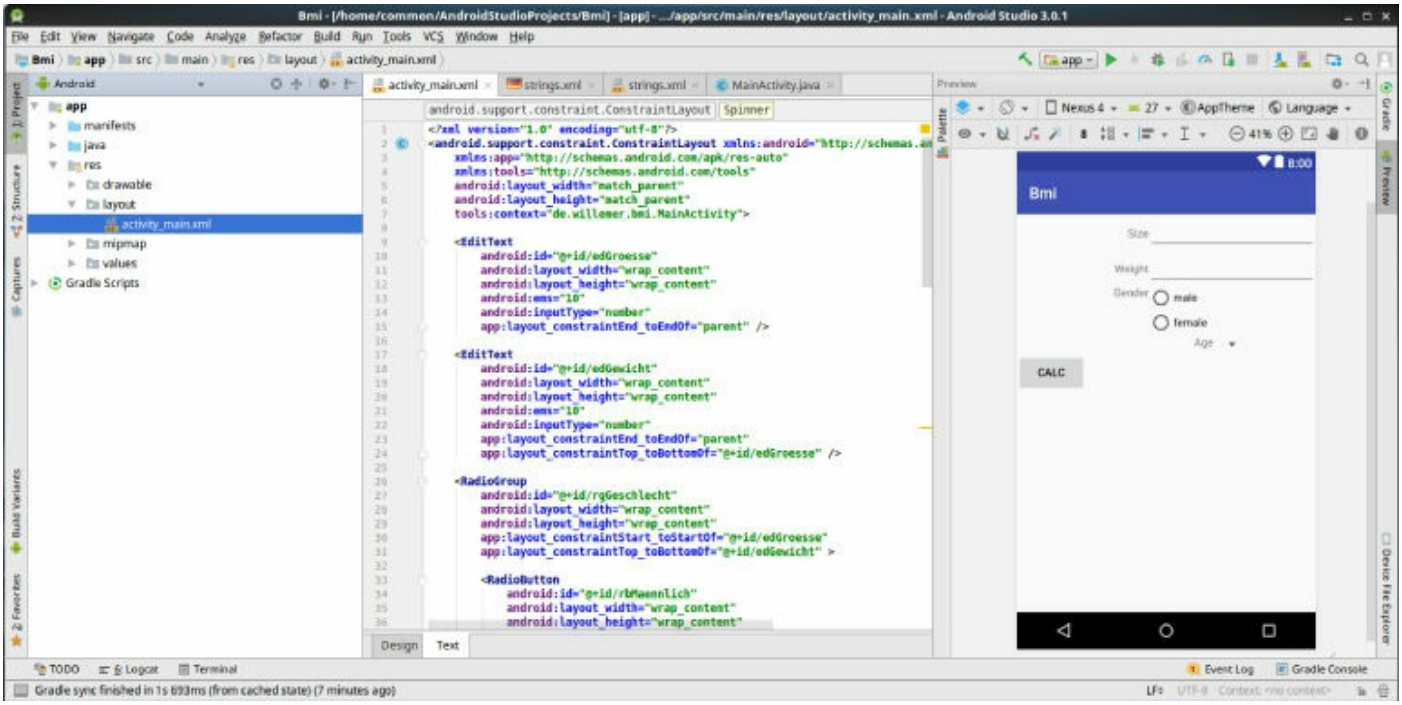


Abbildung 22.10: Die Activity als XML-Code in Android Studio

Leider reicht der Platz nicht, um Ihnen XML in aller Breite darzustellen. Aber ich kann Sie trösten. Sobald man sich an die Struktur erst einmal gewöhnt hat, sind XML-Strukturen relativ einfach zu lesen. Wenn Sie zunächst die Details ignorieren, können Sie vielleicht schon ohne Hilfe ein paar bekannte Punkte wiedererkennen.


Ein Schnellkurs für XML in fünf Schritten:

1. Tag: Ein Tag (gesprochen Täg) befindet sich zwischen spitzen Klammern, die eigentlich aus Kleiner- und Größerzeichen bestehen.
2. Das Tag beginnt mit einem Bezeichner, der beispielsweise das Layout oder das Kontrollelement bezeichnet, das Sie bearbeiten wollen.
3. Ein Tag endet damit, dass vor der schließendem Klammer, also vor dem Größerzeichen, ein Schrägstrich gesetzt wird. Soll das Element mehrere andere

umschließen, dann wird das Tag nicht mit einem Schrägstrich beendet, sondern später mit einem separaten Ende-Tag geschlossen. Das enthält den Bezeichner, dem ein Schrägstrich vorangestellt wird.

4. Innerhalb des Tags können Sie Zuweisungen für Werte vornehmen, die als Attribute bezeichnet werden. Beispielsweise können Sie mit `android:id="@+id/editHugo"` die Kennung eines Elements festlegen. Auch Breite und Höhe oder die Position werden so definiert.
5. Was mache ich als fünfte Zeile? Ach ja! Zwischen Start- und End-Tag können Sie Inhalte eintragen. Das kommt zwar beim Android-Layout nicht vor, aber so rette ich meine fünfte Zeile.

So gewappnet, können Sie die XML-Struktur von *res/layout/activity_main.xml* genauer ansehen.

Der Rahmen

Zu Anfang wird die XML-Version und der Zeichensatz definiert. Letzterer ist typischerweise UTF-8.

```
<?xml version="1.0" encoding="utf-8"?>
```

Gleich zu Anfang wird das Haupt-Layout festgelegt. Das Designer-Tool bevorzugt inzwischen das `ConstraintLayout`.

```
<android.support.constraint.ConstraintLayout ...
```

Der Rahmen der XML-Struktur gibt die »Schemas« vor. Das ist zwar im Deutschen furchtbar falsch, da Sie sprachlich eher Schemata erwarten würden. Dennoch müssen Sie sich vermutlich daran gewöhnen, da die XML-Datei eben diesen Begriff immer wieder in den URLs verwendet:

```
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
```

Die Breite und Höhe wird noch an das Elternelement, also meist den Bildschirmrand, angepasst. Zuletzt wird der Kontext zur Activity hergestellt.

```
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context="de.willemer.bmi.MainActivity">
```

Die Elemente

Für jedes Kontrollelement wird ein neues Tag angelegt, das die Eigenschaften und vor allem Position und Ausdehnung des Elements beschreibt. Neben den Kontrollelementen können Sie auch Layout-Elemente definieren, in denen sich wiederum Kontrollelemente oder weitere Layout-Elemente befinden können.

- `android:id="@+id/element"`: Jedes Element wird mit einer Kennung versehen.
- `android:text="@strings/text"`: Hier wird die Beschriftung hinterlegt.
- `android:layout_width="match_parent"`: Die Breite orientiert sich am Rahmen des Eltenelements.
- `android:layout_height="wrap_content"`: Die Höhe wird durch die Höhe des Inhalts bestimmt.

Sie finden unter der Palette eine große Menge an Kontrollelementen. In der folgenden Übersicht sind die wichtigsten Kontrollelemente aufgezählt.

- `EditText`: Es handelt sich um ein Eingabefeld. Wird es aktiviert, schiebt das Smartphone eine virtuelle Tastatur ins Bild. Es gibt unterschiedliche Eingabetypen, die über das Attribut `inputType` eingestellt werden. Da sie unterschiedliche Eingaben zulassen und andere verwehren, unterscheidet sich oft auch die zugehörige virtuelle Tastatur. Zahlenfelder benötigen beispielsweise keine Buchstaben.
  - `personName` wird als Standard vorgegeben. Wie der Name schon ahnen lässt, werden damit Namen eingegeben.
  - `text` erlaubt alle möglichen Texte.
  - `textPassword` verdeckt die Eingaben von Passwörtern.
  - `number` erlaubt nur Ziffern.
  - `numberSigned` erlaubt Zahlen mit einem Vorzeichen.
  - `numberDecimal` erlaubt Zahlen mit Dezimalstellen.
- `TextView`: Dabei handelt es sich um ein passives Darstellungsfeld (Label) für die Ausgabe statischer Beschriftungen oder auch Ergebnissen des Programms.
- `Button`: Ein Button wird eingesetzt, um Action in das Programm zu bringen, damit das Programm irgendwann einmal in die Hufe kommt.
- `RadioButton`: Diese Elemente können ein- und ausgeschaltet werden. Damit sich die Schalter gegenseitig ausschalten können, müssen alle Schalter, die sich gegenseitig beeinflussen sollen, in einer `RadioGroup` zusammengefasst werden.
- `RadioGroup`: Theoretisch ist es zwar möglich, eine `RadioGroup` im Design-Editor zu erzeugen. Einfacher ist es allerdings im Quellcode. Die `RadioGroup` ist nicht nur ein unsichtbarer Sammelbehälter für Ihre Buttons, sondern auch ein Layout-Element. Zu den enthaltenen Radiobuttons verhält es sich wie ein `LinearLayout`. Es besitzt eine Einstellung unter dem Attribut `android:orientation`, die festlegt, ob die enthaltenen Radiobuttons horizontal (horizontal) oder vertikal (vertical) anzuordnen sind. Weitere

Anordnungselemente hat die `RadioGroup` nicht.

- ✔ `Spinner`: Dieses Element ist eine Klappbox, aus deren Inhalt ein Element gewählt werden kann.

In ganzer Schönheit

Das folgende Layout ist recht intensiv von Hand nachbearbeitet. Lediglich die `@string`-Ressourcen sind noch nicht aufgelöst.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="de.willemer.bmi.MainActivity">
```

```
<EditText
    android:id="@+id/edGroesse"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:ems="10"
    android:inputType="number"
    app:layout_constraintEnd_toEndOf="parent" />

<EditText
    android:id="@+id/edGewicht"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:ems="10"
    android:inputType="number"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/edGroesse" />

<RadioGroup
    android:id="@+id/rgGeschlecht"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintStart_toStartOf="@+id/edGroesse"
    app:layout_constraintTop_toBottomOf="@+id/edGewicht" >

    <RadioButton
        android:id="@+id/rbMaennlich"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/maennlich"
     />

    <RadioButton
        android:id="@+id/rbWeiblich"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/weiblich"
        />
</RadioGroup>

<Spinner
    android:id="@+id/spAlter"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    app:layout_constraintStart_toStartOf="@+id/edGroesse"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/rgGeschlecht"
    />
```

```
<Button
    android:id="@+id/btCalc"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/rechne"
    app:layout_constraintTop_toBottomOf="@+id/spAlter"
    app:layout_constraintStart_toStartOf="parent"
    />

<TextView
    android:id="@+id/tvErgebnis"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text=" "
    app:layout_constraintBaseline_toBaselineOf="@+id/btCalc"
    app:layout_constraintStart_toStartOf="@+id/edGroesse"
    />

<TextView
    android:id="@+id/tvGroesse"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/groesse"
    app:layout_constraintBaseline_toBaselineOf="@+id/edGroesse"
    app:layout_constraintEnd_toStartOf="@+id/edGroesse"
    />

<TextView
    android:id="@+id/tvGewicht"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/gewicht"
    app:layout_constraintBaseline_toBaselineOf="@+id/edGewicht"
    app:layout_constraintEnd_toStartOf="@+id/edGewicht"
    />

<TextView
    android:id="@+id/tvGeschlecht"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/geschlecht"
    app:layout_constraintTop_toTopOf="@+id/rgGeschlecht"
    app:layout_constraintEnd_toStartOf="@+id/rgGeschlecht"
    />
```

```
    <TextView
        android:id="@+id/tvAlter"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/alter"
        app:layout_constraintTop_toTopOf="@+id/spAlter"
        app:layout_constraintEnd_toStartOf="@+id/spAlter"
        />
</android.support.constraint.ConstraintLayout>
```

Listing 22.1: XML-Datei für die `Activity`

Die meisten Attribute gehören zum Layout. Ein Layout umschließt mehrere Kontrollelemente, darum wird es mit einem Tag beispielsweise `<ConstraintLayout>` beginnen und am Ende mit dem passenden Ende-Tag `</ConstraintLayout>` schließen. Dieser Abschluss erscheint nach dem letzten Element, das zum Layout gehört. Sie können auf diesem Wege auch mehrere Layouts in einer `Activity` verwenden und diese sogar ineinander verschachteln. Um flexibel reagieren zu können, werden mehrere Layouts angeboten. Nicht alle Layouts stehen Ihnen in allen Android-Versionen zur Verfügung.

ConstraintLayout

Das `ConstraintLayout` beschreibt den Zusammenhang zwischen den Kontrollelementen mit relativen Positionen zum Rand anderer Kontrollelemente oder den Eltern. Sie können sich dies wie Achsen oder Seile vorstellen, mit denen die Elemente verankert werden. Je nach Display richten sich die Kontrollelemente neu aus und sehen hoffentlich gut aus. Dabei baut sich der Attributname einer solchen Verbindung folgendermaßen auf:

1. `app:layout_constraint`: Dies ist das Präfix, das besagt, dass hier eine Constraint-Verbindung aufgebaut wird.
2. Direkt anschließend wird die Kante des eigenen Elements genannt:
    - `Start`
    - `End`
    - `Top`
    - `Bottom`
3. Mit der Silbe `_to` wird übergeleitet, zu welchem Rand eines Elements die Verbindung aufgebaut wird.
4. Es folgt nun wiederum `Start`, `End`, `Top` oder `Bottom`, je nachdem, an welcher Kante das Element andocken soll.

5. Mit der Silbe `Of` wird die Verbindung abgeschlossen.
6. Es folgt ein Gleichheitszeichen und danach in Anführungszeichen das Ziel
    - `parent`: Sofern das Layout nicht verschachtelt ist, handelt es sich dabei um den Display-Rand.
    - `@+id/element` nennt das Kontrollelement, auf das sich die Verbindung bezieht.

Das folgende Beispiel wird den linken Rand des Elements auf gleiche Höhe des linken Rands von `edPreis` setzen:

```
app:layout_constraintStart_toStartOf="@+id/edPreis"
```

Insbesondere Eingabeelemente und Beschriftungen bieten eine weitere Option. Hier kann eine Beziehung über die Basislinie der Schrift erfolgen. Um eine Beschriftung mit dem Eingabefeld zu kombinieren, würden Sie beispielsweise folgende Anweisung verwenden:

```
app:layout_constraintBaseline_toBaselineOf="@+id/edPreis"
```

Allerdings funktioniert dies nicht mit allen Kontrollelementen. Beispielsweise hat ein `Spinner` oder eine `RadioGroup` keine Baseline.

Ränder und Einrückungen

```
android:layout_marginTop="8dp"
android:layout_marginStart="8dp"
android:layout_marginLeft="8dp"
android:layout_marginEnd="8dp"
android:layout_marginRight="8dp"
```

Damit sind wir bei den Rändern (margin) angekommen. Am einfachsten lernt man die Constraints, indem man mit dem grafischen Editor experimentiert und schaut, welche Ergebnisse in der XML-Datei dabei entstehen. Wenn Sie Änderungen in der XML-Struktur vornehmen, können Sie praktischerweise sofort in der grafischen Darstellung sehen, welche Auswirkungen das hat.

RelativeLayout

Das `RelativeLayout` ist etwas älter als das `ConstraintLayout` und bietet nicht ganz so viele Möglichkeiten. Aber auch mit dem `RelativeLayout` können Sie die Verhältnisse der Elemente zueinander beschreiben. Die Verhältnisse werden durch folgende Attribute beschrieben:

- ✔ `android:layout_above`: Das Element liegt über dem anderen Element.
- ✔ `android:layout_below`: Das Element liegt unter dem anderen Element.
- ✔ `android:layout_toLeftOf`: Das Element liegt links von dem anderen Element.

- `android:layout_toRightOf`: Das Element liegt rechts von dem anderen Element.
- `android:layout_toStartOf`: Das Element richtet sich nach dem Anfang des anderen Elements aus. Die Elemente beginnen also in der gleichen vertikalen Position.
- `android:layout_toEndOf`: Das Element richtet sich nach dem Ende des anderen Elements aus.

LinearLayout

Das `LinearLayout` ordnet einfach alle Elemente untereinander oder nebeneinander an. Die Richtung wird unter `android:orientation` durch die Zeichenkette »horizontal« oder »vertical« festgelegt. Die Elemente innerhalb eines `LinearLayouts` erhalten keine Positionsangaben. Sie werden einfach in ihrer auftretenden Reihenfolge einsortiert.

GridLayout

Mit dem `GridLayout` kann ein Raster definiert werden, in dem die Elemente mit den Parametern `layout_row` und `layout_column` angeben, an welcher Position sie sein möchten.

Das `GridLayout` ist erst ab API 14, also Android 4.0 verfügbar.

Warnungen und Fehler

Sie werden vermutlich auf Warnungen und sogar auf Fehler stoßen. In beiden Fällen werden Sie genauere Meldungen bekommen, wenn Sie mit dem Mauszeiger darüber fahren.

Zu den Fehlern gehören Meldungen, die besagen, dass andere APIs einige Dinge gern anders hätten.

Die meisten Warnungen beziehen sich auf Texte, die Sie in den Kontrollelementen direkt hinterlegt haben. Hier bevorzugt Android die Verwendung einer `@string`-Ressource.

Texte in die String-Datei

Die Beschriftungen der `Activity` sollten nicht direkt in der Layout-Datei stattfinden, sondern wiederum ausgelagert werden. Das wird man schon deshalb tun, weil Android-Apps weltweit verkauft werden und darum mehrsprachig ausgelegt sein sollten oder zumindest diese Option offenhalten sollten. Durch die Trennung der Beschriftung vom Layout kann für jede Sprache eine eigene Beschriftungsdatei angelegt werden.

Editieren der Beschriftungsdatei

Die Texte für die Beschriftungen werden in der Datei *strings.xml* im Verzeichnis

res/values abgelegt. Dort finden Sie normalerweise bereits den Namen der App unter `app_name`. Sinnvollerweise verwendet man in der Standarddatei englische Texte.

```
<resources>
    <string name="app_name">Bmi</string>
    <string name="maennlich">male</string>
    <string name="weiblich">female</string>
    <string name="rechne">Calc</string>
    <string name="groesse">Size</string>
    <string name="gewicht">Weight</string>
    <string name="geschlecht">Gender</string>
    <string name="alter">Age</string>
</resources>
```

Listing 22.2: Beispiel für die Beschriftungstabelle

Speichern Sie Ihre Änderung! Anschließend geht es wieder zurück in die Layout-Datei für die `Activity`.

Verweis im Layout

In der Layout-Datei werden die Einträge der Kontrollemente nun so geändert, dass sie auf die String-Tabelle verweisen. Um den Eintrag Alter in der Datei *string.xml* anzusprechen, wird in der Layout-Datei der Eintrag `"@string/alter"` als String verwendet, typischerweise im Attribut `android:text`. Hier sehen Sie den Ausschnitt aus der Activity-Layout-XML:

```
<TextView
    android:id="@+id/tvAlter"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/alter"
    app:layout_constraintTop_toTopOf="@+id/spAlter"
    app:layout_constraintEnd_toStartOf="@+id/spAlter" />
ndroid:text="@string/anzahl"
```

Das Gleiche kann natürlich bei den anderen Einträgen auch gemacht werden. Ziel ist es, dass die Activity-Datei keine Texte mehr enthält.

String-Arrays

Für den Inhalt der Klappbox werden mehrere Texte benötigt. Dazu ist es hilfreich, dass Sie ein Array von Strings definieren können. Dazu wird statt `string-name` ein Tag mit `string-array-name` definiert, das mehrere Einträge der Art `item` enthält. Als kleine Hausaufgabe überlasse ich es nun Ihnen, den Text englisch zu übersetzen.

```
<string-array name="strAges">
    <item>bis 25 Jahre</item>
    <item>bis 35 Jahre</item>
    <item>bis 45 Jahre</item>
    <item>bis 55 Jahre</item>
    <item>bis 65 Jahre</item>
    <item>über 65 Jahre</item>
</string-array>
```

Listing 22.3: Der `Spinner`-Inhalt in der *strings.xml*

Für die Klappbox, die bei Android `Spinner` heißt, kann nun im Layout ein spezieller Eintrag namens `android:entries` definiert werden, der auf ein solches Array verweist.

```
<Spinner
    android:id="@+id/spAlter"
    android:entries="@array/strAges"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    app:layout_constraintStart_toStartOf="@+id/edGroesse"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/rgGeschlecht" />
```

Listing 22.4: Der Verweis in der Layout-Datei

Danach können Sie bereits die Altersangaben aus der Klappbox wählen.

International

Nun soll die App international werden. Dazu werden weitere String-Dateien benötigt. Im Android Studio legen Sie die zusätzlichen Dateien folgendermaßen an:

1. Öffnen Sie den Pfad *res/values* in Ihrem Projekt.
2. Klicken Sie die Datei *strings.xml* mit der rechten Maustaste an.
3. Dann wählen Sie aus dem Kontextmenü New|Values Ressource File aus.
4. Es erscheint eine Dialogbox. Geben Sie als Dateinamen *strings.xml* ein.
5. Nun klicken Sie unten in der Liste unter Available Qualifiers den Punkt Locale an und schaffen ihn über den Button, der mit Größerzeichen beschriftet ist, in die Spalte Chosen Qualifiers.
6. Es entsteht eine neue Liste der Sprachen und der Regionen, aus der Sie die jeweils gewünschte auswählen.

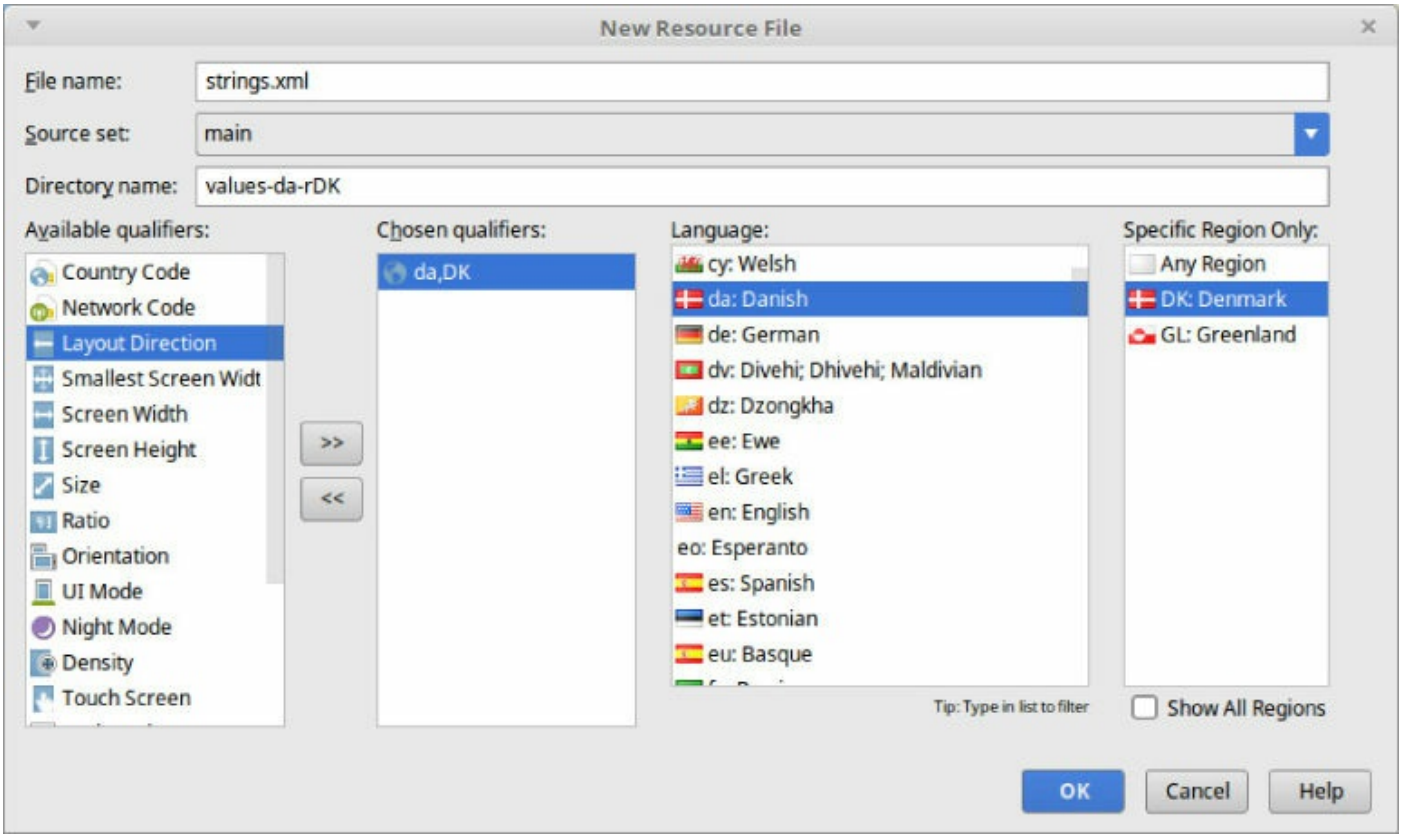


Abbildung 22.11: Neue Sprache hinzufügen

Vermutlich werden Sie, bevor Sie eine dänische Übersetzung erzeugen, erst einmal eine deutsche Variante verwenden. Nach der Bestätigung mit OK erzeugt das Android Studio eine Datei für die neu gewählte Sprache. Sie finden nun unter *res/values/strings.xml* ein Verzeichnis, in dem sich alle Sprachdateien finden, die Ihre App besitzt. Sie müssen nun die Struktur übernehmen und die Beschriftung übersetzen.

Die App wird automatisch beim Start die Sprache wählen, auf die das Smartphone eingestellt ist. Haben Sie mangels Sprachkenntnissen beispielsweise keine japanische Version erstellt, wird das Smartphone auf den Standard zurückfallen. Und da werden Japaner über die englische Beschriftung glücklicher sein als über die deutsche.

Java macht die Activity erst aktiv

Die XML-Dateien können den Aufbau des Bildschirms definieren. Aber um eine App zum Fliegen zu bringen, bedarf es der Programmierung, unter Android eben mit Java.

Die entsprechenden Dateien finden Sie im Projektordner unter *app/java*. Darunter befindet sich ein Package mit dem Domainnamen, den Sie bei der Erstellung des Projekts angegeben haben. Darunter sehen Sie den Quellcode für die erste Activity namens `MainActivity`.

```
package de.willemer.bmi;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

Listing 22.5: Von Android Studio erstellter Programmrahmen

Im Quelltext finden Sie keine Methode `main`. Stattdessen erweitert die App eine von `Activity` abgeleitete Klasse, hier `AppCompatActivity`. Deren Methode `onCreate`, die die Initialisierung übernimmt, wurde bereits überschrieben. Die wichtigste Aufgabe von `onCreate` besteht in der Einbindung der XML-Datei mit dem Layout über die Methode `setContentView`. Weil das Android Studio diesen Teil vorgefertigt hatte, war es möglich, ohne weitere Programmierung die App bereits zu starten.

Ereignisse fangen

Damit die App aktiv wird, müssen die Benutzeraktionen gefangen und mit passenden Aktionen beantwortet werden. Diese Benutzeraktionen sind aus Sicht des Programms Ereignisse (Events), auf die es reagieren soll.

Einen Listener implementieren

Zunächst wollen wir auf den Klick des Buttons reagieren. Dazu müssen wir die Klasse so ändern, dass sie auf Klicks oder besser auf das Tatschen mit Fingern reagiert. Dazu muss die Klasse das Interface `OnClickListener` implementieren. Sie müssen nur die Klassendeklaration ein wenig anpassen.

```
public class MainActivity extends AppCompatActivity
                          implements OnClickListener {
```

Wenn Sie das nachvollziehen, entsteht sofort ein Fehler! Das erkennen Sie an der roten Schrift. Nun lassen Sie sich einfach vom Android Studio helfen.

1. Wenn Sie die Maus über der roten Schrift stehen lassen, sehen Sie den Fehler in einem Tooltipp. Sie sehen nun, dass Java `OnClickListener` nicht kennt.
2. Stellen Sie nun den Eingabe-Cursor direkt links neben oder in das Wort.

Drücken Sie nun Alt + Return und es erscheint eine Liste von möglichen Reparaturen, hier beispielsweise einen passenden Import einzufügen. Android Studio schlägt `android.view.View.OnClickListener` vor. Das scheint eine gute Idee, die Sie akzeptieren.

3. Vom Regen in die Traufe: Noch ein Fehler. Diesmal gefällt es Java nicht, dass `OnClickListener` implementiert wird, aber weit und breit nichts von der Methode `OnClick` zu sehen ist.
4. Verwenden Sie wieder Alt + Return und wählen Sie den Vorschlag, die fehlenden Methoden zu implementieren. Es erscheint noch einmal ein Dialog, der die fehlenden Methoden aufführt. Bestätigen Sie dies und schon erscheint die Methode `onClick` in Ihrem Quelltext.

```
@Override
public void onClick(View view) {
}
```



Alternativ können Sie auch über das Menü Build|Make Project aufrufen. Android Studio klappt unten ein Message-Fenster auf, in dem die Fehler stehen. Zur Bearbeitung klicken Sie die Fehlermeldung an.

Auslöse-Button zuordnen

Der schönste Listener nützt Ihnen nichts, wenn er nicht aktiviert wird. Er muss mit dem Button verbunden werden. Dies passiert sinnvollerweise beim Start der App. Und damit sind wir bei der Methode `onCreate`. Zunächst definieren Sie den Zugriff auf den Button. Dieser wird nicht hier wie unter Java durch `new` erstellt, sondern er wurde ja bereits in der Layout-XML-Datei angelegt. Stattdessen verwendet das Programm die Methode `findViewById`, um den Button mit der Ressource `btCalc` zu verknüpfen.

1. Zunächst erklären Sie, dass Sie einen Button brauchen.
   ```
   Button rechneButton =
   ```
2. Den passenden Button ermitteln Sie über die Methode `findViewById`. Diese liefert aber nicht nur Buttons, sodass Sie deren Rückgabewert casten müssen.
   ```
   Button rechneButton = (Button) findViewById(
   ```
3. Die Ressource des Buttons wird über das Element `id` der Klasse `R` erreicht. Die Klasse `R` erzeugt die Entwicklungsumgebung automatisch, wenn die Layout-Datei gesichert wird. Wenn Sie also `R.id.` eingeben, legt Ihnen Android Studio in einer Liste alle verfügbaren Elemente vor.
   ```
   Button rechneButton = (Button) findViewById(R.id.btCalc);
   ```



Bei der Eingabe der Zeile wird das Wort `Button` rot. Also gehen Sie mit dem Cursor darauf, drücken wieder Alt + Return und lassen Android Studio den Import einbauen.

Nun müssen Sie dem Button sagen, welcher Listener sich um seine Ereignisse kümmern soll. Dazu rufen Sie die Methode `setOnClickListener` über den Button auf und übergeben als Parameter diese Klasse.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Button rechneButton = (Button) findViewById(R.id.btCalc);
    rechneButton.setOnClickListener(this);
}
```

Damit ist alles vorbereitet. Sobald der Benutzer auf den Button tatscht, wird die Methode `onClick` aufgerufen.

Ereignisse fangen

Die Methode `onClick` fängt das Ereignis, das der Button auslöst. Der Button besagt, dass gerechnet werden soll, und dazu müssen Sie natürlich erst einmal alle Elemente auslesen.

In dieser Methode werden Sie die Eingabefelder auslesen und das Ergebnis in der entsprechenden `TextView` zeigen.

Der Zugriff auf die Kontrollelemente erfolgt wieder über die Methode `findViewById`. Das folgende Beispiel ignoriert erst einmal die BMI-Berechnung, sondern holt die Größe aus dem Eingabefeld und platziert die Zahl im Ergebnisfeld.

```
@Override
public void onClick(View view) {
    EditText ed;
    ed = (EditText) findViewById(R.id.edGroesse);
    int groesse = Integer.parseInt(ed.getText().toString());
    TextView tv;
    tv = (TextView) findViewById(R.id.tvErgebnis);
    tv.setText(""+groesse);
}
```

Listing 22.6: `onClick` berechnet aus den Eingabefeldern den BMI.

Lassen Sie sich wieder von Android Studio helfen:

- ✔ Es werden vermutlich `EditText` und `TextView` rot aufleuchten. Klicken Sie den Cursor hinein und drücken Sie Alt + Return und Android Studio baut Ihnen den

Import für die Kontrollelemente.

- ✔ Nach der Eingabe von `findViewById(R.id.` warten Sie einen kleinen Augenblick und Android Studio legt Ihnen eine Liste aller Kontrollelemente vor, aus der Sie sich das passende aussuchen können.

Nun können Sie das Programm ausprobieren und werden feststellen, dass eine Berührung ausreicht, um den Wert aus dem Größeneingabefeld in das Ergebnisfeld zu übernehmen.



Wenn das Feld nur Zahleneingaben zulässt, kann es immer noch sein, dass der Inhalt leer ist. Auf diesem Weg kann also ein Fehler entstehen, insbesondere, wenn das Feld bei einem Fokuswechsel gelesen wird. Am sichersten ist es, die `NumberFormatException` zu fangen, die auftritt, wenn der String unerwartete Inhalte hat. Die folgende Methode ist auf diese Weise gesichert:

```
private int getIntFromEdit(EditText ed) {
    int wert = 0;
    try {
        wert = Integer.parseInt(ed.getText().toString());
    } catch (NumberFormatException e) {
    }
    return wert;
}
```

Die BMI-Berechnung

Die BMI-Berechnung stecken Sie in eine eigene Methode, da sie später nicht nur durch den Button ausgelöst werden soll.

1. Neben der Größe muss auch das Gewicht ausgelesen werden. Auch dies ist ein Zahleneingabefeld.
   ```
   ed = (EditText) findViewById(R.id.edGewicht);
   int gewicht = Integer.parseInt(ed.getText().toString());
   ```
2. Die Grundformel besteht darin, das Gewicht durch das Quadrat der Größe in Metern zu dividieren. Da die Eingabe in Zentimetern erfolgt, müssen wir noch durch 10.000 dividieren. Hier wird ein Dezimalpunkt angegeben, um eine ganzzahlige Division zu verhindern. Am Ende wird die Zahl gerundet. Nachkommastellen sind nicht so spannend.
   ```
   double bmi = 0.0;
   if (groesse>0) {
       bmi = gewicht / (groesse * groesse / 10000.0);
       bmi = Math.round(bmi * 10.0) / 10.0;
   ```
3. Anstatt abzunehmen, können Sie auch einfach warten, bis Sie älter werden. Mit

jedem Jahrzehnt dürfen Sie sich einen BMI-Punkt abziehen. Die Altersgrenzen sind in der Klappbox abgelegt, dem `Spinner`. Der gewählte Index wird durch die Methode `getSelectedItemId` ermittelt. Diesen Index müssen wir einfach nur abziehen. Falls der Benutzer aber nichts gewählt hat, liegt dieser bei -1. Das müssen Sie noch prüfen.

```
Spinner spAge = (Spinner) findViewById(R.id.spAlter);
long alter = spAge.getSelectedItemId();
if (alter < 0) {
    alter = 0;
}
bmi -= alter;
```

Sie können den `Spinner` neben der Nummer auch um das ausgewählte Element selbst bitten. Dazu rufen Sie die Methode `getSelectedItem()` auf.

4. Das Leben ist ungerecht, aber Männer dürfen fetter werden als Frauen. Zumindest erlaubt das der BMI. Wenn sich also ein Männchen als solches in den Radiobuttons zu erkennen gab, ziehen Sie noch einen Punkt ab. Ob ein Radiobutton markiert ist, können Sie mit dessen Methode `isChecked()` prüfen.

```
RadioButton rbMaennlich = (RadioButton) findViewById
  (R.id.rbMaennlich);
if (rbMaennlich.isChecked()) {
    bmi++;
}
```

5. Zum Schluss wird das Ergebnis, wie oben beschrieben, in das Ergebnisfeld geschrieben.

Der Parameter von onClick

Die Methode `onClick` beschränkt sich nun auf den Aufruf der Methode zur Ermittlung des BMI.

```
@Override
public void onClick(View view) {
    ermittleBmi();
}
```

Es lohnt sich ein kleiner Blick auf den Parameter von `onClick`. Die als Parameter übergebene `View` liefert über die Methode `getId()` die Kennung des auslösenden Elements. So können Sie beispielsweise ermitteln, welcher Button gedrückt wurde, falls mehr als einer aktiv ist.

```
@Override
public void onClick(View view) {
    if (view.getId() == R.id.btCalc) {
        // Der Button btCalc wurde gedrückt
    } else if (view.getId() == R.id.btAbbruch) {
        // Der Button btAbbruch wurde gedrückt
    }
}
```

Listing 22.7: `onClick` ermittelt den auslösenden Button.

Ereignisse der Kontrollelemente

Der Benutzer muss nun nacheinander alle Felder bestücken. Zum Schluss löst er den Button aus und sieht das Ergebnis. Falls ihm auffällt, dass er doch gewachsen ist, kann er die Größe ändern und zur Berechnung noch einmal den Button anklicken.

Der Button und sein Tatsch

Der Button bringt häufig das Leben in die Bude. Darum ist das Erstellen eines Buttons nur sinnvoll, wenn Sie auch dessen Klickverhalten beobachten. Diese Rolle kann leicht die eigene `Activity` übernehmen. Sie muss dazu das Interface `OnClickListener` implementieren. Sofort wird die `Activity` rot, weil sie noch nicht die Methode `onClick()` implementiert hat. Allein das Implementieren reicht aber nicht. Es muss jedem Button, der eine Wirkung haben soll, auch eine Verknüpfung zum `Listener` gebaut werden. Und dazu gibt es die Methode `setOnClickListener()`. Hier noch einmal im Überblick:

```
public class MainActivity extends Activity
                          implements View.OnClickListener {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        Button button = (Button) findViewById(R.id.btCalc);
        button.setOnClickListener(this);

        ...
   }
    @Override
    public void onClick(View v) {
        ...
    }
}
```

Listing 22.8: Der Button und der Listener

Den Fokuswechsel fangen

Häufig müssen Sie bei Eingabefeldern nur die Texte mit der Methode `getText()` auslesen, wenn mal ein Button gedrückt wurde. Seltener kommt es vor, dass Sie mit der Methode `setText()` einen neuen Text hineinschieben müssen. Insofern wirkt das EditView beinahe träge. Aber auch der `EditText` feiert Ereignisse, wenn das Feld verlassen oder betreten wird. Diese Ereignisse fasst man als Fokuswechsel zusammen. So kann gleich nach dem Verlassen des Eingabefelds die Benutzereingabe in das Gesamtergebnis einbezogen werden. Dazu muss `OnFocusChangeListener` implementiert werden.

```
import android.view.View.OnFocusChangeListener;

public class MainActivity extends AppCompatActivity
        implements View.OnFocusChangeListener {
```

Listing 22.9: Ergänzung des Programmkopfs

Für jedes Eingabefeld können Sie in `onCreate` die Methode `setOnFocusChangeListener()` aufrufen. Sie können als Parameter `this` verwenden, weil die Activity ja den `OnFocusChangeListener` implementiert.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    // ...
    EditText ed;
    ed = (EditText) findViewById(R.id.edGroesse);
    ed.setOnFocusChangeListener(this);
```

Anschließend wird noch die Methode onFocusChange implementiert, die bei jedem Fokuswechsel aufgerufen wird. Nun wird bei Betreten oder Verlassen jeweils der BMI neu berechnet.

```
@Override
public void onFocusChange(View view, boolean b) {
    ermittleBmi();
}
```

Listing 22.10: Ein Fokuswechsel löst eine Neuberechnung aus.

Sie können nicht nur Eingabefelder, sondern alle Kontrollelemente mit einem Fokuswechsel beglücken. Wenn Sie es geschickt anstellen, benötigen Sie für Ihre App gar keinen Button mehr, weil die Werte jedes Mal neu berechnet werden, wenn ein Kontrollelement verlassen wird.

Radiobuttons und ihre Gruppe

Für die Verarbeitung der Ereignisse rund um die Radiobuttons greift Java vor allem auf die `RadioGroup` zu. Will das Programm informiert werden, sobald jemand einen der Buttons anklickt, braucht es einen Lauscher und der heißt `OnCheckedChangeListener`. Der Einfachheit halber kann die `Activity` diesen gleich selbst implementieren.

```
import android.widget.RadioGroup.OnCheckedChangeListener;

public class MainActivity extends AppCompatActivity
        implements OnCheckedChangeListener {
```

Listing 22.11: Ergänzung des Programmkopfs

Nun muss die abstrakte Methode `onCheckedChanged()` des Interface implementiert werden. Die Methode erhält als Parameter die betroffene `RadioGroup` und die Nummer des Radiobuttons übergeben.

```
@Override
public void onCheckedChanged(RadioGroup radioGroup, int i) {
    ermittleBmi();
}
```

Um den Listener aktiv für die Radiobuttons scharf zu schalten, muss über die `RadioGroup` die Methode `setOnCheckedChangeListener()` aufgerufen werden.

```
RadioGroup rg = (RadioGroup) findViewById(R.id.rgGeschlecht);
rg.setOnCheckedChangeListener(this);
```

Die Klappbox ist ein Spinner

Wenn Sie auf dem Laufenden sein wollen, ob der Benutzer an der Klappbox spielt, sollten Sie `OnItemSelectedListener` implementieren. Dazu müssen Sie zwei Methoden überschreiben: `onItemSelected()` und `onNothingSelected()`.

```
import android.widget.AdapterView;

public class MainActivity extends AppCompactActivity
        implements AdapterView.OnItemSelectedListener {
```

Damit alles scharf wird, müssen Sie dann noch dem Spinner mit der Methode `setOnItemSelectedListener()` den Listener zuordnen.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    // ...
    Spinner spAge= (Spinner) findViewById(R.id.spAlter);
    spAge.setOnItemSelectedListener(this);
}
```

Sie müssen in diesem Fall zwei Methoden implementieren, nämlich `onItemSelected` und `onNothingSelected`.

```
@Override
public void onItemSelected(AdapterView<?> adapterView,
        View view, int i, long l) {
    ermittleBmi();
}
@Override
public void onNothingSelected(AdapterView<?> adapterView) {
    ermittleBmi();
}
```

Texte aus dem Java-Programm lesen

Die Texte für die Beschriftungen werden in der Datei *strings.xml* im Verzeichnis *res/values* abgelegt.

```
<string name="alter">Age</string>
```

Die definierten Zeichenketten können Sie auch von Java aus lesen. Verwenden Sie dazu den folgenden Aufruf:

```
String str = getRessources().getString(R.id.anzahl);
```

Für unsere BMI-Auswertung sind die reinen BMI-Werte wenig hilfreich, weil die meisten Nutzer mit einem BMI-Wert von 29 nicht viel anfangen können. Also benötigen Sie eine Beschriftung, die natürlich auch in die String-Datei gehört.

```
<string-array name="strMasse">
    <item>Geh zum Arzt!</item>
    <item>Normalgewicht</item>
    <item>Übergewicht</item>
    <item>Adipositas</item>
    <item>Schwanger?</item>
</string-array>
```

Auf das Array kann mit der folgenden Befehlszeile zugegriffen werden:

```
String[] strMasse = getResources().getStringArray(
        R.array.strMasse);
```

Eine Umwandlung des BMI-Wertes in den passenden String erledigt dann die Methode `getBmiString`:

```
private String getBmiString(int bmi) {
    int[] grenze = {20, 25, 30, 40};
    int index = 0;
    while (index < grenze.length && grenze[index] < bmi) {
        index++;
    }
    String[] strMasse = getResources().getStringArray(
            R.array.strMasse);
    String str = "";
    if (index<strMasse.length) {
        str = strMasse[index];
    }
    return str;
}
```

Listing 22.12: Umwandlung des BMI in einen String

Die MainActivity zusammengefasst

Nachdem Sie alle Puzzle-Teile gesehen haben, setzen wir alles zusammen. Die fertige `MainActivity` zur Ermittlung des BMI!

```
package de.willemer.bmi;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.Button;
import android.widget.EditText;
import android.widget.RadioButton;
import android.widget.RadioGroup;
import android.widget.Spinner;
import android.widget.TextView;

import java.util.concurrent.ExecutionException;

public class MainActivity extends AppCompatActivity
        implements View.OnClickListener,
        View.OnFocusChangeListener,
        RadioGroup.OnCheckedChangeListener,
        AdapterView.OnItemSelectedListener  {
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Button rechneButton = (Button) findViewById(R.id.btCalc);
    rechneButton.setOnClickListener(this);
    EditText ed;
    ed = (EditText) findViewById(R.id.edGroesse);
    ed.setOnFocusChangeListener(this);
    ed = (EditText) findViewById(R.id.edGewicht);
    ed.setOnFocusChangeListener(this);
    RadioGroup rg = (RadioGroup) findViewById(
            R.id.rgGeschlecht);
    rg.setOnCheckedChangeListener(this);
    Spinner spAge= (Spinner) findViewById(R.id.spAlter);
    spAge.setOnItemSelectedListener(this);
}

private int getIntFromEdit(EditText ed) {
    int wert = 0;
    try {
        wert = Integer.parseInt(ed.getText().toString());
    } catch (NumberFormatException e) {
    }
    return wert;
}

private String getBmiString(int bmi) {
    int[] grenze = {20, 25, 30, 40};
    int index = 0;
    while (index < grenze.length && grenze[index] < bmi) {
        index++;
    }
    String[] strMasse = getResources().getStringArray(
            R.array.strMasse);
    String str = "";
    if (index<strMasse.length) {
        str = strMasse[index];
    }
    return str;
}

private void ermittleBmi() {
    EditText ed;
    ed = (EditText) findViewById(R.id.edGroesse);
    int groesse = getIntFromEdit(ed);
```

```
        ed = (EditText) findViewById(R.id.edGewicht);
        int gewicht = getIntFromEdit(ed);
        double bmi = 0.0;
        // Verhindere Division durch 0
        if (groesse>0) {
            bmi = gewicht / (groesse * groesse / 10000.0);
            bmi = Math.round(bmi * 10.0) / 10.0;

            // Man muss nicht abnehmen, sondern nur alt werden
            Spinner spAge = (Spinner) findViewById(R.id.spAlter);
            long alter = spAge.getSelectedItemId();
            if (alter < 0) {
                alter = 0;
            }
            bmi -= alter;

            // Maenner dürfen fetter werden.
            RadioButton rbMaennlich =
                    (RadioButton)findViewById(R.id.rbMaennlich);
            if (rbMaennlich.isChecked()) {
                bmi++;
            }
        }
        TextView tv;
        tv = (TextView) findViewById(R.id.tvErgebnis);
        tv.setText(getBmiString((int)bmi));
    }

    @Override
    public void onClick(View view) {
        if (view.getId() == R.id.btCalc) {
            ermittleBmi();
        }
    }

    @Override
    public void onFocusChange(View view, boolean b) {
        ermittleBmi();
    }

    @Override
    public void onCheckedChanged(RadioGroup radioGroup, int i) {
        ermittleBmi();
    }
```

```
    @Override
    public void onItemSelected(AdapterView<?> adapterView,
                               View view, int i, long l) {
        ermittleBmi();
    }

    @Override
    public void onNothingSelected(AdapterView<?> adapterView) {
        ermittleBmi();
    }
}
```

Listing 22.13: Das Listing der `MainActivity`

Bermuda auch auf Android

Um Bermuda auf Android zu portieren, benötigt man ein Zeichenbrett und ein paar Grafikbefehle. Als Zeichenbrett können Sie eine `View` einsetzen. Eine `View` ist die Mutter aller Kontrollelemente und kann wie ein `Button` oder ein `Spinner` in ein Layout eingebaut werden.

Eine eigene View als Spielfeld

Der Bildschirm des Spiels wird durch eine eigene `View` geprägt, die vom Programm selbst gestaltet wird und `BermudaView` heißen soll. Hinzu kommt ein `Button`, um ein neues Spiel zu starten, und eine `TextView`, die den Spielstand anzeigen soll. Damit ergibt sich eine übersichtliche Layout-XML-Datei:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="de.willemer.bermuda.MainActivity">

    <de.willemer.bermuda.BermudaView
        android:id="@+id/bermudaview"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintBottom_toTopOf="@+id/newgame"
        />
    <TextView
        android:id="@+id/score"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/message"
        android:textAppearance="?android:attr/textAppearanceMedium"
        app:layout_constraintBaseline_toBaselineOf="@+id/newgame"
        app:layout_constraintStart_toStartOf="parent"
         />

    <Button
        android:id="@+id/newgame"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/newgame"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        />

</android.support.constraint.ConstraintLayout>
```

Listing 22.14: Die Layout-Datei *activity_main.xml*

Die eigene View muss mit vollem Package-Namen angegeben werden. Unter gleichem Namen muss nun eine Java-Klasse erstellt werden, die die Aktivitäten der View übernimmt.

Sie sehen auch, dass die Texte gar nicht erst direkt in die Layout-Datei geschrieben

wurden, sondern bereits für die String-Tabelle in *res/values/strings.xml* vorbereitet wurden.

```
<resources>
    <string name="app_name">Bermuda</string>
    <string name="newgame">New Game</string>
    <string name="message">Search for four lost ships</string>
    <string name="markmodus">Mark</string>
</resources>
```

Listing 22.15: Die Beschriftung in *strings.xml*

Eine Java-Klasse für die BermudaView

Der Zauber beginnt in unserer Klasse `BermudaView`, die die Klasse `View` erweitert. Sie wird wie jede normale Java-Klasse erzeugt, muss dann aber die `View` erweitern und dazu einen Konstruktor bekommen.

1. Im Projekt öffnen Sie *app/java*.
2. Klicken Sie das Package mit der rechten Maustaste an und wählen Sie New|Java Class.
3. In dem Dialog geben Sie `BermudaView` als Name an.
4. Mit OK schließen Sie den Dialog ab.
5. Im Quelltext ergänzen Sie die Klassendefinition um `extends View`.
   ```
   public class BermudaView extends View {
   ```
6. Das Wort `View` wird rot. Setzen Sie den Cursor hinein und drücken Sie Alt + Return und wählen Sie Import Class.
7. Anschließend wird die gesamte Zeile rot. Wieder klicken Sie hinein und drücken Alt + Return. Wählen Sie Creating constructor matching super.
8. Sie erhalten mehrere Vorschläge für den Konstruktor. Sie benötigen einen Konstruktor mit mindestens zwei Parametern: `Context` und `AttributeSet`. Das Ergebnis sollte etwa so aussehen:

```
package de.willemer.bermuda;

import android.content.Context;
import android.support.annotation.Nullable;
import android.util.AttributeSet;
import android.view.View;

public class BermudaView extends View {

    public BermudaView(Context context,
                       @Nullable AttributeSet attrs) {
        super(context, attrs);
    }
}
```

Nun müssten Sie die App bereits starten können. Es erscheint der Text in der `TextView`, ein Button für ein neues Spiel und eine große, weiße Fläche.



Abbildung 22.12: Die leere `BermudaView`

Zeichnen in der View

Eine `View` zeichnet ihren Inhalt selbst. Dazu wird sie von Android aufgefordert, indem die Methode `onDraw` aufgerufen wird. Wollen Sie also das stahlende Weiß durch eigene Grafiken ersetzen, müssen Sie die Methode `onDraw` in der Klasse `BermudaView` überschreiben.

Als Parameter erhält `onDraw` ein Objekt der Klasse `Canvas`. Das ist die Zeichenumgebung von Android, die diverse Zeichenoperationen zur Verfügung stellt, wie beispielsweise `drawLine`, um eine Linie zu zeichnen.

Zum Zeichnen wird auch ein Objekt von `Paint` benötigt. Darüber werden Farben und andere Attribute gesteuert. Die folgende Implementierung von `onDraw` zieht eine rote Diagonale durch die `BermudaView`.

```
@Override
public void onDraw(Canvas canvas) {
    Paint paint = new Paint();
    paint.setColor(Color.RED);
    canvas.drawLine(0, 0,
        canvas.getWidth(), canvas.getHeight(), paint);
}
```

Listing 22.16: Eine Diagonale durch die `View`

Ein paar Grafikprimitive des Canvas

Gezeichnet wird über ein `Canvas`, das Ihnen in der Methode `onDraw` als Parameter bereitgestellt wird. Die Zeichenprimitive beginnen alle mit der Silbe `draw`. Die folgende Übersicht zeigt die wichtigsten Grafikprimitive des Canvas:

- Punkt: Unter Android kann auch ein Punkt erstellt werden.
  `void drawPoint(float x, float y, Paint paint)`
- Linie: Es wird eine Linie von den angegebenen Anfangs- zu den Endkoordinaten gezogen.
  ```
  void drawLine(float vonX, float vonY, float bisX, float bisY,
                Paint paint)
  ```
- Kreis: Ein Kreis wird durch die Koordinaten des Mittelpunkts und den Radius als Parameter festgelegt.
  `void drawCircle(float x, float y, float radius, Paint paint)`
- Rechteck: Als Parameter werden die vier Koordinaten links, oben, rechts und unten angegeben, also nicht etwa Breite und Höhe.
  ```
  void drawRect(float links, float oben,
                float rechts, float unten, Paint paint)
  ```

Als Parameter für `drawRect` kann statt der Koordinaten auch ein Objekt der Klasse `RectF` verwendet werden, das die öffentlichen Attribute `left`, `top`, `right` und `bottom` besitzt.

- Oval: Die Größe wird durch ein Objekt der Klasse `RectF` festgelegt, das den äußeren Umfang bestimmt.
  `void drawOval(RectF oval, Paint paint)`
- Text: Das Zeichnen von Text erfolgt über die Methode `drawText`. Die Parameter sind der String und die Position, wo der Text erscheinen soll.
  `void drawText(String text, float x, float y, Paint paint)`

Mit der Methode `drawColor()` können Sie den Canvas-Bereich löschen und gleich mit einer Hintergrundfarbe besetzen. Um einen blauen Hintergrund zu bekommen, reicht die folgende Zeile:

```
canvas.drawColor(Color.BLUE);
```

Alternativ können Sie natürlich auch ein Rechteck über die Ausmaße des Canvas ziehen.

```
canvas.drawColor(Color.BLUE);
canvas.drawRect(0, 0, canvas.getWidth(), canvas.getHeight(), paint);
```

Paint, die Art des Zeichnens

Vielleicht haben Sie schon die `fill`-Methoden vermisst, aber ob eine Figur gezeichnet oder gefüllt wird, wird durch ein Objekt der Klasse `Paint` festgelegt. Eine Referenz darauf wird jeder Grafikprimitiven als Parameter übergeben.

Die Klasse `Paint` stellt diverse Zeichenattribute zur Verfügung. Für unsere Zwecke reicht uns erst einmal das Setzen der Farbe und des Zeichenstils.

- Der Zeichenstil wird mit `setStyle` gesetzt. Der Parameter kann folgende Werte annehmen:
  - `FILL` wird die Grafikprimitive füllen.
  - `STROKE` wird nur die Außenlinie zeichnen.
  - `FILL_AND_STROKE` lässt beides in einem Schritt erfolgen.
- Die Farbe wird mit der Methode `setColor` gesetzt. Ihre Parameter sind die bekannten Farbkonstanten aus `Color`. Sie können sie aussuchen, indem Sie `Color.` eingeben und warten, was Ihnen Android Studio so anbietet.

Da nach der offiziellen Dokumentation innerhalb der Methode `onDraw()` das Erzeugen von Objekten vermieden werden soll, wird in der View üblicherweise ein lokales Attribut der Klasse `Paint` angelegt, das auch gern privat sein darf.

Die folgende Sequenz zeichnet zunächst einen grün gefüllten Kreis, um ihn dann in

Rot zu umrahmen. Weil die Farbkombination so ungewöhnlich scheußlich ist, verzichte ich auf einen Screenshot.

```
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.view.View;

public class GrafikView extends View {

    private Paint paint = new Paint();

    public GrafikView(Context context,
            @Nullable AttributeSet attrs) {
        super(context, attrs);
    }

    @Override
    public void onDraw(Canvas canvas) {
        paint.setStyle(Paint.Style.FILL);
        paint.setColor(Color.GREEN);
        float x=100, y=100, radius=100;
        canvas.drawCircle(x, y, radius, paint);
        paint.setStyle(Paint.Style.STROKE);
        paint.setColor(Color.RED);
        canvas.drawCircle(x, y, radius, paint);
    }
}
```

Listing 22.17: Fill und Draw per Paint

Übernahme der Bermuda-Ressourcen

Bei den grafischen Oberflächen haben wir bereits die Sourcen aus Kapitel 12 übernommen. Das können wir nun auch tun. Ziehen Sie die Dateien der Klassen `Flotte`, `Position`, `Schiff` und `Spielfeld` einfach in den Projektordner in das Package der `MainActivity` hinein. Android Studio wird automatisch den Package-Namen ergänzen.

Damit ist die Java-Seite geklärt. Nun wäre es schön, wenn wir auch die Bilder übernehmen könnten. Dazu gehen Sie ganz ähnlich vor. Sie ziehen die Bilddateien von *mark0.png* bis *mark8.png*, sowie *schiff.png* und *leer.png* in das Verzeichnis *res/drawable*.

Das Programm kommt an diese Bilder heran, indem es für die Datei *schiff.png* auf `R.drawable.schiff` zugreift. Mithilfe einer `BitmapFactory` zimmert es daraus eine `Bitmap`, die sich mit der Methode `drawBitmap` dann auf dem Bildschirm darstellen lässt.

```
@Override
public void onDraw(Canvas canvas) {
    canvas.drawColor(Color.BLUE);
    Bitmap schiff = BitmapFactory.decodeResource(
            getResources(), R.drawable.schiff);
    canvas.drawBitmap(schiff, 20, 20, paint);
}
```

Listing 22.18: Ein Schiff auf blauem Hintergrund

Allerdings ist es nicht besonders klug, jede `Bitmap` bei jedem Aufruf von `onDraw` neu zu generieren. Besser ist es, die Bitmaps bereits vorher einzulesen und in `onDraw` nur noch darzustellen. Stattdessen kann sich der Konstruktor um die Erzeugung der Bitmaps kümmern. Und dann kann der Konstruktor auch gleich die Breite und Höhe der Symbole für `xDist` und `yDist` bestimmen.

```
package de.willemer.bermuda;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.support.annotation.Nullable;
import android.util.AttributeSet;
import android.view.View;

public class BermudaView extends View {

    private Paint paint = new Paint();
    private Bitmap leer, schiff, ziffer[];
    private int xDist=1, yDist=1;

    public BermudaView(Context context,
                       @Nullable AttributeSet attrs) {
        super(context, attrs);
        leer = BitmapFactory.decodeResource(
                getResources(), R.drawable.leer);
        schiff = BitmapFactory.decodeResource(
                getResources(), R.drawable.schiff);
        ziffer = new Bitmap[9];
        for (int i=0; i<9; i++) {
            ziffer[i] = BitmapFactory.decodeResource(
                    getResources(),
                    getResources().getIdentifier(
                        "mark"+i, "drawable",
                        context.getPackageName()));
        }
        xDist = schiff.getWidth();
        yDist = schiff.getHeight();
    }

    @Override
    public void onDraw(Canvas canvas) {
        canvas.drawColor(Color.WHITE);
        for (int x=0; x<Spielfeld.MAXX; x++) {
            for (int y=0; y<Spielfeld.MAXY; y++) {
                canvas.drawBitmap(
                        ziffer[x], x*xDist, y*yDist, paint);
            }
        }
        canvas.drawBitmap(leer, 0, Spielfeld.MAXY*xDist, paint);
```

```
            canvas.drawBitmap(schiff, Spielfeld.MAXX*xDist,
                    Spielfeld.MAXY*yDist, paint);
        }
    }
```

Listing 22.19: Ein Test für die Darstellung der Bitmaps

Einbindung des Spielfelds

Um das Spielfeld einzubinden, muss es erst einmal erzeugt werden. Es bietet sich an, ein privates Attribut in der Klasse anzulegen, das sich gleich initialisiert.

```
private Spielfeld spiel = new Spielfeld();
```

Die Methode `onDraw` braucht nun nur so weit angepasst zu werden, dass der Inhalt des Spielfelds angezeigt wird.

```
@Override
public void onDraw(Canvas canvas) {
    canvas.drawColor(Color.WHITE);
    for (int x=0; x<Spielfeld.MAXX; x++) {
        for (int y=0; y<Spielfeld.MAXY; y++) {
            Position pos = new Position(x, y);
            char c = spiel.get(pos);
            Bitmap img = null;
            if (c >= '0' && c <= '8') {
                img = ziffer[c-'0'];
            } else if (c == Spielfeld.SCHIFF) {
                img = schiff;
            } else {
                img = leer;
            }
            canvas.drawBitmap(img, x*xDist, y*yDist, paint);
        }
    }
}
```

Listing 22.20: `onDraw` wertet das Spielfeld aus.

Den Tatsch fangen

`BermudaView` soll auch den echten Touch implementieren oder zumindest das Interface `OnTouchListener`, damit es den Ort der Fingerabdrücke auch sehen kann. Sobald Sie dies dem Kopf der Klasse hinzufügen, wird die IDE auch die Methode `onTouch()` einbauen wollen. Die Methode `onTouch` wird bei Berührung aufgerufen. Die Kopfzeile von `BermudaView` wird ergänzt:

```
public class BermudaView extends View
        implements View.OnTouchListener {
```

Nun trennen Sie nur noch das zweimalige Drücken von Alt + Return davon, dass Sie eine funktionsfähige Methode `onTouch` besitzen.

```
@Override
public boolean onTouch(View view, MotionEvent motionEvent) {
    return false;
}
```

Damit der Listener aktiv wird, muss er für den Auslöser gestartet werden. Der Auslöser ist in diesem Fall die komplette View. Darum muss im Konstruktor die Methode `setOnTouchListener()` gestartet werden.

```
public BermudaView(Context context,
                   @Nullable AttributeSet attrs) {
    super(context, attrs);
    this.setOnTouchListener(this);
```

Wird das Display betatscht, werden folgende Schritte durchgeführt:

1. Die Position wird aus dem `MotionEvent` mit den Methoden `getX` und `getY` ermittelt.
2. Durch eine Division mit der Symbolbreite wird die Spielkoordinate errechnet.
3. Es wird geprüft, ob der Aufschlag innerhalb des Spielfelds erfolgte.
4. Nun wird die Methode `suche` des Spielfelds aufgerufen.
5. Das Ergebnis wird dem Spielfeld über die Methode `set` mitgegeben.
6. Durch Aufruf von `invalidate` wird das System gezwungen, das Display neu anzuzeigen und damit `onDraw` aufzurufen.

Im Quelltext sieht das dann so aus:

```
@Override
public boolean onTouch(View view, MotionEvent me) {
    int suchX = (int)me.getX()/xDist;
    int suchY = (int)me.getY()/yDist;
    if (suchX>=0 && suchY>=0 && suchX<Spielfeld.MAXX
                && suchY<Spielfeld.MAXY) {
        Position pos = new Position(suchX, suchY);
        char c = spiel.suche(pos);
        spiel.set(pos, c);
        invalidate();
    }
    return false;
}
```

Listing 22.21: Der Tatsch wird mit dem Spiel verkoppelt.

Activity und View

Zwei Aufgaben sind noch nicht gelöst:

- ✔ Das Spiel muss feststellen, ob der Spieler gewonnen hat.
- ✔ Der Spieler muss ein neues Spiel starten können.

Wenn der Spieler alle vier Schiffe gefunden hat, muss etwas passieren. Ein gutes Spiel spricht das Belohnungszentrum im Gehirn an. Da Bermuda nicht zum Suchtspiel werden soll, beschränkt es sich darauf, den Sieg anzuzeigen. Dazu wird eine Nachricht im `TextView` der `MainActivity` ausgegeben. Um den Jubel zu unterstützen, wird es rot hinterlegt.

Hier entsteht ein Problem. Die `BermudaView` wird den Sieg über die Methode `alleSchiffeGefunden` der Klasse `Spielfeld` feststellen, aber die `TextView` liegt außerhalb ihres Wirkungsbereichs. Innerhalb der `BermudaView` wird die Anweisung `findViewById(R.id.score)` den Wert `null` zurückgeben.

Activity beschafft die TextView

Die `TextView` für die Nachrichtenzeile gehört der `MainActivity`. Die kann die `TextView` leicht ermitteln und einfach an die `BermudaView` weiterleiten. Mit folgenden Anweisungen innerhalb der Methode `onCreate` wird das erreicht:

```
BermudaView bermuda = (BermudaView) findViewById(
        R.id.bermudaview);
bermuda.transferMessageTV(
        (TextView) findViewById(R.id.score));
```

Listing 22.22: Übergabe der Nachrichtenzeile an die `BermudaView`

Dazu passend müssen Sie nun eine Methode `transferMessageTV` in der `BermudaView` erstellen, die die `TextView` in einer globalen Variablen hinterlegt, damit aus der `BermudaView` Nachrichten an den Benutzer gemeldet werden können.

```
private TextView messageTV = null;
// ...
public void transferMessageTV(TextView tv) {
    messageTV = tv;
}
```

Spielstand anzeigen

Den Besitz der Nachrichtenzeile nutzt die `BermudaView` schamlos aus, um in der Methode `onTouch` anzugeben, wie oft gepeilt wurde. Bei einem Gewinn können endlich rot hinterlegte Lobeshymnen erschallen.

```
public boolean onTouch(View view, MotionEvent me) {
    // ...
        spiel.set(pos, c);
        score++;
        String str = getResources().getString(R.string.versuche);
        str = "" + score + " " + str + " ";
        messageTV.setText(str);
        if (c==Spielfeld.SCHIFF) {
            if (spiel.alleSchiffeGefunden()) {
                str += getResources().getString(R.string.won);

         messageTV.setText(str);
         messageTV.setBackgroundColor(Color.RED);
    }
}
invalidate();
```

Neues Spiel

Zu guter Letzt muss noch der `Button` zur Anforderung eines neuen Spiels aktiviert werden. Dazu muss, wie im ersten Teil des Kapitels demonstriert, die `MainActivity` das Interface `onClickListener` implementieren, den `Button` per `setOnClickListener` scharf schalten und die Methode `onClick` implementieren.

```
public class MainActivity extends AppCompatActivity
                          implements View.OnClickListener {
// ..
    protected void onCreate(Bundle savedInstanceState) {
        // ...
        Button neuesSpiel = (Button) findViewById(R.id.newgame);
        neuesSpiel.setOnClickListener(this);
    }

    @Override
    public void onClick(View view) {
        BermudaView bermuda = (BermudaView) findViewById(
                R.id.bermudaview);
        bermuda.neuesSpiel();
    }
}
```

Listing 22.23: Den Button in `MainActivity` scharf machen

Der Druck auf den `Button` ruft nun die Methode `neuesSpiel` in der `BermudaView`, die natürlich geschrieben werden muss. Sie erzeugt ein neues Spielfeld, setzt den Punktestand auf 0 und gibt die Anfangsnachricht auf der Nachrichtenzeile aus.

```
public void neuesSpiel() {
    spiel = new Spielfeld();
    String str = getResources().getString(R.string.message);
    messageTV.setText(str);
    messageTV.setBackgroundColor(Color.LTGRAY);
    score = 0;
    invalidate();
}
```

Listing 22.24: Neues Spiel in der `BermudaView`

Die Programme im Überblick

Die `MainActivity` bewacht den einzigen `Button`, damit der Anwender ein neues Spiel einläuten kann. Außerdem muss sie zusehen, dass die `BermudaView` den Zugriff auf die Nachrichtenzeile bekommt. Ansonsten überlässt sie alles der `BermudaView`.

```
package de.willemer.bermuda;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity
        implements View.OnClickListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button neuesSpiel = (Button) findViewById(R.id.newgame);
        neuesSpiel.setOnClickListener(this);
        BermudaView bermuda = (BermudaView) findViewById(
                R.id.bermudaview);
        bermuda.transferMessageTV(
                (TextView) findViewById(R.id.score));
    }

    @Override
    public void onClick(View view) {
        BermudaView bermuda = (BermudaView) findViewById(
                R.id.bermudaview);
        bermuda.neuesSpiel();
    }
}
```

Listing 22.25: MainActivity

Die Hauptarbeit liegt in der `BermudaView`. Sie übernimmt den Aufbau des Bildschirms über die Methode `onDraw` und die Reaktionen auf die Aktionen des Spielers in der Methode `onTouch`. Für die Schnittstelle zur `MainActivity` stellt sie die Methoden `neuesSpiel` und `transferMessageTV` zur Verfügung.

```
package de.willemer.bermuda;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
```

```
import android.graphics.Color;
import android.graphics.Paint;
import android.support.annotation.Nullable;
import android.util.AttributeSet;
import android.view.MotionEvent;
import android.view.View;
import android.widget.TextView;

public class BermudaView extends View
        implements View.OnTouchListener {

    private Paint paint = new Paint();
    private Bitmap leer, schiff, ziffer[];
    private int xDist=1, yDist=1;

    private Spielfeld spiel = new Spielfeld();
    private TextView messageTV = null;
    private int score = 0;

    public BermudaView(Context context,
                       @Nullable AttributeSet attrs) {
        super(context, attrs);
        this.setOnTouchListener(this);
        leer = BitmapFactory.decodeResource(getResources(),
                R.drawable.leer);
        schiff = BitmapFactory.decodeResource(getResources(),
                R.drawable.schiff);
        ziffer = new Bitmap[9];
        for (int i=0; i<9; i++) {
            ziffer[i] = BitmapFactory.decodeResource(
                    getResources(),
                    getResources().getIdentifier(
                        "mark"+i, "drawable",
                        context.getPackageName()));
        }
        xDist = schiff.getWidth();
        yDist = schiff.getHeight();
    }

    @Override
    public void onDraw(Canvas canvas) {
        canvas.drawColor(Color.WHITE);
        for (int x=0; x<Spielfeld.MAXX; x++) {
            for (int y=0; y<Spielfeld.MAXY; y++) {
                Position pos = new Position(x, y);
                char c = spiel.get(pos);
                Bitmap img = null;
                if (c >= '0' && c <= '8') {
                    img = ziffer[c-'0'];
```

```
            } else if (c == Spielfeld.SCHIFF) {
                img = schiff;
            } else {
                img = leer;
            }
            canvas.drawBitmap(img, x*xDist, y*yDist, paint);
        }
    }
}

@Override
public boolean onTouch(View view, MotionEvent me) {
    int suchX = (int)me.getX()/xDist;
    int suchY = (int)me.getY()/yDist;
    if (suchX>=0 && suchY>=0 && suchX<Spielfeld.MAXX
                && suchY<Spielfeld.MAXY) {
        Position pos = new Position(suchX, suchY);
        char c = spiel.suche(pos);
        spiel.set(pos, c);
        score++;
        String str = getResources().getString(
                R.string.versuche);
        str = "" + score + " " + str + " ";
        messageTV.setText(str);
        if (c==Spielfeld.SCHIFF) {
            if (spiel.alleSchiffeGefunden()) {
                str += getResources().getString(
                        R.string.won);
                messageTV.setText(str);
                messageTV.setBackgroundColor(Color.RED);
            }
        }
        invalidate();
    }
    return false;
}

public void neuesSpiel() {
    spiel = new Spielfeld();
    String str = getResources().getString(
            R.string.message);
    messageTV.setText(str);
    messageTV.setBackgroundColor(Color.LTGRAY);
    score = 0;
    invalidate();
}
```

```
    public void transferMessageTV(TextView tv) {
        messageTV = tv;
    }
}
```

Listing 22.26: BermudaView

Das Ergebnis des Programms sehen Sie in Abbildung 22.13.

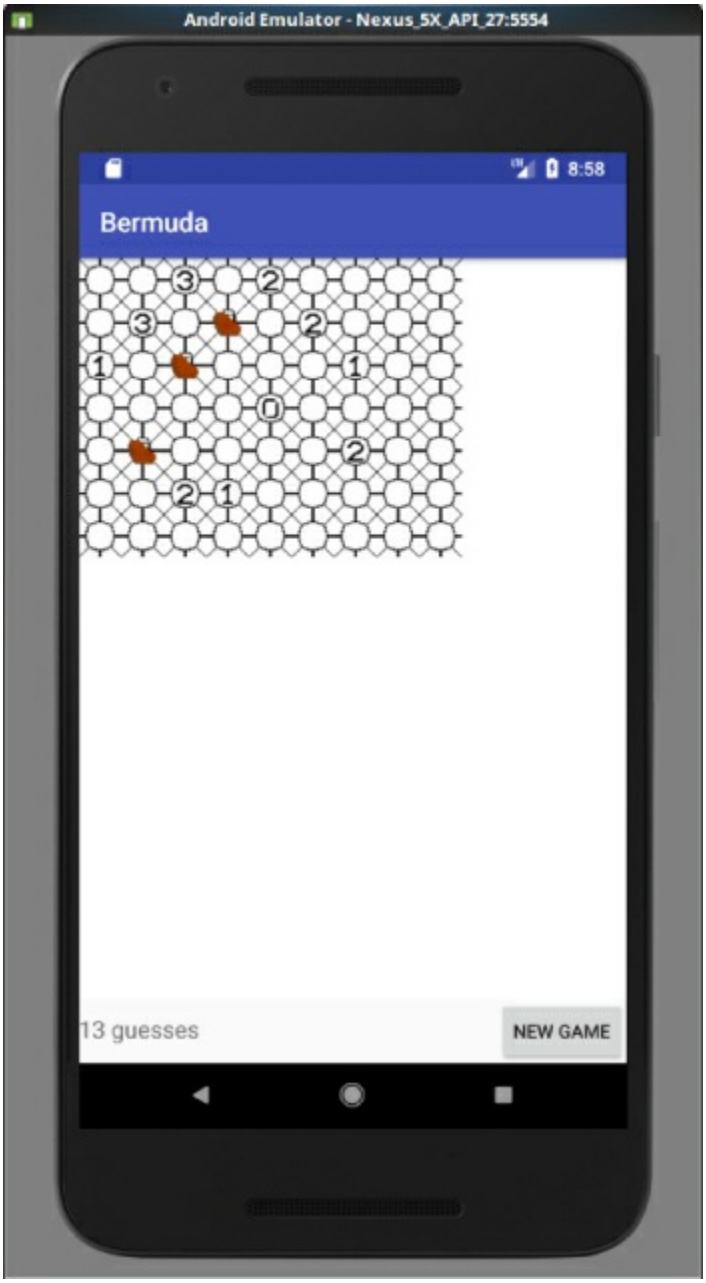


Abbildung 22.13: Bermuda mit Bitmaps unter Android

Und nun?

Das Kapitel ist hier zu Ende. Aber vielleicht wollen Sie jetzt ganz viele Android-Apps schreiben und es bleibt noch die ein oder andere Frage. Da gibt es die Möglichkeit des Zweitbuchs. Oder vielleicht auch der Blick ins Internet. Dort gibt

es eine Website, die sich nur mit Android-Programmierung beschäftigt: http://developer.android.com

Teil V

Daten längerfristig speichern



Kapitel 23

Dateien halten die Daten frisch

IN DIESEM KAPITEL

- Daten aus Dateien lesen und in Dateien schreiben
- Serialisierung: Ein Objekt nach dem anderen in die Datei
- Per Random Access beliebige Dateistrukturen lesen und schreiben
- Das Dateisystem auslesen

Es ist immer wieder erfreulich, dass sich Computer auch über längere Zeit noch erinnern können. Dazu verwenden sie in der Regel Dateien. Das sollten Ihre Java-Programme nutzen und in der Lage sein, ihre Erkenntnisse dort abzulegen. Denn der Inhalt der Variablen ist spätestens beim nächsten Stromausfall verloren.

Damit dieses Kapitel mehr als nur einen Abschnitt hat, gibt es mehrere Arten, um Daten in Dateien abzulegen. Und auch Ihre Programme werden unterhaltsamer, wenn Sie alle Varianten im Wechsel einsetzen.

Daten aus Textdateien lesen

Eine sehr einfache Art, Daten einzulesen, beschert der `Scanner`, den wir schon missbraucht haben, um Tastatureingaben einzulesen. Dazu wurde der Datenstrom `System.in` angezapft und die dort von der Tastatur eingegebenen Werte als Daten interpretiert.

```
import java.util.Scanner;
// ...
Scanner scanner = new Scanner(System.in);
double zahl = scanner.nextDouble();
```

Listing 23.1: Einlesen von der Tastatur

Wenn Sie aber niemanden finden, der bereit ist, Ihre Versuchsdaten über das Verhalten rosa gefärbter Dackel im Verlaufe des Kirchenjahres bei jedem Programmstart erneut einzutippen, ist es Zeit, über eine Alternative nachzudenken. Und diese kann darin bestehen, dass Sie diese Daten in einer Datei vorhalten, die das Programm immer wieder einlesen kann. Dazu kann der Datenstrom der Tastatur auf eine Datei umgesetzt werden.

```
import java.io.File;
import java.util.Scanner;
// ...
File datei = new File("datei.txt");
if (datei.exists()) {
    Scanner scanner;
    try {
        scanner = new Scanner(datei);
        double zahl = scanner.nextDouble();
    // ...
    scanner.close();
```

Listing 23.2: Einlesen aus der Datei

Mit dem Programm in Listing 23.2 können Sie einen Wert aus der Datei *datei.txt* lesen. Der Ordnung halber sollten Sie den Scanner nach Gebrauch wieder schließen.



Wenn Sie `double`-Werte über den Scanner einlesen, berücksichtigt er beim Dezimaltrennzeichen die voreingestellte Sprache des Computers. Bei einem deutschen Computer will der Scanner also Kommata sehen und weist Dezimalpunkte als `InputMismatchException` angewidert von sich.

Wo liegt die Datei?

Wenn Dateinamen einen vollständigen Pfad enthalten, beginnen sie mit einem Schrägstrich. Leute, die noch Windows benutzen, sprechen zwar oft vom *Backslash*, also dem rückwärtsgerichteten Schrägstrich, aber auch Windows verkraftet inzwischen den normalen Schrägstrich.

Beginnt also ein Dateiname mit einem Schrägstrich, spricht man von einem *absoluten Pfad*. Die Position der Datei wird exakt beschrieben. Bei Windows käme nun noch der Laufwerksbuchstabe hinzu. Aber das lassen wir mal weg. Es macht die Sache nur noch komplizierter. Beginnt der Dateiname nicht mit einem Schrägstrich (oder Backslash), so ist es ein *relativer Pfad* und bezieht sich auf die aktuelle Arbeitsposition im Dateisystem.

Die aktuelle Arbeitsposition ist der Ort, von dem aus das Programm gestartet wurde. Wenn das Programm aus Eclipse heraus gestartet wird, setzt Eclipse diese Position auf das Wurzelverzeichnis des Projekts. Und da trifft es sich gut, dass Eclipse eine neue Datei genau dorthin legen kann.

1. Klicken Sie das Projekt mit der rechten Maustaste an.
2. Wählen Sie New|File.
3. Geben Sie im Folgedialog den Namen *datei.txt* an.

Und schon finden Sie die Datei im Editor vor, um die Datenwerte Ihrer rosa Dackel einzutragen.



Die grafischen Oberflächen bieten eine Dateiauswahlbox an. Dann wählt der Benutzer für Sie die passende Datei aus. Das ist für beide Seiten ein Gewinn. Der Benutzer kann die Datei frei wählen und hat dazu den vertrauten Dialog zur Verfügung und Sie müssen sich nicht mit den Spezialitäten des Computers bei der Dateinamensgestaltung herumschlagen.

Eine solche Auswahlbox finden Sie bei Swing (Kapitel 16). Nur beim Elternfenster müssen Sie etwas schummeln und eine `null` übergeben, da Konsolenprogramme ja keinen eigenen Frame haben.

```
import javax.swing.JFileChooser;
import java.io.File;
import java.util.Scanner;
// ...
    File datei = new File("datei.txt");
    if (datei.exists()) {
        Scanner scanner;
        JFileChooser chooser = new JFileChooser();
        if (chooser.showOpenDialog(null)
                ==JFileChooser.APPROVE_OPTION) {
            datei = chooser.getSelectedFile();
        }
        try {
            scanner = new Scanner(datei);
            double zahl = scanner.nextDouble();
```

Listing 23.3: Auswahlbox von Swing klauen

Ausnahmezustände

Bei kaum einer anderen Gelegenheit gibt es so viele Möglichkeiten, dass etwas schiefgehen kann, wie beim Lesen und Schreiben von Dateien, wenn man mal von einer Hochzeit absieht. Während man bei der Hochzeit ohne Netz und doppelten Boden arbeitet, können Java-Programmierer den Befehl `try` in Stellung bringen und alle Ausnahmesituationen behandeln. Um genau zu sein, müssen Sie das sogar, weil der Compiler sonst rot anläuft.

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.InputMismatchException;
import java.util.NoSuchElementException;

public class Zahlenlesen {

    public static void main(String[] args) {
        java.io.File datei = new java.io.File("datei.txt");
        if (datei.exists()) {
            java.util.Scanner eingabe;
            try {
                eingabe = new java.util.Scanner(datei);
                while(eingabe.hasNext()) {
                    double zahl = eingabe.nextDouble();
                    System.out.println(zahl);
                };
            } catch (FileNotFoundException e) {
                System.out.println("Keine Datei?");
            } catch (InputMismatchException e) {
                System.out.println("Zahlenformat!");
            } catch (NoSuchElementException e) {
                // Datei ist ausgelesen
            } catch (Exception e) {
                e.printStackTrace(); // Zeige den Fehler an
            }
            if (eingabe!=null) {
                eingabe.close();
            }
        }
    }
}
```

Listing 23.4: Exception-Handling

Das Programm in Listing 23.4 liest so lange Daten, bis die Datei keine mehr hergibt. Um festzustellen, ob noch auswertbare Daten vorliegen, kann die Methode `hasNext` befragt werden. Alternativ kann das Programm einfach immer weiter Daten lesen, bis die Ausnahme `NoSuchElementException` auftritt.

Listing 23.4 zeigt die Behandlung gleich mehrerer Ausnahmen:

- ✔ Die `FileNotFoundException` tritt ein, wenn das Programm die Datei nicht

finden kann.

- Führt der eingelesene Zahlenwert zu Verdauungsstörungen bei der Methode `nextDouble`, wird sie sich mit dem Wurf einer `InputMismatchException` rächen.
- `NoSuchElementException` wird ausgelöst, wenn in der Datei kein Element mehr vorliegt, das weiterverarbeitet werden kann.
- Im letzten Fall wird die grundlegende Exception aufgefangen.

Der Scanner und seine Methoden

Der `Scanner` bietet für die verschiedenen Datentypen Methoden an, um sie mundgerecht einlesen zu können:

- `nextInt`: Der Aufruf dieser Methode wird die nächste Eingabe als ganze Zahl interpretieren und zurückgegeben.
- `nextDouble`: Die Methode liefert einen `double` zurück, wenn der nächste Wert der Datei so interpretierbar ist. Achtung: Das Dezimalzeichen richtet sich nach der Sprache des Computers. Auf deutschsprachig eingestellten Computern wird also ein Komma statt eines Dezimalpunkts erwartet.
- `next`: Diese Methode liefert den nächsten Wert, der durch Leerzeichen, Tabulatoren oder Zeilenendemarken getrennt ist, als `String` zurück.
- `nextLine`: Die Methode liefert den Rest der Zeile in einem `String`. Die Leseposition setzt auf den Anfang der nächsten Zeile auf.

Passend zum jeweiligen Aufruf zum Einlesen der Daten gibt es die Möglichkeit, vorab zu prüfen, ob ein solcher Wert in der Datei überhaupt vorliegt. Die folgenden Methoden liefern logischerweise einen Wert vom Typ `boolean` zurück:

- `hasNext`: Gibt es einen weiteren Wert?
- `hasNextInt`: Ist der nächste Wert eine ganze Zahl?
- `hasNextDouble`: Ist der nächste Wert eine gültige Fließkommazahl?



Das Auslesen von Dateien per `Scanner` ermöglicht es, mit einem Texteditor oder auch mit einem Programm Daten zu erzeugen, die von einem Java-Programm ohne großen Aufwand eingelesen werden können.

Daten in einer Textdatei

Wenn Sie die Ergebnisse Ihrer Rosa-Dackel-Forschung nicht in die Datei eintippen,

sondern durch ein Programm erzeugen wollen, benötigen Sie eine Möglichkeit, von Java aus in Dateien zu schreiben. Diese Dateien können dann wieder von anderen Programmen oder gar von Menschen gelesen werden. So könnten Sie auch ein Programm schreiben, das aus Textbausteinen Liebesbriefe erstellt. Diese Textbausteine können Sie leicht aus ein paar Vorabendserien oder Schlagern zusammenstellen.

Die abstrakte Klasse `Writer` ist die Basis für sequenzielle Ausgaben. Die Klasse `FileWriter` stellt eine Implementierung dar. Im Konstruktor kann der Pfadnamen gleich als Parameter übergeben werden. Existiert sie noch nicht, wird sie angelegt. Anschließend schreiben Sie mit der Methode `write` in die Datei. Die Methode `close` schließt die Datei nach getaner Arbeit.

Das folgende Beispiel schreibt ein paar zufällige Werte für die Rosa-Dackel-Studie. Sie erfahren bei dieser Gelegenheit, wie einfach es ist, Werte für solche Studien zu ermitteln.

```
import java.io.FileWriter;
import java.io.IOException;
import java.util.Random;

public class Zahlenschreiben {

    public static void main(String[] args) {
        Random random = new Random();
        String linefeed
                 = System.getProperty("line.separator");
        try {
            FileWriter datei = new FileWriter("datei.txt");
            for (int i=0; i<10; i++) {
                double wert = random.nextDouble();
                String strWert = "" + wert;
                strWert = strWert.replace('.', ',');
                datei.write(strWert+linefeed);
            }
            datei.close();
        } catch (IOException e) {
        }
    }
}
```

Listing 23.5: Daten in eine Textdatei schreiben

Das Zeilentrennzeichen

Eine Besonderheit ist der Zeilentrenner. Leider ist der bei jedem Betriebssystem etwas anders. UNIX und Linux verwenden das Linefeed-Zeichen (ASCII 10), das alte MacOS verwendete Carriage Return (ASCII 13) und MS-DOS/Windows eine Kombination aus beiden. Um das Trennzeichen des aktuellen Systems zu ermitteln, verwenden Sie die folgende Anweisung:

```
String linefeed = System.getProperty("line.separator");
```

Gepufferte Ausgabe

Festplatten sind blockorientierte Geräte. Beim Schreiben und Lesen wird immer ein kompletter Speicherblock übertragen. Wenn Sie wie in Listing 23.5 immer nur kleine Datenmengen schreiben, wird bei jedem Aufruf von `write` ein kompletter Block geschrieben. Das verzögert Programme ganz enorm.

Der zusätzlich eigentlich notwendige Aufwand, die Daten vor dem Speichern zu sammeln, bleibt Ihnen erspart, wenn Sie einfach einen `BufferedWriter` vor die Ausgabe in den `FileWriter` schieben. Die Methoden `write` und `close` werden dann auf den `BufferedWriter` angewendet.

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Random;

public class Zahlenschreiben {

    public static void main(String[] args) {
        Random random = new Random();
        String linefeed
                = System.getProperty("line.separator");
        try {
            FileWriter fw = new FileWriter("datei.txt");
            BufferedWriter datei = new BufferedWriter(fw);
            for (int i=0; i<10; i++) {
                double wert = random.nextDouble();
                String strWert = "" + wert;
                strWert = strWert.replace('.', ',');
                datei.write(strWert+linefeed);
            }
            datei.close();
        } catch (IOException e) {
        }
    }
}
```

Listing 23.6: Dateischreiben mit Puffer

Daten mit dem FileReader lesen

Als Gegenstück zum `FileWriter` gibt es den `FileReader`, der konsequenterweise keine Methode `write` besitzt. Sie vermuten richtig, dass es hier eine Methode `read` gibt. Die Methode liefert ein Zeichen, das allerdings in einer `int`-Variablen zurückgegeben wird. Die Methode `read` liefert den Inhalt der Datei also nicht scheibchen-, sondern zeichenweise.

```
import java.io.FileReader;
import java.io.IOException;

public class DateiLesen {
    public static void main(String[] args) {
        FileReader datei;
        int zeichen;
        try {
            datei = new FileReader("datei.txt");
            while ((zeichen = datei.read()) != -1) {
                System.out.print((char)zeichen);
            }
            datei.close();
        } catch (IOException e) {
            System.err.println("Dateilesefehler");
        }
    }
}
```

Listing 23.7: Textdatei auslesen

BufferedReader

Zeichenweises Auslesen ist alles andere als effizient. Bei größeren Dateien wird der Lesevorgang darum extrem langsam vonstattengehen. Glücklicherweise gibt es einen `BufferedReader`, der die Leseoperationen blockweise puffert. So wird nicht jedes einzelne Byte von der Festplatte übertragen, sondern erst einmal eine Wagenladung mit Daten im Hauptspeicher gesammelt, bevor die Reise zur tausendmal langsameren Festplatte losgeht. Sie können die Kombination von `FileReader` und `BufferedReader` gleich in einer Anweisung zusammenfassen.

```
BufferedReader datei
    = new BufferedReader(new FileReader("ausgabe.txt"));
```

Die Klasse `BufferedReader` stellt nun auch die Methode `readLine` für das Auslesen ganzer Zeilen zur Verfügung. Die zurückgegebene Referenz verweist auf den eingelesenen String. Gibt es keine weitere Zeile mehr, liefert die Methode null zurück.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class DateiBufferedLesen {
    public static void main(String[] args) {
        BufferedReader datei;
        String zeile;
        try {
            datei = new BufferedReader(
                    new FileReader("datei.txt"));
            while ((zeile = datei.readLine()) != null) {
                System.out.println(zeile);
            }
            datei.close();
        } catch (IOException e) {
            System.err.println("Dateilesefehler");
        }
    }
}
```

Listing 23.8: Gepuffertes Lesen

Serialisierte Klassen

Wenn Sie statt einfacher Zeilen oder Zahlen komplexe Klassen speichern und lesen möchten, wird es mit `FileWriter` und Co schnell sehr kompliziert. Für diese Zwecke bietet Java an, dass die Klassen serialisiert werden. Das bedeutet, dass alle Elemente einer Klasse in eine Reihe gestellt werden. Dann passen sie auch gut in eine Datei. Wenn Objekte serialisiert ausgegeben werden, spricht man von einem *Stream*.

Die Serialisierung von Objekten scheint zunächst nicht trivial zu sein. Immerhin kann ein Objekt andere Objekte und vielleicht auch Arrays oder Strings enthalten, die alle über Referenzen erreicht werden. Diese Referenzen greifen wie Äste aus dem Objekt auf weitere Objekte zu, die wiederum solche Äste haben können. Es entsteht ein Gestrüpp. Eine Freude für Biologen, ein Graus für Dateien. Denn diese haben einen Anfang und ein Ende und dazwischen gibt es einen geradlinigen Datenstrang. Die Objekte müssen also so angeordnet werden, dass sie in Reih und Glied in die Datei passen.

Glücklicherweise müssen Sie diese Arbeit gar nicht selbst übernehmen. Es reicht, wenn Sie Ihren Klassen erklären, dass sie bitte das Interface `Serializable`

implementieren sollen. Erfreulicherweise müssen dazu nicht einmal abstrakte Methoden implementiert werden. Allerdings müssen alle Referenzen der Klasse wiederum auf Klassen verweisen, die selbst `Serializable` implementieren. Strings und Arrays erfüllen diese Bedingung glücklicherweise von Haus aus.

Das Landkreis-Beispiel

Lassen Sie mich das Beispiel der Landkreise aus Kapitel 15 noch einmal aufwärmen. Sie erinnern sich? Die um eine Serialisierung aufgedonnerte Klasse sieht so aus:

```
import java.io.Serializable;

public class Kreis implements Serializable {
    private String wo;
    private long einwohner;
    Kreis(String w, long e) {
        wo = w; einwohner = e;
    }
    String getWo() {
        return wo;
    }
    long getEinwohner() {
        return einwohner;
    }
}
```

Listing 23.9: Serialierbarer Landkreis

Einen solchen Kreis können Sie mit dem Programm aus Listing 23.10 in der Datei *einkreis.data* ablegen.

```
import java.io.*;
public class SchreibKreis  {
    public SchreibKreis () {
        Kreis kreis = new Kreis("Schleswig-Flensburg",197903);
        try {
            FileOutputStream datei
                    = new FileOutputStream("einkreis.data");
            ObjectOutputStream out
                    = new ObjectOutputStream(datei);
            out.writeObject(kreis);
            out.close();
        } catch(IOException ex) {
            System.err.println("Schreibfehler");
        }
    }
    public static void main(String args[]) {
        new SchreibKreis ();
    }
}
```

Listing 23.10: Landkreis in eine Datei schreiben

Die Klasse `FileOutputStream` stellt die Verbindung zum Dateisystem her, die dann von der Klasse `ObjectOutputStream` genutzt wird, um die Daten zu serialisieren und zu schreiben. Die Methode `writeObject` erwartet ein `Object` und kann damit beliebige Objekte speichern. Hier zum Beispiel einen Kreis mit einer Zahl und einer String-Referenz.

Sie kennen vermutlich Murphys Gesetz: Alles, was schiefgehen kann, wird auch schiefgehen. Und damit ist klar: Die Exceptions sind auch dabei. Die Dateioperationen zeigen deutlich, dass Murphy ein Optimist war. Gerade im Umgang mit Dateien kann ja so viel Unerwartetes passieren: Die Festplatte ist voll, die Datei hatte einen etwas anderen Namen oder der USB-Stick, der eben noch da war, ist nun in der Hosentasche. All dies fängt für uns die `IOException`.

Objekte wieder einlesen

Ich nehme an, es interessiert Sie auch, wie man die Daten aus der Datei wieder in ein Objekt zurückbekommt.

```
import java.io.*;
public class LeseKreis {
    @SuppressWarnings({"unchecked"})
    public LeseKreis () {
        Kreis kreis=null;
        try {
            FileInputStream datei
                    = new FileInputStream("einkreis.data");
            ObjectInputStream in
                    = new ObjectInputStream(datei);
            kreis = (Kreis) in.readObject();
            in.close();
        } catch(ClassNotFoundException ex) {
            System.err.println("Dateistrukturfehler");
        } catch(IOException ex) {
            System.err.println("Dateilesefehler");
        }
        System.out.println(kreis.getWo() + " "
                    + kreis.getEinwohner());
    }
    public static void main(String args[]) {
        new LeseKreis ();
    }
}
```

Listing 23.11: Landkreis aus einer Datei lesen

Nachdem Sie gesehen haben, wie man einen Kreis in die Datei schreibt, birgt das Lesen sicher keine Überraschung mehr. Neu wäre da höchstens noch die `ClassNotFoundException`. Da die Methode `in.readObject` nur eine Referenz auf `Object` zurückliefert, muss sie auf `Kreis` gecastet werden. Da in der Datei auch vermerkt ist, welche Klasse das gesicherte Objekt hatte, könnte es sein, dass das lesende Programm die Daten in einem falschen Objekt ablegen will. Das würde dann zu einer solchen Ausnahme führen.

Spielstandsicherung für Bermuda

Mithilfe der Serialisierung kann das Bermuda-Programm leicht um eine Sicherung des Spielstands erweitert werden. Tatsächlich sind die Eingriffe erstaunlich gering. Als Beispiel verwenden wir hier die Umsetzung in Swing aus Kapitel 19.

Zunächst wird bei den beiden Klassen `Schiff` und `SpielFeld` jeweils `java.io.Serializable` importiert und dann deklariert, dass sie `Serializable`

implementieren.

```
import java.io.Serializable;
public class Schiff implements Serializable {
```

Listing 23.12: Schiffsserialisierung

Nun müssen die Spielstände geschrieben und gelesen werden. Wie bauen wir die Sicherung ein? Sobald der Spieler das Spiel verlässt, muss das Programm aktiv werden. Also fliegt die `setDefaultCloseOperation` raus und wir setzen einen ordentlichen `WindowsListener` auf:

```
addWindowListener(
    new WindowAdapter() {
        public void windowClosing(WindowEvent ev) {
            if (!spielEnde) { // Nur Spielunterbrechungen
                try {
                    FileOutputStream datei
                        = new FileOutputStream("bermuda.game");
                    ObjectOutputStream out
                        = new ObjectOutputStream(datei);
                    out.writeObject(sf);
                    out.close();
                } catch(Exception ex) {
                    System.out.println(ex);
                }
            } // if !spielEnde
            dispose(); // Fenster schließen
            System.exit( 0 ); // Programm verlassen
        }
    }
);
```

Listing 23.13: Spielstand sichern

Damit dürfte alles geregelt sein. Habe ich noch etwas vergessen? Aber klar! Was passiert bei einem Neustart, um den Spielstand einzulesen? Das steht in Listing 23.14. Und natürlich muss der auch im Konstruktor ausgeführt werden.

```
try {
    FileInputStream datei
            = new FileInputStream("spielstand.game");
    ObjectInputStream in
            = new ObjectInputStream(datei);
    spielfeld = (SpielFeld) in.readObject();
    in.close();
} catch(Exception ex) {
    spielfeld = new Spielfeld();
}
```

Listing 23.14: Spielstand zurückholen

Und dann muss für den Fall, dass tatsächlich mal ein Spiel beendet wird, die Datei *bermuda.game* wieder gelöscht werden. Um herauszufinden, wie man das macht, lesen Sie weiter hinten im Kapitel.

Mit diesen paar Zeilen können Sie nun das Spiel schnell beenden, wenn der Chef schaut, ob Sie nicht heimlich bei der Arbeit spielen. Sobald er weg ist, können Sie das Spiel wieder starten und dort weitermachen, wo Sie aufgehört hatten.

Properties als Konfigurationsdatei

In `Properties` lassen sich Strings assoziativ über andere Strings zugreifen. Wenn Sie darin die Autokennzeichen speichern, können Sie über »HH« auf »Hamburg« zugreifen. Das wird Sie nicht besonders beeindrucken, das können Sie mit einer einfachen `Map` aus den Java Collections, wie sie in Kapitel 15 beschrieben ist, auch erreichen. Die `Map` ist sogar flexibler, weil sie nicht auf String-Pärchen beschränkt ist.



Wie praktisch die `Properties` sind, haben auch andere Programmierer entdeckt. Und darum finden Sie diese bei verschiedenen Gelegenheiten, wenn es darum geht, Konfigurationen oder Eigenschaften zu verwalten.

Das wirklich Praktische an `Properties` ist es aber, dass Sie mit einem Aufruf alle Pärchen in eine Textdatei speichern können oder eben eine solche Textdatei einlesen können. Damit eignen sich die `Properties` ideal für die Verwaltung von Konfigurationdateien von Programmen.

`Properties` importieren Sie aus `java.util` und legen Sie per `new` an.

```
import java.util.Properties;
// ...
Properties prop = new Properties();
```

Auf Strings über Strings zugreifen

Für den Zugriff auf die Einträge in `Properties` stehen folgende Methoden zur Verfügung:

- ✔ `setProperty(schluessel, wert)`

  Es wird ein Wert unter einem Schlüssel abgelegt.
- ✔ `wert = getProperty(schluessel)`

  Es wird der Wert zurückgegeben, der für den Schlüssel hinterlegt ist. Existiert der Schlüssel nicht, gibt die Methode `null` zurück.

  ```
  String inhalt = prop.getProperty("Farbe");
  ```
- ✔ `wert = getProperty(schluessel, vorgabe)`

  Damit der Rückgabewert nicht `null` wird, kann ein Vorgabe-String als zweiter Parameter angegeben werden. Gibt es den Schlüssel nicht, wird dieser Parameter zurückgegeben.

  ```
  String inhalt = prop.getProperty("Farbe", "grau");
  ```

Andere Typen als String

`Properties` bieten nur Strings als Schlüssel und Werte. Es lassen sich aber natürlich auch auf den bekannten Wegen andere Typen verwenden, die in Strings oder aus Strings konvertiert werden können.

```
String strZahl = prop.getProperty("Breite");
double breite = 0;
try {
    if (strZahl!=null) { // gar nicht vorhanden?
        breite = Double.parseDouble(strZahl);
        System.out.println("Wert: "+breite);
    }
} catch (NumberFormatException e) {
    System.out.println("Zahlenformat verkehrt");
}
```

Sie können Werte mit der Methode `setProperty` einspeisen. `Properties` nehmen zwar nur Zeichenketten auf, aber durch Umwandlung ist aus einer Zahl schnell ein String erzeugt. Hier wird die Breite auf 12 gesetzt.

```
String inhalt = "" + 12;
prop.setProperty("Breite", inhalt);
```

Listing 23.15 zeigt, wie Sie alle Elemente aus den `Properties` auslesen können. Dazu rufen Sie mit der Methode `keySet()` das Set, das alle Schlüssel enthält. Darüber können Sie eine Schleife drehen und so die Inhalte ausgeben. Sie müssen

allerdings das `Object` zu einem `String` casten.

```
import java.util.Properties;
public class Props {

    public static void main(String[] args) {
        Properties prop = new Properties();
        prop.setProperty("user", "isaac");
        prop.setProperty("name", "Isaac Newton");
        prop.setProperty("email", "isaac@newton.de");
        for (Object obj : prop.keySet()) {
            String key = (String) obj;
            System.out.println(key + ": "
                    + props.getProperty(key));
        }
    }
}
```

Listing 23.15: über Schlüssel auf die Werte zugreifen

Properties als Konfigurationsdatei

Nun kommen wir zum interessantesten Punkt: Sie können die `Properties` einfach in eine Datei schreiben. Dazu dient die Methode `store`, die einen Output-Stream als Parameter erwartet. Hier können wir den bereits bekannten `FileOutputStream` einsetzen. Ein paar Exceptions gehören bekanntlich beim Umgang mit Dateien dazu. So können Sie die aktuelle Konfiguration eines Programms einfach als Datei speichern.

```
public static void schreibeProps(Properties prop, String name) {
    try {
        FileOutputStream fos = new FileOutputStream(name);
        props.store(fos, "");
        fos.close();
    } catch (FileNotFoundException e) {
    } catch (IOException e) {
    }
}
```

Listing 23.16: Schreiben der `Properties`

Nun entsteht eine Datei, die alle Werte des `Properties`-Objekts enthält.

```
#
#Sat Mar 18 20:44:49 CET 2017
user=isaac
email=isaac@newton.de
name=Isaac Newton
```

Datei einlesen

Nun können Sie mit einem einfachen Text-Editor eine weitere Zeile hinzufügen, die dann beim Lesen genauso ausgewertet wird wie die anderen Zeilen.

```
public static Properties leseProps(String name) {
    Properties prop = new Properties();
    try {
        FileInputStream fos = new FileInputStream(name);
        prop.load(fos);
        fos.close();
    } catch (FileNotFoundException e) {
    } catch (IOException e) {
    }
    return prop;
}
```

Listing 23.17: Lesen einer `Properties`-Datei

Random-Access-Dateizugriffe

Wie bei einem Array oder in einer Map möchte man manchmal wahlfrei auf einzelne Elemente in einer Datei zugreifen. Beispielsweise werden Sie das Telefonbuch nicht komplett durchlesen, nur um die Nummer von Herrn Zuckermeier zu ermitteln.

Für solch ein Telefonbuch werden Sie eine Datei anlegen, in der für jeden Telefonbesitzer ein Datenblock gleicher Größe reserviert wird. Nehmen wir an, dass Sie 200 Bytes pro Teilnehmer benötigen. Um den 128. Block zu holen, müssen Sie nur auf die Position 25.600 zugreifen und die nächsten 200 Bytes einlesen. Sie haben also drei wichtige Funktionen:

- ✔ `seek`: Rücke zu einer bestimmten Position innerhalb einer Datei.
- ✔ `read`: Lese einen Datenblock an der aktuellen Position.
- ✔ `write`: Schreibe einen Datenblock an der aktuellen Position.

Damit haben Sie die Basiselemente für einen wahlfreien Zugriff gelegt. Man spricht auch von einem *Random-Access-Zugriff*.

Byte-Array statt Klassen

Leider können Sie nicht einfach eine Klasse definieren und die Elemente per wahlfreiem Zugriff verwenden, da die Größe der meisten Objekte nicht starr ist. Sie benötigen einen immer gleich großen Datenblock und darum erwartet `RandomAccessFile` für diese Zwecke ein Array von Byte-Werten.


Wenn Sie in erster Linie wahlfreien Zugriff für Anwendungsprogramme benötigen, sollten Sie sich die Datenbanken anschauen. Sie sind Spezialisten in diesem Themenbereich.

Datenblöcke lesen und schreiben

Das Programm in Listing 23.18 erzeugt eine Datei, die mit 16 Byte großen Blöcken arbeitet. In jeden Block werden andere Buchstaben geschrieben. Im ersten Block an Position 0 stehen lauter As, im zweiten an Position 1 lauter Bs und so fort. Auf diese Weise können Sie sich die Datei später leichter anschauen.

1. Erstellen Sie in Eclipse ein neues Java-Projekt.
2. Legen Sie eine neue Klasse an und nennen Sie sie beispielsweise `RandomAccess`.
3. Geben Sie das Programm in Listing 23.18 ein.
4. Starten Sie das Programm.
5. Klicken Sie das Projekt mit der Maus an und drücken Sie die Taste F5.
6. Im Projekt erscheint die Datei *test.data*.
7. Klicken Sie sie doppelt an.

Anschließend sehen Sie im Editor die Buchstabenblöcke.

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomAccess {

    public static void main(String[] args) {
        final int SIZE = 16;
        final int OFFSET = 0;
        final int ANZAHL = 5;
        byte[] testDaten = new byte[SIZE];
        byte[] leseDaten = new byte[SIZE];

        try {
            RandomAccessFile datei
                = new RandomAccessFile("test.data", "rw");
            for (int i=0; i<ANZAHL; i++) {
                for (int j=0; j<SIZE; j++) {
                    testDaten[j] = (byte)('A'+i);
                }
                datei.seek(i*SIZE);
                datei.write(testDaten);
            }
            // Als Beispiel hole dir den vierten Block:
            datei.seek(3*SIZE);
            datei.read(leseDaten);
            datei.close();
            System.out.println((char)leseDaten[0]);
        } catch(FileNotFoundException e) {
        } catch(IOException e) {
        }
    }
}
```

Listing 23.18: Umgang mit einer Random-Access-Datei

Zunächst legt das Programm ein Objekt der Klasse `RandomAccessFile` an. Der Dateiname wird dem Konstruktor übergeben. Ein zweiter Parameter enthält einen String mit den gewünschten Schreib- und Leserecht. Ein r steht für read und ein w für write.

Die Klasse `RandomAccessFile` stellt die folgenden Methoden zur Verfügung:

- Die Methode `read` liest einen Datenblock. Der Parameter ist der Speicherbereich, der gelesen oder geschrieben werden soll. Er besteht für gewöhnlich aus einem Array von Bytes.
- Die Methode `write` schreibt einen Datenblock und verwendet genau wie `read` typischerweise ein Array von Bytes als Parameter.
- Die Methode `seek` positioniert den Dateizeiger. Damit wird die Stelle in der Datei angefahren, ab der `read` und `write` lesen und schreiben.
- Wenn man die Datei nicht mehr braucht, macht man sie ordentlich zu, damit kein Schmutz hineinkommt. Dazu verwenden Sie die Methode `close`.

Auslesen einer fremden Dateistruktur

Als Beispiel will ich in den alten Zeiten von MS-DOS und CP/M herumkramen. Damals gabe es dBase, ein sehr einfaches Datenbanksystem. Eine Datei entsprach einer Datenbanktabelle. Eine dBase-Datei endet namentlich auf *.dbf*. Innerlich beginnt sie mit einem Header, in dem die Struktur der Tabelle abgelegt wird. Es folgen in gleich großen Blöcken die eigentlichen Daten.

Erzeugen einer dBase-Datei

Falls Sie nicht zufällig noch aus alten Zeiten eine dBase-Datei irgendwo herumfliegen haben, können Sie leicht eine erzeugen. Die meisten Tabellenkalkulationen sind in der Lage, ihre Tabellen als dBase-Dateien zu schreiben. Für das Beispiel verwenden wir LibreOffice. Das ist kostenlos und besitzt eine tadellose Tabellenkalkulation. Sie können sie unter `de.libreoffice.org` herunterladen.

1. Starten Sie die Tabellenkalkulation von LibreOffice.
2. Geben Sie die Highscore-Tabelle aus Abbildung 23.1 ein.

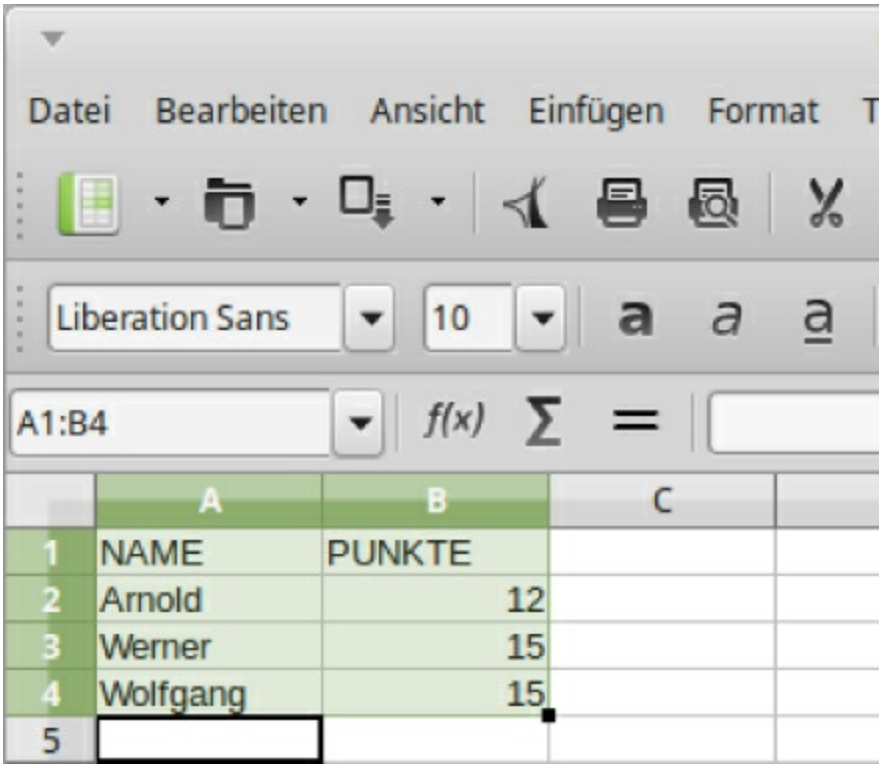


Abbildung 23.1: LibreOffice Calc erzeugt eine dBase-Datei.

3. Markieren Sie alle gefüllten Felder.
4. Wählen Sie im Menü Daten|Bereich Festlegen.
5. Wählen Sie im Menü Datei|Speichern Unter.
6. Unten im Dialog finden Sie eine Combobox mit der Beschriftung Alle Formate. Wählen Sie daraus Dbase (.Dbf) aus.
7. Als Name verwenden Sie *test*.
8. LibreOffice fragt, ob es weiterhin das dBase-Format verwenden soll. Das ist eigentlich egal. Die Datei wurde erzeugt und mehr wollen wir im Moment gar nicht.

Aufbau der Struktur

Mit dieser Datenstruktur wollen wir ein wenig spielen. Zunächst liest das Beispielprogramm in Listing 23.19 den ersten Block der Datei ein. Er ist immer 16 Byte lang. In den ersten Bytes befinden sich die Informationen, die Sie in Tabelle 23.1 sehen.

Position	Inhalt
Bytes 4-7	Anzahl der Zeilen der Tabelle
Bytes 8-9	Offset, an dem die Daten anfangen
Bytes 10-11	Die Größe eines Datensatzes

Tabelle 23.1: Der erste Block einer dBase-Datei

Die Positionen beginnen bei 0 und die Bytes sind Little-Endian-codiert. Das

kleinstwertige Byte kommt also zuerst.

Um den Header zu lesen, muss zunächst die Datenstruktur im Hauptspeicher angelegt werden. Sie bestimmt, wie groß der zu lesende Block ist. Wir erreichen das durch ein Array von `byte` mit 16 Elementen. Wird darauf `read` oder `write` angewandt, wird ein entsprechend großer Block gelesen oder geschrieben.

```
import java.io.RandomAccessFile;
import java.io.FileNotFoundException;
import java.io.IOException;

public class DBase {

    static final int DBFHEADER = 16;
    public static int getBetrag(byte wert) {
       return (wert < 0 ? -wert : wert);
    }

    static int getLittleEndian(byte[] data, int offset, int size) {
        int wert = 0;
        for (int i=size-1; i>=0; i--) {
            wert *= 256;
            wert += getBetrag(data[offset+i]);
        }
        return wert;
    }

    static public void main(String[] args) {
        // Speicherblock für den dBase-Header-Block
        byte [] dbfHeader = new byte[DBFHEADER];

        try {
            // Lies den dBase-Header ein
            RandomAccessFile datei
                = new RandomAccessFile("test.dbf", "r");
            datei.read(dbfHeader);
            datei.close();
        } catch(FileNotFoundException e) {
            System.out.println("Datei nicht vorhanden.");
            System.exit(1);
        } catch(IOException e) {
            System.out.println("Dateifehler.");
            System.exit(2);
        }

        // Wie viele Daten stehen in der DBF-Datei?
        int recordZahl = getLittleEndian(dbfHeader, 4, 4);
        System.out.println("Anzahl der Records: " + recordZahl);
        // Wo beginnt der Datenbereich in der Datei?
        int offset = getLittleEndian(dbfHeader, 8, 2);
        System.out.println("Offset: " + offset);

        // Wie groß ist jeder Datenblock?
        int recordGroesse = getLittleEndian(dbfHeader, 10, 2);
```

```
        System.out.println("Record-Größe: " + recordGroesse);
    }
}
```

Listing 23.19: dBase-Dateien auslesen

Ein paar kleine Hilfsmethoden erleichtern den Umgang mit dem Byte-Array:

- Die Methode `getBetrag` sorgt dafür, dass aus den Bytes Werte ohne Vorzeichen gelesen werden können. Der normale Typ `byte` ist in Java vorzeichenbehaftet, was in dieser Anwendung sehr störend ist.
- Die Methode `getLittleEndian` liest aus einem Byte-Array einen Wert, der mehrere Bytes umfasst, und zwar im »Little-Endian«-Format. Das bedeutet, dass das niederstwertige Byte zuerst kommt.

So wird der Datenoffset, die Recordgröße und die Anzahl der Records ermittelt. Mit diesen Angaben kann im Datenbereich der Datei positioniert werden. Da in dBase-Dateien alle Daten in ASCII abgelegt werden, also auch die Zahlen, kann man die Sätze einfach per Casting auf `char` ausgeben. Allerdings passen die Umlaute nicht ganz, weil MS-DOS einen anderen Code verwendete als UTF. Aber das ist eine andere Baustelle ...

```
static void leseRecords(long recordZahl,
                        long recordGroesse, long offset) {
    try {
        byte[] dbfRecord = new byte[(int)recordGroesse];
        RandomAccessFile datei
                = new RandomAccessFile("test.dbf", "rw");
        for (int i=0; i<recordZahl; i++) {
            datei.seek(i*recordGroesse+offset);
            datei.read(dbfRecord);
            for (int j=0; j<recordGroesse; j++) {
                System.out.print((char)dbfRecord[j]);
            }
        }
        datei.close();
    } catch(FileNotFoundException e) {
        System.out.println("Datei nicht vorhanden.");
    } catch(IOException e) {
        System.out.println("Dateifehler.");
    }
}
```

Listing 23.20: Auslesen der Blöcke der dBase-Datei

Das Zerlegen der Datensätze

Wer sich näher für die dBase-Struktur interessiert: Ab dem Byte 32 beginnen die jeweils 32 Byte langen Spaltendefinitionen. Dabei stehen in jeder Spaltendefinition die Informationen, die Sie im Folgenden aufgeführt sehen:

- Position 0–11: Der Namen des Feldes
- Position 12: Typkennzeichen: (‘C’, ‘N’, ‘D’, ‘L’, ‘M’)
  - C: Zeichenfeld
  - N: Numerisches Feld
  - D: Datumsfeld
  - L: Logisches Feld, also boolean
  - M: Memofelder
- Position 13–14: Länge des Feldes
- Position 15–16: Davon Dezimalstellen (nur bei Feldtyp ‘D’)
- Position 17–18: Offset des Feldes innerhalb des Records

Mit diesen Informationen dürfte es also kein Problem sein, die Daten in den dBase-Dateien zu zerlegen und spaltenweise anzuzeigen.

Umgang mit dem Dateisystem

Sie können mit Java auch im Dateisystem herumpfuschen. Sie können Dateien umbenennen, heimlich löschen, durch Verzeichnisse wechseln oder Verzeichnisse anlegen.

Zugriff per File

Für die Dateisystemoperationen stellt Java die Klasse `File` zur Verfügung. Diese Klasse befindet sich im Package `java.io`. Dem Konstruktor wird der Dateiname als Parameter übergeben. Nun können die Methoden von `File` aufgerufen werden. Ein kleines Beispiel zeigt, wie ein Verzeichnis mit der Methode `mkdir` angelegt wird.

```
import java.io.File;
public class MachDir {
    public static void main(String[] args) {
        File verzeichnis = new File("testdir");
        verzeichnis.mkdir();
    }
}
```

Listing 23.21: Ein Verzeichnis anlegen

Der Konstruktor von `File` kann auch zwei Strings erhalten, um Pfad- und Dateiname getrennt zu übergeben.

Es folgen ein paar Funktionalitäten, die nicht wirklich spannend sind, aber einfach in die Werkzeugkiste eines Programmierers gehören. Sie werden vielleicht einmal froh sein, sie hier zu finden, wenn Sie sie einmal benötigen.

- `mkdir`: Die Methode `mkdir` legt ein Verzeichnis unter dem Namen des `File`-Objekts an. Der Rückgabewert ist `true`, wenn das Verzeichnis erzeugt werden konnte.
```
File verzeichnis = new File("testdir");
boolean wurdeErzeugt = verzeichnis.mkdir();
```
- `mkdirs`: Die Methode `mkdirs` erzeugt einen kompletten Pfad. Bei Aufruf der Methode `mkdir` muss das Elternverzeichnis bereits existieren. Dagegen erzeugt `mkdirs` alle benötigten Elternverzeichnisse automatisch.
```
File verzeichnis = new File("test/dir/komplett");
boolean wurdeErzeugt = verzeichnis.mkdirs();
```
- `renameTo`: Die Methode benennt eine Datei oder ein Verzeichnis um. Dazu benötigt sie ein Objekt von `File` mit dem neuen Namen.
```
File verzeichnis = new File("test/dir/komplett");
verzeichnis.mkdirs();
File neuName = new File("test/dir/huhu");
boolean wurdeErzeugt = verzeichnis.renameTo(neuName);
```
- Die Methode `delete` löscht die Datei oder das Verzeichnis, für das das Objekt von `File` steht.
```
boolean geloescht = datei.delete();
```
- Namensermitlung: Sie können auf vielfältige Art den Namen einer Datei oder eines Verzeichnisses ermitteln:
  - `getName`: liefert den Namen.
  - `getPath`: liefert den Namen inklusive des relativen Pfades.
  - `getAbsolutePath`: liefert den Namen inklusive des absoluten Pfades.
  - `getParent`: liefert den Pfad des Elternverzeichnisses.
```
System.out.println(verzeichnis.getName());
System.out.println(verzeichnis.getPath());
System.out.println(verzeichnis.getAbsolutePath());
System.out.println(verzeichnis.getParent());
```
- Die folgenden Methoden geben boolesche Informationen einer Datei preis:
  - `exists`: Gibt es diese Datei oder dieses Verzeichnis überhaupt?

- `isFile`: Handelt es sich um eine Datei?
- `isDirectory`: Handelt es sich um ein Verzeichnis?
- `canWrite`: Darf die Datei geschrieben werden?
- `canRead`: Darf die Datei gelesen werden?
- `isHidden`: Ist die Datei verborgen?
- `isAbsolute`: Wurde bei der Erzeugung der Datei ein absoluter Pfad verwendet?

- Die Methode `lastModified` liefert einen Wert vom Typ `long`, der den Zeitpunkt der letzten Änderung in Millisekunden seit dem 1.1.1970 darstellt. Die Klasse `Calendar` hilft bei der Umrechnung in Datum und Uhrzeit.
- Mit der Methode `length` ermitteln Sie die Größe der Datei.
- Die Methode `list` liefert, auf ein Verzeichnis angewendet, die Namen aller Dateien, die sich in dem Verzeichnis befinden. Das Beispiel liefert die Namen aller Dateien und Verzeichnisse in meinem Heimatverzeichnis auf meinem Linux-Computer.

```
File verzeichnis = new File("/home/arnold");
String[] dateien = verzeichnis.list();
for (String name : dateien) {
    System.out.println(name);
}
```

- Die Methode `listFiles` ist der Methode `list` nicht unähnlich. Auch sie liefert den Inhalt eines Verzeichnisses. Im Gegensatz zu `list` liefert `listFiles` allerdings ein Array von `File`-Objekten zurück. Das ist in vielen Fällen sehr viel praktischer.

  Optional können Sie der Methode `listFiles` als Parameter ein Objekt der Klasse `FileFilter` übergeben. Wenn Sie einen eigenen `FileFilter` erstellen wollen, müssen Sie die Methode `accept` überschreiben. Sie erhält als Parameter jede gefundene Datei als `File`. Wenn Sie diese Datei in der Liste aufnehmen wollen, geben Sie `true` zurück, ansonsten `false`.

```
File dir = new File(".");
File[] dateiliste = dir.listFiles(
    new FileFilter() {
        @Override
        public boolean accept(File datei) {
            return true;
        }
    }
);
```

Verzeichnisse auslesen

Ein Verzeichnis enthält ein ganzes Rudel von Dateien. Und es gibt immer wieder Anlässe, um so ein Verzeichnis auslesen und die darin enthaltenen Dateien anschauen zu wollen. Das klingt nach einer einfachen Schleife, einer Fingerübung. Legen wir los!

Wir lesen ein Verzeichnis aus, indem wir ein Objekt der Klasse `File` anlegen. Die Methode `listFiles` liefert ein Array von `File` zurück, das die Dateien des Verzeichnisses enthält. Ja, ich gebe zu: Das ist noch nicht so spannend.

```
import java.io.File;
import java.io.FileFilter;

public class SimpleDir {

    public static void main(String[] args) {
        File dir = new File(".");
        File[] dateiliste = dir.listFiles();
        for (File datei : dateiliste) {
            System.out.println(datei.getName());
        }
    }
}
```

Listing 23.22: Verzeichnis auslesen

Verzeichnisse rekursiv auslesen

Verzeichnisse können außer Dateien natürlich auch wieder Verzeichnisse enthalten. Und wenn man Verzeichnisse wirklich vollständig durchlaufen will, muss man auch die darin enthaltenen Verzeichnisse und deren Unterverzeichnisse durchsuchen.

Wenn die Methode `durchsuche` ein Verzeichnis ausliest und dort wiederum auf ein Verzeichnis stößt, ruft sie wiederum die Methode `durchsuche` auf. Das funktioniert tatsächlich! Diesen Selbstaufruf nennt man *Rekursion*. Sie erinnern sich? In Kapitel 5 hatten Sie noch überlegt, was ich mir für einen hirnverbrannten Quatsch aushecke, als Methoden sich selbst aufriefen. Und jetzt verrate ich Ihnen: Sie werden wohl kaum eine effizientere Art finden, eine Verzeichnisstruktur zu durchlaufen.

```
import java.io.File;

public class RekursivDir {

    public static void main(String[] args) {
        File dir = new File(".");
        durchsuche(dir);
    }

    static private int tiefe = 0;

    static void durchsuche(File dir) {
        if (dir.isDirectory()) {
            tiefe++;
            zeige(dir, tiefe);
            File[] liste = dir.listFiles();
            for (File datei : liste) {
                if (datei.isDirectory()) {
                    durchsuche(datei);
                } else {
                    zeige(datei, tiefe+1);
                }
            }
        }
        // Verzeichnis ausgelesen. Tiefe reduzieren
        tiefe--;
    }

    private static void zeige(File dir, int tiefe) {
        for (int i=0; i<tiefe; i++) {
            System.out.print("  ");
        }
        System.out.println(dir.getName());
    }
}
```

Listing 23.23: Rekursives Auslesen eines Verzeichnisses

Das Hauptprogramm erzeugt ein Objekt der Klasse `File` für das aktuelle Verzeichnis, das bei allen Betriebssystemen einfach nur . heißt. Damit ruft es die rekursive Methode `durchsuche` auf.

Diese prüft den Parameter und prüft, ob es ein Verzeichnis ist. In diesem Fall liest

es dessen Kinder aus. Ist eines davon wieder ein Kind, ruft `durchsuche` sich selbst auf. Ansonsten zeigt es die Datei. Dabei verwendet es die Variable `tiefe`, um eine Einrückung zu erstellen. Das Ergebnis sieht beispielsweise so aus:

```
java ListDir k18
  k18
    Flaschen
    strings.xml
    MainActivity.java
    activity_main.xml
  GrafikView.java
    BermudaImage
      res
        drawable
        mark4.png
        schiff.png
        mark3.png
        mark5.png
        leer.png
        mark6.png
        mark2.png
        mark8.png
        mark0.png
        mark7.png
        mark1.png
    BermudaView.java
    BermudaText
    BermudaView.java
    MainActivity.java
```

Mit dem Visitor durch die Verzeichnisse

Es kann verschiedene Gründe geben, ein Verzeichnis auszulesen. Beispielsweise tun dies einige Leute, weil sie in einem Java-Buch der Aufforderung des Autors folgen. Im Beispiel oben haben wir die Dateinamen gelesen und in einem Baumdiagramm angezeigt.

In anderen Fällen kann es sein, dass man den Speicherbedarf für das Verzeichnis ermitteln will. Dazu müsste man jede enthaltene Datei nach ihrer Größe fragen. Wenn Sie sich den Befehl `find` unter UNIX oder Linux ansehen, werden Sie sehen, was man mit einem Verzeichnisbaum alles anstellen kann.

Anstatt nun jedes Mal das Programm neu zu schreiben, ist es eleganter, einen *Visitor* (Besucher) zu bauen. Ein solcher Besucher folgt dem Programm und besucht die Datenstruktur beim Betreten, beim Durchlaufen und beim Verlassen jedes Verzeichnisses. Was der Besucher im Einzelnen dort tut, legt die Programmierung des Visitors fest. Der Visitor gehört zu den Entwurfsmustern, die auch als *design patterns* bekannt sind.

Wir legen den Besucher ganz abstrakt aus und erstellen das Interface eines Visitors.

```
import java.io.File

public interface DirVisitor {
    public void betreten(File dir);
    public void verlassen(File dir);
    public void besuchen(File datei);
}
```

Listing 23.24: Das Interface des Visitors

Nun schreiben wir eine Implementierung. Der Einfachheit halber verwenden wir zunächst das, was wir schon haben, nämlich die Ausgabe der Dateinamen.

```
import java.io.File;

public class DirAnzeigeVisitor implements DirVisitor {
    int tiefe;

    @Override
    public void betreten(File dir) {
        tiefe++;
        zeigName(dir.getName(), tiefe);
    }

    @Override
    public void verlassen(File dir) {
        tiefe--;
    }

    @Override
    public void besuchen(File datei) {
        zeigName(datei.getName(), tiefe);
    }

    private void zeigName(String name, int tiefe) {
        for (int i=0; i<tiefe; i++) {
            System.out.print("  ");
        }
        System.out.println(name);
    }
}
```

Listing 23.25: Implementierung als Anzeige-Visitor

Nun muss das bisherige Programm auf die Verwendung des Visitors angepasst werden. Die Hauptproblematik besteht darin, den Visitor durch die Parameter bei der Rekursion zu schleifen.

```
import java.io.File;

public class DirMainVisitor {

    public static void main(String[] args) {
        File file = new File(".");
        durchsuche(file, new DirAnzeigeVisitor());
    }

    static void durchsuche(File dir, DirVisitor visitor) {
        // Ist der übergebene Name ein Verzeichnis?
        if (dir.isDirectory()) {
            visitor.betreten(dir);
            File[] liste = dir.listFiles();
            for (File datei : liste) {
                if (datei.isDirectory()) {
                    // Verzeichnis! Aufruf der Rekursion!
                    durchsuche(datei, visitor);
                } else {
                    visitor.besuchen(datei);
                }
            }
            visitor.verlassen(dir);
        }
    }
}
```

Listing 23.26: Verzeichnis auslesen mit Visitor

Das Programm ist nicht sehr viel länger geworden, aber tut immer noch dasselbe. Dafür hätten Sie sich den Stress also gar nicht machen müssen. Interessant ist aber, was passiert, wenn der Visitor eine andere Aufgabe erledigen soll. Änderungen der Aufgabenstellung beziehen sich nun nämlich einzig und allein auf den Visitor. Wenn wir beispielsweise wissen wollen, wie viele Bytes ein Verzeichnisbaum belegt, wird ein Visitor geschrieben, der beim Besuch jeder Datei deren Methode `length` aufruft und die Ergebnisse summiert.

Ein Visitor ermittelt die Verzeichnisgröße

Nehmen wir die Vorlage und passen sie ein wenig an. Zunächst bauen wir eine

neue Visitor-Klasse. Wir brauchen ein Attribut, das die Größe speichert. Wir nennen sie einfach `groesse`. Beim Betreten und Verlassen des Verzeichnisses muss der Visitor nichts tun. Während des Besuchs ermittelt er mit der Methode `length` die Größe der übergebenen Datei und summiert sie auf das Attribut `groesse`.

```
import java.io.File;

public class DirGroesseVisitor implements DirVisitor {
    private long groesse = 0;

    @Override
    public void betreten(File dir) {
    }

    @Override
    public void verlassen(File dir) {
    }

    @Override
    public void besuchen(File datei) {
        groesse += datei.length();
    }

    public long getGroesse() {
        return groesse;
    }
}
```

Listing 23.27: Ein Visitor zur Größenermittlung

Zum Schluss brauchen Sie noch eine Methode, um die Summe zu ermitteln. Dazu schreiben Sie eine `get`-Methode für das Attribut `groesse`. Ja, und damit ist der Besucher schon fertig. Nun müssen wir das Hauptprogramm anpassen. Die `main`-Methode muss um die Ausgabe der Gesamtgröße ergänzt werden.

```
public static void main(String[] args) {
    File file = new File(".");
    durchsuche(file, new DirAnzeigeVisitor());
    DirGroesseVisitor visitor = new DirGroesseVisitor();
    durchsuche(file, visitor);
    System.out.println(visitor.getGroesse());
}
```

Aufgaben

Kreis-Datei

Langweilig? Dann sollten Sie dringend selbst programmieren. Schreiben Sie doch ein Programm, das Landkreise wie in Listing 15.12 in eine Datei schreibt und später wieder einliest.

Kombinieren Sie das doch mit einer `ArrayList`, die in Kapitel 15 beschrieben ist, und versuchen Sie einmal, ob Sie ein komplettes `ArrayList`-Objekt speichern und lesen können.

Highscore

Mit Dateien ist es möglich, dass Ihre Spiele einen Highscore verwalten, der nicht bei jedem Spielende verloren geht. Ihre Aufgabe besteht darin, mindestens drei Spieler mit je einem Spielstand (ganze Zahlen) und seinem Namen (String) in einer Datei zu speichern und diese auch wieder zu laden und anzuzeigen. Welche Technik Sie dazu verwenden, bleibt Ihnen überlassen. Optional können Sie den Highscore in Bermuda einbauen. Je weniger Versuche der Spieler benötigt, desto besser.

Kapitel 24

Datenbankzugriff aus Java

IN DIESEM KAPITEL

Die Bonbons in der Derby-Datenbank

Mit JDBC gegen die Datenbanken

Transaktionen bei Banken und bei Datenbanken

Klassen statt Tabellen

Wenn Ihre Programme nicht nur in den Tag hineinleben, sondern Daten für die Ewigkeit oder zumindest bis zur nächsten Eiszeit aufbewahren sollen, dann benötigen Sie Dateien. Deren Anwendung haben Sie in Kapitel 23 bereits kennengelernt. Bei größeren Programmen haben Sie es mit strukturierten Daten zu tun. Und da ist es fast immer sinnvoller, die Speicherung der Daten denjenigen in die Schuhe zu schieben, die sich damit wirklich auskennen. Und das sind Datenbanken.

Für die Kommunikation mit Datenbanken stellt Java die Standardschnittstelle *Java Database Connectivity (JDBC)* zur Verfügung. Damit ist es möglich, die Datenbanken unterschiedlicher Hersteller gleichartig zu bearbeiten.

Über JDBC können Sie Datenbankbefehle an die Datenbanken senden. Die Java Persistence API (JPA) macht es möglich, die Datenbanksprache zu ignorieren und mit Klassen Datenbanktabellen zu modellieren.

Datenbanken und ihre Tabellen

Datenbanken basieren in den meisten Fällen auf Tabellen. Man spricht vom *Relationalen Datenmodell*. Das bedeutet, dass die Tabellen miteinander so in Beziehung gesetzt werden, dass man damit die Wirklichkeit modellieren kann. Der Grundgedanke wird Ihnen bekannt vorkommen. So etwas Ähnliches habe ich Ihnen bereits im Zusammenhang mit der objektorientierten Programmierung erzählt. Und ja, wir werden beides irgendwie miteinander kombinieren müssen.

Tabellen für Bestände

Jeder Kaufmann hat einen Warenbestand und wird diesen irgendwie erfassen wollen. Ohne Computer würde der Kaufmann eine Tabelle erstellen, die mehrere Spalten erhielte. Da könnte die Artikelnummer, die Bezeichnung, der

Einkaufspreis, der Verkaufspreis und die derzeitige Lagermenge drinstehen.

Schon allein, um seine Werbung an die richtigen Leute zu senden, wird er vielleicht auch eine Tabelle mit seinem Kundenbestand haben. Jeder Kunde, der etwas auf sich hält, besitzt eine Kundennummer. Aber natürlich hat er auch einen Namen und eine Adresse.

Tabellen zeigen auch Beziehungen

Beide Tabellen stehen in Relation zueinander. Ein Kunde kauft Waren. Jeder kann beliebig viele Waren kaufen. Die Anzahl der Waren, die ein Kunde kauft, könnte der Variablen n entsprechen. Und jede Ware kann an verschiedene Kunden verkauft werden, beispielsweise in der Variablen m erfasst. Die Leute, die mit Datenbanken umgehen, nennen so etwas eine n:m-Beziehung und verwenden, um dies darzustellen, wieder eine Tabelle. In dieser Tabelle wird jeder Kauf gespeichert. Um darzustellen, wer was kauft, muss man nur die Kundennummer und die Artikelnummer eintragen. Wenn nun noch die Menge, der Preis und das Datum hineinkommen, hat der Kaufmann alle Informationen zusammen.

Schlüssel als Bezeichner

Eine Artikelnummer bezeichnet eindeutig einen Artikel. Die Artikelnummer wird darum als *primärer Schlüssel* für die Warentabelle bezeichnet. Genauso ist die Kundennummer der primäre Schlüssel für die Kundentabelle. Die Verkaufstabelle verwendet beide Schlüssel zur Identifizierung. Dort werden sie *Sekundärschlüssel* genannt.

Datenbanksprache SQL

Die Datenbankexperten sind mit der Sprache SQL gestraft, die sich seit den 1960er Jahren immer mehr zum Standard mauserte. Bevor Sie hämisch lachen, sollten Sie wissen, dass SQL auch die Basis für die JDBC-Schnittstelle von Java ist, und Sie wohl auch nicht davon verschont bleiben. Es ist aber nicht so dramatisch. Die Befehle werden Sie im Zuge der Beispiele kennenlernen.

Eine Datenbank für die Bonbonbude

Sie werden es vielleicht schon der einschlägigen Literatur entnommen haben: Superman fliegt nicht mehr. Stattdessen hat er einen Kiosk am Hauptbahnhof von Norgaardholz eröffnet. Die ganzen Heldentaten waren finanziell gesehen ein Desaster und irgendwann bekommen auch Superhelden Superzipperlein und dann ist die eigene Rente näher als die Rettung der Welt.

Nun verkauft er Süßigkeiten und Comic-Hefte. Der Plunder kostet nichts im Einkauf und hat eine irre Gewinnspanne. Er muss nur zusehen, dass sich die Bude in der Nähe einer Schule befindet, und alles ist geritzt. Leider hat Superman an dieser Stelle nicht aufgepasst, aber der alte Kumpel Batman kauft seine Lakritze

nun nur noch bei ihm und auch Spiderman unterstützt ihn, wo immer er kann.

Seine Süßigkeiten hat Superman bisher in einem Rechenheft verwaltet. Darin vermerkte er für jeden seiner Verkaufsschlager eine Zeile mit dem Namen des Artikels, dem Preis und der Menge, die noch auf Lager ist (siehe Tabelle 24.1).

Artikel	Preis	Menge
Knoblauch-Nougat-Riegel	1,50	150
Lakritzschimmelbonbons	0,05	494
Kümmelsenflutscher	0,25	76

Tabelle 24.1: Eine Liste für die Artikel

Durch den gezielten Einsatz von Bleistift und Radiergummi wird der aktuelle Stand verwaltet. Das Papier wird allerdings langsam dünner. Superman braucht also eine Alternative. Papier ist teuer, Einen Computer gab es bei eBay als billigen Leasingausläufer. Und Spiderman kann programmieren.

Spiderman legt eine Datenbank `Bonbonbude` an. Darin kann er beliebig viele Tabellen anlegen. Neben den Artikeln passen auch Kunden oder Lieferanten hinein. Zunächst bleiben wir aber bei einer Tabelle für die Artikel.

Die Datenbank Derby

Für die Bonbonbude wählt Spiderman die Datenbank Derby. Derby wird auch JavaDB genannt, wenn man es mit den Pferden nicht so hat. Derby ist eine Kleinstdatenbank, die vollständig in Java geschrieben ist und einfach innerhalb der Anwendung laufen kann. Und das Beste daran ist: Es ist Open Source und kostet auch keine Lizenzen. Das ist gut, weil Superman bisher noch nicht einen Lolly verkauft hat, auch nicht an Kojak.

Wenn Sie mitspielen wollen, müssen Sie Derby aus dem Internet herunterladen. Die URL von Derby lautet:

`http://db.apache.org/derby`

Dort suchen Sie nach den Downloads. Selbst alte Java-Versionen werden von Derby noch unterstützt.

- Suchen Sie sich die aktuellste aus, die auf Ihre Java-Version passt, und laden Sie sie herunter.
- Sie erhalten eine ZIP-Datei.
- Öffnen Sie diese durch Doppelklick. Darin befindet sich ein Verzeichnis für das ganze Derby-Paket mit allen Tools und Bibliotheken.
- Darunter befindet sich auch ein Verzeichnis *lib*, das unter anderem die Datei *derby.jar* enthält.

- Holen Sie die Datei *derby.jar* aus der ZIP-Archiv heraus. Mehr brauchen Sie erst einmal nicht.

Einbinden einer JAR-Datei

Die Datenbankverwaltung Derby steckt vollständig in der Datei *derby.jar*. Sie müssen nur erreichen, dass die Datei in Ihr Programm eingebunden wird, und schon können Sie alle Bestandteile nutzen. Von Eclipse aus binden Sie eine JAR-Datei in Ihr Projekt folgendermaßen ein:

- Klicken Sie das Projekt im Project-Explorer mit der rechten Maustaste an und wählen Sie dort den Eintrag Properties. Es erscheint ein Dialog.
- Im Properties-Dialog sind auf der linken Seite die Themen der Eigenschaften aufgeführt. Hier wählen Sie Java Build Path aus.
- Auf der rechten Seite wenden Sie sich an den Tab Libraries.
- Dort sehen Sie das JRE aufgeführt und klicken auf den Button Add External Jars. Es öffnet sich eine Dateiauswahlbox, mit der Sie die zuvor heruntergeladene Datei *jersey.jar* auswählen.
- Klicken Sie auf den Button Apply And Close. Der Dialog schließt sich. Danach finden Sie *jersey.jar* unter dem Projekt als weitere Bibliothek.

Verbindung zu derby.jar

Im Programm muss diese Datei eingebunden werden. Sie enthält den JDBC-Treiber unter dem Namen `org.apache.derby.jdbc.EmbeddedDriver`. Dazu wird die Klasse über die Methode `Class.forname` eingebunden.

```
public bindeDerbyEin() throws ClassNotFoundException {
    Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
}
```

Listing 24.1: Programmtechnisches Einbinden von Derby

Wenn die Datei *derby.jar* nicht da ist, gibt es keine Klasse, aber eine Exception `ClassNotFound`. Schlägt die zu, ist alles schiefgegangen. Dann ist es Zeit, dem Benutzer zu erzählen, dass die Situation hoffnungslos ist. Das Programm kann nun nur noch einsam und allein sterben.

Ein Programm erzeugt die Tabelle

Für die Bonbonbude benötigen wir, wie oben schon ausgeführt, eine Tabelle `Artikel` und diese wird mit dem folgenden Befehl erstellt:

```
create table artikel (
    name  varchar(32),
    preis dec(10,2),
    menge int
)
```

Sieht nicht nach Java aus? Richtig. Nun müssen Sie stark sein: Es gibt noch eine andere Programmiersprache außer Java. Das allein ist schon bitter. Aber es kommt noch härter: Wir werden uns diese Sprache SQL anschauen müssen, weil es die Standardsprache für Datenbankoperationen ist. SQL steht für *Standard Query Language*, also Standardabfragesprache. Aber keine Angst: So schlimm wird es nicht. Außerdem bin ich ja bei Ihnen.

- ✔ Der Befehl erzeugt eine Tabelle namens `artikel`. Groß- und Kleinschreibung ist SQL übrigens egal.
- ✔ In Klammern werden dann die Spalten aufgezählt. Hier sind es `name`, `preis` und `menge`. Sie sind durch Kommata getrennt.
- ✔ Die Typen stehen bei SQL hinter den Namen.
  - Dabei entspricht `varchar` etwa dem String. Die Zahl in den Klammern gibt den Vorgabewert für die Länge an.
  - Mit `dec` werden Zahlen mit Nachkommastellen bezeichnet. Die Zahlen in Klammern beschreiben die Stellenzahl und die Zahl der Nachkommastellen.
  - Und `int` dürfte Ihnen bekannt vorkommen.

Ein Java-Programm zum Einrichten der Tabelle

Um den SQL-Code an die Datenbank zu senden, schreiben wir ein Java-Programm und erzeugen damit diese Tabelle. Eigentlich muss es nur einmal laufen. Dann existiert die Tabelle und wir können sie nutzen. Der Konstruktor bindet wie schon erwähnt Derby ein.

Klicken Sie also das Projekt an und wählen Sie New|Class. Geben Sie der neuen Klasse den Namen `CreateBonbonbude`.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;

public class CreateBonbonbude {
    final static String dbName = "Bonbonbude";
    final static String tableName = "Artikel";

    public CreateBonbonbude()
            throws ClassNotFoundException {
        // Binde die Datenbank Derby ein
        Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
    }

    public static void main(String args[]) {
        try {
            CreateBonbonbude bude = new CreateBonbonbude();
            Connection datenbank = bude.oeffneDB();
            bude.erzeugeTabelle(datenbank);
            datenbank.close();
        } catch (SQLException e) {
            System.out.println(e.toString());
            System.out.println(e.getSQLState());
            System.out.println(e.getErrorCode());
        } catch (ClassNotFoundException e) {
            System.out.println("Derby-Jar fehlt.");
            e.printStackTrace();
        }
    }

    public Connection oeffneDB() throws SQLException {
        // Die Verbindungseigenschaften
        Properties properties = new Properties();
        properties.put("user", "superman");
        properties.put("password", "geheim");
        // Verbindung mit automatischer Erzeugung der DB
        Connection dbConnect = DriverManager
                .getConnection(
                "jdbc:derby:" + dbName + ";create=true",
                properties);
        return dbConnect;
    }

    public void erzeugeTabelle(Connection connection)
            throws SQLException {
        // Sende SQL-Befehl zur Erzeugung der Tabelle
```

```
        Statement statement = connection.createStatement();
        statement.execute("create table " + tableName
                + " ( " + " name  varchar(32), "
                + " preis dec(10,2), "
                + " menge int " + " )");
        statement.close();
    }
}
```

Listing 24.2: Erzeugen der Tabelle `Artikel` in der Datenbank Bonbonbude

Die Datenbank entsteht in einem Verzeichnis

Wenn Sie das Programm ausführen, sehen Sie hoffentlich nichts. Denn die einzigen Ausgaben des Programms auf dem Bildschirm sind Fehlermeldungen. Ist das Programm sauber gelaufen, entsteht an der Startstelle ein Verzeichnis mit dem Namen *Bonbonbude*, wie er im Programm in der Konstanten `dbName` festgelegt wurde.



Unter Eclipse sehen Sie dieses Verzeichnis im Projekt, wenn Sie Ihr Projekt anklicken und die Taste F5 drücken.

Übersicht über das Programm

- Die Methode `main` legt ein Objekt der eigenen Klasse an. So kommt es zum Aufruf des Konstruktors. Im Konstruktor wird die Datei *derby.jar* eingebunden. Ein Scheitern wird durch eine `ClassNotFoundException` gemeldet.
- Wenn das Programm nicht gestorben ist, kann es die Methode `oeffneDB` aufrufen, die ein Objekt der Klasse `Connection` liefert, die die Basis aller Datenbankoperationen ist.
- Die Methode `erzeugeTabelle` sendet den SQL-Befehl zur Erzeugung der Artikel-Tabelle an die Datenbank.

Stimmt etwas mit der Datenbank nicht, wird eine Exception `SQLException` geworfen. Die fangen wir hier auf und melden dem Benutzer, was der Datenbank nicht schmeckt.

Verbindung aufnehmen

Über ein Objekt der Klasse `Connection` kommuniziert das Programm in Listing 24.2 mit der Datenbank. Der Name `Connection` erscheint merkwürdig, da die Datenbank Derby Bestandteil des eigenen Programms ist und eine Kontaktaufnahme eher einem Methodenaufruf ähnelt. Bei großen Datenbanken kann die Verbindung mit einer hohen Wahrscheinlichkeit durch eine Kommunikation über ein Netzwerk erfolgen.

- Das Objekt der Klasse `Connection` wird nicht per `new` erzeugt, sondern über den `DriverManager` und dessen Methode `getConnection` ermittelt.
- Die Methode `getConnection` erwartet als Parameter einen String, der datenbankspezifisch ist und auch Verbindungs-String genannt wird.
  - Bei Derby beginnt er mit `jdbc:derby:`.
  - Es folgt der Datenbankname, der eigentlich der Name des Verzeichnisses ist, in dem die Datenbank liegt.
  - Gefolgt von einem Semikolon folgen weitere Optionen. Im Beispiel besagt `create=true`, dass die Datenbank angelegt werden kann. Wird auf eine existierende Datenbank zugegriffen, kann diese Option entfallen.
- Der zweite Parameter ist ein Objekt der Klasse `Properties`, das den Benutzernamen und das Passwort des Datenbankbenutzers enthält.

Auch hier kann eine `SQLException` auftreten. Sie wird durch den Befehl `throw` an die aufrufende Methode weitergeleitet.

Anlegen der Tabelle

In der Methode `erzeugeTabelle` in Listing 24.2 ist beispielhaft zu sehen, wie SQL-Befehle an die Datenbank weitergereicht werden. Auf der Basis der Verbindung wird ein Objekt der Klasse `Statement` erzeugt.

Über dieses Objekt wird dessen Methode `execute` ausgeführt, die einen SQL-Befehl als String-Parameter übergeben bekommt. Darin befindet sich der Befehl zum Anlegen der Tabelle `Artikel`.

Zum Abschluss wird das Statement mit der Methode `close` geschlossen, weil sich das einfach so gehört und die Datenbank es gar nicht schätzt, wenn jemand die Tür offen stehen lässt.

In die Datenbank schreiben

Eine Datenbank macht erst dann richtig Spaß, wenn auch Daten darin sind. Prinzipiell benutzen wir die gleiche Technik wie beim Anlegen der Tabelle: Wir schicken der Datenbank den SQL-Befehl zum Einfügen eines Datensatzes. Um eine neue Zeile in einer Tabelle anzulegen, verwendet SQL den Befehl `INSERT`:

```
INSERT INTO artikel (name, preis, menge)
       VALUES ('Blutwurstlutscher', 2.54, 120)
```

Listing 24.3: Einfügen eines neuen Artikels per SQL

- Der Befehl beginnt mit `INSERT INTO` gefolgt von dem Tabellennamen, hier `artikel`.

- In Klammern werden kommasepariert die Spalten aufgezählt, hier sind es `name`, `preis` und `menge`.
- Nach dem Schlüsselwort `VALUES` folgen in einer weiteren Klammer wiederum kommasepariert die Werte. Wie Sie sehen, wünscht sich auch SQL als Dezimalkomma einen Punkt, verwendet aber im Gegensatz zu Java Hochkommata statt Anführungszeichen für Zeichenkettenliterale.

Der Artikel

Als Java-Programmierer werden Sie sofort anmelden, dass Sie die Artikel in der Datenbank als Objekte handhaben wollen. Also legen wir gleich eine Klasse `Artikel` an. Da wir immer wieder neue Artikel anlegen müssen, erstellen wir gleich einen Konstruktor, der Name, Preis und Menge akzeptiert. Dazu noch einen Standardkonstruktor, damit es später keine Probleme gibt, leere Artikel anzulegen, die wir aus der Datenbank erst lesen wollen.

```
public class Artikel {
    private String name;
    private double preis;
    private int menge;

    public Artikel() { }

    public Artikel(String name, double preis, int menge) {
        this.name = name;
        this.preis = preis;
        this.menge = menge;
    }
}
```

Listing 24.4: Rohbau für die Klasse `Artikel`

Nun brauchen wir noch Getter und Setter, um die privaten Attribute ändern und lesen zu können. Dazu verwenden wir die Hilfe von Eclipse.

- Sie klicken in den Editor mit der rechten Maustaste und wählen dort Source|Generate Getters And Setters.
- Es erscheint ein Dialog. Dort wählen Sie den Button Select All an. In der Liste erhalten alle Attribute eine Markierung.
- Nach dem Klick auf OK schließt der Dialog und alle Getter und Setter sind erzeugt.

Insert per SQL-Befehls-Manipulation

Das Programmbeispiel unterscheidet sich kaum von dem Programm zum Anlegen

einer Tabelle, allerdings kann die Option `create=true` beim Aufbau der Verbindung weggelassen werden. Nun gibt es eine Methode `insert`, die den SQL-Befehl `INSERT` mit einem Objekt der Klasse `Artikel` verknüpft. Dazu erzeugt die Methode aus den übergebenen Daten einen SQL-String, der mit der Methode `execute` eines Statement-Objekts an die Datenbank übergeben wird.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;

public class InsertBonbon {
    final static String DBNAME = "Bonbonbude";
    final static String TABLENAME = "Artikel";

    public InsertBonbon()
            throws ClassNotFoundException {
        // Binde die Datenbank Derby ein
        Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
    }
    public static void main(String args[]) {
        Artikel lolly = new Artikel(
                "Blutwurstlutscher", 2.54, 120);
        try {
            InsertBonbon bude = new InsertBonbon();
            Connection datenbank = bude.oeffneDB();
            bude.insert(datenbank, lolly);
            datenbank.close();
        } catch (SQLException e) {
            System.out.println(e.toString());
            System.out.println(e.getSQLState());
            System.out.println(e.getErrorCode());
        } catch (ClassNotFoundException e) {
            System.out.println("Derby-Jar fehlt.");
            e.printStackTrace();
        }
    }

    void insert(Connection db, Artikel artikel)
            throws SQLException {
        Statement statement = db.createStatement();
        statement.execute("insert into "+ TABLENAME
                + " (name, preis, menge) VALUES "
                + " (\'"+artikel.getName()+"\',"
                + artikel.getPreis()+","
                + artikel.getMenge()+" )");
        statement.close();
    }

    public Connection oeffneDB() throws SQLException {
```

```
        // Die Verbindungseigenschaften
        Properties properties = new Properties();
        properties.put("user", "superman");
        properties.put("password", "geheim");
        // Verbindung mit automatischer Erzeugung der DB
        Connection db = DriverManager.getConnection(
                "jdbc:derby:"+DBNAME, properties);
        return db;
    }
}
```

Listing 24.5: Artikel in die Datenbank aufnehmen

Das funktioniert recht gut, hat aber ein paar Schönheitsfehler.

- ✔ Die Übergabe der Werte ist nicht typsicher. Der Compiler kann nicht feststellen, ob der übergebene Wert ein `double`, ein `int` oder ein `String` ist.
- ✔ Bei der Übergabe von Zeichenketten muss die Zeichenkette in Hochkommata eingeschlossen werden, damit sich SQL nicht verschluckt.
- ✔ Es bleibt das Risiko einer SQL-Injection. Dieser Angriff auf Datenbanken erfolgt dadurch, dass man Werteingaben mit SQL-Befehlen so kombiniert, dass die Datenbank Operationen ausführt, die vom Programmierer nicht vorgesehen waren.

PreparedStatement

Um diese Ziele zu erreichen, verwendet man ein `PreparedStatement`. Darin werden die Werte durch Fragezeichen als Platzhalter ersetzt. Nachfolgende Set-Funktionen können diese dann füllen.

```
void insert(Connection db, Artikel artikel)
        throws SQLException {
    PreparedStatement befehl = db.prepareStatement(
            "insert into "+ TABLENAME
            + " (name, preis, menge) VALUES (?,?,?)");
    befehl.setString(1, artikel.getName());
    befehl.setDouble(2,  artikel.getPreis());
    befehl.setInt(3, artikel.getMenge());
    befehl.executeUpdate();
    befehl.close();
}
```

Listing 24.6: Die Einfügemethode mit `PreparedStatement`

Mehrere Inserts und Updates

Das Einfügen sollte inzwischen geklappt haben. Wir wollen darauf aufbauend mehrere Sätze direkt nacheinander einfügen. Der präparierte Befehl kann nun mit immer neuen Werten gefüllt werden und wird durch Aufruf von `executeUpdate` in die Datenbank gebracht.

```
Artikel[] liste = {
        new Artikel("Oreganodrops", 0.48, 30),
        new Artikel("Senfpralinen", 0.78, 40),
        new Artikel("Seifenschoko", 1.28, 20)
    };
bude.insert(datenbank, liste);

// ...

void insert(Connection db, Artikel[] artikelliste)
        throws SQLException {
    PreparedStatement befehl = db.prepareStatement(
            "insert into "+ TABLENAME
            + " (name, preis, menge) VALUES (?,?,?)");
    for (Artikel artikel : artikelliste) {
        befehl.setString(1, artikel.getName());
        befehl.setDouble(2,  artikel.getPreis());
        befehl.setInt(3, artikel.getMenge());
        befehl.executeUpdate();
    }
    befehl.close();
}
```

Listing 24.7: Einfügen einer Liste in die Datenbank

Da läuft einem das Wasser im Munde zusammen, nicht wahr?



Wie Ihnen längst aufgefallen ist, beginnt die Zählung der Spalten nicht etwa mit 0, sondern mit 1. Auf nichts ist Verlass in der Programmierung.

Datenbank auslesen

Natürlich werden Sie mir blind vertrauen, dass die Einträge in der Datenbank angekommen sind. Dennoch bestehe ich darauf, Ihnen zu zeigen, wie Sie es selbst prüfen können.

Wieder benötigen Sie einen SQL-Befehl, den Sie an die Datenbank schicken können. Für das Auslesen der Tabelle verwendet SQL den Befehl `SELECT`.

```
SELECT name,preis,menge FROM artikel
```

Listing 24.8: Auslesen einer Datenbank mit `SELECT`

Im Gegensatz zum Einfügen von Daten oder dem Anlegen einer Tabelle müssen wir beim Auslesen der Tabelle die Ergebnisse des SQL-Befehls auffangen und diese in Java-Variablen überführen. Etwas schwieriger wird es dadurch, dass wir bei Aufruf des Befehls nicht einmal wissen, wie viele Zeilen wir zurückbekommen werden.

1. Über die Datenbankverbindung `Connection` fordern Sie ein `Statement` über die Methode `createStatement` an.
2. Über das `Statement` können Sie die Methode `executeQuery` aufrufen. Diese liefert alle Treffer in der Datenbank in einer Menge, also einem Set, genauer ausgedrück in einem `ResultSet`.
3. Über diese Ergebnismenge drehen Sie eine Schleife. Jede Schleifenrunde liefert einen Satz. Die Schleifenbedingung wird über die boolesche Methode `next` ermittelt.
4. Innerhalb der Schleife kann über die Methode `getString`, `getInt` oder `getDouble` der Inhalt jedes Elements ausgelesen werden. Als Parameter erwarten diese Methoden den Namen der Spalte als String oder die Nummer der Spalte.

Um die Datenbankergebnisse weiterzuverarbeiten, gibt unsere Methode `leseAlle` eine `ArrayList` für Artikel zurück.

```
public ArrayList<Artikel> leseAlle(Connection db)
        throws SQLException {
    ArrayList<Artikel> liste = new ArrayList<>();
    Statement statement = db.createStatement();
    ResultSet ergebnis = statement.executeQuery(
            "SELECT name, preis, menge FROM artikel");
    while (ergebnis.next()) {
        liste.add(new Artikel(
                ergebnis.getString("name"),
                ergebnis.getDouble("preis"),
                ergebnis.getInt("menge")));
    }
    ergebnis.close(); // Ergebnismenge schließen
    return liste;
}
```

Listing 24.9: Ein Blick in die Datenbank

Die Methode wird über die folgende Anweisungsfolge aufgerufen und ausgewertet:

```
ArrayList<Artikel> list = bude.leseAlle(datenbank);
for (Artikel artikel : list) {
    System.out.print(artikel.getName()+", ");
    System.out.print(artikel.getPreis()+", ");
    System.out.println(artikel.getMenge());
}
```

Werte in der Datenbank ändern

Aus irgendeinem unerfindlichen Grund verkaufen sich die Blutwurstlutscher nicht wie erwartet. Nur zehn Stück wurden verkauft. Vielleicht hilft eine Preissenkung? Dazu muss nun der Preis in der Datenbank geändert werden. Solche Änderungen sind aus Sicht von SQL ein Update. Superman kann mit dem SQL-Befehl `UPDATE` seine Blutwurstlutscher auf 87 Cent reduzieren und den Bestand auf 110 setzen.

```
UPDATE artikel SET preis=0.87,menge=110 WHERE name='Blutwurstlutscher'
```

- Nach dem Befehl `UPDATE` wird die Tabelle genannt, in der die Operation erfolgen soll.
- Auf den Befehl `SET` folgen kommasepariert Zuweisungen, wie Sie sie aus Java kennen. Dabei wird jede zu ändernde Spalte genannt und der neue Wert per Gleichheitszeichen zugewiesen.
- Hinter `WHERE` wird festgelegt, welche Zeilen der Tabelle geändert werden sollen. Im Beispiel werden alle Einträge geändert, deren Spalte `name` den Inhalt »Blutwurstlutscher« hat.

Hier selektieren Sie über den Namen des Artikels. Das kann sich als unpraktisch erweisen, wenn Superman Schokoladengurken in verschieden großen Gläsern anbieten will. Darum verwenden die meisten Firmen Artikelnummern, die eindeutig sind. So etwas haben wir aber nicht. Pech gehabt! Denken Sie aber noch an diesen Vorschlag, wenn Sie den Preis Ihres Joghurts mit Hering/Maracuja-Geschmack anpassen wollen.



Die `WHERE`-Klausel können Sie übrigens auch an einen `SELECT`-Befehl anhängen, um nur bestimmte Zeilen auszulesen.

Ein kurzer Blick auf WHERE

Auf `WHERE` folgt immer eine Bedingung, die festlegt, welche Datensätze verwendet werden.

- Die Abfrage von Zahlenwerten erfolgt fast wie in Java.
  - `a > b`: Ist a größer als b?

- `a < b`: Ist a kleiner als b?
- `a = b`: Ist a gleich b? SQL verwendet nur ein Gleichheitszeichen.

✔ Bei Strings gibt es ein paar Unterschiede. So stehen String-Literale in Hochkommata. Und man kann nach einfachen Mustern abfragen, indem `LIKE` verwendet wird. Als Platzhalter dient:

- Der Unterstrich für ein Zeichen
- Das Prozentzeichen für mehrere Zeichen.

Die folgende Bedingung trifft zu, wenn in `name` die Wörter »Maus«, »Haustelefon« oder »rausgeworfen« stehen.

```
WHERE name LIKE '_aus%'
```

Die Methode `aendereNachName` ändert den Artikel in der Datenbank nach dem Namen des Artikels und verwendet die anderen Attribute des Artikels für die Änderungen in den Spalten `preis` und `menge`.

```
void aendereNachName(Connection db, Artikel artikel)
        throws SQLException  {
    Statement statement = db.createStatement();
    statement.execute("update "+ TABLENAME
            + " set preis=" + artikel.getPreis()
            + ",menge=" + artikel.getMenge()
            + " where name=\'"
            + artikel.getName() + "\'");
    statement.close();
}
```

Listing 24.10: Änderungen

Nun können die Blutwurstlutscher im Preis gesenkt werden. Sollten sich die Lutscher immer noch nicht verkaufen, dann sollte Superman vielleicht einen Hinweis »aphroditierende Wirkung« aufkleben. Wie man hört, jagen aus diesem Grund ganze Horden von Idioten die Nashörner, um deren Hörner zu verpulvern. So wird es für Nashörner noch schwieriger, ein Bankkonto zu eröffnen, denn der Bankangestellte könnte sich ja etwas nebenher verdienen wollen.

Löschen von Einträgen

Für das Löschen von Einträgen verwendet SQL den Befehl `DELETE`. Der folgende Befehl würde also alle Produkte löschen, die 2000 Euro kosten oder teurer sind.

```
DELETE FROM artikel WHERE preis>1999.99
```

Damit nicht die gesamte Tabelle gelöscht wird, ist hier die `WHERE`-Klausel unverzichtbar. Die Umsetzung per JDBC bringt nichts wirklich Neues, darum ist

dies eine ideale Aufgabe für den Leser, und das sind in diesem Falle Sie.

Wenn Datenbanken einmal groß werden

Supermans kleine Bonbonbude benötigt nur eine einzige Tabelle. Aber es ist ja auch nur eine kleine Bude. Größere Buden wie Banken oder Versandhäuser haben neben ihrer Artikel-Tabelle noch mindestens eine Tabelle für ihre Kunden. Die hohe Kunst der Datenbankdefinition liegt darin, Tabellen nicht nur als Liste für Objekte zu verwenden, sondern Tabellen für Beziehungen zwischen Tabellen einzusetzen.

Eigenständige Datenbanksysteme

Große Datenbanksysteme werden nicht mehr in die Applikation eingebunden, sondern laufen als eigenständige Systeme, die oft über ein Netzwerk von vielen Computern aus verwendet werden. Solche Datenbank-Management-Systeme kennen Sie vielleicht unter den Namen PostgreSQL, Oracle, MySQL oder DB/2.

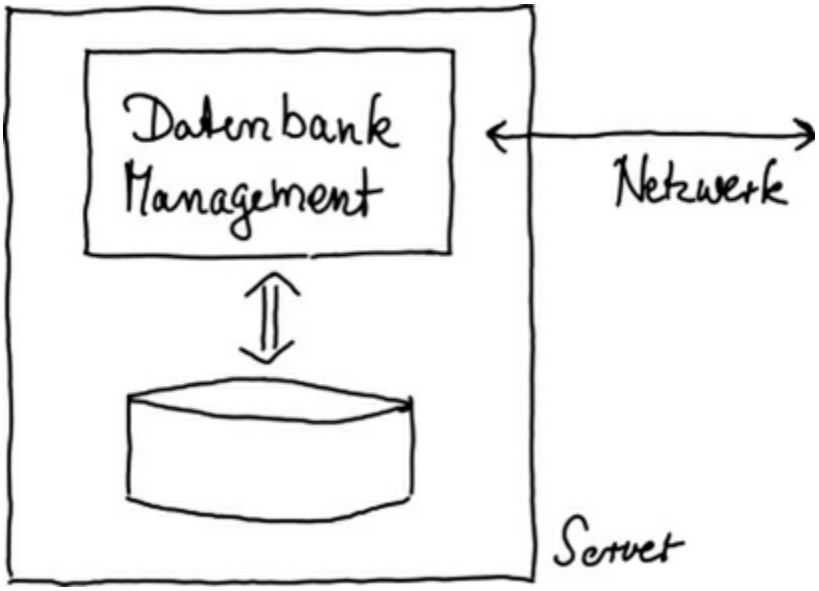


Abbildung 24.1: Das Datenbank-Management-System

Aus Sicht der Java-Programmierung sind die Unterschiede zwischen den Datenbankensystemen nicht so gewaltig. Tatsächlich können Sie die Beispielprogramme für die Derby-Datenbank auch in anderen Datenbanken so weiterverwenden. Lediglich die Verbindungsaufnahme unterscheidet sich, da jede Datenbank ihren eigenen JDBC-Treiber verwendet.

Datenbank, Datenbankbenutzer und Datenbankverbindung

Ein Datenbank-Management-System kann parallel ganz unterschiedliche

Datenbestände verwalten. So kann sie gleichzeitig eine Warenwirtschaft, eine Bonbonbude und den Bestand der Vegetation im Amazonasgebiet verwalten.

Jeder dieser Datenbestände kann viele Tabellen umfassen und wird als *Datenbank* bezeichnet. Jede Datenbank erhält einen eigenen Namen.

Da es ganz schrecklich wäre, wenn der Bonbonbudenbesitzer sich heimlich über die Vegetation im Amazonasgebiet informiert, kann ein Datenbank-Management-System Benutzerkennungen und Passwörter verwalten. So kann für den Bonbonbudenbesitzer ein anderer Benutzer angelegt werden als für die Amazonas-Datenbank.

Selbst innerhalb der Datenbank kann es mehrere Datenbankbenutzer geben. So möchte der Bonbonbudenbesitzer zwar seiner Aushilfe erlauben, ihre Verkäufe zu erfassen, aber nicht das Löschen der Bestände oder gar eine Änderung der Datenstruktur. Man kann also den Datenbankbenutzern unterschiedliche Rechte einräumen.

Die folgende Befehlsfolge richtet eine Datenbank `bonbonbude` ein und gewährt alle Rechte daran dem Benutzer `superman`, der das Passwort »geheim« hat.

```
create user superman password 'geheim' login;
create database bonbonbude;
grant all on database bonbonbude to superman;
```

Anschließend kann sich der Benutzer `superman` bei der Datenbank `bonbonbude` anmelden und die notwendigen Tabellen für seine Datenbestände anlegen.



Eine Datenbank wird typischerweise folgende Parameter für den Zugriff erfordern:

- HOST: Der Name oder die IP-Adresse des Computers, auf dem sich die Datenbank befindet. Ist die Datenbank auf dem gleichen Computer wie das Programm, kann der Name `localhost` verwendet werden.
- PORT: Auf einem Computer können mehrere Netzwerkdienste nebeneinander eine friedliche Koexistenz betreiben. Sie werden über die jeweilige Portnummer von außen angesprochen. Die Datenbankhersteller verwenden üblicherweise unterschiedliche Portnummern.
- DBMS: Der Datenbankhersteller.
- DATABASE: Der Name der Datenbank
- USER: Der Benutzer, mit dem die Datenbank geöffnet wird.
- PASSWORD: Das passende Passwort.
- TABLE: Die Tabellen der Datenbank haben natürlich auch Namen.

Der JDBC-Treiber

Der JDBC-Treiber klemmt zwischen der Datenbank und der Java-Anwendung. Die Datenbankhersteller stellen ihn meist in Form einer JAR-Datei zur Verfügung. Diese muss dann in die Anwendung eingebunden werden, damit der Treiber vom Java-Programm verwendet werden kann.



Von Derby her kennen Sie ja schon eine solche JAR-Datei. Und tatsächlich enthält auch die Datei *derby.jar* einen JDBC-Treiber. Bei Derby hängt zusätzlich noch das ganze Datenbanksystem dran. Bei den großen Datenbanken ist das Datenbanksystem ein unabhängig laufender Prozess, vermutlich sogar auf einem ganz anderen Computer, der über ein Netzwerk erreicht wird. Die JAR-Datei bereitet dann nicht mehr selbst die Daten auf, sondern sendet die SQL-Befehle über die Verbindung an diesen separaten Prozess.

Das Einbinden der JAR-Datei erfolgt wie bei der Derby-Datenbank durch Aufruf von `Class.forName`. Hier wird der JDBC-Treiber von PostgreSQL eingebunden:

```
Class.forName("org.postgresql.Driver");
```

Der String, der den Ort und Namen des JDBC-Treibers anspricht, ist bei jeder Datenbank ein klein wenig anders. Hier ein paar Beispiele:

Datenbank	JDBC-Treiberklasse
Oracle	`oracle.jdbc.driver.OracleDriver`
HSQL	`org.hsqldb.jdbcDriver`
MySQL	`com.mysql.jdbc.Driver`
DB/2	`COM.ibm.db2.jdbc.app.DB2Driver`
Derby	`org.apache.derby.jdbc.EmbeddedDriver`
PostgreSQL	`org.postgresql.Driver`

Tabelle 24.2: Datenbanken und ihre JDBC-Treiberklassen

Der JDBC-Treiber muss über den Datenbankenhersteller bezogen werden. Entweder er ist bereits im Lieferumfang der Datenbank enthalten oder Sie finden ihn zum Download auf der Homepage des Datenbankherstellers. Den JDBC-Treiber der Datenbank PostgreSQL finden Sie beispielsweise auf der Website

`https://jdbc.postgresql.org/download.html`

Vergessen Sie nicht, die JAR-Datei des JDBC-Treibers in das Projekt einzubinden.

Verbindung mit der Datenbank

Ist der Treiber eingebunden, kann die Verbindung zur Datenbank aufgenommen werden.

1. Die Methode `DriverManager.getConnection` wird aufgerufen.
2. Als Parameter erhält sie in einem String die Verbindungsdaten zu der Datenbank, die für den Zugriff notwendig sind. Dazu können gehören:
    - Der Hostname des Computers, auf dem die Datenbank läuft. Der Name `localhost` steht dabei für den eigenen Rechner.
    - Der Name der Datenbank selbst. Ein Datenbanksystem steht oft nicht nur für einen Zweck zur Verfügung. Im Sonderfall von Derby ist dies der Verzeichnisname, wo sich das Datenbankverzeichnis befindet.
    - Der Benutzername des Datenbankbenutzers. Datenbanken lassen nicht unbedingt jeden hinein, der Daten lesen oder verändern will.
    - Das Passwort des Benutzers. Eine Benutzerverwaltung ist wenig sinnvoll, wenn sie nicht durch ein Passwort abgesichert wird.

    Der Verbindungsstring ist datenbankspezifisch. In manchen Fällen ist er auf mehrere Parameter verteilt.
3. Der Rückgabewert von `DriverManager.getConnection` ist ein Objekt der Klasse `Connection`. Dieses Objekt bildet die Basis für die späteren Datenbank-Operationen.

```
Connection dbConnect = DriverManager.getConnection(
    "jdbc:postgresql://debianserver/"+dbName,
    "datenbankuser",
    "geheim");
```

Listing 24.11: Verbindungsaufbau zu PostgreSQL

Betrachten wir die Parameter von `DriverManager.getConnection` genauer:

- ✔ `jdbc` heißt das Protokoll.
- ✔ `postgresql` ist das Datenbanksystem.
- ✔ `debianserver` heißt der Datenbankserver oder Host.
- ✔ Es folgt der Name der Datenbank.
- ✔ Der zweite String ist der Datenbankbenutzer.
- ✔ Der dritte String enthält das passende Passwort.

Es gibt, wie schon angedeutet, einige Unterschiede zwischen den Datenbanken. Sie erinnern sich? Die Verbindung zu Derby arbeitete über ein Objekt der Klasse `Properties`.

```
Properties properties = new Properties();
properties.put("user", "arnold");
properties.put("password", "geheim");
Connection dbConnect = DriverManager.getConnection(
    "jdbc:derby:"+dbName, properties);
```

Listing 24.12: Verbindungsaufbau zu Derby

Transaktionen

Eine der großen Leistungen von Datenbanken ist, dass sie Möglichkeiten anbieten, Daten immer konsistent zu halten. Betrachten wir dazu den Ablauf einer Bestellung:

- Die Bestellung bekommt einen Vermerk, dass die Ware ausgeliefert wird, aber noch nicht bezahlt ist.
- Die Statistik der verkauften Waren wird um die Bestellmenge erhöht.
- Der Warenbestand wird um die Bestellmenge reduziert.

Problematisch wird es, wenn der Ablauf der Bestellung unterbrochen wird. Je nach Reihenfolge stimmt dann der Bestand nicht, eine Bestellung geht verloren oder es gibt eine Doppellieferung, wenn die abgebrochene Bestellung noch einmal aufgesetzt wird. Komplizierter wird es dadurch, dass oft mehrere Benutzer auf die Datenbank parallel zugreifen. Hier kann es durch nicht sauber abgegrenzte Aktionen zu Fehlern in den Datenbankbeständen führen. Als Gegenmittel verwenden Datenbanken Transaktionen.



Eine Transaktion umschließt einen solchen Ablauf wie die oben erwähnte Bestellung. Wird die Transaktion vollständig durchlaufen, wird mit einem *Commit* der Ablauf der Transaktion bestätigt und alle Änderungen werden in einer ununterbrechbaren Aktion durchgeführt. Gibt es während des Ablaufs eine Störung oder Unterbrechung, wird die gesamte Aktion per *Rollback* wieder auf den Ausgangsstatus zurückgeführt, als sei sie nie ausgeführt worden. Das Ergebnis ist, dass die Datenbank in jedem Fall die Konsistenz ihrer Daten garantiert.

Die kritische Reisebuchung

Als kritisches Beispiel wird immer wieder gern die Buchung von Flugreisen gewählt. In zwei Reisebüros steht je ein Paar und möchte eine Reise buchen, bei der nur noch drei Plätze frei sind.

Beide Paare nicken und lassen ihre Namen in die Passagierliste eintragen. Es werden gleichzeitig je 2 von den freien Plätzen abgezogen und damit beide Male

eine 1 in die Anzahl der freien Plätze geschrieben, während gleichzeitig aber vier Personen in die Liste der Passagiere eingetragen wurden.

Die Inkonsistenz liegt darin, dass die Anzahl der gebuchten Personen die Anzahl der Plätze übersteigt und sogar noch ein freier Platz ausgewiesen wird. Inkonsistenz hat mit Inkontinenz übrigens nichts zu tun, ist aber mindestens genauso unangenehm.

Auch dieses Problem ist durch eine Transaktion zu sichern. Nur wenn Abfrage, Platzreservierung und Freiplatzverwaltung komplett funktioniert hat, wird die Transaktion per Commit bestätigt. Das Paar, das eine Millisekunde später nach den freien Plätzen fragt, bekommt also einen Rollback und den Hinweis, dass nur noch ein Platz frei ist. Da hilft dann nur noch die Scheidung oder die Suche nach einer anderen Reise.

Automatisches Commit oder Selbstkontrolle

Da die meisten einfachen Datenbankanwendungen nicht unbedingt auf Transaktionen angewiesen sind, wird standardmäßig jede SQL-Anweisung als Transaktion angesehen und sofort unwiederbringlich durchgeführt. Die Datenbankverbindung hat aber eine Methode `setAutoCommit`, mit der Sie dieses Verhalten abschalten können. Dann setzen Sie so lange Befehle ab, bis ein neuer konsistenter Zustand erreicht ist. Wenn Sie zufrieden sind, rufen Sie die Methode `commit` auf. Erst dann werden die Daten in die Datenbank geschrieben. Sollten Sie aber Probleme bekommen und keinen konsistenten Zustand mehr erreichen können, reißen Sie die Hände hoch und rufen: »Es war alles nur Spaß!« Der passende Aufruf lautet `rollback`.

```
Connection db = DriverManager.getConnection( // ...
db.setAutoCommit(false);
// Ausführung mehrerer Statements
if (schiefgelaufen) {
    db.rollback();
} else {
    db.commit();
}
```

Datenbankunterstützung durch Eclipse

Manchmal ist es nett, in die Datenbanken hineinzuschauen, ohne gleich ein Programm schreiben zu müssen. Dazu stellt Eclipse eine Database-Development-Perspektive zur Verfügung. Zunächst müssen Sie die Verbindung zur Datenbank aufbauen.

1. Sie rufen Window|Preferences auf. Es erscheint ein Dialog, in dessen linker

Spalte Sie DataManagement|Connectivity|Driver Definitions auswählen.

2. Falls in der Liste nicht zufällig bereits ein Treiber für Ihre Datenbank existiert, klicken Sie auf den Button Add. Es erscheint ein neuer Dialog.
    a. Unter dem Label Name/Type wird der Hersteller der Datenbank ausgewählt.
    b. Unter dem Label JAR List wird die JAR-Datei des Treibers angezeigt. Vermutlich werden Sie über den Button Add JAR/Zip eine JAR vom Dateisystem einbinden müssen.

Damit haben Sie nun den Treiber eingebunden und können auf die Datenbank schauen. Dies erreichen Sie über die Datenbank-Perspektive.

1. Aus dem Eclipse-Menü wählen Sie Window|Perspective|Open Perspective|Other.
2. Aus dem Dialog wählen Sie Database Development.
3. Links oben sehen Sie den Data Source Explorer und an erster Stelle einen Eintrag für die Database Connections.
4. Diese klicken Sie mit der rechten Maustaste an und wählen aus dem Kontextmenü New.
5. Es erscheint ein Dialog, aus dem Sie zunächst Ihre Datenbank auswählen. Mit Next kommen Sie weiter.
6. Im Folgedialog können Sie unter General oder auch Optional die Verbindungsdaten Ihrer Datenbank eingeben.
7. Sie finden in diesem Dialog auch den Button Test Connection. Das ist sehr hilfreich, weil Sie sofort erkennen können, ob alle Zugangsdaten korrekt sind.
8. Wenn alles passt, klicken Sie auf Finish.

Sie finden unter den Database Connections nun die gerade festgelegte Datenbank. Wenn Sie diese auffalten, können Sie über den Datenbanknamen die Schemas zu den Tables herunterklicken, bis Sie Ihre Tabellen finden. Mit einem Rechtsklick können Sie den Inhalt der Tabellen direkt bearbeiten.

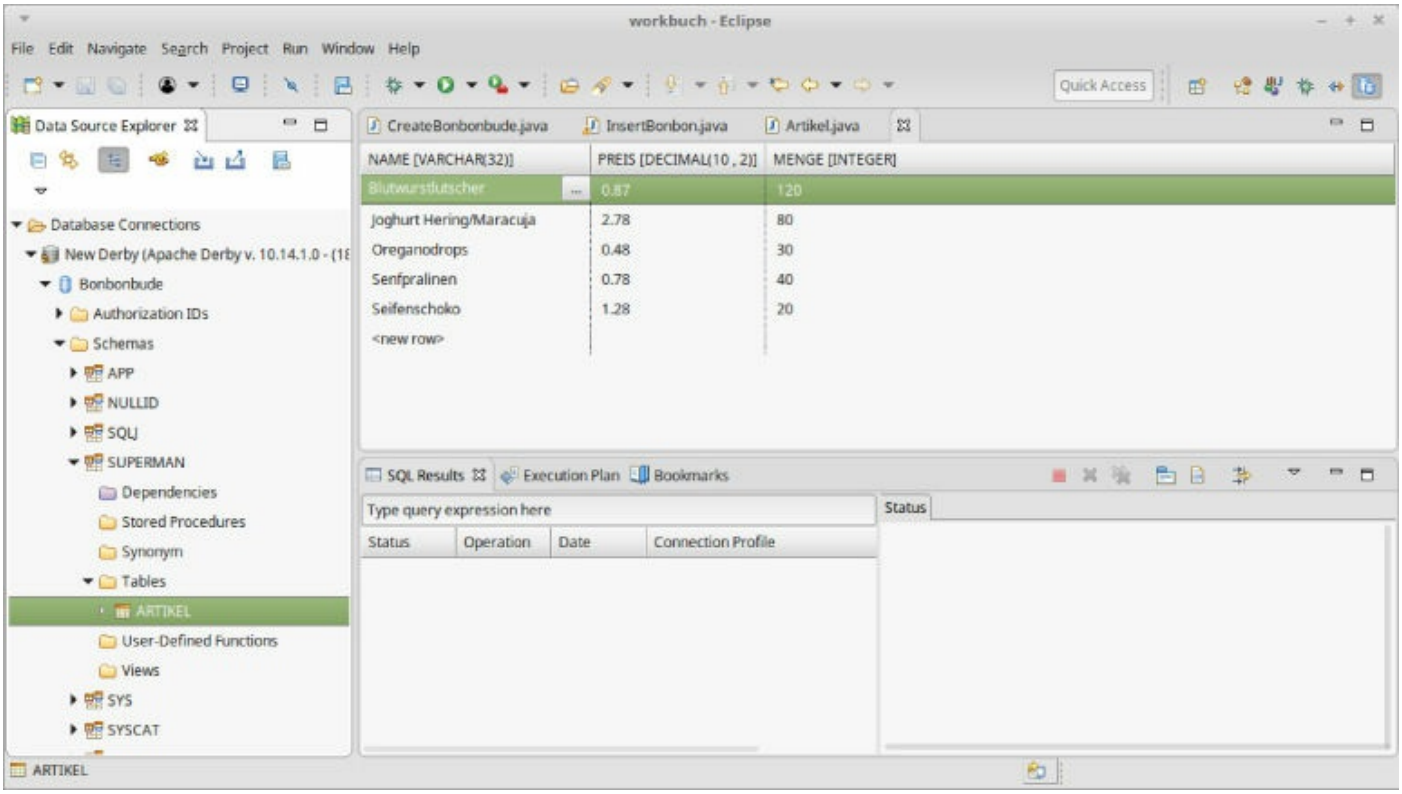


Abbildung 24.2: Eclipse in der Database-Development-Perspektive

Aufgaben

Löschen von Einträgen

Erstellen Sie doch selbst einmal eine Methode `delete`, die den SQL-Befehl `DELETE` umsetzt. Damit können Sie endlich auch Ihre Blutwurstlutscher aus dem Sortiment entfernen. Der Befehl `DELETE` bestimmt über die `WHERE`-Klausel, welche Sätze zu löschen sind.

```
DELETE FROM artikel WHERE preis>1999.99
```

Das ist eine sehr kundenfreundliche Aktion: Alle Artikel, deren Preis über 1999,99 liegt, werden aus der Tabelle `artikel` gelöscht. Und hier noch ein paar weitere Aufgaben für Sie:

- ✔ Ergänzen Sie Ihre Datenbank so, dass sie Artikelnummern enthält.
- ✔ Entwickeln Sie eine Methode, die nur den Bestand erhöht oder erniedrigt.
- ✔ Bauen Sie eine grafische Oberfläche um Ihr Datenbankprogramm.

Java Persistence API: JPA

Bei Zugriffen auf die Datenbank stört den geplagten Java-Programmierer, dass er sich dazu in die Datenbanksprache SQL einarbeiten muss. Viel netter wäre es, eine Tabelle durch eine Klasse zu modellieren und der Datenbank zu erzählen, sie möchte bitte das davon erstellte Objekt in eine passende Tabelle schreiben.

JPA verwendet Java-Klassen als Stellvertreter für Datenbanktabellen und nennt diese *Entity*. Diese Vorgehensweise, die dem Java-Programmierer viele Details an Datenbankwissen erspart, nennt man »Object-Relational Mapping« oder kurz *ORM*.



JPA kommt aus der Java Enterprise Edition, die vor allem die Entwicklung von verteilten Anwendungen mithilfe eines Application-Servers betreibt. Zwar kann JPA auch in der Standard-Edition verwendet werden, ihre Bibliothek gehört allerdings nicht zu deren Standardumfang. Eclipse stellt für solche Zwecke jedoch ein fertiges JPA-Projekt zur Verfügung.

Erstellen eines Eclipse-JPA-Projekts

Eclipse unterstützt die Arbeit mit JPA. Sie können statt eines Java-Projekts ein JPA-Projekt anlegen, das für die Verwendung von JPA vorbereitet ist. Dazu gehen Sie folgendermaßen vor:

1. Sie wählen aus dem Menü File|New|Project. Es erscheint ein Dialog.
2. Unter dem Punkt JPA wählen Sie JPA Project aus. Weiter geht es mit Next.
3. Auf der Folgeseite des Dialogs geben Sie unter Project Name den Namen des Projekts an. Als Target Runtime kann ihr aktuelles JDK gewählt werden. Im Falle, dass Sie für Java Enterprise Edition entwickeln, sollten Sie allerdings die Umgebung des Application-Servers angeben.
4. Unter Configuration wählen Sie die Basic JPA Configuration. Mit Next geht es weiter.
5. Klicken Sie noch einmal auf Next!
6. Unter Platform wählen Sie Eclipselink aus.
7. Darunter wird die JPA-Implementierung festgelegt. Dazu muss unter User Library die Liste mit mindestens einer Implementierung bestückt werden.
8. Eclipse stellt dazu die JPA-Referenzimplementierung `EclipseLink` zur Verfügung, die Sie über den Download-Button auf der rechten Seite herunterladen können.
9. Es erscheint ein neuer Dialog mit den verschiedenen Versionen von `EclipseLink`. Wählen Sie die aktuellste aus. Bestätigen Sie die Lizenz und klicken Sie auf Finish. Nach dem Download verschwindet der Dialog und Sie befinden sich wieder im Projektdialog.
10. Die geladene Version von EclipseLink erscheint in der Liste.
11. Im gleichen Dialog finden Sie den Eintrag Connection. Damit ist die Verbindung zu einer Datenbank gemeint.
    - Über den Link Add Connection bauen Sie eine neue Verbindung auf. Es

erscheint ein neuer Dialog:

- Wählen Sie zunächst Derby als Datenbankprofil.
- Als Name verwenden Sie einen Namen mit Wiedererkennungswert, beispielsweise `DerbyBonbonbude`. Sie kommen weiter mit Next.
- Auf der nächsten Seite geben Sie die Parameter für die Datenbankverbindung an. Bei Derby ist dies der Pfadname, der Benutzername und das Passwort. Die URL wird bereits vorgegeben.
- Der Haken bei Create Database ist für unser Projekt wichtig, da das Java-Programm die Tabelle automatisch erstellen soll.
- Mit Test Connection können Sie prüfen, ob die Datenbank funktioniert.
- Der Klick auf Next liefert Ihnen eine Übersicht. Der Klick auf Finish erstellt die Datenbankverbindung.

- Falls Sie unter Eclipse sogar schon eine JDBC-Verbindung zu einer Datenbank eingerichtet haben, können Sie diese unter der Klappbox finden. Hoffentlich hatten Sie zuvor einen Namen mit Wiedererkennungswert verwendet.
- In beiden Fällen können Sie mit einem Haken bei Add Driver Library TO Build Path eine Datenbank einbinden.

12. Sie können nun den Button Finish anklicken und damit ein Projekt anlegen.

Einhängen der Derby-Verbindung in die persistence.xml

Die Datei findet sich im Projekt unter dem Verzeichnis *JPA Content*. Wenn Sie darauf doppelt klicken, erscheint ein Assistent für die Bearbeitung (siehe Abbildung 24.3).

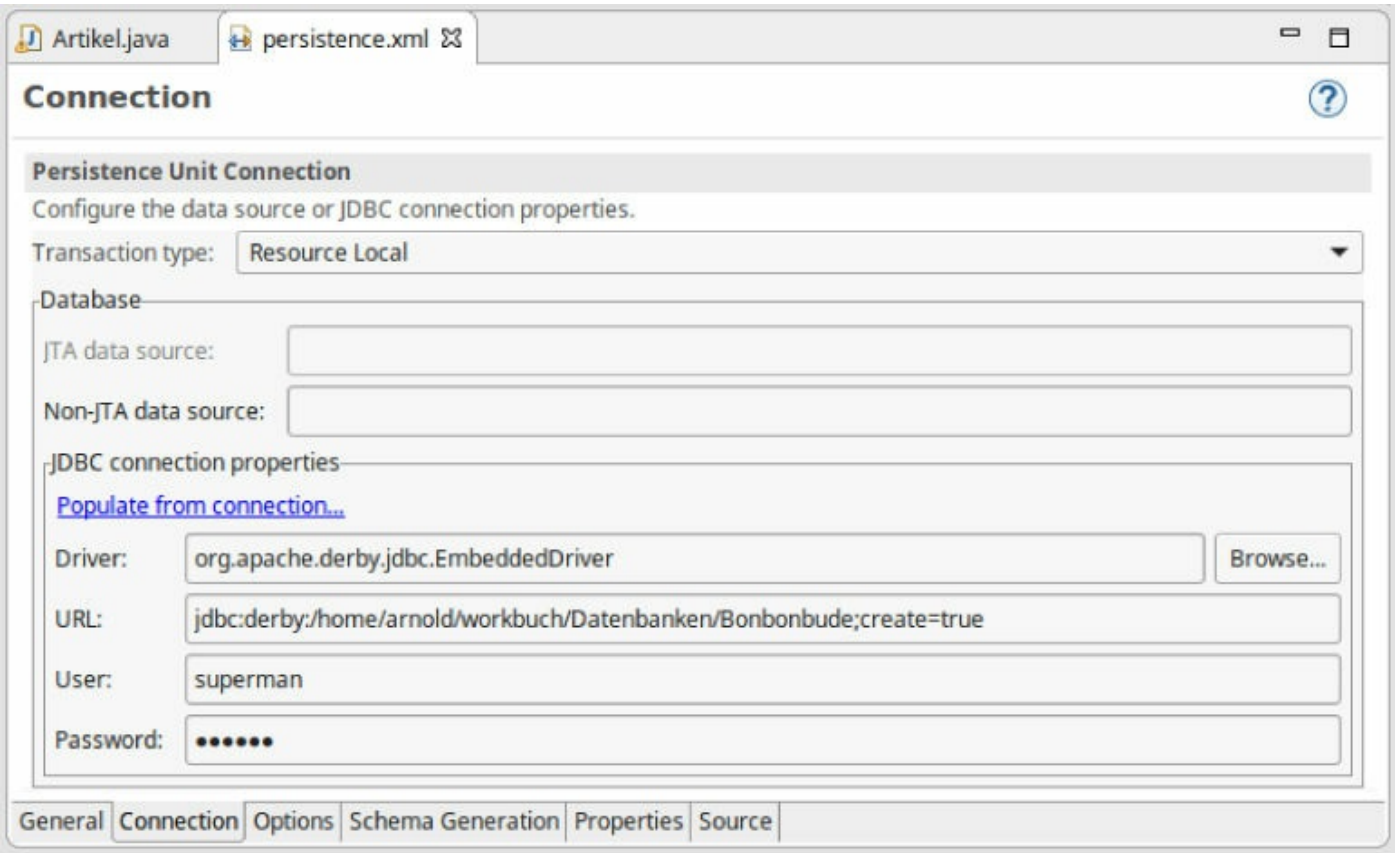


Abbildung 24.3: Der Eclipse-Editor für *persistence.xml*

1. Wählen Sie den Reiter Connection.
2. Transaction Type stellen Sie auf Resource Local.
3. Daraufhin werden die Verbindungseigenschaften aktiv und können bearbeitet werden.
4. Klicken Sie auf Populate From Connection. Es erscheint ein Dialog, aus dem Sie Ihre Datenbankverbindung zu Derby auswählen können, der Sie ja glücklicherweise einen Namen mit hohem Wiedererkennungswert gegeben haben.
5. Im Hauptfeld können Sie User (beispielsweise `superman`) und Password (beispielsweise »geheim«) eingeben.

Erstellen einer Entity

Die Basis für jede Datenbanktabelle stellt eine normale Java Bean dar, also eine einfache Klasse mit einem oder mehreren Attributen. Daneben benötigt die Klasse für ihre privaten Attribute die passenden `set`- und `get`-Methoden und einen Standardkonstruktor. Durch die Markierung mit der Annotation `@Entity` wird sie zur Entity-Klasse.

In einem JPA-Projekt erhalten Sie die Möglichkeit, über das Kontextmenü des Projekts per New eine JPA Entity anzulegen.

1. Auf der ersten Seite geben Sie den Klassennamen an. Wenn Sie nun Finish

wählen, erhalten Sie ein Gerüst für eine Entity, allerdings ohne jegliche Felder. Einfacher ist es, die Felder im Dialog anzugeben.

2. Per Next kommen Sie in den entsprechenden Dialog.
3. Hier können Sie nun per Add Felder und damit Spalten hinzufügen. Nach dem Anlegen ist es auch möglich, eine Spalte als Schlüssel (Key) zu markieren.
4. Wählen Sie nacheinander immer über den Button Add:
    - Type: int Name: id
    - Type: String Name: name
    - Type: int Name: menge
    - Type: double Name: preis

    Zum Schluss versehen Sie `id` mit einem Haken bei Key.
5. Mit Finish schließen Sie die Definition der Entity ab.

Eclipse erzeugt Ihnen eine Java-Entity-Datei:

```java
import java.io.Serializable;
import java.lang.String;
import javax.persistence.*;

/**
 * Entity implementation class for Entity: Artikel
 *
 */
@Entity

public class Artikel implements Serializable {

    @Id
    private int id;
    private String name;
    private int menge;
    private double preis;
    private static final long serialVersionUID = 1L;

    public Artikel() {
        super();
    }
    public int getId() {
        return this.id;
    }

    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }
    public int getMenge() {
        return this.menge;
    }

    public void setMenge(int menge) {
        this.menge = menge;
    }
    public double getPreis() {
        return this.preis;
    }
```

```
    }
}
```

Listing 24.13: Von Eclipse erzeugte Entity-Datei



Es ist wahrscheinlich, dass Eclipse die Anweisung `@Entity` rot färbt und meint, dass es ein Fehler sei, dass das Schema in der Tabelle nicht aufgelöst werden kann. Dieses Henne-Ei-Problem können Sie umgehen, indem Sie die Validierung abschalten:

- Über das Menü Window|Preferences gelangen Sie in den Dialog für die Einstellungen von Eclipse.
- In der linken Spalte suchen Sie den Punkt Validation und wählen ihn an.
- Suchen Sie auf der rechten Seite den Punkt JPA Validator und schalten Sie ihn für Build aus.

Nun suchen Sie die neu erstellte Klasse im Projektverzeichnis *src* und seinem Package-Ordner.

- Klicken Sie sie mit der rechten Maustaste an.
- Wählen Sie JPA-Tools|Add TO Persistence Unit.

In Listing 24.13 sind einige Annotationen gestreut. Sie deklarieren die Bestandteile der Klasse, die mit der Datenbanktabelle korrespondieren. Hier einige der wichtigsten Annotationen, die JPA zur Verfügung stellt.

- `@Entity`: Diese Klasse wird von JPA verwaltet und repräsentiert eine Tabelle. Üblicherweise wird der Klassenname auch als Tabellenname verwendet. Wenn das nicht passen will, können Sie durch die Namenszuweisung einen anderen Tabellennamen wählen.
- `@Id`: Damit wird der Primärschlüssel der Tabelle gekennzeichnet. Datenbanken benötigen solche Schlüssel, um schnell einen Datensatz der Tabelle hervorzaubern zu können. Der Wert muss innerhalb der Tabelle eindeutig sein.
- `@GeneratedValue`: Der Wert eines Schlüssels ist eigentlich weitgehend egal, da er nur dazu dient, den richtigen Eintrag schnell aus der Tabelle zu holen. Darum kann man ihn mit dieser Option automatisch erzeugen lassen. Wenn das Programm aber beispielsweise Waren mit eigenen Artikelnummern belegen will, darf diese Option dort nicht stehen.
- `@Column`: Das markierte Attribut entspricht einer Spalte in der Datenbanktabelle. Das versteht sich allerdings normalerweise von selbst.

- `@Transient`: Wenn Sie ein Attribut in der Tabelle verwenden, das nicht in die Datenbank gehört, können Sie es mit dieser Annotation markieren.

Ein Blick auf die persistence.xml

Die Konfiguratonsdatei *persistence.xml* enthält die Parameter eines JPA-Projekts. Sie finden Sie im Verzeichnis *META-INF* des Projekts. In einem JPA-Projekt von Eclipse wird sie ein zweites Mal unter JPA-Content aufgeführt. Unter Eclipse können Sie viele Einträge über JPA-Tools oder über den eingebauten Assistenten erledigen. Sie können allerdings auch auf die XML-Struktur direkt schauen, wenn Sie auf Source unterhalb des Assistenten klicken.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
 xmlns="http://xmlns.jcp.org/xml/ns/persistence"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
 http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="JpaProjekt"
     transaction-type="RESOURCE_LOCAL">
        <class>Artikel</class>
        <properties>
            <property name="javax.persistence.jdbc.url"
             value="jdbc:derby:/home/arnold/MyDB;create=true"/>
            <property name="javax.persistence.jdbc.user"
             value="superman"/>
            <property name="javax.persistence.jdbc.password"
             value="geheim"/>
            <property name="javax.persistence.jdbc.driver"
             value="org.apache.derby.jdbc.EmbeddedDriver"/>
        </properties>
    </persistence-unit>
</persistence>
```

Listing 24.14: *persistence.xml*

- Zu Anfang stehen einige Verweise auf die Definition der XML-Struktur.
- Die `persistence-unit` wird durch den Projektnamen vorgegeben. Sie wird später von der `EntityManagerFactory` aufgegriffen.
  - `name`: Über den Namen wird die Persistenz mit Java gekoppelt.
  - `transaction-type`:
    - `RESOURce_LOCAL`: Standard bei einer JSE-Anwendung

- `JTA`: Automatismus bei einer JEE-Anwendung

- ✔ Sie sehen, wie die Klasse der Entity eingebunden ist.
- ✔ Für jede Entity werden Eigenschaften (properties) definiert.
  - Der JDBC-Treiber wird eingebunden und mit Parametern versehen.
  - Die Zugangsdaten der Datenbank wie Benutzer, Passwort und URL werden aufgeführt.

Der Entity-Manager und seine Factory

Die Entity-Klasse dient nun als Verbindung zu einer Datenbanktabelle. Aber noch modelliert sie nur. Interessant wird es, wenn Sie Objekte der Entity-Klasse in die Datenbank schreiben oder lesen können. Für diese Aktionen bietet der `EntityManager` passende Methoden an.

Die Verbindung zu Ihrer Datenbank und den Entities erhält der `EntityManager` eher indirekt durch die `EntityManagerFactory`. Sie erzeugt, wie der Name schon sagt, `EntityManager` und verwendet als Parameter die Persistence Unit, die in *persistence.xml* festgelegt wurde und sowohl die Entities als auch den JDBC-Treiber und dessen Parameter kennt.



Es darf nur genau eine `EntityManagerFactory` geben. Auf der anderen Seite muss sie bei Ende des Programms geschlossen werden, damit bei einem Neustart nicht eine zweite `EntityManagerFactory` angelegt wird.

Das Hauptprogramm

```
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class JpaHauptprogramm {

    public static void main(String[] args) {
        EntityManagerFactory factory =
                Persistence.createEntityManagerFactory("Jpa4");
        EntityManager manager = factory.createEntityManager();
        // Testdaten erzeugen
        manager.getTransaction().begin();
        manager.persist(createArtikel("Lutscher", 12, .45));
        manager.persist(createArtikel("Bonbon", 15, .28));
        manager.getTransaction().commit();
        // Daten auslesen und anzeigen
        Query query = manager.createQuery("select t from artikel t");
        List<Artikel> liste = query.getResultList();
        for (Artikel artikel : liste) {
            System.out.println(artikel);
        }
        manager.close();
    }

    private static Artikel createArtikel(String name,
            int menge, double preis) {
        Artikel artikel = new Artikel();
        artikel.setName(name);
        artikel.setMenge(menge);
        artikel.setPreis(preis);
        return artikel;
    }
}
```

Alle Schreiboperationen werden mit Transaktionsbeginn und -ende umrahmt.

Die Methode `persist` erhält als Parameter die Entity, die in der Datenbank abgespeichert werden soll.

Für eine Anfrage wird die Methode `createQuery` aufgerufen. Sie liefert eine `List` von Entities zurück.

Kapitel 25

XML-Dateien

IN DIESEM KAPITEL

Strukturen in Text abbilden

XML-Datei zu einem Objekt machen und umgekehrt

Rekursives Auslesen von anonymen XML-Dateien

An der einen oder anderen Stelle sind Sie sicher bereits über die Abkürzung XML gestolpert. XML steht für *Extensible Markup Language*, also eine erweiterbare Auszeichnungssprache. XML ist der Standard, wenn es darum geht, strukurierte Daten portabel zu speichern oder zu übertragen.

- Strukturiert bedeutet, dass Texte, Zahlen, Arrays und Objektattribute so abgelegt werden, dass sie später wieder in die Ausgangsstrukturen zurückgeführt werden können.
- Portabel bedeutet, dass die Daten von jedem beliebigen Computersystem gelesen und geschrieben werden können.

Das macht XML ideal für das Speichern von Konfigurationen, aber auch für die Übertragung von Daten in Netzwerken.

Die Strukturen von XML-Dateien

- Ein Element besitzt einen Bezeichner, also einen Namen, und einen Inhalt und entspricht damit dem Konzept einer Variablen. Ein Element eines Autos könnte die Marke sein. Der Bezeichner wäre dann vielleicht `Marke` und der Inhalt beispielsweise Renault, Opel oder Ford.
- Ein Element wird durch ein *Tag* eingerahmt. Ein Tag wird als Täg ausgesprochen, weil es so englisch ist. Ein Tag taucht als Start-Tag und als Ende-Tag auf. Beide werden von spitzen Klammern umschlossen, die der Fachmann sofort als Kleiner- und Größerzeichen identifiziert. Das Ende-Tag erkennt man daran, dass ein Schrägstrich der öffnenden Klammer folgt.
  ```
  <Marke>Renault</Marke>
  ```
- Elemente können ineinander verschachtelt werden. Beispielsweise kann ein

Auto eine Marke, einen Typ, einen Preis und Hubraum haben.

```
<Auto>
    <Marke>Renault</Marke>
    <Typ>R4</Typ>
    <Preis>6798.99</Preis>
    <Hubraum>845</Hubraum>
</Auto>
```

- Ein Tag kann Attribute haben. Ein Attribut sieht aus wie eine String-Zuweisung, besteht also aus einem Bezeichner, einem Gleichheitszeichen und einem Wert, der in Anführungszeichen eingeschlossen ist.

```
<Hubraum einheit="ccm">845</Hubraum>
```

- Ein XML-Dokument besitzt genau ein Wurzelelement, im Beispiel ist das `Fuhrpark`. Alle anderen Elemente können beliebig oft in anderen Elementen enthalten sein.

```
<Fuhrpark>
<Auto>
    <Marke>Renault</Marke>
    <Typ>R4</Typ>
    <Preis>6798.99</Preis>
    <Hubraum>845</Hubraum>
</Auto>
<Auto>
    <Marke>Opel</Marke>
    <Typ>Zafira</Typ>
    <Preis>8798.99</Preis>
    <Hubraum>1598</Hubraum>
</Auto>
</Fuhrpark>
```

Beispiel Vokabelkarten

Als Beispiel wollen wir ein Programm betrachten, das dem Anwender beim Vokabellernen hilft. Das Programm bildet einen Vokabelkasten nach, der über mehrere Fächer verfügt, in dem sich kleine Karten zum Lernen befinden. Auf der einen Seite der virtuellen Karte befindet sich die Frage, auf der anderen die Antwort. Das Programm zählt mit, wie oft der Lernende die Antwort gekannt oder nicht gekannt hat.

Das Beispiel in Listing 25.1 stellt die Daten eines Lernprogramms dar. Der fleißige Schüler hat bereits drei Vokabeln erfasst: Heim, Haus und Hund. Das Gesamtkunstwerk ist ein `vokabelkasten`. Darin befinden sich drei Exemplare des Typs `vokabel`.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<vokabelkasten>
   <vokabel fachnr="0" bekannt="0" unbekannt="0">
      <frage>Heim</frage>
      <antwort>home</antwort>
   </vokabel>
   <vokabel fachnr="1" bekannt="23" unbekannt="2">
      <frage>Haus</frage>
      <antwort>house</antwort>
   </vokabel>
   <vokabel fachnr="3" bekannt="22" unbekannt="15">
      <frage>Hund</frage>
      <antwort>dog</antwort>
   </vokabel>
</vokabelkasten>
```

Listing 25.1: XML-Darstellung zweier Vokabeln

Neben den Datenbereichen `frage` und `antwort` besitzt eine Vokabel die Attribute `fachnr`, `bekannt` und `unbekannt`. Prinzipiell könnte man die Attribute auch als Datenfelder verwenden. Der Unterschied liegt darin, dass man Datenfelder wiederum mit eigenen Unterknoten versehen könnte. Das geht bei Attributen nicht.

Letztes Lernen

Nehmen wir an, dass die Vokabelkarte bei jedem Lernen einen Datumsstempel erhält, an welchem Tag sie das letzte Mal betrachtet wurde. Dann müsste in das Element `vokabel` noch ein Element `zuletzt` eingebettet werden, das Tag, Monat und Jahr aufnimmt.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<vokabelkasten>
<vokabel fachnr="1" known="23" unknown="2">
<zuletzt>
  <tag>31</tag>
  <monat>5</monat>
  <jahr>2017</jahr>
</zuletzt>
<frage>Haus</frage>
<antwort>house</antwort>
</vokabel>
...
</vokabelkasten>
```

Listing 25.2: Die Vokabel erhält einen Zeitstempel.

Analoge Klasse

Die XML-Datei würde zu der Klasse in Listing 25.3 passen.

```
class Vokabel {
    String frage;
    String antwort;
    int bekannt;
    int unbekannt;
    int fachnr;
}
```

Listing 25.3: Die passende Klasse Vokabel

Erweiterung

Sollte das Programm in einer späteren Version auch die Lautschrift einer Vokabel aufnehmen wollen, ist das kein Problem. Neben `frage` und `antwort` ist noch Platz für weitere Einträge. Das Programm kann auch die XML-Dateien der älteren Versionen lesen, die Felder für die Lautschrift wären dann eben leer. Es wäre auch möglich, dass das ältere Programm die Dateien der neueren liest. Es wird nur nach Frage und Antwort suchen und das Feld für die Lautschrift einfach ignorieren.

Auch das Verschachteln von Klassen lässt sich per XML nachbilden. So könnte die Vokabel sich vielleicht merken, wann sie das erste Mal erfasst wurde. In der Klasse sähe das so aus:

```
class Datum {
    int tag=1, monat=1, jahr=2000;
}

class Vokabel {
    Datum zuletzt;
    String frage;
    String antwort;
    int bekannt=0;
    int unbekannt=0;
    int fachnr=1;
}
```

Listing 25.4: Die Klasse `Vokabel` enthält ein `Datum`.

Die Aufgabe liegt nun darin, die XML-Datei einzulesen und in Objekte der Klasse zu überführen. Und später wollen Sie vermutlich auch gern die Objekte wieder zu XML-Dateien machen.

Einlesen einer XML-Datei

Theoretisch können Sie eine XML-Struktur natürlich auch selbst auswerten. Dazu müssen Sie die Tags identifizieren, die Ende-Tags zu den passenden Start-Tags synchronisieren und natürlich Fehler in der Struktur aufdecken. Das Auslesen der Klammerstrukturen führt unweigerlich zu einer Rekursion. Ganz ehrlich: Das wollen Sie nicht.

Glücklicherweise gibt es XML schon etwas länger und so gab es vor uns schon Programmierer, die sich mit dem Auslesen von XML-Dateien befasst und Bibliotheken für diese Arbeit vorbereitet haben.

JAXP

Neben vielen anderen Bibliotheken gibt es JAXP. JAXP ist bereits in Java eingebaut und muss nicht als Sonderausstattung geordert werden.

Von der Datei zur Knotenliste

Im ersten Schritt wollen wir eine XML-Datei einlesen, die Elemente identifizieren und auf dem Bildschirm ausgeben.

- Holen Sie sich eine `DocumentBuilderFactory`.
- Sie ahnen schon, dass eine `DocumentBuilderFactory` einen `DocumentBuilder` erzeugen kann.
- Der `DocumentBuilder` verfügt über die Methode `parse`. Sie erwartet als Parameter den Dateinamen der XML-Datei und liefert als Ergebnis ein `Document`. Damit haben Sie den Zugriff auf die XML-Struktur.
- Über den Aufruf `getDocumentElement` erhalten Sie das Wurzelelement, in diesem Fall den Vokabelkasten.
- Das Wurzelelement können Sie durch Aufruf von `getChildNodes` bitten, die Liste aller Elemente zu liefern, und damit haben Sie die Vokabelliste in der Hand, die Sie mit einer Schleife durchlaufen können.

```
DocumentBuilderFactory factory =
        DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse(new File("vokabel.xml"));
// Wir holen das root-Element
Node rootNode = document.getDocumentElement();
NodeList nodeList = rootNode.getChildNodes();
// Durchsuchen aller Unterknoten
for (int i = 0; i < nodeList.getLength(); i++) {
    Node nodeChild = nodeList.item(i);
    // Verarbeite die Knoten hier!
}
```

Listing 25.5: Von der XML-Datei zur Knotenliste

Das Programm wird die Datei *vokabel.xml* einlesen. Es wird sie in dem Verzeichnis suchen, wo es gestartet wurde. Falls Sie das Programm aus Eclipse heraus starten, sucht es die Datei im Wurzelverzeichnis des Projekts. Um die Datei zu erzeugen, klicken Sie das Projekt mit der rechten Maustaste an und wählen New|File und geben den Dateinamen an.

Wie das beim Umgang mit Dateien so ist, wird es nicht ohne Exceptions ablaufen. Darum müssen Sie noch ein `try` mit verschiedenen `catch`-Zweigen einbauen. Die Methode unterscheidet den übergebenen Knoten (`Node`) nach dem Typ. Hier sind in der Regel nur drei Typen wirklich interessant:

- Element: Ein Element ist der Rahmen für den Inhalt. Er hat einen Namen, über den er identifizierbar ist. Dieser steht in den spitzen Klammern und rahmt gegebenenfalls weitere Elemente ein. Er kann also Unterelemente enthalten. Darüber hinaus kann er auch Attribute und natürlich einen Wert haben.
  ```
  <element>inhalt</element>
  <element/>
  <element><unterelement>inhalt</unterelement></element>
  ```
- Attribut: Ein Attribut steht direkt neben dem Elementnamen wie eine Zuweisung einer Stringvariablen.
  ```
  <element attribut="attributwert">inhalt</element>
  ```
- Text oder Daten: Der Wert steht zwischen den Tags eines Elements.
  ```
  <element>text</element>
  ```

XML-Knoten werden Objekte

Aus dem Text mit den vielen spitzen Klammern sollen Objekte werden, die wir in Java weiterverarbeiten können. Der Vokabelkasten, den unsere XML-Struktur beschreibt, besteht in Java aus der Klasse `Vokabel`.

```
public class Vokabel {
    Datum zuletzt = new Datum();
    String frage;
    String antwort;
    int bekannt;
    int unbekannt;
    int fachnr;
    public String toString() {
        return ""+frage+"/"+antwort
            +"("+fachnr+","+bekannt+","+unbekannt+")"
            +"["+zuletzt.tag+"."+zuletzt.monat+"."
            +zuletzt.jahr+"]";
    }
}
```

Listing 25.6: Die Klasse `Vokabel`

Die Struktur bietet keine Überraschung. Allerdings muss für das Datum noch eine weitere Klasse `Datum` geschaffen werden.

```
public class Datum {
    int tag, monat, jahr;
}
```

Listing 25.7: Die Klasse `Datum`

Eine Methode für jede Klasse

Für das Auslesen eines Knotens erstellen Sie jeweils eine eigene Methode. Da die Klasse `Datum` so angenehm übersichtlich ist, zeige ich Ihnen zuerst die Methode `leseDatum`, auch wenn diese erst von der vokabellesenden Methode aufgerufen wird.

```
static void leseDatum(Node node, Datum datum) {
    NodeList datumList = node.getChildNodes();
    for (int i=0; i<datumList.getLength(); i++) {
        Node datumChild = datumList.item(i);
        String wert = datumChild.getTextContent();
        wert = wert.trim();
        String nodeName = datumChild.getNodeName();
        if (nodeName.equals("tag")) {
            datum.tag = Integer.parseInt(wert);
        } else if (nodeName.equals("monat")) {
            datum.monat = Integer.parseInt(wert);
        } else if (nodeName.equals("jahr")) {
            datum.jahr = Integer.parseInt(wert);
        }
    }
}
```

Listing 25.8: Methode zum Auslesen eines Datums

Die Methode beginnt bei einem Datumsknoten.

- ✔ Die Methode `getChildNodes` ermittelt die Liste aller Unterknoten. Die Unterknoten werden die Bestandteile des Datums, nämlich Tag, Monat und Jahr enthalten.
- ✔ Mit der Methode `getTextContent` holen Sie zunächst den Inhalt zwischen den Tags.
- ✔ Damit Sie wissen, ob es sich um den Tag, den Monat oder das Jahr handelt, müssen Sie den Knotennamen mit `getNodeName` ermitteln.
- ✔ Mit einer gestaffelten Abfrage ordnen Sie den Wert dem richtigen Objekt-Attribut zu.

Nachdem Sie die Auslesemethode für das Datum gesehen haben, werden Sie die Bausteine in der Vokabel-Methode wiedererkennen. Sie beginnt genau wie die Datumsmethode aus Listing 25.8 und wertet die Unterknoten nach den Feldern `frage`, `antwort` und `zuletzt` aus. Letzeres ist das Datum, für das die Methode `leseDatum` aus Listing 25.8 aufgerufen wird.

Im zweiten Teil liest die Methode die Attribute des Knotens durch den Aufruf von `getAttributes` aus. Im Unterschied zu den Unterknoten wird hier eine Map verwendet. Der Mechanismus ist aber prinzipiell gleich.

```
static void leseVokabel(Node node, Vokabel vokabel) {
    NodeList nodeList = node.getChildNodes();
    for (int i = 0; i < nodeList.getLength(); i++) {
        Node childNode = nodeList.item(i);
        String str = childNode.getTextContent();
        str = str.trim();
        if (!str.isEmpty()) {
            if (childNode.getNodeName().equals("frage")) {
                vokabel.frage = str;
            } else if (childNode.getNodeName().equals("antwort")) {
                vokabel.antwort = str;
            } else if (childNode.getNodeName().equals("zuletzt")) {
                leseDatum(childNode, vokabel.zuletzt);
            }
        }
    }
    NamedNodeMap attribs = node.getAttributes();
    for (int i = 0; i < attribs.getLength(); i++) {
        Node attChild = attribs.item(i);
        String name = attChild.getNodeName();
        String value = attChild.getNodeValue();
        if (name.equals("fachnr")) {
            vokabel.fachnr = Integer.parseInt(value);
        } else if (name.equals("bekannt")) {
            vokabel.bekannt = Integer.parseInt(value);
        } else if (name.equals("unbekannt")) {
            vokabel.unbekannt = Integer.parseInt(value);
        }
    }
}
```

Listing 25.9: Auslesen der Vokabeln

Zu guter Letzt muss die Methode `leseVokabel` noch mit dem Hauptprogramm verbunden werden. In Listing 25.5 hatte ich Sie mit einem Kommentar vertröstet, den wir nun füllen können.

```
for (int i = 0; i < nodeList.getLength(); i++) {
    Node nodeChild = nodeList.item(i);
    Vokabel vokabel = new Vokabel();
    leseVokabel(nodeChild, vokabel);
    System.out.println("Vokabel: " + vokabel);
}
```

Listing 25.10: Aufruf von `leseVokabel`

Zu guter Letzt setzen wir die Fragmente zu einem kompletten Programm zusammen.

```
import java.io.File;
import java.io.IOException;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

public class ReadVokXml {
    public static void main(String[] argv) {
        try {
            DocumentBuilderFactory factory =
                    DocumentBuilderFactory.newInstance();
            DocumentBuilder builder =
                    factory.newDocumentBuilder();
            Document document = builder.parse(
                    new File("vokabel.xml"));
            NodeList nodeList =
                    document.getElementsByTagName("vokabel");
            for (int i = 0; i < nodeList.getLength(); i++) {
                Node nodeChild = nodeList.item(i);
                Vokabel vokabel = new Vokabel();
                leseVokabel(nodeChild, vokabel);
                System.out.println("Vokabel: " + vokabel);
            }
        } catch (SAXParseException spe) {
            System.out.println("Parse-Fehler Zeile: "
                    + spe.getLineNumber() + ", uri "
                    + spe.getSystemId());
        } catch (SAXException sxe) {
            sxe.printStackTrace();
        } catch (ParserConfigurationException pce) {
            pce.printStackTrace();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }

    static void leseVokabel(Node node, Vokabel vokabel) {
        NodeList nodeList = node.getChildNodes();
        for (int i = 0; i < nodeList.getLength(); i++) {
```

```
            Node childNode = nodeList.item(i);
            String str = childNode.getTextContent();
            str = str.trim();
            if (!str.isEmpty()) {
                if (childNode.getNodeName().
                        equals("frage")) {
                    vokabel.frage = str;
                } else if (childNode.getNodeName().
                        equals("antwort")) {
                    vokabel.antwort = str;
                } else if (childNode.getNodeName().
                        equals("zuletzt")) {
                    leseDatum(childNode, vokabel.zuletzt);
                }
            }
        }
        NamedNodeMap attribs = node.getAttributes();
        for (int i = 0; i < attribs.getLength(); i++) {
            Node attChild = attribs.item(i);
            String name = attChild.getNodeName();
            String value = attChild.getNodeValue();
            if (name.equals("fachnr")) {
                vokabel.fachnr = Integer.parseInt(value);
            } else if (name.equals("bekannt")) {
                vokabel.bekannt = Integer.parseInt(value);
            } else if (name.equals("unbekannt")) {
                vokabel.unbekannt = Integer.parseInt(value);
            }
        }
    }

    static void leseDatum(Node node, Datum datum) {
        NodeList datumList = node.getChildNodes();
        for (int i = 0; i < datumList.getLength(); i++) {
            Node datumChild = datumList.item(i);
            String wert = datumChild.getTextContent();
            wert = wert.trim();
            String nodeName = datumChild.getNodeName();
            if (nodeName.equals("tag")) {
                datum.tag = Integer.parseInt(wert);
            } else if (nodeName.equals("monat")) {
                datum.monat = Integer.parseInt(wert);
```

```
            } else if (nodeName.equals("jahr")) {
                datum.jahr = Integer.parseInt(wert);
            }
        }
    }
}
```

Listing 25.11: Auslesen von Vokabeln aus der XML-Datei

XML-Dateien schreiben

Das Erzeugen von XML-Dateien ist wesentlich einfacher als das Lesen. Sie könnten es mit ein paar gut platzierten Schreibbefehlen vermutlich auch ohne Bibliotheksunterstützung hinbekommen. Der Ablauf erinnert an vielen Stellen an das Auslesen eines XML-Dokuments, nur halt andersherum. Wirklich verwunderlich ist das aber nicht.

1. Das Programm erzeugt ein leeres XML-Dokument. Dazu wird die `DocumentBuilderFactory` gebeten, einen `DocumentBuilder` zu liefern. Dieser erzeugt durch den Aufruf von `newDocument()` das XML-Dokument, das die Daten aufnehmen soll.
2. Für das XML-Dokument wird das Root-Element erzeugt. Es stellt für alle Dateneinträge den Sammelbehälter dar, in den per `appendChild()` alle Vokabeln eingehängt werden. Zum Schluss wird das Root-Element wieder per `appendChild()` an das Dokument gehängt.
3. Es folgt die Schleife, die alle Vokabeln aus der Vokabelliste abarbeitet und jeweils in ein Element verpackt. Ein Element ist eine Sonderform eines `Node`. Die Arbeit übernimmt die Methode `createVokabel`.
4. Nachdem das Root-Element an das Dokument gehängt wurde, kann nun das Schreiben der XML-Datei eingeleitet werden. Dazu wird ein Transformer von der `TransformerFactory` geliefert. Dieser benötigt nun als Datenquelle eine `DOMSource`, die aus dem XML-Dokument gewonnen wird, und eine Datei, in die alles geschrieben wird, um aus dem Dokument eine XML-Datei zu erzeugen.

```
public class WriteVokXml {

    static Vokabel createVok(String frage, String antwort) {
        Vokabel vok = new Vokabel();
        vok.frage = frage;
        vok.antwort = antwort;
        vok.zuletzt.jahr = 2018;
        return vok;
    }
```

```
public static void main( String[] argv ) {
    ArrayList<Vokabel> vokabeln = new ArrayList<Vokabel>();
    vokabeln.add(createVok("Uhr", "clock"));
    vokabeln.add(createVok("Katze", "Igitt"));
    try {
        DocumentBuilderFactory factory
                = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder;
        builder = factory.newDocumentBuilder();
        Document doc = builder.newDocument();
        Element root = doc.createElement("vokabelkasten");
        for (Vokabel vokabel : vokabeln) {
            Element element = createVokabel(doc, vokabel);
            root.appendChild(element.cloneNode(true));
        }
        doc.appendChild(root);
        // Schreibe Document in eine Datei
        Transformer transformer = TransformerFactory
                .newInstance().newTransformer();
        DOMSource source = new DOMSource(doc);
        FileOutputStream os = new FileOutputStream(
                new File("ausgabe.xml"));
        StreamResult result = new StreamResult(os);
        transformer.setOutputProperty(
                OutputKeys.INDENT, "yes");
        transformer.setOutputProperty(
                OutputKeys.ENCODING, "UTF-8");
        transformer.transform(source, result);
    } catch (TransformerConfigurationException e) {
        e.printStackTrace();
    } catch (TransformerFactoryConfigurationError e) {
        e.printStackTrace();
    } catch (TransformerException e) {
        e.printStackTrace();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (ParserConfigurationException e) {
        e.printStackTrace();
    }
}
```

Listing 25.12: Das Hauptprogramm für das Schreiben einer XML-Datei

Vokabeln in die XML-Struktur einbetten

In der Schleife wird für jede Vokabel der Liste die Methode `createVokabel`

aufgerufen.

1. Es werden die Datenfelder für die Elemente eingesetzt.
    - Dazu wird zunächst ein Element für jedes Datenfeld per `createElement()` erstellt.
    - Der Dateninhalt wird über die Methode `setTextContent()` an das Element gehängt.
    - Das Element wird mit dem Vokabel-Element verknüpft.
2. Es werden die Attribute für die Vokabel mit dem Aufruf von `setAttribute()` festgelegt.
3. Die Unterstruktur für das Datum wird erzeugt, sofern sie vorhanden ist. Das Datum wird prinzipiell genauso behandelt wie eine Vokabel, nur dass sie an das Vokabel-Element statt an das Root-Element mit `appendChild()` gehängt wird.

```
static Element createVokabel(Document doc, Vokabel vokabel) {
    Element frage = doc.createElement("frage");
    frage.setTextContent(vokabel.frage);
    Element antwort = doc.createElement("antwort");
    antwort.setTextContent(vokabel.antwort);
    Element element = doc.createElement("vokabel");
    element.appendChild(frage.cloneNode(true));
    element.appendChild(antwort.cloneNode(true));
    element.setAttribute("fachnr", "" + vokabel.fachnr);
    element.setAttribute("bekannt", "" + vokabel.bekannt);
    element.setAttribute("unbekannt", ""+vokabel.unbekannt);
    Element zuletzt = createDatum(doc, vokabel.zuletzt);
    element.appendChild(zuletzt.cloneNode(true));
    return element;
}
```

Listing 25.13: Die Vokabel in die XML-Struktur bringen

In den letzten Zeilen ruft die Methode `createVokabel` die Methode `createDatum`, um das Datum der letzten Bearbeitung in die XML-Struktur zu schreiben. Da passiert allerdings nichts wirklich Neues.

```
static Element createDatum(Document doc, Datum datum) {
    Element zuletzt = doc.createElement("zuletzt");
    Element tag = doc.createElement("tag");
    tag.setTextContent("" + datum.tag);
    zuletzt.appendChild(tag.cloneNode(true));
    Element monat = doc.createElement("monat");
    monat.setTextContent("" + datum.monat);
    zuletzt.appendChild(monat.cloneNode(true));
    Element jahr = doc.createElement("jahr");

    jahr.setTextContent("" + datum.jahr);
    zuletzt.appendChild(jahr.cloneNode(true));
    return zuletzt;
}
```

Listing 25.14: Die Methode `createDatum`

Wunderbar! Nun haben Sie eine XML-Datei erzeugt. Zum Test können Sie sie ja wieder einlesen. Bei mir hat es jedenfalls geklappt.

Eine beliebige XML-Datei auslesen

Die XML-Datei repräsentiert normalerweise Daten, die in Objekten gespeichert werden. Sie können JAXP aber auch verwenden, um eine beliebige XML-Datei auszulesen.

Das Eröffnen der XML-Datei erfolgt auf dem gleichen Weg. Wirklich neu ist der Aufruf der Methode `zeigeNode`. Ihr Parameter ist ein Knoten. Im Hauptprogramm wird ihr jeder Kindknoten der Wurzel übergeben.

```
import java.io.File;
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

public class ReadXml {

    public static void main(String[] argv) {
        try {
            DocumentBuilderFactory factory =
                    DocumentBuilderFactory.newInstance();
            DocumentBuilder builder = factory.newDocumentBuilder();
            Document document = builder.parse(
                    new File("vokabel.xml"));
            Node rootNode = document.getDocumentElement();
            NodeList nodeList = rootNode.getChildNodes();
            for (int i = 0; i < nodeList.getLength(); i++) {
                Node nodeChild = nodeList.item(i);
                zeigeNode(nodeChild);
            }
        } catch (SAXParseException spe) {
            System.out.println("Parse-Fehler Zeile: " +
                    spe.getLineNumber() + ", uri " +
                    spe.getSystemId());
        } catch (SAXException sxe) {
            sxe.printStackTrace();
        } catch (ParserConfigurationException pce) {
            pce.printStackTrace();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
    // ...
}
```

Listing 25.15: Das vollständige Hauptprogramm

Wenn Knoten Unterknoten haben und darunter wieder Unterknoten denkbar sind,

dann spricht der Informatiker von einer Baumstruktur. Baumstrukturen lassen sich eigentlich nur rekursiv vernünftig auswerten, also durch Methoden, die sich selbst aufrufen.

Die Methode `zeigeNode` ist rekursiv und lässt sich auf alle XML-Dateien anwenden. Sie zeigt die Knoten und Attribute einer XML-Struktur auf dem Bildschirm an.

```
static void zeigeNode(Node node) {
    short typ = node.getNodeType();
    switch (typ) {
    case Node.ELEMENT_NODE:
        String sNodeName = node.getNodeName();
        System.out.println("NodeName: " + sNodeName);
        if (node.hasChildNodes()) { // Unterknoten?
            NodeList nodeList = node.getChildNodes();
            for (int i=0; i<nodeList.getLength(); i++) {
                zeigeNode(nodeList.item(i));
            }
        }
        NamedNodeMap attribs = node.getAttributes();
        for (int i=0; i<attribs.getLength(); i++) {
            Node attChild = attribs.item(i);
            zeigeNode(attChild); // rekursiv
        }
        break;
    case Node.ATTRIBUTE_NODE:
        System.out.print("Attribut: " + node.getNodeName());
        System.out.println(" Wert: " + node.getNodeValue());
        break;
    case Node.TEXT_NODE:
        String str = node.getTextContent();
        str = str.trim(); // Störende Leerzeichen entfernen
```

```
            if (!str.isEmpty()) {
                System.out.println("Textinhalt: " + str);
            }
            break;
        case Node.NOTATION_NODE:
        case Node.COMMENT_NODE:
        case Node.DOCUMENT_NODE:
        default:
        }
}
```

Listing 25.16: Anzeigen einer XML-Datei

Als Ausgabe erhalten Sie eine Liste aller Knoten, Attribute und ihrer Werte.

Kapitel 26

JSON

IN DIESEM KAPITEL

- JSON als portables Austauschformat von Daten
- Java und die JSON-Bibliothek Org.Json
- Datentausch per Konstruktor
- JSON als Internetstandard

Das Format JSON hat sich in den letzten Jahren als schlanke Alternative zu XML immer mehr verbreitet. Ursprünglich scheint es aus dem JavaScript-Umfeld zu kommen, jedenfalls deutet der Name *JavaScript Object Notation* darauf hin. Dennoch ist JSON auch von anderen Sprachen nutzbar und auch Java unterstützt es.

Da JSON nicht so viele spitze Klammern verwendet und nicht überall ein Tag herumliegt, wirkt es schlanker und leichter lesbar, wenn man das Pech hat, in die Datei hineinschauen zu müssen.

Der JSON-Sprachschatz

Aus Arrays und Objekten lassen sich die Java-Datenstrukturen nachbilden.

- ✔ Ein Objekt wird durch geschweifte Klammern eingeschlossen.
- ✔ Ein Element des Objekts besteht aus einem Namen, gefolgt von einem Wert. Beide werden getrennt durch einen Doppelpunkt und sind jeweils von Anführungszeichen umgeben.
- ✔ Ein Objekt kann mehrere dieser Elemente aufnehmen, sie müssen dazu durch Kommata getrennt werden.

Als Beispiel betrachten wir eine Person mit der Neigung zu falschen Angaben.

```
{ "name": "Arnold Willemer", "alter": "12", "ort": "Norgaardholz" }
```

Ein Array nimmt mehrere gleichartige Elemente auf. Das Array erhält einen Namen, die Elemente allerdings enthalten lediglich Werte.

- ✔ Ein Array wird durch eckige Klammern eingeschlossen.

- ✔ Ein Array kann beliebig viele Werte aufnehmen. Alle Werte werden in Anführungszeichen eingeschlossen.
- ✔ Enthält das Array mehrere Werte, werden sie durch Kommata getrennt.

Als Beispiel sehen Sie die großartigsten Bands dieses Planeten.

```
[ "The Beatles", "ABBA", "Beatwatt" ]
```

Ein Objekt kann als Wert wiederum ein Objekt oder ein Array verwenden. Dadurch sind beliebige Verschachtelungen möglich.

```
{
  "name": "Arnold Willemer",
  "instrumente" [ "Gitarre", "Mundharmonika", "Kazoo" ]
  "ort":  "Norgaardholz",
  "sprachen" [
    { "sprache": "deutsch", "kenntnis": "gut" }
    { "sprache": "englisch", "kenntnis": "gut" }
    { "sprache": "plattdeutsch", "kenntnis": "schlecht" }
  ]
}
```

Ein Syntaxdiagramm auf http://json.org stellt die komplette Beschreibung dar.

JSON in Java

Wenn Sie JSON in Java einsetzen wollen, benötigen Sie eine Bibliothek. Tatsächlich gibt es sogar mehrere Implementationen von JSON. Für die Beispiele habe ich Org.Json verwendet. Eine passende JAR-Datei können Sie unter folgender URL herunterladen.

http://mvnrepository.com/artifact/org.json/json

Leider ist die JAR-Datei unter dem Begriff Bundle verborgen. Sobald Sie darauf klicken, bietet die Seite den Download an.

Einbindung in Eclipse

Die JAR-Datei muss in das Projekt eingebunden werden und über den Build Path dem Compiler bekannt gemacht werden. Nach dem Download der JAR-Datei sind also die folgenden Schritte durchzuführen:

1. Legen Sie ein normales Java-Projekt an.
2. Klicken Sie es anschließend mit der rechten Maustaste an und wählen Sie New|Folder. Geben Sie dem Verzeichnis den Namen *lib* und bestätigen Sie es mit Finish.

3. Kopieren oder ziehen Sie die heruntergeladene JAR-Datei in das Projektverzeichnis *lib*.
4. Klicken Sie das Verzeichnis *lib* an, damit Sie die JAR-Datei sehen können.
5. Klicken Sie die JAR-Datei mit der rechten Maustaste an und wählen Sie Build Path|Add TO Build Path.

Aufbau der JSON-Struktur

Für JSON wird im Speicher ein Objekt beziehungsweise für das Array ein JSON-Array erzeugt. Die Attribute werden ähnlich wie bei einer `Map` über einen Schlüssel gespeichert.

Das Objekt und das JSONObject

- Ein Objekt, das in JSON codiert werden soll, wird durch ein `JSONObject` repräsentiert.
- Die Attribute des Objekts werden jeweils durch die Methode `put` hinzugefügt. Die Methode `put` erwartet zwei Parameter:
  - Der erste Parameter ist ein String, der den Schlüssel darstellt, über den das Attribut angesprochen werden kann. Typischerweise wird dazu der Attributname verwendet.
  - Der zweite Parameter erwartet den Attributwert vom Typ `Object`. Damit können alle Objekte übernommen werden. Allerdings müssen primitive Datentypen mit einem Wrapper umgeben werden.

Auf diese Weise lässt sich schnell eine Struktur für eine Vokabelkarte erstellen, wie wir sie schon in Kapitel 25 über XML verwendet haben. Das folgende Beispiel erzeugt aus dem `JSONObject` einen String per `toString`, der sich als JSON-String zur Datenübertragung eignet.

```
import org.json.JSONObject;

public class SimpleVokabelJson {
    public static void main(String[] args) {
        JSONObject jobj = new JSONObject();
        jobj.put("frage", "Buch");
        jobj.put("antwort", "book");
        jobj.put("fachnr", new Integer(2));
        String jsonString = jobj.toString();
        System.out.println(jsonString);
    }
}
```

Listing 26.1: Erzeugen eines JSON-Objekts

Das Wiedereinlesen oder Parsen eines JSON-Strings erfolgt dadurch, dass man den String dem Konstruktor von `JSONObject` als Parameter übergibt. Das Auslesen eines Wertes erfolgt mit der typangepassten `get`-Methode. Als erster Parameter muss der Schlüssel angegeben werden.

```
JSONObject jvok = new JSONObject(jsonString);
Vokabel vok = new Vokabel();
vok.frage = jvok.getString("frage");
vok.antwort = jvok.getString("antwort");
vok.fachnr  = jvok.getInt("fachnr");
System.out.println(vok);
```

Listing 26.2: Auslesen eines JSON-Strings

Das Array und das JSONArray

Für die Codierung und Decodierung eines Arrays verwenden Sie die Klasse `JSONArray`. Dieses wird dann die Array-Elemente aufnehmen. Dennoch benötigen Sie hier zunächst ein `JSONObject`, um für das Array selbst einen Schlüssel angeben zu können. Schließlich soll das Array ja auch einen Namen erhalten.

1. Beim Erzeugen der JSON-Struktur verwenden Sie das Array als Parameter für den Konstruktor von `JSONArray`.
2. Nun erzeugen Sie ein `JSONObject`.
3. Ihm fügen Sie das `JSONArray`-Objekt über die Methode `put` hinzu und geben dabei als ersten Parameter den Namen des Arrays an.

Beginnen wir mit einem einfachen Array aus Zahlen.

```
import org.json.JSONArray;
import org.json.JSONObject;

public class JsonArray {

    public static void main(String[] args) {
        int[] quadrate = { 1, 4, 9, 16, 25 };
        JSONArray qarr = new JSONArray(quadrate);
        System.out.println(qarr.toString());
        // Ergibt: [1,4,9,16,25]
        JSONObject qobj = new JSONObject();
        qobj.put("quadrate", qarr);
        String js = qobj.toString();
        System.out.println(js);
        // Ergibt: {"quadrate":[1,4,9,16,25]}
    }
}
```

Listing 26.3: Erzeugen eines JSON-Arrays

1. Zum Auslesen des JSON-Strings mit einem Array erzeugen Sie wie zuvor ein `JSONObject`, dem Sie als Parameter den JSON-String übergeben.
2. Sie geben den Schlüssel des Arrays an, um mit der Methode `getJSONArray` das `JSONArray` zu bekommen.
3. Darüber erzeugen Sie eine Schleife. Mit der Methode `length` erhalten Sie die Anzahl der Elemente.
4. Innerhalb der Schleife können Sie die Elemente mit der typspezifischen `get`-Methode ermitteln. Als Parameter übergeben Sie den Index.

```
JSONObject obj = new JSONObject(js);
JSONArray jarr = obj.getJSONArray("quadrate");
int neuquad[] = new int[quadrate.length];
for (int i=0; i<jarr.length(); i++) {
    neuquad[i] = jarr.getInt(i);
}
for (int qzahl : neuquad) {
    System.out.println(qzahl);
}
```

Listing 26.4: Auslesen eines JSON-Arrays

Klassen sorgen selbst für JSON

Um eine Klasse dazu zu bringen, sich selbst um die JSON-Codierung zu kümmern, bedarf es nicht viel. Man kann einen Konstruktor mit einem Parameter der Klasse `JSONObject` erstellen. Auf der anderen Seite kann man eine Methode `toJSONObject` definieren, die analog zu `toString` eine JSON-Repräsentation herausgibt.

Der Vokabelkasten

Wie schon in Kapitel 25 greifen wir den Vokabelkasten auf, der viele Vokabelkarten enthält, die jeweils wieder ein Datum des letzten Zugriffs enthalten. Und da das Datum das einfachste ist, fangen wir damit an:

```
public class Datum {
    int tag=1, monat=1, jahr=2000;
}
```

Listing 26.5: Ein Datum mit JSON-Fähigkeiten

Das ist natürlich das absolute Minimum für eine Datumsklasse. Wenn es nur darum geht, eine Klasse für das Modellieren zu bauen, kann man das tolerieren. Aber Sie werden intuitiv natürlich beklagen, dass die Attribute nicht privat sind und Getter und Setter fehlen.

- ✔ Versetzen Sie also die Attribute in den Zustand `private`.
- ✔ Klicken Sie mit der rechten Maustaste in den Editor und wählen Sie Source|Generate Getters And Setters.
- ✔ Klicken Sie im Dialog auf Select All und dann auf OK.

Nun bauen Sie einen Konstruktor, der als Parameter ein `JSONObject` übernimmt und daraus die Attribute erzeugt. Einen leeren Standardkonstruktor sollten Sie gleich dazubauen, sonst können Sie kein Datum mehr ohne diesen Parameter erzeugen.

```
public Datum() {
}

public Datum(JSONObject obj) {
    this.tag = obj.getInt("tag");
    this.monat = obj.getInt("monat");
    this.jahr = obj.getInt("jahr");
}
```

Listing 26.6: Ein Datum aus einem `JSONObject` konstruieren

Eine so erstellte Klasse können Sie dem Konstruktor von `JSONObject` übergeben,

um daraus direkt ein `JSONObject` zu generieren. Anders herum können Sie ein `JSONObject` als Parameter für den Konstruktor verwenden, um ein fertig bestücktes Datum zu erzeugen:

```
Vokabel vokabel = new Vokabel();
JSONObject vokJsonObj = new JSONObject(vokabel);
// Und auch andersherum
Vokabel nochEineVokabel = Vokabel(vokJsonObj);
```

Ein paar Bedingungen müssen erfüllt sein, damit `JSONObject` eine Klasse im Konstruktor akzeptiert:

- ✔ Alle öffentlichen Methoden ohne Parameter, die mit `get` beginnen, werden verwendet.
- ✔ Am Rückgabetyp werden auch Arrays erkannt.
- ✔ Ist der Rückgabetyp aus einer anderen Klasse gebildet, werden auch dort die `get`-Methoden herangezogen.
- ✔ Als Schlüssel wird der Name der Methode ohne das `get` verwendet. Der erste Buchstaben wird in Kleinbuchstaben übersetzt. Aus `getTag` wird also »tag«.

Die Klasse `Vokabel` wird etwas komplexer, weil sie ein Objekt einer anderen Klasse enthält, nämlich eines der Klasse `Datum`.

```
import org.json.JSONObject;

public class Vokabel {
    private Datum zuletzt = new Datum();
    private String frage;
    private String antwort;
    private int fachnr=1;

    public Vokabel() {
    }

    public Vokabel(JSONObject obj) {
        this.frage   = obj.getString("frage");
        this.antwort = obj.getString("antwort");
        this.fachnr  = obj.getInt("fachnr");
        this.zuletzt = new Datum((JSONObject)obj.get("zuletzt"));
    }

    public Datum getZuletzt() {
        return zuletzt;
    }

    public void setZuletzt(Datum zuletzt) {
        this.zuletzt = zuletzt;
    }

    public String getFrage() {
        return frage;
    }

    public void setFrage(String frage) {
        this.frage = frage;
    }

    public String getAntwort() {
        return antwort;
    }

    public void setAntwort(String antwort) {
        this.antwort = antwort;
    }

    public int getFachnr() {
        return fachnr;
    }

    public void setFachnr(int fachnr) {
```

```
        this.fachnr = fachnr;
    }

    public String toString() {
        return ""+frage+"/"+antwort+"("+fachnr+")"
            +"["+zuletzt.getTag()+"."+zuletzt.getMonat()+"."
            +zuletzt.getJahr()+"]";
    }
}
```

Listing 26.7: Die Vokabel mit JSON-Fähigkeiten

Mit dem folgenden Programm wird nun ein Vokabelkasten erstellt, der dann in einen JSON-String umgewandelt wird. Anschließend wird aus dem String wieder eine Liste von Vokabeln gemacht.

```
import java.util.ArrayList;
import org.json.JSONArray;
import org.json.JSONObject;

public class JsonWrite {

    static Vokabel createVok(String frage, String antwort) {
        Vokabel vok = new Vokabel();
        vok.setFrage(frage);
        vok.setAntwort(antwort);
        vok.getZuletzt().setJahr(2018);
        return vok;
    }

    public static void main(String[] args) {
        ArrayList<Vokabel> vokabeln = new ArrayList<Vokabel>();
        vokabeln.add(createVok("Uhr", "clock"));
        vokabeln.add(createVok("Katze", "Igitt"));
        JSONArray jlist = new JSONArray();
        for (Vokabel vokabel : vokabeln) {
            jlist.put(new JSONObject(vokabel));
        }
        JSONObject jkasten = new JSONObject();
        jkasten.put("vokabelkasten", jlist);
        String jvokkasten = jkasten.toString();

        System.out.println(jvokkasten);

        JSONObject vokkasten = new JSONObject(jvokkasten);
        JSONArray voks = vokkasten.getJSONArray("vokabelkasten");
        for (int i=0; i<voks.length(); i++) {
            JSONObject jvok = (JSONObject) voks.get(i);
            Vokabel vok = new Vokabel(jvok);
            System.out.println(vok);
        }
    }
}
```

Listing 26.8: Testprogramm für Vokabel und Datum

Eine Person wird zur Datei

Der Vokabelkasten demonstriert ein Array von Objekten. Allerdings fehlt der umgekehrte Ansatz, nämlich ein Objekt, das ein Array enthält. Darum hier das Beispiel einer Person, die mehrere Instrumente spielt und mehrere Sprachen beherrscht.

Für jede Sprache gibt es ein Objekt, das den Namen der Sprache und den Kenntnisstand in Schulnoten aufnimmt. Natürlich sind die Attribute privat und alle Getter und Setter wurden durch Source|Generate Getters And Setters erzeugt. Schließlich wurde ein Konstruktor eingebaut, der ein `JSONObject` als Parameter auswertet. Der Standardkonstruktor muss erwähnt werden, sonst ist eine einfache Erzeugung der Sprache nicht mehr möglich, und ein Konstruktor, der die Sprache und den Kenntnisstand erlaubt, verkürzt das Testprogramm und spart die Setter.

```
import org.json.JSONObject;

public class Sprache {
    private String sprache = "";
    private int kenntnis = 6;
    public Sprache() {
    }
    public Sprache(String sprache, int kenntnis) {
        this.sprache = sprache;
        this.kenntnis = kenntnis;
    }
    public Sprache(JSONObject jobj) {
        sprache = jobj.getString("sprache");
        kenntnis = jobj.getInt("kenntnis");
    }
    public String getSprache() {
        return sprache;
    }
    public int getKenntnis() {
        return kenntnis;
    }
}
```

Listing 26.9: Die Klasse Sprache

Bei den Instrumenten, die die Person beherrscht, machen wir weniger Aufwand. Wir verwenden einfach eine `ArrayList` von Strings. Für die JSON-Auswertung ist der Konstruktor mit dem Parameter `JSONObject` besonders interessant.

```
import java.util.ArrayList;
import java.util.Iterator;

import org.json.JSONArray;
import org.json.JSONObject;
public class Person {
    private String name = "";
    private int alter = 0;
    ArrayList<String> instrumente = new ArrayList<String>();
    ArrayList<Sprache> sprachen = new ArrayList<Sprache>();

    public String getName() {
        return name;
    }
    public ArrayList<String> getInstrumente() {
        return instrumente;
    }
    public int getAlter() {
        return alter;
    }
    public ArrayList<Sprache> getSprachen() {
        return sprachen;
    }
    public Person() {
    }
    public Person(JSONObject jsonObject) {
        this.name = jsonObject.getString("name");
        this.alter = jsonObject.getInt("alter");
        JSONArray instrJson = jsonObject.getJSONArray("instrumente");
        for (int i=0; i<instrJson.length(); i++) {
            instrumente.add(instrJson.getString(i));
        }
        JSONArray sprachenJson = jsonObject.getJSONArray("sprachen");
        for (int i=0; i<sprachenJson.length(); i++) {
            JSONObject jsonSprache = sprachenJson.getJSONObject(i);
            sprachen.add(new Sprache(jsonSprache));
        }
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAlter(int alter) {
        this.alter = alter;
    }
}
```

Listing 26.10: Die Klasse Person mit ihren Fähigkeiten

Das Hauptprogramm erzeugt nun eine Person, beurteilt deren Eigenschaften, wobei

es zur Schwindelei neigt. Viel interessanter ist aber, dass es aus der Person eine JSON-Datei generiert, diese anschließend wieder einliest und ein Duplikat dieser Person erstellt.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import org.json.JSONObject;

public class JsonSpielerei {
    public static void main(String[] args) {
        Person person = new Person();
        // Wir verändern nun ein paar Elemente der Person
        person.setName("Arnold Willemer");
        person.instrumente.add("Gitarre");
        person.instrumente.add("Mundharmonika");
        person.setAlter(13);
        person.sprachen.add(new Sprache("deutsch", 2));
        person.sprachen.add(new Sprache("englisch", 4));
        JSONObject personobj = new JSONObject(person);
        System.out.println(personobj.toString());
        // Die JSON-Struktur wird in eine Datei geschrieben
        try {
            FileWriter file = new FileWriter("jsontest.json");
            file.write(personobj.toString());
            file.close();
        } catch (IOException e) {
        }
        // JSON-Struktur aus Datei lesen und Duplikat erzeugen
        try {
            BufferedReader reader = new BufferedReader(
                  new FileReader("jsontest.json"));
            String zeile = "";
            String jsonString = "";
            while ((zeile = reader.readLine()) != null) {
                jsonString += zeile;
            }
            reader.close();
            Person duplikat = new Person(
                    new JSONObject(jsonString));
            JSONObject dupobj = new JSONObject(duplikat);
            System.out.println(dupobj.toString());
        } catch (IOException e) {
        }
    }
}
```

Listing 26.11: Ein Beispiel

Frag mal Google Maps

Wer in Deutschland eine SIM-Karte kauft, muss Namen und Adresse angeben. Das hat der Innenminister beschlossen. Da der Beruf des Innenministers bereits nach wenigen Monaten eine Allergie gegen Freiheiten hervorruft und Humorlosigkeit Bestandteil der Stellenausschreibung ist, wird er nie verstehen, warum es normale Bürger gibt, die bei Demonstrationen auch auf dem Umweg über ihre Smartphone-SIM ihre persönlichen Daten nicht hinterlassen wollen. Dafür wird er sich vielleicht wundern, wie oft in den Datenbeständen der Polizei Albert Einstein und Donald Duck auftauchen.

Ich wollte bei einer Prepaid-Karte als Prinz Eisenherz auftreten und wählte eine fiktive Adresse. Aber die Prüf-Software hat das sofort gemerkt! Vielleicht fragt sie bei Google-Maps nach. Das kann ich auch!

Google stellt eine API zur Verfügung, mit der Sie vor allem die korrekte Schreibweise einer Adresse und zusätzlich die Position auf unserem Planeten ermitteln können. Eine ausführliche Beschreibung der API finden Sie unter folgender URL:

`developers.google.com/maps/documentation/geocoding`

Google bietet Ihnen zwei Formate an: XML und JSON, empfielt aber ausdrücklich JSON. Daran werden wir uns also halten. Eins muss ich noch beichten: Der Zugriff über das Internet läuft über die Klassen `URL` und `URLEncoder`, mit denen wir uns erst später bei den Netzwerken befassen.

Das Programm ermittelt die genaue Position von Munkbrarup. Sie wollten sicher schon immer wissen, wo das liegt. Dazu holt sich das Programm zunächst die Antwort von Google und wertet anschließend den JSON-String aus. Das Hauptprogramm sieht also so aus:

```
import java.io.IOException;
import java.net.URL;
import java.net.URLEncoder;
import java.util.Scanner;
import org.json.JSONArray;
import org.json.JSONObject;

public class GeoGoogle {
    public static void main(String[] args) {
        try {
            String antwort = fragGoogleMaps("Munkbrarup");
            String adresse = werteJsonAntwortAus(antwort);
            System.out.println(adresse);
        } catch (IOException e) {
        }
    }
}
```

Listing 26.12: Das Hauptprogramm

Die Methode `fragGoogleMaps` hat wenig mit JSON zu tun, sondern ist einfach nur ein Vorgriff auf die Netzwerkprogrammierung mit der `URL`. Diese wird aus der Google-Adresse gebildet und enthält nach dem Fragezeichen die Optionen, die Google auswertet. Dabei bildet `address=Munkbrarup` die entscheidende Anfrage.

```
static String fragGoogleMaps(String adresse)
        throws IOException {
    String urlStr =
        "http://maps.google.com/maps/api/geocode/json"
        + "?sensor=false&address=";
    urlStr += URLEncoder.encode(adresse, "UTF-8");
    URL url = new URL(urlStr);
    // Hole die Antwort von Google-Maps ab
    Scanner scanner = new Scanner(url.openStream());
    String antwort = new String();
    while (scanner.hasNext()) {
        antwort += scanner.nextLine();
    }
    scanner.close();
    return antwort;
}
```

Listing 26.13: Eine Abfrage an Google stellen

Über den `Scanner`, der die `URL` als Parameter bekommt, wird die Antwort zeilenweise in einen String gelesen. Darin befindet sich die in JSON codierte Antwort von Google.

Nun kommen wir zu dem Teil, der auch zum Kapitel passt: Die Auswertung der JSON-Antwort!

```
static String werteJsonAntwortAus(String antwort) {
    String ort = "";
    JSONObject jsonObj = new JSONObject(antwort);
    if (jsonObj.getString("status").equals("OK")) {
        JSONArray ergebnisse = jsonObj.getJSONArray("results");
        for (int i = 0; i < ergebnisse.length(); i++) {
            JSONObject obj = ergebnisse.getJSONObject(i);
            String ausgabe = obj.getString("formatted_address");
            JSONObject pos = obj.getJSONObject("geometry")
                    .getJSONObject("location");
            ausgabe += " (lat: " + pos.getDouble("lat");
            ausgabe += " lng: " + pos.getDouble("lng") + ")\n";
            ort += ausgabe;
        }
    }
    return ort;
}
```

Listing 26.14: Auswertung der JSON-Antwort

Im String wird nach verschiedenen Schlüsseln gesucht. Den Namen und den Wert dieser Schlüssel habe ich mir nicht ausgedacht, sondern sie sind auf der oben erwähnten Website von Google Maps ausführlich erläutert.

- ✔ Nur wenn `status` den Wert »OK« enthält, wurde Google fündig und das Ergebnis kann ausgewertet werden.
- ✔ Das Array `results` enthält die Liste aller Treffer. Zum Glück ist Munkbrarup einzigartig auf der Welt. So enthält das Array nur ein Element.
- ✔ Jedes Element des Arrays besitzt einen Eintrag in `formatted_address`. Das ist der Name von Munkbrarup, wie Google ihn kennt. Google fügt noch Germany hinzu.
- ✔ Über `geometry` und `location` erreichen Sie die Weltkoordinate, die aus Breitengrad und Längengrad besteht, also `lat` und `lng`.

Aus diesen Informationen bastelt sich die Methode einen Antwort-String zusammen, den sie zurückgibt.

Teil VI

Praktisches



Kapitel 27

Javabohnen als Datenmodell

IN DIESEM KAPITEL

Private Attribute mit Gettern und Settern

Aktionen bei Änderungen

Klassen mit Veto-Rechten

Eine Klasse kann in verschiedenen Rollen auftreten. Besonders häufig wird sie verwendet, um einen eigenen Datentyp zu modellieren. Dabei stehen die Attribute einer Klasse im Vordergrund, die transportiert, verarbeitet, gespeichert und wieder geladen werden sollen. Solche Datenobjekte werden oft als *Plain Old Java Object (POJO)* oder als *Bean* bezeichnet. Es gibt zwar kein spezielles Schlüsselwort für eine Bean, aber klare Vorgaben und ein ganzes Package voller Unterstützung. Eine Bean ist eine Klasse mit folgenden Eigenschaften:

- Eine Bean hat einen öffentlichen parameterlosen Konstruktor. Dieser wird bekanntlich von Java erstellt, wenn Sie keinen eigenen beitragen wollen. Sollten Sie aber einen Konstruktor mit Parametern anbieten, muss eine Bean zusätzlich auch einen Standardkonstruktor ohne Parameter besitzen.
- Die Bean muss serialisierbar sein. Das erreichen Sie durch die Implementierung des Interface `Serializable`.
- Alle Attribute sind privat, für den Zugriff existieren öffentliche Getter und Setter.

In Listing 27.1 sehen Sie das Beispiel einer Person mit einem Namen und dem Attribut, ob er noch Single ist. Auch Getter und Setter gibt es.

```
public class Person {
    private String name = "";
    private boolean single = true;

    public String getName() {
        return this.name;
    }
    public void setName(String str) {
        this.name = str;
    }
    // ...
}
```

Listing 27.1: Beispiel für ein Attribut mit Getter- und Setter-Methode

Serialisierbarkeit

Eine Bean muss das Interface `Serializable` implementieren. Damit das klappt, müssen Sie vor Definition der Klasse den Import von *java.io.Serializable* festlegen. Weitere Konsequenzen hat die Implementierung auf den ersten Blick nicht.

```
import java.io.Serializable;

public class Person implements Serializable {
```

Die Serialisierbarkeit ist nicht mehr zu halten, wenn die Klasse Objekte anderer Klassen enthält, die nicht serialisierbar sind.

Die Serialisierbarkeit bedeutet, dass die Elemente einer Klasse in irgendeiner Form so hintereinander aufgeführt werden können, dass man sie später aus dieser Reihenfolge wieder zu dem gleichen Objekt zusammensetzen kann. Diese Fähigkeit wird benötigt, wenn das Objekt in einer Datei gespeichert oder an einen Empfänger gesendet werden soll.

Mit der Implementierung von `Serializable` wird also überwacht, ob es nicht doch Klassenbestandteile gibt, die nicht serialierbar sind. Quertreiber können Klassen wie `Thread` oder `Socket` sein.

Sollen Attribute nicht serialisiert werden oder enthält die Klasse Attribute, die nicht serialisierbar sind, so können Sie diese ausschließen, indem Sie den Modifizierer `transient` voranstellen.

Attributänderungen und Meldung

Damit nicht jeder Hans und Franz in den Attributen einer Klasse herumpfuscht,

halten Sie sich einfach an die Regel, dass Attribute immer privat sind und mit sogenannten Getter- und Setter-Methoden verändert oder gelesen werden können. Auf diese Weise können Sie bei einem Zugriff auf ein Attribut aktiv werden, die Änderung verhindern oder bei einer Änderung Aktionen auslösen.

Die Java-Beans-Bibliothek stellt Mechanismen zur Verfügung, die es anderen Objekten ermöglichen, Änderungen an den Attributen zu überwachen.

PropertyChangeListener

Eine Klasse kann einen Rückruf einrichten, damit Änderungen an den Attributen von interessierter Seite verfolgt werden.

Die meisten Mütter legen beispielsweise einen gewissen Wert darauf, zu erfahren, wenn ihre Kinder heiraten. Sie werden zu Recht einwenden, dass man schon verlangen kann, dass die Kinder in diesem Fall ihre Mutter anrufen. Neben der Mutter gibt es aber weitere Interessierte, wie beispielsweise das Finanzamt. Bevor man jemanden vergisst, sollte man lieber einen `PropertyChangeListener` in Anspruch nehmen. Leider gibt es die im wahren Leben nicht. Java hat da Vorteile. Die Vorgehensweise:

1. Wir haben das Attribut `single` als `private` gut versteckt. Natürlich gibt es eine Getter- und eine Setter-Methode.
2. Irgendein Objekt möchte informiert werden, wenn sich an diesem Status etwas ändert, beispielsweise die Klasse für die Heiratskandidaten.
3. Dazu wird in der Klasse `Person` eine Liste der Interessierten angelegt. Natürlich ganz privat. Diese Liste ist vom Typ `PropertyChangeSupport`.
4. Hier kann jede Klasse eingetragen werden. Zu diesem Zweck bietet die Liste die Methode `addPropertyChangeListener` an. Als Parameter erwartet sie einen String, der das Attribut beschreiben sollte, und den Interessenten, der informiert werden soll.
5. Wird nun tatsächlich das Attribut geändert, wird dies ja nur über die Methode `setSingle` passieren können. Dort kann nun die Methode `firePropertyChange` der Interessentenliste aufgerufen werden. Parameter ist die Attributkennung, der bisherige und der zukünftige Zustand des Attributs.

Diese Vorgänge sehen Sie in der Klasse `Person` realisiert.

```
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public class Person {
    private String name;
    private boolean single = true;

    private PropertyChangeSupport singleinfo
            = new PropertyChangeSupport(this);

    public static final String SINGLE = "single";

    public void addSingleListener(
            PropertyChangeListener listener) {

        singleinfo.addPropertyChangeListener(SINGLE, listener);
    }

    public void setSingle(boolean neuerStatus) {
        singleinfo.firePropertyChange(SINGLE,
                this.single, neuerStatus);
        this.single = neuerStatus;
    }

    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
}
```

Listing 27.2: Die Klasse `Person` erlaubt die überwachung.

Um eine Überwachung zu erreichen, müssen folgende Schritte durchgeführt werden:

1. Als interessierte Klasse kann sich nur anmelden, wer das Interface `PropertyChangeListener` implementiert. Ein Objekt einer solchen Klasse kann bei der Person in der Liste der Interessierten registriert werden.
2. Wer eine `PropertyChangeListener`-Klasse sein will, muss die Methode `propertyChange` implementieren. Diese erhält ein `PropertyChangeEvent` per Parameter. Diese wiederum liefert folgende Informationen über das Wechselereignis:

- `getSource` liefert das auslösende überwachte Objekt.
- `getPropertyName` liefert die Bezeichnung des Attributs.
- `getOldValue` liefert den bisherigen Wert.
- `getNewValue` liefert den zukünftigen Wert.

Die Methode `HeiratsKandidaten` erlaubt es, Personen zu registrieren. Dabei meldet sie sich selbst als Beobachter an. Im Falle eines Wechsels gibt die Methode `propertyChange` Informationen über den Wechsel auf dem Bildschirm aus.

```
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;

public class HeiratsKandidaten implements PropertyChangeListener {

    public void add(Person person) {
        person.addSingleListener(this);
    }

    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        Person person = (Person) evt.getSource();
        System.out.println(person.getName()
                + " Attribut: "
                + evt.getPropertyName() + " bisher: "
                + evt.getOldValue() + " nun: "
                + evt.getNewValue());
    }
}
```

Listing 27.3: Die Klasse `HeiratsKandidaten` überwacht die Klasse `Person`.

Im letzten Schritt wird das Hauptprogramm eine Heiratskandidatenliste und zwei Personen erstellen, die sie der Liste hinzufügt, und demonstrieren, dass sich bei Änderung des Single-Status die Kandidatenliste meldet.

```
public class Main {

    public static void main(String[] args) {
        HeiratsKandidaten liste = new HeiratsKandidaten();
        Person anna = new Person();
        anna.setName("Anna");
        liste.add(anna);
        Person peter = new Person();
        peter.setName("Peter");
        liste.add(peter);
        anna.setSingle(true);
        anna.setSingle(false);
        peter.setSingle(false);
        anna.setSingle(false);
        anna.setSingle(true);
    }
}
```

Listing 27.4: Das Testprogramm

Die Ausgabe des Programms zeigt die Zeilen, die auf Änderungen des Zustands reagiert haben.

```
Anna Attribut: single bisher: true nun: false
Peter Attribut: single bisher: true nun: false
Anna Attribut: single bisher: false nun: true
```

Einspruch!

Nun können sich zwar alle zum Mitlauschen anmelden. Verhindern können sie aber nichts, auch wenn sie gern gewollt hätten. Der Braut wird in den zivilisierten Ländern nicht nur ein Informationsrecht eingeräumt, sie kann sogar ein Veto einlegen. Falls der Bräutigam darüber hinaus auch der Mutter ein Veto einräumt, sollte sich die Braut vielleicht überlegen, ob sie nicht von ihrem Veto Gebrauch machen will.

Da sich in diesem Fall bereits ein Familiendrama abzeichnet, wollen wir es beim Single-Dasein bei der Informationsliste belassen. Stattdessen soll es möglich sein, ein Veto einzulegen, wenn der Name der Person geändert wird.

Der Einspruch, also das Veto, funktioniert ganz ähnlich wie der `PropertyChangeListener`. Der Veto-Interessierte meldet sich an und in der Setter-Methode werden alle Veto-Mächte informiert, welche Änderung geplant ist. Nun aber kann der Einspruchsberechtigte eine Unterbrechung beantragen, indem er eine `PropertyVetoException` auslöst. Damit wird der Ablauf unterbrochen und die

Zuweisung verhindert.

Für die Veto-Mächte wird natürlich eine andere Liste benötigt als die Interessentenliste.

```
public class Person {
    // ...
    private String name;
    public static final String NAME = "name";
    private VetoableChangeSupport vetos
            = new VetoableChangeSupport(this);

    public void meldeVetoRechtAn(
            VetoableChangeListener listener) {
        vetos.addVetoableChangeListener(listener);
    }

    public void setName(String name) {
        try {
            vetos.fireVetoableChange(NAME, this.name, name);
            this.name = name;
        } catch (PropertyVetoException e) {
        }
    }
}
```

Listing 27.5: Ergänzung der Klasse `Person` um Veto-Rechte

Nun schreiben wir die Klasse für die Verwandtschaft, die ja an jedem Namen etwas zu nörgeln hat. Die Klasse `Verwandtschaft` meldet sich als Veto-Macht an und wird dann durch die Methode `vetoableChange` geweckt, wenn der Name geändert werden soll. Sollte das Programm das Veto einlegen wollen, wirft es die Exception `PropertyVetoException` und verhindert damit die Übernahme des Namens.

```
import java.beans.PropertyChangeEvent;
import java.beans.PropertyVetoException;
import java.beans.VetoableChangeListener;

public class Verwandtschaft implements VetoableChangeListener {

    @Override
    public void vetoableChange(PropertyChangeEvent evt)
            throws PropertyVetoException {

        if (Person.NAME.equals(evt.getPropertyName())) {
            String neuName = (String) evt.getNewValue();
            if (neuName.equals("Arnold")) {
                throw new PropertyVetoException(
                        "Alles, nur nicht Arnold!", evt);
            }
        }
    }
}
```

Listing 27.6: Es wird ein Veto eingelegt.

Nun gilt es, die Veto-Möglichkeit zu testen:

```
Verwandtschaft onkel = new Verwandtschaft();
Verwandtschaft tante = new Verwandtschaft();
peter.meldeVetoRechtAn(onkel);
peter.meldeVetoRechtAn(tante);
peter.setName("Arnold");
System.out.println("Peter heißt nun "+peter.getName());
peter.setName("Karl");
System.out.println("Peter heißt nun "+peter.getName());
```

Listing 27.7: Klappt der Widerspruch?

Als Ergebnis kommt heraus:

```
Peter heißt nun Peter
Peter heißt nun Karl
```



Da sich alle möglichen Objekte in der Liste der `ChangeListener` eintragen können, sind die Querschläge durch das Ändern eines Attributs nur schwer verfolgbar. Das kann sich bei Wartungsarbeiten als Horror erweisen. Verwenden Sie dieses Verfahren also zurückhaltend.

Kapitel 28

Parallelverarbeitung mit Threads

IN DIESEM KAPITEL

- Der Schönheitsschlaf für wache Computer
- Leichtgewichtsprozesse und parallele Abläufe
- Kritische Bereiche gegen Scherben
- Warten auf den Produzenten
- Ewiges Warten unterbrechen

Java ermöglicht es Ihnen, Programme so zu schreiben, dass die linke Hand nicht weiß, was die rechte tut. Und weil das in der Wirklichkeit schon für Verwirrung sorgt, muss das als letzter Schrei auch in Ihr Programm.

Bei genauerer Beobachtung stellen Sie fest, dass auf Ihrem Computer bereits mehrere Prozesse parallel laufen. Im Vordergrund bedienen Sie die Textverarbeitung und im Hintergrund sehen Sie, wie die Tabellenkalkulation abstürzt. Um dieses Erlebnis noch zu steigern, hat die Industrie Prozessoren mit mehreren Kernen gebaut. Spötter behaupten, die Zweikerntechnologie wäre nur geschaffen worden, damit neben einem Virenscanner noch eine nützliche Anwendung halbwegs performant laufen kann. Aber wie beschäftigen Sie in einem System mit vier oder mehr Kernen die gelangweilte Hardware? Programmieren Sie doch ein paar Threads!

Threads ermöglichen die Parallelverarbeitung innerhalb eines Prozesses. Ein Prozess ist ein laufendes Programm, das durch das Betriebssystem einen gegenüber anderen Prozessen geschützten Speicherbereich erhält. In den modernen Multitaskingsystemen können mehrere Prozesse parallel nebeneinander laufen wie besagte Tabellenkalkulation und die Textverarbeitung.

Ein Thread wird dagegen unter der Kontrolle des Programms erzeugt. Auch Threads können parallel zueinander ablaufen, teilen sich aber den Speicherbereich und alle anderen Ressourcen. Zugriffskollisionen muss das Programm regeln. Sie ahnen schon, dass das wohl Ihre Aufgabe sein dürfte.



Prozesse werden vom Betriebssystem voreinander geschützt. Threads müssen sich selbst untereinander einigen.

Erst einmal Schlaf nachholen

Das war jetzt aber reichlich verwirrend. Grund genug, sich erst einmal auszuruhen. Und falls Sie dazu gerade keine Zeit haben, lassen Sie das doch Ihren Computer erledigen. Das Programm in Listing 28.1 wird sich nach dem Start für zehn Sekunden schlafen legen und dann enden.

```
class LassMichSchlafen {
    public static void main(String[] args) {
        System.out.println("Ich schlafe 10 Sekunden.");
        try {
            Thread.sleep(10*1000);
        } catch(InterruptedException e) {
        }
        System.out.println("Ich bin wieder wach.");
    }
}
```

Listing 28.1: Lass mich bitte zehn Sekunden schlafen!

Falls Sie mir nicht glauben, prüfen Sie es doch mit einer Stoppuhr. Der Parameter der Methode `Thread.sleep` enthält die Schlafenszeit in Millisekunden. Wer `Thread.sleep` aufruft, muss sich um die `InterruptedException` kümmern. Allerdings gibt es in diesem einfachen Beispiel dafür keinen zwingenden Handlungsbedarf.

Die Methode `Thread.sleep` werde ich immer wieder nutzen, wenn ich die Parallelität von Threads demonstrieren will. Aber auch für andere Gelegenheiten wie beispielsweise Time-outs ist diese Methode praktisch.

Tatsächlich kann eine Schlafpause zur Entspannung und damit zur Wellness des Prozessors führen. Steht ein Vorgang in Ihrem Programm allzusehr unter Last, kann ein gut positionierter Schönheitsschlaf von ein paar Millisekunden Wunder wirken. Ich werde darauf noch zurückkommen.

Leichtgewichtige Threads

Ein Thread wird durch die gleichnamige Klasse Thread repräsentiert. Die Referenz auf Thread ermöglicht es, den Thread zu ... Das ist Käse! Da Thread und Thread leicht zu verwechseln sind, verwende ich den etwas sperrigen Begriff *Leichtgewichtsprozess*. In der Informatik ist dieser Begriff nicht nur bei figurbetonten Programmierern beliebt. Viel verständlicher ist es, dass ein Leichtgewichtsprozess durch die Klasse `Thread` repräsentiert wird. Über die Referenz auf das Thread-Objekt kann der Leichtgewichtsprozess gesteuert werden.

Erweiterung von Thread

Auch ein Leichtgewichtsprozess benötigt einen Quelltext. Dieser befindet sich bei einem Objekt von `Thread` in der Methode `run`. Sie erstellen Ihren eigenen Leichtgewichtsprozess, indem Sie eine eigene Klasse von `Thread` ableiten und die Methode `run` überschreiben. Gestartet wird der Leichtgewichtsprozess von außen durch Aufruf der Methode `start`.

```
public class ThreadErweiterung extends Thread {

    @Override
    public void run() {
        for (int i=0; i<5; i++) {
            try {
                Thread.sleep(250);
                System.out.println("    "+i);
            } catch(InterruptedException e) {
            }
        }
    }

    static public void main(String[] args) {
        Thread thread = new ThreadErweiterung();
        thread.start();
        for (int i=7; i>0; i--) {
            try {
                Thread.sleep(200);
                System.out.println(i);
            } catch(InterruptedException e) {
            }
        }
    }
}
```

Listing 28.2: Erweiterung der Klasse `Thread`

Der Leichtgewichtsprozess in Listing 28.2 zählt im Hintergrund von 0 bis 4 und pausiert dazwischen jeweils für eine Viertelsekunde. Das Hauptprogramm zählt währenddessen von 7 bis 1 herunter, lässt sich aber nur jeweils 200 Millisekunden Zeit. Da der Leichtgewichtsprozess bei der Ausgabe ein paar Leerzeichen einrückt, können Sie die Verläufe wunderbar beobachten.


Einen Thread gibt es immer: das Hauptprogramm, das mit `main` startet.

Implementierung von Runnable

Es gibt eine zweite Variante, einen Leichtgewichtsprozess zu erzeugen. Hier wird nicht die Klasse `Thread` erweitert, sondern das Interface `Runnable` implementiert. Diese Implementierung muss eine Methode `run` enthalten, die der Klasse `Thread` verdächtig ähnlich sieht. Sie starten ein Objekt einer Klasse, die `Runnable` implementiert, indem Sie es als Parameter dem Konstruktor von `Thread` übergeben. Ansonsten bleibt alles gleich. Sie starten den Leichtgewichtsprozess, indem Sie die Methode `start` von `Thread` aufrufen. Schauen Sie selbst:

```
public class RunnableImpl implements Runnable{

    @Override
    public void run() {
        for (int i=0; i<5; i++) {
            try {
                Thread.sleep(250);
                System.out.println("   "+i);
            } catch(InterruptedException e) {
            }
        }
    }

    static public void main(String[] args) {
        Thread thread = new Thread(new RunnableImpl());
        thread.start();
        for (int i=7; i>0; i--) {
            try {
                Thread.sleep(200);
                System.out.println(i);
            } catch(InterruptedException e) {
            }
        }
    }
}
```

Listing 28.3: Implementierung von `Runnable`

Glockenläuten

Nun wollen wir mit mehreren Leichtgewichtprozessen arbeiten, die unabhängig voneinander arbeiten. Als Beispiel betrachten wir einen Kirchturm mit drei Glocken. Jede der Glocken läutet, wie sie will. Was die anderen Glocken treiben, interessiert sie nicht.

Damit Sie die Glocken unterscheiden können, erhält jede Glocke als Konstruktorparameter einen String, mit dem sie sich später auf dem Bildschirm melden soll. Ein weiterer Parameter bestimmt, wie oft sie läuten soll, und schließlich gibt ein Parameter die Zeit an, die zwischen den Schlägen vergehen soll.

Die Methode `run` durchläuft eine Schleife und schlägt so oft, wie es in der Variablen `anzahl` festgelegt wurde. In der Zeit zwischen den Schlägen schläft der Leichtgewichtsprozess und gibt dann die vorbereitete Meldung aus.

```
public class Glocken implements Runnable {

    private String meldung = "";
    private int anzahl = 0;
    private long zeit = 0;

    public Glocken(String meldung, int anzahl, long zeit) {
        this.meldung = meldung;
        this.anzahl = anzahl;
        this.zeit = zeit;
    }

    @Override
    public void run() {
        for (int i=0; i<anzahl; i++) {
            try {
                Thread.sleep(zeit);
                System.out.println(meldung);
            } catch(InterruptedException e) {
            }
        }
    }

    public static void main(String[] args) {
        new Thread(new Glocken("bing", 10, 600)).start();
        new Thread(new Glocken("  bang", 15, 400)).start();
        new Thread(new Glocken("    bong", 8, 800)).start();
    }
}
```

Listing 28.4: Parallele Glocken

Die Methode `main` stecken wir der Einfachheit halber in die Klasse `Glocken` hinein. Sie erzeugt drei Glocken-Threads, die unterschiedlich läuten, und startet sie sofort. Durch die Einrückung der Meldungen können Sie die Glocken optisch leicht

unterscheiden. Das Hauptprogramm endet sofort, aber das ist ja nicht schlimm. Das Gesamtprogramm endet erst, wenn der letzte Thread keine Lust mehr hat.

Warten auf die Glocken

Der Gottesdienst sollte erst beginnen, wenn die Glocken aufgehört haben zu läuten. Ansonsten könnte es sein, dass der Pfarrer sagt: »Ich wünsche Ihnen einen guten BANG.« Jeder der gestarteten Leichtgewichtsprozesse wird so lange laufen, bis die Methode `run` ein Ende findet. Während Leichtgewichtsprozesse im Hintergrund rackern, kann das Hauptprogramm tun und lassen, was es will.

Häufig will das Hauptprogramm fortfahren, wenn es sicher ist, dass die Aufgaben erledigt sind, die es an einen Leichtgewichtsprozess ausgelagert hat. Damit es weiß, dass der Leichtgewichtsprozess sicher fertig ist, kann es dessen Methode `join` aufrufen. Dann wird der Aufrufer bis zum Ende des Threads schlafen.

Wir verändern nun das Programm `Glocken` dahingehend, dass wir die Threads in einer `ArrayList` speichern. So können wir auf alle Threads warten, bis sie fertig sind.

```
import java.util.ArrayList;
import java.util.List;

public class GlockenJoin implements Runnable {

    private String meldung = "";
    private int anzahl = 0;
    private long zeit = 0;
    static List<Thread> gelaeut = new ArrayList<Thread>();

    public GlockenJoin(String meldung,
            int anzahl, long zeit) {

        this.meldung = meldung;
        this.anzahl = anzahl;
        this.zeit = zeit;
    }

    @Override
    public void run() {
        for (int i = 0; i < anzahl; i++) {
            try {
                Thread.sleep(zeit);
                System.out.println(meldung);
            } catch (InterruptedException e) {
            }
        }
    }

    public static void main(String[] args) {
        gelaeut.add(new Thread(new GlockenJoin(
                "bing", 10, 600)));
        gelaeut.add(new Thread(new GlockenJoin(
                "  bang", 15, 400)));
        gelaeut.add(new Thread(new GlockenJoin(
                "    bong", 8, 800)));
        for (Thread thread : gelaeut) {
            thread.start();
        }
        for (Thread thread : gelaeut) {
            try {
                thread.join();
            } catch (InterruptedException e) {
            }
        }
        System.out.println("Geläute zu Ende");
    }
```

```
}
```

Listing 28.5: Die Glocken kommen in die Liste.

Kollisionen in der Vasenfabrik

Glocken haben die nette Eigenheit, dass sie schwingen und klingen, ohne sich gegenseitig wehzutun. Leider ist das aber nicht immer im Leben so. Schon der Versuch, zwei wohlerzogene Menschen durch die Tür eines Restaurants zu bekommen, kann eine abendfüllende Angelegenheit sein. Immerhin erkennen die Personen die Engstelle und werden hoffentlich eine zufriedenstellende Lösung finden, bevor sie verhungern.

Computer sind da etwas unflexibler. Wir stellen uns eine Fabrik für Vasen vor. Vasen müssen nach dem Formen gebrannt werden, damit sie nicht von den ersten Blumen, die man hineinsteckt, deformiert werden. Dieser Prozess dauert eine Weile. Darum hat der Fabrikbesitzer beschlossen, mehrere Öfen zu kaufen, um den Brennvorgang zu parallelisieren. Ein Ofen wird in der Klasse in Listing 28.6 simuliert und unterscheidet sich nur geringfügig von den Glocken aus dem vorigen Abschnitt.

```
public class VasenOfen implements Runnable {

    private int anzahl;
    private Verpacker verpacker;

    public VasenOfen(Verpacker verpackung, int anzahl) {
        this.verpacker = verpackung;
        this.anzahl = anzahl;
    }

    @Override
    public void run() {
        for (int i = 0; i < anzahl; i++) {
            // Von Zeit zu Zeit wird eine Vase fertig
            try {
                Thread.sleep((long) (100 * Math.random()));
            } catch (InterruptedException e) {
            }
            // Verpacken ist ein kritischer Bereich!
            verpacker.verpacke();
        }
    }
}
```

Listing 28.6: Der Ofen für die Vasen

Nach dem Brennen liefert der Vasenofen seine Vase an den Verpacker. Dieser wird in Listing 28.7 simuliert. Da Simulationen realitätsnah sein sollten, stellt die Simulation auch gleich fest, ob zwei Vasen gleichzeitig verpackt werden sollen. Nach dem Zusammenstoß passen zwar mehr Vasen in einen Karton, langfristig benötigen wir aber eine Lösung, die das verhindert, weil wir davon ausgehen müssen, dass der Kunde nicht glücklich wäre.

```
public class Verpacker {

    private boolean bereitsAktiv = false;

    public void verpacke() {
        if (bereitsAktiv) {
            System.out.println("Klirr");
        } else {
            bereitsAktiv = true;
            // karton.verbraucheKarton();
            System.out.print("Verpacke ...");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
            System.out.println("... Verpackt");
        }
        bereitsAktiv = false; // Fertig!
    }
}
```

Listing 28.7: Der Verpacker verpackt Vasen.

Spielen wir die Fabrik einmal durch. Das Hauptprogramm in Listing 28.8 ist nicht übertrieben groß oder originell. Es wird ein Verpacker erzeugt. Danach entstehen drei Öfen, die fleißig Vasen produzieren.

```
public class VasenMain {

    public static void main(String[] args) {
        Verpacker verpacker = new Verpacker();
        for (int i=0; i<3; i++) {
            (new Thread(new VasenOfen(
                    verpacker, 10))).start();
        }
    }
}
```

Listing 28.8: Starten der Öfen und des Verpackers

Wenn Sie das Programm starten, sehen Sie, wie die Vasen verpackt werden, und auch, dass es offenbar reichlich Bruch gibt. Das würde nicht passieren, wenn die Vasenöfen ein wenig aufeinander Rücksicht nehmen würden.

Auch wenn Sie nicht an Zauberei glauben, gibt es ein magisches Wort, das

ausreicht, um der sinnlosen Zerstörung von Steingut Einhalt zu gebieten. Sprechen Sie mir langsam nach: `synchronized`. Das Wort setzen Sie als Modifizierer vor die Methode `verpacke`.

```
public synchronized void verpacke(VasenOfen vase) {
```

Ein neuer Start und schon gibt es kein Geklirre mehr. Was ist passiert? Das Zauberwort sorgt dafür, dass immer nur ein Leichtgewichtsprozess auf einmal die Methode ausführen kann. Und schon werden alle Vasen nacheinander verpackt. Einen Bereich, den immer nur ein Leichtgewichtsprozess gleichzeitig betreten darf, nennt man in der Informatik einen *kritischen Bereich*.

Warten auf Kartons

Nachdem nun eine Vase nach der anderen verpackt wird, stellt sich heraus, dass auch der Verpacker ein Problem hat. Immer wieder gehen ihm die Kartons aus. Ohne Karton gestaltet sich das Verpacken der Vasen etwas schwierig. Also wartet er von Zeit zu Zeit auf Nachschub. In dieser Zeit kann keine Vase verpackt werden.

Diese Situation bilden wir in der Klasse `Karton` in Listing 28.9 nach. Sie enthält eine Methode für das Auffüllen der Kartons und eine für das Entnehmen eines Kartons. Bei der Entnahme eines Kartons wird der ausführende Leichtgewichtsprozess durch die Methode `wait` gestoppt, wenn kein Karton mehr da ist.

Wird der Kartonstapel aufgefüllt, wird die Methode `notify` aufgerufen. Damit wird der wartende Leichtgewichtsprozess wieder freigegeben. Er holt sich den Karton und die Verpackung kann weitergehen.

```
public class Karton {
    private int kartons = 0;

    public void neuerKarton(int anzahl) {
        synchronized(this){
            kartons += anzahl;
            notify();
        }
    }

    public void verbraucheKarton() {
        synchronized(this) {
            while (kartons<=0) {
                try {
                    wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            kartons--;
        }
    }
}
```

Listing 28.9: Karton

Sowohl `wait` als auch `notify` kann nur prinzipiell über jedes Objekt laufen. Allerdings muss das Objekt zuvor synchronisiert werden. Und da ist es wieder, das magische Wort `synchronized`. Im Falle des Kartons wird als Synchronisationsobjekt einfach das aktuelle Objekt über `this` verwendet.

In unserem Programm haben wir nur einen Verpacker. Insofern funktioniert `notify` wunderbar. Wenn mehrere Leichtgewichtsprozesse aktiv sind, können Sie auch `notifyAll` verwenden.

Nun müssen Sie natürlich den Verpacker noch so umstellen, dass er bei der Verpackung immer einen Karton anfragt. Der Kartonlieferant wird dem Verpacker im Konstruktor übergeben und gespeichert. Beim Verpacken wird dessen Methode `verbraucheKarton` aufgerufen.

```
public class Verpacker {

    private boolean bereitsAktiv = false;
    private Karton karton;

    public Verpacker(Karton karton) {
        this.karton = karton;
    }

    public synchronized void verpacke(VasenOfen vase) {
        if (bereitsAktiv) {
            System.out.println("Klirr");
        } else {
            bereitsAktiv = true;
            karton.verbraucheKarton();
            System.out.print("Verpacke ...");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
            System.out.println("... Verpackt");
        }
        bereitsAktiv = false; // Fertig!
    }
}
```

Aber wer sorgt dafür, dass es genügend Kartons gibt? Das überlassen wir einfach dem Chef. Der tut eh nie etwas Vernünftiges.

```
public class VasenMain {

    private static void liefereKartons(Karton karton) {
        for (int i=0; i<10; i++) {
            System.out.println("Liefere Kartons");
            karton.neuerKarton(4);
            try {
                Thread.sleep(800);
            } catch (InterruptedException e) {
            }
        }
    }
    public static void main(String[] args) {
        Karton karton = new Karton();
        Verpacker verpacker = new Verpacker(karton);
        for (int i=0; i<3; i++) {
            (new Thread(new VasenOfen(verpacker, 10))).start();
        }
        liefereKartons(karton);
    }
}
```

Listing 28.10: `VasenMain` mit Kartonlieferung

Wenn Sie das Programm nun starten, können Sie an der Ausgabe sehen, wie die Produktion immer wieder stockt, wenn die Kartons ausgehen.

Wir warten nicht auf Godot: Time-out

Insbesondere in der Netzwerkprogrammierung kann es sein, dass eine Verbindung lange auf sich warten lässt und vielleicht nie zustande kommt. Ein Computer mag an dieser Stelle geduldig sein, der Anwender ist es vermutlich eher nicht.

Solch langweilige Aufgaben wie das Warten auf Ereignisse überträgt man gern auf Leichtgewichtsprozesse. Die können sich nämlich nicht wehren. Darüber hinaus bleibt das Hauptprogramm noch aktiv für andere Zwecke.

Im Beispiel simulieren wir eine hängende Netzwerkverbindung durch den Aufruf von `wait`. Dazu benötigen wir irgendein Objekt, das synchronisierbar ist. Glücklicherweise sind das alle Objekte, sodass wir hier einfach `Object` verwenden.

```
public class TimeOut implements Runnable {

    static private Object object = new Object();

    private void warte() {
        try {
            // Wir können nur auf ein Objekt zugreifen,
            // wenn wir es zuvor synchronisieren
            synchronized(object) {
                object.wait();
            }
            System.out.println("Ich ende regulär");
        } catch (InterruptedException e) {
            System.out.println("Ich wurde unterbrochen");
        }
    }

    @Override
    public void run() {
        warte();
    }

    public static void main(String[] args) {
        Thread thread = new Thread(new TimeOut());
        thread.start();
        try {
            Thread.sleep(2000);
            // Unterbreche den Hintergrund-Thread:
            thread.interrupt();
            Thread.sleep(2000);
            // Löse das Ereignis aus, auf das der Thread wartet
            synchronized(object) {
                object.notify();
            }
        } catch (InterruptedException e) {
        }
    }
}
```

Listing 28.11: Ein Thread wartet oder wird dabei unterbrochen.

Während der Leichtgewichtsprozess nasebohrend wartet, übernimmt das Hauptprogramm den Abbruch durch einen Time-out. Er schläft dazu zwei Sekunden und ruft dann die Methode `interrupt` des Threads auf. Das löst im Thread eine `InterruptedException` aus, die letztlich zum Ende des

Leichtgewichtsprozess führt.

In Listing 28.11 können Sie den Aufruf von `interrupt` auskommentieren. Dann wird das Hauptprogramm weiterlaufen und ruft `notify` über das Objekt auf, auf das der Leichtgewichtsprozess mit `wait` wartet.

Prozessoren rösten

Sie kennen vielleicht diese Hilfsprogramme, mit denen man einen Computer unter Last setzt, um zu prüfen, ob die Lüftung einwandfrei läuft. So ein Programm schreiben wir jetzt selbst. Mein Verlag bittet mich aber um den folgenden Hinweis:



Wenn das Programm auf einem Rechner längere Zeit läuft, dessen Kühlsystem nicht in Ordnung ist, führt es zu einem plötzlichen Abschalten des Rechners ohne Vorwarnung. Sollten darüber hinaus noch die Temperatursensoren defekt sein, die für diese Abschaltung verantwortlich sind, grillen Sie mit diesem Programm Ihre CPU zu Tode. Wenn Ihr System aber tatsächlich derartig in den Seilen hängt, ist dessen Hinscheiden kein großer Verlust. Dennoch werde weder ich noch der Verlag dafür haften. Es geht alles auf Ihr eigenes Risiko!

Hört sich spaßig an? Okay, dann los! Es ist ja Ihr Computer und nicht meiner! Um einen Computer richtig heiß zu machen, muss man ihn beschäftigen. Sie könnten ihn einen Film konvertieren lassen oder ein paar Grafiken rendern oder die folgende Zeile ausführen lassen:

```
for (;;);
```

Was das ist? Eine `for`-Schleife, die nichts tut und das ganz schnell und immer wieder. Der Informatiker nennt das vornehm »busy waiting«, also »beschäftigtes Warten«. Eine solche Endlosschleife lässt die CPU ständig im Kreis rennen. Sie bekommt nie eine Pause und wird heiß.

Damit das Ganze nicht so karg ist, habe ich mir ein Button-Programm aus Kapitel 16 ausgeliehen und darin die Zeile versteckt. Eintippen können Sie es schon einmal. Aber vor dem Start lesen Sie erst einmal weiter.

```
import java.awt.event.*;
import javax.swing.*;

public class SwingGrillBlock extends JFrame
                        implements ActionListener {
    private JButton click;
    public void brateCPU() {
        for (;;);
    }
    @Override
    public void actionPerformed(ActionEvent ev) {
        if (click.getText().equals("Stopp")) {
            System.exit(0);
        } else {
            brateCPU();
            click.setText("Stopp");
        }
    }
    public SwingGrillBlock() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        click = new JButton("Grill mich!");
        click.addActionListener(this);
        add(click);
        pack();
        setVisible(true);
    }
    public static void main(String[] args) {
        new SwingGrillBlock();
    }
}
```

Listing 28.12: CPU-Heizung

Wenn Sie nachher das Programm starten, wird ein Fenster mit einem Button Grill Mich! erscheinen. Klicken Sie darauf wie gewünscht, wird die Methode `brateCPU` aufgerufen, die unseren CPU-Bräter startet. Das Listing besagt, dass Ihr Button nun eine neue Beschriftung Stopp bekommen soll und bei einem erneuten Klicken das Programm beenden soll.

Tatsächlich passiert aber gar nichts mehr. Der eingedrückte Button kommt nicht mehr nach oben. Die Beschriftung wechselt nicht. Das Programm wird auch nicht beendet. Das passiert auch nicht, wenn Sie das Schließkreuz des Fensters anklicken. Allerdings brät nach wie vor Ihre CPU und macht dem Lüfter richtig Beine.



Spätestens jetzt sollten Sie eine Idee haben, wie Sie das Programm anderweitig stoppen können. Sie können zum Beispiel den Task-Manager verwenden oder das Programm `top` oder was auch immer Ihr System so hergibt, um Amok laufende Prozesse zu stoppen. Falls nichts mehr hilft, können Sie einfach das System herunterfahren, aber möglichst, bevor die CPU Rauchzeichen gibt.

Wie kommt es zu diesem Versagen der Kontrollelemente? Da Ihre Endlosschleife direkt aufgerufen wurde, ist Swing blockiert. Das Fenster nimmt keine Aktionen mehr entgegen. Spätestens wenn Ihr Rechner raucht, dürften Sie gelernt haben, dass man längerwährende Vorgänge immer in einen Leichtgewichtsprozess auslagert.

Bedienbar durch Threads

Das hat Spaß gemacht. Allerdings wollen wir nun eine Version haben, die man noch bedienen kann, auch wenn die CPU gerade gebraten wird. Die Lösung dazu ist ein Leichtgewichtsprozess. Fegen Sie die Asche von Ihrem Schreibtisch, kaufen Sie sich einen neuen Computer und tippen Sie Listing 28.13 ein. Dieses teilt das Programm in Leichtgewichtsprozesse auf, die parallel zueinander laufen. So können wir mit dem einen Thread die CPU braten, während der andere Thread die Benutzersteuerung übernimmt und nicht blockiert ist.

```
import javax.swing.*;
import java.awt.event.*;

public class SwingGrillPro extends JFrame
                  implements ActionListener, Runnable {
    JButton click;

    @Override
    public void run() {
        System.out.println("Thread startet");
        for (;;);
    }

    @Override
    public void actionPerformed(ActionEvent ev) {
        if (click.getText().equals("Stopp")) {
            System.exit(0);
        } else {
            Thread thread = new Thread(this, "braten");
            thread.start();
            click.setText("Stopp");
        }
    }

    public SwingGrillPro() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        click = new JButton("Grill mich!");
        click.addActionListener(this);
        add(click);
        pack();
        setVisible(true);
    }

    public static void main(String[] args) {
        new SwingGrillPro();
    }
}
```

Listing 28.13: CPU-Heizung im Thread

In diesem Programm läuft die CPU-Braterei in einem eigenen Thread. Dazu implementiert die Klasse das Interface `Runnable`. Das bedeutet, dass sie eine

Methode `run` zur Verfügung stellt, die dann später in einem eigenen Thread gestartet werden kann. Dorthin ist der Inhalt der Methode `brateCPU` gewandert. In der Methode `actionPerformed` wird nun, anstatt `brateCPU` direkt aufzurufen, ein Thread erzeugt. Als Parameter wird eine Referenz des Objekts übergeben, dessen Methode `run` als Thread laufen soll. In diesem Fall ist das `this`. Danach wird über das Thread-Objekt die Methode `start` aufgerufen, um den Thread zu starten.

Wenn Sie nun das Programm in Listing 28.13 starten, können Sie den Button jederzeit anklicken. Das Programm bleibt bedienbar, weil die Swing-Komponenten einen eigenen Thread haben. Sie können den Button ein zweites Mal anklicken und das Programm ganz normal über den Schließbutton beenden. Solange Sie aber nicht klicken, läuft die Prozessorbelastung weiter. Sie merken das, weil der Lüfter irgendwann startet. Sie können sich die Last auch per Task-Manager (Windows) oder top (Linux, Mac) anzeigen lassen.

Entspannte Schleifen

Falls Sie in Ihrem Programm eine Schleife haben, die ebenfalls dazu neigt, das Lüftersystem des Computers zu belasten und selbst neuere Geräte träge erscheinen zu lassen, können Sie erstaunliche Effekte erzielen, indem Sie in diese Schleife einen Sleep-Aufruf von wenigen Millisekunden schmuggeln. Eine solche Pause hat gleich mehrere Vorteile:

- Der Prozessor wird mal kurz aus der ersten Reihe genommen und kann ein wenig entspannen.
- Der Prozess gibt seine Zuständigkeit für einen Augenblick ab und ermöglicht so anderen Prozessen, zwischendurch Luft zu schnappen und ihre persönlichen Dinge zu regeln.
- Es gibt keinen Ärger mit der Gewerkschaft, weil die gesetzlichen Ruhezeiten eingehalten werden.

Die wenigen Millisekunden werden der Performance nicht unbedingt wehtun, aber können dem Anwender den Eindruck vermitteln, dass der Computer nicht am Rande der Erschöpfung steht.

Multitasking in der GUI

Beim Grillen der CPU haben Sie gesehen, dass grafische Oberflächen besonders darunter leiden, wenn sie keine freie Fahrt haben. Das hängt damit zusammen, dass ein Fensterprogramm ereignisgesteuert ist. Die verschiedenen Ereignisse erhält es durch einen Nachrichtenkanal. Ist der verstopft, kann es nicht mehr atmen. Solch verstopfte Programme werden über kurz über lang vom System beschimpft, weil sie keine Rückmeldungen liefern. Das ist nicht verwunderlich, weil man schlecht

antworten kann, wenn man die Anfrage gar nicht erst empfangen hat.

Da die grafischen Oberflächen ihren Nachrichtenkanal frei halten sollen, ist es nicht verwunderlich, dass sie eigene Mittel zur Verfügung stellen, um Hintergrundaktivitäten zu erlauben. Das Beispiel in Listing 28.14 zeigt den internen Timer von Swing. Er kann aufgezogen werden und wird nach Ablauf der Zeit über den Swing-Nachrichtenkanal eine Action auslösen, die der `ActionListener` auf die gleiche Weise empfängt wie den Klick auf einen Button.



Viele grafische Bibliotheken sind nicht *threadsafe*. Das bedeutet, dass Sie beispielsweise Dialoge oder Fenster nicht in unterschiedlichen Threads parallel laufen lassen sollten, schon weil Fensterprogramme auf einen gemeinsamen Nachrichtenkanal angewiesen sind. Es ist allerdings völlig in Ordnung und oft auch dringend erforderlich, dass Sie länger währende Aufgaben in einen parallelen Thread auslagern.

Das Programm in Listing 28.14 startet beim Klick auf den Button eine Zeitbombe, die im Sekundentakt tickt, sobald Sie auf den Button mit der Beschriftung Start drücken. Das Timer-Ereignis wird ebenso in der Methode `actionPerformed` gefangen wie der Button-Klick. Mit der Methode `getSource` ermittelt sie den Verursacher der Störung und kann angemessen reagieren.

- Wurde auf den Button geklickt, wird der Timer gestartet oder gestoppt und die Beschriftung des Buttons angepasst.
- Ansonsten muss es wohl der Timer gewesen sein. Also schauen wir, ob die Zeit abgelaufen ist oder ob noch getickt werden muss.

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingConstants;
import javax.swing.Timer;

public class SwingTimerDemo extends JFrame
                            implements ActionListener {

    private Timer timer;
    private JButton btStartStopp = new JButton("Start");
    private JLabel ticker = new JLabel("Tick",
            SwingConstants.CENTER);
    private int sekunden = 10;

    public SwingTimerDemo() {
        setLayout(new BorderLayout());
        add(BorderLayout.SOUTH, btStartStopp);
        add(BorderLayout.CENTER, ticker);
        timer = new Timer(1000, this);
        timer.setRepeats(true); // Immer wieder feuern!
        btStartStopp.addActionListener(this);
        setSize(200, 100);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        // Wer hat die Action losgetreten?
        if (e.getSource() == btStartStopp) { // Der Button!
            sekunden = 10;
```

```
            if (timer.isRunning()) {
                timer.stop();
                btStartStopp.setText("Start");
            } else {
                timer.start();
                btStartStopp.setText("Stopp");
                ticker.setText("" + sekunden);
            }
        } else { // Es muss wohl der Timer gewesen sein
            sekunden--;
            if (sekunden == 0) {
                ticker.setText("Bumm");
                timer.stop();
                btStartStopp.setText("Start");
            } else {
                ticker.setText("" + sekunden);
            }
        }
    }

    public static void main(String[] args) {
        new SwingTimerDemo();
    }
}
```

Listing 28.14: Swing hat einen eigenen Timer.

Natürlich können Sie den gleichen Effekt mit einem Thread erzielen. Wenn Sie allerdings nur einen Timer realisieren müssen, ist es sinnvoll, das Angebot von Swing anzunehmen.



Lassen Sie sich nicht mit diesem Buch in der Nähe eines Flughafens erwischen. Die Geheimdienste könnten das Listing lesen und als Anleitung zum Zeitbombenbau auslegen. Sie sollten auch in freiheitlichen Demokratien damit rechnen, dass Geheimdienste paranoid sind und keinen Sinn für Humor besitzen.

Aufgabe

Ändern Sie das Programm in Listing 28.14 zu einer Klasse `ZeitbombeThread`, die statt des Swing-Timers einen Thread verwendet.

Teil VII

Netzwerke und Internet



Kapitel 29

Netzwerkprogrammierung

IN DIESEM KAPITEL

Das Netzwerk kommt in die Sockets.

Java schreibt und empfängt Briefe.

Java spricht mit Java per RMI.

Ein Netzwerk verbindet Computer durch Kabel. Und wenn kein Kabel zur Hand ist, dann darf es auch gern Funk sein. Das nennt sich dann WLAN. Auch Mobilfunkübertragungen lassen sich für die Netzwerkverbindungen nutzen und heißen dann UMTS oder LTE. Ein Netzwerk bildet eine Einheit, in der alle Computer miteinander kommunizieren können. Durch die Verbindung vieler Netzwerke entstand ein Hypernetzwerk, das wir Internet nennen. Wie die Computer rein technisch verbunden sind, interessiert den Programmierer weniger. Viel spannender ist, wie Programme sich unterhalten. Und letztlich sprechen die Programme miteinander.

Die Netzwerke werden immer wichtiger. Smartphones und Tablets haben oft keinen USB-Anschluss, um Daten bequem an die Außenwelt zu liefern. Das Einzige, was die kleinen Kerlchen prima können, ist der Datenverkehr über das Netzwerk. Und genau darum ist es wichtig, dass sich Programmierer damit beschäftigen.

Das Liebesgeflüster der Sockets

Die Netzwerkwelt spricht heute TCP/IP. Das IP hinter dem Bruchstrich steht für *Internet Protocol* und damit wird auch klar, wo dieses Protokoll seinen Anfang nahm. Gab es bis in die 1990er Jahre auch noch andere Protokolle, sind diese inzwischen dem Protokoll TCP/IP gewichen. Und ganz ehrlich: Seien Sie froh drum. Die Vielfalt kann ganz schön anstrengend sein.

Die Basis eines Gesprächs unter TCP/IP-Freunden sind die Sockets. Tatsächlich sind diese kleinen Racker immer involviert, auch wenn sie in komplexeren Protokollen manchmal nicht mehr zu sehen sind. Alle Dienste im TCP/IP-Netzwerk und damit im Internet lassen sich mit Sockets nachbilden. Und da Sockets relativ simpel arbeiten, können Sie daran auch lernen, wie ein Netzwerk so tickt.

Zur Unterhaltung gehören immer zwei. Und zwar zwei Programme. Wenn die Programme gleichzeitig anfangen zu sprechen, können Sie damit zwar eine tolle Kaffeeklatschsimulation bauen, aber Kommunikation ist etwas anderes. Höchstens Gerüchte lassen sich so erzeugen.

Begriffsklärung

Ein paar Begriffe tauchen bei der Programmierung von Client-Server-Pärchen immer wieder auf:

- *Client:* Ein Client ist ein Programm. Es beginnt die Kommunikation mit einer Anfrage.
- *Server:* Auf dem Ziel-Computer sitzt wiederum ein Server-Programm und wartet die ganze Zeit darauf, dass ihm jemand eine Anfrage vorlegt. Solange niemand fragt, tut der Server nichts. Aber wenn jemand fragt, ist er sofort zu Diensten und liefert eine Antwort. Auch wenn viele Leute von dem Server als Hardware sprechen, ist der Server eigentlich ein Programm.
- *Host:* Das ist der Computer, auf dem sich einer der Spieler befindet. Dieser kann einen Namen haben oder eine IP-Adresse, wie beispielsweise 192.168.1.17. Findet die gesamte Kommunikation auf ein und demselben Rechner statt, kann man den Namen `localhost` oder 127.0.0.1 verwenden.
- *Socket:* Socket ist die Steckdose, durch die die Programme auf den Draht gehen. Die Sockets sind durchnummeriert.
- *Port:* Ein Port ist ein Socket, dessen Nummer sich lohnt zu merken, weil dort das Serverprogramm Anfragen entgegennimmt. Beispielsweise nimmt ein Webserver standardmäßig Anfragen auf Port 80 entgegen.

Romeo hat eine Anfrage

Um eine Kommunikation über das Netzwerk zu programmieren, benötigt man sinnigerweise zwei Programme: einen Anfrager und einen Diener, also Client und Server. Sie benötigen aber nicht zwingend auch zwei Computer. Sie können die Kommunikation auf demselben PC durchführen. Die Adresse des eigenen Rechners heißt immer `localhost`. Und falls Sie das als IP-Adresse ausdrücken wollen: 127.0.0.1.

Der Datenverkehr ist simpel: Der Client schickt eine kurze Nachricht an den Server, der diese zur Kontrolle auf dem Bildschirm ausgibt. Dann schickt der Server eine Antwort an den Client, die dieser ebenfalls auf dem Bildschirm auswirft. Der Client in Listing 29.1 wird von Romeo verwendet, um Julia eine Botschaft der Liebe zu senden.

```
import java.io.DataInputStream;
import java.io.IOException;
import java.io.PrintStream;
import java.net.Socket;

public class Client {

    public static void main(String[] args) {
        Socket socket = null;
        try {
            System.out.println("Client sendet");
            socket = new Socket("localhost", 33030);

            PrintStream output = new PrintStream(
                        socket.getOutputStream());
            output.println("Hab mich lieb, Julia!");
            System.out.println("Gesendet ... Client wartet");

            DataInputStream input = new DataInputStream(
                        socket.getInputStream());
            String str = input.readLine();
            System.out.println("Empfangen: " + str);
            input.close();
            output.close();
            socket.close();
        } catch (IOException e) {
            System.out.println("IO-Probleme ...");
            e.printStackTrace();
        }
    }
}
```

Listing 29.1: Der Client fragt ganz lieb.

Romeos Client arbeitet auf folgende Weise:

1. Der Konstruktor des Sockets erwartet als Parameter die Adresse des Computers und den Port, an den sich die Anfrage richtet. Im Beispiel wurde 33030 verwendet, weil zu hoffen steht, dass diesen sonst niemand belegt.
2. Über den Socket wird nun ein Ausgabestrom gebildet, wie Sie es schon von Dateien her kennen. Tatsächlich verhalten sich Sockets an manchen Stellen auch durchaus dateiähnlich. Selbst der Ausnahmefall ist die Regel: Bei Fehlern wird eine `IOException` ausgelöst. Das Senden an einen Socket erfolgt ohne

Blockade des Prozesses.

3. Nun will der Client wissen, was der Server antwortet. Solange noch keine Antwort vorliegt, blockiert das Programm. Auch hier wird ein Eingabestrom über den Socket gebildet.
4. Im Abgang wird die Servernachricht ausgegeben, die Datenströme und schließlich auch der Socket per `close` geschlossen.

Wenn Romeo auf einen leeren Balkon stößt, wird er nicht lange warten, sondern sich seine Liebesbekundungen für einen besetzten Balkon aufsparen. Während es Romeo vielleicht noch passieren kann, dass er eine Weile die Rollos anschmachtet, weil er das Fehlen von Julia nicht bemerkt, wird ein Computer einen unbesetzten Port sofort erkennen und sich mit einem »Connection refused« abwenden.

Anders ist die Lage, wenn der Server nicht so schnell antwortet. Romeo wird länger warten, wenn Julia überlegt, ob sie sich wirklich einen haarigen Typ antun möchte, der jeden Samstagnachmittag biertrinkend vor der Sportschau zubringt. Dabei denkt sie vielleicht an ihren Vater und vermutet, dass sie wohl dem Schicksal ihrer Mutter nicht entkommen kann. Romeo wiederum hält das Zittern in ihrer Stimme für das Tremolo unbändiger Liebe.

Julia wartet und antwortet

Wer sich um schmachtende Männerherzen kümmern will, muss früh aufstehen und bereits den Balkon besetzt haben, bevor die Anfrage kommt. So ist es auch beim Serverprozess.

1. Der Server startet, belegt einen Port, den er als Konstruktorparameter an `ServerSocket` übergibt. Sollte bereits ein anderes Programm diesen Port belegen, führt das zu einer Exception.
2. Der Aufruf von `accept` blockiert den Server, bis eine Anfrage eintrifft.
3. Der Inhalt der Nachricht des Clients wird über einen Eingabe-Stream gelesen.
4. Der Server verarbeitet die Anfrage. In unserem Beispiel ist dies einfach nur eine Ausgabe auf der Konsole.
5. Anschließend sendet der Server seine Anfrage an den Client zurück.
6. Während Julia nach ihrem Kontakt mit Romeo bedient ist und keine weiteren Männer mehr sehen will, sind Serverprogramme weniger wählerisch. Sie laufen in einer Endlosschleife und bedienen jeden dahergelaufenen Client, der ihre Nummer kennt.



Bei der Entwicklung von Servern ist es ein typischer Fehler, dass man den alten Server noch nicht beendet hat und sich darüber wundert, warum der

Server jetzt plötzlich nicht mehr starten will.

```
import java.io.DataInputStream;
import java.io.IOException;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;

public class Server {

    public static void main(String[] args) {
        try {
            ServerSocket server = new ServerSocket(33030);
            while (true) {
                System.out.println("warte ...");
                Socket socket = server.accept();
                System.out.println("accepted");

                DataInputStream input = new DataInputStream(
                        socket.getInputStream());
                String str = input.readLine();
                System.out.println("Empfangen: " + str);

                PrintStream output = new PrintStream(
                        socket.getOutputStream());
                output.println("Du mich auch, Romeo!");

                input.close();
                output.close();
                socket.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Listing 29.2: Und der Server antwortet.



Wenn Sie die beiden Programme unter Eclipse starten, erzeugt Eclipse für jedes Programm eine eigene Console. Sie können zwischen den beiden umschalten, indem Sie den Pfeil neben dem Monitorsymbol anklicken. Dort finden Sie eine Liste mit Einträgen für den Server und den Client.

Den Server müssen Sie abschießen, wenn Sie ihn nicht mehr brauchen. Unter Eclipse klicken Sie auf das rote Quadrat unter Console. Wenn Sie den Server von einer Eingabeaufforderung gestartet haben, können Sie ihn mit der Tastenkombination Strg + C beenden.



Viele Messgeräte sind über TCP/IP ansprechbar. In diesem Fall finden Sie in der Dokumentation, wie Sie das Gerät ansprechen können. Dazu können Sie den Socket-Client aus diesem Abschnitt anpassen und die Ergebnisse weiterverarbeiten.

Mails versenden

Bei der Durchsicht meiner E-Mails beschleicht mich manchmal der Verdacht, dass die eine oder andere Nachricht gar nicht persönlich an mich gerichtet ist, sondern von einem Computer an ganz viele Leute verteilt worden ist.

Mail-Protokolle

Wenn Sie von Ihrem Programm aus Mails versenden wollen, benötigen Sie einen Mail-Server, der Ihre Mails akzeptiert und an den Adressaten weiterleitet. Dementsprechend müssen Sie sich an das Protokoll des Servers halten. Dieses heißt SMTP und basiert natürlich auch auf TCP/IP und damit auf der Socket-Programmierung. Wenn Sie das Protokoll also nachprogrammieren, können Sie direkt mit einem Mail-Server kommunizieren.

Allerdings ist es einfacher, wenn Sie eine Bibliothek zur Hilfe nehmen, die SMTP beherrscht. Dafür gibt es die JavaMail-API. Und im Umgang damit ist das Senden von Mails schon deutlich einfacher.

Der Mail-Server, den Sie beauftragen, wird in den meisten Fällen nur Absender der gleichen Domain akzeptieren. Wenn Sie also, wie im Beispiel beschrieben, den GMX-Mail-Server verwenden, müssen Sie auch eine GMX-Mail-Adresse als Absender benutzen. Sie müssen außerdem damit rechnen, dass Sie sich mit Benutzer und Passwort authentifizieren und verschlüsselt senden müssen.

Eclipse mit JavaMail-API ausstatten

Tatsächlich können Sie Ihr Java-Programm dazu bringen, E-Mails zu versenden. Dazu benötigen Sie die JavaMail-API, die allerdings nicht Bestandteil der Java Standard Edition ist. Zum Glück können Sie aber nachrüsten. Unter der URL `javamail.java.net` finden Sie unter Downloads die Datei *javax.mail.jar*, die Sie herunterladen.

1. Klicken Sie das Projekt mit der rechten Maustaste an und wählen Sie

`Properties`.

2. In der linken Spalte klicken Sie Java Build Path an.
3. Auf der rechten Seite klicken Sie auf den Reiter Libraries.
4. Rechts neben der Liste erscheinen Buttons. Klicken Sie auf Add External Jars.
5. Mit der Dateiauswahlbox bewegen Sie sich dorthin, wo Sie die Datei *javax.mail.jar* nach dem Download geparkt haben.
6. Beenden Sie den Dialog mit OK.

Wenn Sie in Ihr Projekt schauen, finden Sie die JAR-Datei unter dem Verzeichnis *Referenced Libraries*. Eine ausführlichere Beschreibung für den Umgang mit JAR-Dateien finden Sie in Kapitel 11.

Wir senden eine Mail

Das folgende Programm sendet eine Mail über den GMX-Server. Die Adresse im String `absender` und das Passwort in `passwort` müssen Sie auf Ihre eigene GMX-Adresse anpassen. Natürlich müssen Sie auch die Adresse in `empfaenger` setzen. Letztere muss selbstverständlich nicht von GMX stammen.

Für den Inhalt füllen Sie die Variablen `betreff` und `inhalt`. Nachdem Sie das Programm gestartet haben, müsste kurze Zeit später eine E-Mail in Ihrem Mail-Programm auftauchen.

```
import java.util.Date;
import java.util.Properties;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.NoSuchProviderException;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

public class Versenden {

    static String smtpHost = "smtp.gmx.de";
    static String absender = "absenderadresse@gmx.de";
    static String passwort = "xxxxxxx";
    static String empfaenger = "zieladresse@provider.de";
    static String betreff = "Automatische Antwort";
    static String inhalt = "Dies schreibt mein Java-Programm.";

    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("mail.smtps.host", smtpHost);
        props.put("mail.smtps.auth", "true");
        props.put("mail.smtp.starttls.enable", "true");
        props.put("mail.smtp.port", "587");
        Session session = Session.getInstance(props, null);
        Message msg = new MimeMessage(session);
        try {
            msg.setFrom(new InternetAddress(absender));
            msg.setRecipients(Message.RecipientType.TO,
                    InternetAddress.parse(empfaenger, false));
            msg.setSubject(betreff);
            msg.setText(inhalt);
            msg.setHeader("X-Mailer", "JavaProgramm");
            msg.setSentDate(new Date());
            Transport transport;
            transport = session.getTransport("smtp");
            transport.connect(smtpHost, absender, passwort);
            transport.sendMessage(msg, msg.getAllRecipients());
            transport.close();
        } catch (NoSuchProviderException e) {
            e.printStackTrace();
        } catch (MessagingException e) {
            e.printStackTrace();
        }
    }
```

```
}
```

Listing 29.3: Versenden einer Mail per Java

Das Programm arbeitet in den folgenden Schritten:

1. Session: Die Session enthält alle Parameter, die für den Datentransport erforderlich sind, wie den Host, den Port und ob Passwörter und Verschlüsselung erforderlich sind.

   Diese Informationen erhält die Session in Form von Properties, die an anderer Stelle im Buch beschrieben werden.

2. Message: Eine Message und insbesondere eine `MimeMessage` enthält vor allem die Informationen, die versandt werden sollen, also den Inhalt, den Betreff und die Empfängerliste. Zur Erstellung der Message wird die Session als Konstruktorparameter übergeben.

3. Transport: Der Transport ist der Arbeiter. Er sorgt für den Kontakt zum Mail-Server und sendet alles, was in `Message` enthalten ist, weiter.

Als Beispiel habe ich GMX verwendet. Dafür bekomme ich von GMX leider keine Provisionen. Aber GMX reagiert am zickigsten, was die Akzeptanz von Mails betrifft. Jedenfalls gibt es mehrere Dinge, die eine erfolgreiche Sendung behindern können:

- ✔ Der Benutzername und das Passwort müssen stimmen. Das scheint trivial, aber je nach Provider besteht der Benutzername aus der kompletten Absender-Adresse oder nur dem Teil vor dem @.
- ✔ Die meisten Provider weigern sich inzwischen glücklicherweise, Passwörter unverschlüsselt über die Leitung zu lassen. Damit ist der Port 25 so gut wie tot. Übrigens betrifft die Verschlüsselung nur die Strecke zwischen dem Client und dem Server. Auf dem Server selbst liegen alle Mails unverschlüsselt herum, damit die Nachrichtendienste nicht so viel Arbeit haben.
- ✔ Wenn der Absender nicht mit dem Konto übereinstimmt, wird die Mail nicht angenommen. Das gilt allerdings nicht für alle Provider. Sie können also mit diesem Programm sehr leicht eine Mail mit dem offiziellen Absender des Bundespräsidenten an jedermann senden. Sie müssen nur den richtigen Provider finden.

Jetzt noch einmal mit Anhang

Wenn Sie Rechnungen versenden wollen, erwartet heute jeder Kunde eine schicke PDF-Datei im Anhang. Eigentlich ist das Mail-Protokoll für binäre Anhänge nicht gedacht gewesen. Also wurde eine Codierung der Binärdaten in Text gebastelt. Sie bläht einen Anhang noch einmal gut um ein Drittel auf.

Das Programm zum Versenden der Mail muss für die Anhänge noch um MIME ergänzt werden:

1. Sie erzeugen ein Objekt der Klasse `MimeMultipart`, der mehrere `MimeBodyPart`-Objekte aufnehmen kann.
2. Der bisherige Inhalt wird als ein `MimeBodyPart` angelegt.
3. Der Anhang muss in einer Datei vorliegen. Deren Pfad muss in der String-Variablen `anhangdatei` stehen.
4. Für den Anhang wird ebenfalls ein Objekt der Klasse `MimeBodyPart` erzeugt.
5. Schließlich wird das `MimeMultipart`-Objekt der Message als Inhalt durch den Aufruf von `setContent` übergeben.
6. Weiter geht es wie gehabt mit dem Transport.

```
// Beim Anhang kommt der Text in den MimeBodyPart
// msg.setText(inhalt);
msg.setSentDate(new Date());
// Neu bei Versenden eines Anhangs:
MimeMultipart mimeMultipart = new MimeMultipart();
MimeBodyPart text = new MimeBodyPart();
// Bei einem Anhang muss der Inhalt in den MimeBodyPart
text.setText(inhalt);
text.setDisposition(MimeBodyPart.INLINE);
// Der eigentliche Anhang:
File file = new File(anhangdatei);
MimeBodyPart attachement = new MimeBodyPart();
attachement.setDataHandler(new DataHandler(new FileDataSource(file)));
attachement.setFileName(anhangdatei);
attachement.setDisposition(MimeBodyPart.ATTACHMENT);
mimeMultipart.addBodyPart(text);
mimeMultipart.addBodyPart(attachement);
msg.setContent(mimeMultipart);
```

Listing 29.4: Ergänzung um einen Anhang

Java als Web-Client

Wenn es schon so viele Webserver im Internet gibt, möchte man sie natürlich auch gern von Java aus ansprechen.

Als Client für einen Webserver tritt in der Regel ein Browser auf, also Firefox oder einer seiner Kollegen Safari, Chromium oder wie sie auch immer heißen mögen. Sie können aber auch mit einem Java-Programm einem Webserver Löcher in den Bauch fragen.

Da ein Webserver TCP/IP als Protokoll verwendet und dementsprechend auch

Sockets verwendet, könnten Sie ihn auch per Socket-Kommunikation ansprechen. Sie müssten sich lediglich an dessen Protokoll HTTP halten. Das ist auch gar nicht so schwierig. Einfacher ist es jedoch, wenn Sie eine `HttpURLConnection` aufmachen.

Das folgende Programm verbindet sich zu der Web-Adresse, die in der Variablen `url` steht. Viele Web-Anbieter sind inzwischen auf HTTPS umgestiegen. Dazu müssen Sie lediglich den Inhalt von `http` auf `https` umstellen.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class WebLeser {

    public static void main(String[] args) {
        String url = "http://www.willemer.de";
        String charset = "UTF-8";
        HttpURLConnection httpVerbindung;
        try {
            httpVerbindung =
                    (HttpURLConnection) new URL(url).openConnection();
            httpVerbindung.setRequestProperty("Accept-Charset",
              charset);
            httpVerbindung.setRequestMethod("GET");
            InputStream inStream = httpVerbindung.getInputStream();
            BufferedReader in = new BufferedReader(
                    new InputStreamReader(inStream));
            String empfangeneZeile;
            StringBuffer antwort = new StringBuffer();
            while ((empfangeneZeile = in.readLine()) != null) {
                antwort.append(empfangeneZeile+"\n");
            }
            in.close();
            System.out.println(antwort.toString());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Listing 29.5: Java-Client zum Auslesen einer Web-URL

Das Programm arbeitet in folgenden Schritten:

1. Zunächst wird ein Objekt der Klasse `URL` für die im Parameter des Konstruktors

angegebene Zieladresse erzeugt.

2. Über deren Methode `openConnection` wird eine Verbindung vom Typ `HttpURLConnection` gebildet.
3. Über deren Methode `setRequestProperty` können Einstellungen der Verbindung aufgenommen werden. Hier im Beispiel wird der Zeichensatz voreingestellt. Auf diese Weise können selbst veraltete Windows-PCs noch korrekte Umlaute bekommen.
4. Die Methode `setRequestMethod` kann das HTTP-Kommando einstellen. Am bekanntesten dürften POST für das Senden von Formularinhalten und GET für das Lesen von Seiten sein.
5. Über die so vorkonfigurierte Verbindung wird nun ein `InputStream` angefordert. Diesen kennen Sie schon aus Kapitel 23 über Dateien. Und tatsächlich kann die HTTP-Verbindung wie eine Datei ausgelesen werden.
6. In einer Schleife wird Zeile für Zeile eingelesen und diese in der Variablen `antwort` gesammelt und zum Schluss auf dem Bildschirm ausgegeben.

Java ruft Java: RMI

Wenn Sie eine Verbindung zwischen zwei Teilen Ihrer Anwendung bauen wollen, können Sie natürlich die Sockets verwenden. Wenn auf beiden Computern ein Java-Programm werkelt, wird es in den meisten Fällen einfacher sein, RMI (Remote Method Invocation) zu verwenden. RMI ermöglicht es Ihnen, eine Methode auf dem anderen Computer über das Netzwerk aufzurufen.

Das Interface des Servers

Der Fernaufruf basiert auf einem Interface, das `Remote` erweitert. Der Server wird es implementieren und der Client aufrufen. Auf beiden Seiten muss das Interface also vorhanden sein.

In Listing 29.6 wird ein solches Interface erstellt, das zwei Methoden deklariert.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ServerInterface extends Remote {
    public String bitteMeldeMich(String name)
            throws RemoteException;
    public int gibZahl() throws RemoteException;
}
```

Listing 29.6: RMI-Interface

Sie sehen schon, dass das Package `java.rmi` die benötigten Klassen für RMI zur Verfügung stellt. Dieses Interface wird nun vom Server implementiert.

Der RMI-Server

Der RMI-Server muss gestartet sein, bevor ihn der Client aufruft.

```
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class MeinServer extends UnicastRemoteObject
                        implements ServerInterface {

    MeinServer() throws RemoteException {
        super();
    }

    public static void main(String[] args) {
        try {
            LocateRegistry.createRegistry(
                    Registry.REGISTRY_PORT);
            Naming.rebind("MeinServer", new MeinServer());
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    @Override
    public int gibZahl() throws RemoteException {
        return 42;
    }

    @Override
    public String bitteMeldeMich(String name)
            throws RemoteException {
        return "Hallo "+name;
    }
}
```

Listing 29.7: Ein einfacher RMI-Server

In der Methode `main` präpariert sich der Server für einen späteren Aufruf durch den Client.

1. Der Server erzeugt eine Registry auf der Standard-Portnummer.
2. Der Server bindet sich unter dem Namen `MeinServer` an diesen Port.

Der Server implementiert die beiden Methoden des Interface. Beide müssen angeben, dass sie gegebenenfalls eine `RemoteException` werfen könnten.

Der Server wird von der Konsole einfach gestartet und blockiert, ohne einen Ton von sich zu geben, sofern alles in Ordnung ist.

Ein RMI-Client

Der RMI-Client sucht nach dem Namen `MeinServer` in der Registry. Der URL-String hat den allgemeinen Aufbau: *rmi://host:port/objekt* Dann ruft er die beiden Methoden des Servers nacheinander auf und gibt dessen Rückgaben auf dem Bildschirm aus.

```
import java.rmi.Naming;

public class MeinClient {
    public static void main(String[] args) {
        String url = "rmi://127.0.0.1/MeinServer";
        try {
            ServerInterface server = (ServerInterface)
                    Naming.lookup(url);
            String meldung = server.bitteMeldeMich("Arnold");
            System.out.println(meldung);
            int zahl = server.gibZahl();
            System.out.println(zahl);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Vor dem Start des Clients muss der Server natürlich gestartet sein. Der Start des Clients bewirkt einen Aufruf der Server-Methoden, deren Ergebnisse er auf dem Bildschirm ausgibt.

Objekte als Parameter

Wenn Sie simple Typen in einer RMI-Schnittstelle verwenden, gibt es keinen Ärger. Wenn es sich allerdings um komplexere Objekte handelt, wird das Leben für eine Übertragung per Netzwerk schwieriger. Das Objekt muss klein gehackt, in Stückchen über den Draht gesendet und auf der anderen Seite wieder zusammengesetzt werden. Dieser brachiale Umgang mit filigranen Daten wird in Java freundlich mit *serialisieren* umschrieben.

Immerhin müssen Sie sich als Programmierer nicht weiter darum kümmern. Die Klasse muss nur mit dem Hinweis ausgestattet werden, dass sie `Serializable` implementiert. Sowohl für den Server als auch für den Client muss eine gleichlautende Klasse zur Verfügung stehen.

```
import java.io.Serializable;

public class Person implements Serializable {
    String name = "";
    String adresse = "";
    private int gebJahr = 1990;

    public void setGebJahr(int gebJahr) {
        this.gebJahr = gebJahr;
    }
    public int getGebJahr() {
        return gebJahr;
    }
}
```

Listing 29.8: Eine Klasse `Person` für die Parameter

Mit dieser Person sollen nun zwei Dinge geschehen:

- ✔ Der Server soll eine neu erzeugte Person erstellen und mit Daten vorbesetzen.
  `public Person lieferePerson() throws RemoteException;`
- ✔ Der Server soll eine Person vom Client übernehmen und ändern. Als Beispiel wird die Person einem Alterungsprozess unterworfen, den wir alle an unserem Geburtstag durchleiden.
  `public Person altere(Person person) throws RemoteException;`

Beide Zeilen müssen dem Interface hinzugefügt werden. Natürlich muss `MeinServer` sie implementieren:

```
@Override
public Person lieferePerson() throws RemoteException {
    Person person = new Person();
    person.name ="Alexander";
    person.setGebJahr(1996);
    return person;
}

@Override
public Person altere(Person person) throws RemoteException {
    int jahr = person.getGebJahr();
    person.setGebJahr(jahr+1);
    return person;
}
```

Listing 29.9: Implementierung der Personen-Methoden

Zum Test wird der Client nun um folgende Zeilen ergänzt:

```
Person person = server.lieferePerson();
System.out.println(person.name);
System.out.println(person.getGebJahr());
person = server.altere(person);
System.out.println(person.getGebJahr());
```

Listing 29.10: Nachfrage nach Personen



Ein Unterschied ist noch relevant: Lokal wird der Parameter als Referenz übergeben. Manipulationen am Objekt wirken sich auf das übergebene Objekt aus. Bei RMI werden die Parameterobjekte nicht an den Aufrufer zurückgegeben. Stattdessen werden die veränderten Objekte als Rückgabewert an den Aufrufer gesandt.

Time-out und Parallelität

Jedes Warten auf Antwort führt im Netzwerk zum Blockieren des Programms. Falls der Client eine grafische Oberfläche hat, muss dann ein Thread her. Denn wenn die Oberfläche mitblockiert, funktioniert kein Button und kein Neuzeichnen der Fenster mehr.

Darüber hinaus will der Client in der Regel nicht ewig auf eine Antwort warten. Nach einer gewissen Schamfrist sollte der Versuch der Kontaktaufnahme mit dem tiefsten Ausdruck des Bedauerns abgebrochen werden.

Aber auch der Server kann keine Anfrage beantworten, solange er die letzte

Anfrage nicht fertig bearbeitet hat. Das ist kein Problem, solange es geduldige und vor allem nur wenige Clients gibt. Ansonsten wird es notwendig sein, dass die Anfragen parallel bearbeitet werden müssen. Und dann gibt es sofort das Problem, wenn sich die parallelen Anfragen gegenseitig in die Daten greifen.

Dies und alle anderen Fragen rund um das Thema Threads sind für die Netzwerkprogrammierung so wichtig, dass sie in Kapitel 28 behandelt werden.

Kapitel 30

Heidis Berge und die Enterprise Edition

IN DIESEM KAPITEL

- Von Glassfischen und Katern zu Application Servern
- HTTP als Protokollbasis
- Servlets als Antwort auf Formulare
- Java Server Pages oder Java Server Faces
- Datenbanken im Hintergrund mit JDBC oder JPA
- REST als Verwalter für Ressourcen
- Mit SOAP Remote Procedure Calls realisieren

Heidi ist zurück aus der Großstadt in ihren geliebten Bergen. Das Leben dort ist karg, aber inzwischen hat es sich unter Rollstuhlfahrern herumgesprochen, dass in Heidis Bergen schon einmal ein Mädchen aus dem Rollstuhl aufstand und plötzlich laufen konnte. So ist Heidis Alm zum Wallfahrtsort der Rollstuhlfahrer geworden, der selbst von der Allgemeinen Ortskrankenkasse anerkannt ist. Nachdem Heidi überrascht feststellte, wie viele Häuser selbst auf ihrer kleinen Alm Platz finden, wühlen dort nun die Bagger und bauen Ferienhütten und Heidi träumt vom Reichtum.

Aber die Buchungen bekommt Heidi nicht in den Griff. Sie schleppt ständig einen Zettelkasten mit sich herum. Schließlich erklärt ihr Peter, der sich auf der Suche nach seinen Ziegen ins Silcon Valley verlaufen hatte, dass man dort Application Server mit Java Enterprise Edition programmiert. Damit könnte man doch die Buchungen zentral auf einem Server verwalten. Heidi kann über ihren Laptop im Büro die Anfragen erfassen, während Peter beim Reinigen der Ferienhütte auf dem Smartphone nachsehen kann, wie viele Gäste als Nächstes die Hütte belegen wollen.

Peter hat daraufhin beim Alm-Öhi billig einen ausrangierten Server gekauft und darauf Glassfish installiert. Der Alm-Öhi findet die Idee auch gut und erklärt sich bereit, die komplette IT-Ausstattung zu liefern. Heidi hat schon oft erlebt, dass Peter verrückte Ideen hat. Vom Alm-Öhi weiß sie, dass er das Tannenrauschen

hinterm Haus zu gern gegen das Meeresrauschen an der Flensburger Förde tauschen will. Ob sie ihm trauen kann?

Sie telefoniert mit Klara, die seit ihrer wundersamen Heilung in Frankfurt als Freelancer arbeitet. Klara erklärt Heidi, dass sie irre Stundensätze bekommt, seit sie JEE (Java Enterprise Edition) beherrscht. Dazu brauche sie nur einen Applikationsserver wie beispielsweise Glassfish. Dieser nutzt das Protokoll HTTP des World Wide Webs, um Client-Server-Architekturen zu bauen. Und das könne Heidi auch lernen. Immerhin hätte sie ja an der Hochschule Flensburg zwei Semester Medieninformatik studiert. Sie bräuchte nur das Buch »Java für Dummies – Alles in einem Band« zu kaufen. Darin wäre alles ganz toll erklärt.

Das Konzept des Application Servers

Die Java Enterprise Edition soll Java für das Unternehmen darstellen. Der Unterschied zwischen einem Patience-Spiel und einer Unternehmens-Software liegt vor allen darin, dass Unternehmen zentrale Daten und Abläufe kennen, die von mehreren Arbeitsplätzen aus genutzt werden sollen. Darum setzt die Java Enterprise Edition auf einen sogenannten *Application Server*, der folgende Gedanken berücksichtigt:

- Eine zentrale Datenbank ist schon ein guter Anfang. Damit sind die Datenbestände, aber noch nicht die Abläufe zentralisiert. Der Application Server unterstützt darum den Zugriff auf Datenbanken.

  Natürlich ist es leicht möglich, Programme für den Application Server zu schreiben, die die Tabellen einer Datenbank einfach an den Client durchreichen, beispielsweise mit einem REST-Service.
- Das andere Extrem ist ein Terminalserver, auf dem die komplette Anwendung wie bei den alten Hostservern der 1960er Jahre auf dem zentralen Server stattfindet. Rhetorisch wird dieser Ansatz durch den Kampfbegriff vom »Fat Client« unterfüttert. An diesem Konzept sind vor allem Hersteller teurer Hochleistungsrechner und der entsprechenden Software interessiert. Häufig werden Terminalserver eingesetzt, wenn sich der Software-Hersteller um die Umsetzung einer Client-Server-Architektur drückt. Darum wird sie auch vom Application Server nicht unterstützt.
- Ein dazwischen liegender Ansatz ist die Verwendung eines Browsers als Client. Diese Lösung bietet den Charme, dass alles als Client eingesetzt werden kann, was einen Browser starten kann, vom Smartphone bis zur Workstation, unabhängig von Betriebssystem und Programmiersprache.

  Hier tummelt sich seit langer Zeit die Programmiersprache PHP, deren Beliebtheit bei Vollblut-Informatikern gern mit Metaphern unangenehmer Krankheiten beschrieben wird. Dieser Ansatz lässt sich durch Java Server Pages

leicht realisieren und durch die moderneren Java Server Faces übertreffen.

- Eine gelungene Client-Server-Architektur sorgt dafür, das der Server so weit entlastet wird, wie die Arbeit auch dezentral vom Client übernommen werden kann. Schließlich sind heutzutage selbst die billigsten Computer extrem leistungsfähig.

  Die Optimierung der Lastverteilung sollte die über das Netzwerk übertragene Datenmenge im Blick behalten. Zwar sind lokale Netzwerke inzwischen sehr schnell, aber gerade, wenn die Daten zu Filialen über das Internet gesendet werden, fallen hohe Latenzzeiten auf die Qualität der Software zurück.

- Natürlich können Sie eine Socket- oder eine RMI-Kommunikation selbst bauen. Aber es ist viel praktischer, einen fertigen Server vorzuhalten, der die komplette Serverkommunikation übernimmt. Damit muss sich die Anwendung nicht um so gemeine Dinge wie Parallelität und Lastverteilung kümmern.

- Statt ein neues Protokoll zu erfinden, wird das HTTP der Webserver übernommen. Es ist erprobt, funktioniert mit diversen Client-Umgebungen und kommt, wenn es den Port der Webserver verwendet, sogar perfekt durch die verschiedenen Firewalls, weil den meisten Arbeitsplätzen das Surfen erlaubt ist. Darüber hinaus besitzt HTTP vier Kommandos, die wunderbar zu den Anforderungen der Anwendungsprogrammierung passen:

  - GET: Damit holt ein Browser Texte und Bilder von einem Webserver. Der Client sendet eine URL und der Server liefert die zugehörigen Daten. In der URL kann durch die Pfadangabe die Anwendung und die Ressource bestimmt werden. Daran können sogar noch Parameter oder kleinere Datenbestände angehängt und so an den Server übergeben werden.
  - POST: Der Client sendet Daten an den Server. Dieses Kommando wird von HTTP vor allem benutzt, um HTML-Formularinhalte an den Server zu senden. Da die Daten nicht über die URL versendet werden, werden sie bei HTTPS-Verbindungen auch automatisch verschlüsselt. Im Gegensatz zu den GET-Parametern ist die Datenmenge auch nicht beschränkt.
  - PUT: Mit diesem Befehl steht eine zweite Variante für das Senden von Daten zur Verfügung. REST verwendet den Befehl gern für das Erzeugen neuer Datenobjekte.
  - DELETE: Damit steht auch ein Löschbefehl zur Verfügung.

- HTTP-Server senden üblicherweise HTML-Dokumente, die schon optisch XML ähneln. Mit XHTML steht sogar eine HTML-Variante zur Verfügung, die den strengeren Vorgaben von XML genügt. Wenn sich Client und Server also auf XML einigen, können damit beliebige Datenstrukturen ausgetauscht

werden.

Aber es ist gar nicht zwingend erforderlich, XML vorzuschreiben. In manchen Fällen kann JSON sehr viel praktischer sein. Und auch HTML bietet Vorteile, wenn ein Browser als Client eingesetzt werden soll.

- Auf der Basis dieser Technologien lassen sich zwei sehr unterschiedliche Kommunikationsmuster realisieren:
  - Der Client ruft Funktionen des Servers auf. Wie bei einem Methodenaufruf werden die Daten als Parameter einer Aktivität betrachtet. Dieses Konzept ist als *Remote Procedure Call* bekannt und wird beispielsweise bei SOAP eingesetzt.
  - Die Ressource steht im Mittelpunkt. Dieser Ansatz steht insofern der objektorientierten Denkweise näher, als dass Datenobjekte gestaltet werden, für die Methoden aufgerufen werden. Dieser Ansatz wird beispielsweise von REST umgesetzt.
- Der Application Server unterstützt den Anwendungsprogrammierer auch in Fragen der Gültigkeitsdauer des Status. Dabei unterscheidet man zwischen Client und Server.
  - Application: Die Daten stehen allen Clients zur Verfügung und bleiben erhalten, bis der Server das nächste Mal gestartet wird.
  - Session: Ein typisches Beispiel ist der Einkauf in einem Online-Shop. Sie wollen einerseits, dass der Server nicht vergisst, welche Waren Sie bereits in den Einkaufskorb gelegt haben. Andererseits möchten Sie Ihren Einkaufskorb nicht mit einem anderen Kunden teilen.
  - Request: Sie möchten beispielsweise alle Spielzeuge für Dreijährige sehen. Die Antwort mag sich über mehrere Seiten hinziehen, muss aber nicht bis zum Ende Ihrer Sitzung zur Verfügung stehen.
  - Page: Der Server muss sich keine Daten merken, sondern nur eine Seite liefern.

Durch die Verwendung offener Standards wird zu guter Letzt nicht einmal die Verwendung von Java als Client erzwungen. Denn XML und HTTP sind grundsätzlich in fast jeder Sprache beherrschbar.

Die Entwicklung von Java-Programmen für den Application Server ist etwas anders als in der Standard Edition. Während ein normales Programm unter der Fuchtel des Runtime Environment typischerweise über die statische Methode `main` startet, verwendet eine JEE-Anwendung den Application Server als Runtime, der irgendwann vor dem Start der Anwendung gestartet wird. Abgesehen von einer eventuellen Initialisierung startet das Programm durch das Eintreffen einer Nachricht.

Die Java-EE-Anwendungen werden dem Server als WAR- oder EAR-Dateien zur Verfügung gestellt. Man spricht von einem *Deploy*. Eine WAR- oder EAR-Datei ist also vergleichbar mit einer JAR-Datei im Zusammenhang mit dem Standard Java Runtime Environment. Während das Java Runtime Environment nur die Verbindung zum Betriebssystem schafft, übernimmt der Application Server die Netzwerkkommunikation, die Parallelabwicklung der Anwendungen und die Datenbankoperationen.

Die Java Enterprise Edition bietet einen bunten Blumenstrauß an Konzepten und Technologien. Einige bauen aufeinander auf, einige ergänzen sich gegenseitig und andere konkurrieren sogar miteinander. Im Zuge seiner Entwicklung hat die Java Enterprise Edition immer mehr hinzugelernt. Für Ihre Anwendung werden Sie darum in der Regel nur einen Teil der JEE benötigen.

Vorbereitung und Installation

Heidi will mit Klara zusammen ihre Software selbst schreiben. Klara schlägt vor, dass Heidi erst einmal zur Erholung nach Frankfurt kommt, um dort die Installation der Entwicklungsumgebung zu lernen. Anschließend wollen sie die Möglichkeiten der Java Enterprise Edition austesten.

Eclipse

Klara wirft einen Blick auf Heidis Laptop. Dort hat sie noch eine Version von Eclipse installiert, mit dem sie damals ein Ballerspiel für Peters Geburtstag entwickelt hat. Peter hatte sich riesig gefreut. Nun konnte er mit der Schleuder auf Ziegen schießen, ohne dass es dem Bestand der Herde schadete.

Leider ist diese Version von Eclipse nicht für JEE gedacht. Klara macht es sich einfach: Sie entfernt das alte Eclipse und ersetzt es durch eine Version, die auch für JEE geeignet ist. Das ist kein Problem, da Heidi auch mit der JEE-Version die Level ihres Ziegenspiels weiterentwickeln kann.

Die JEE-Version von Eclipse installieren

Eine ausführliche Beschreibung zur Installation von Eclipse finden Sie bereits in Kapitel 2. Prinzipiell ist der Ablauf identisch, Sie müssen nur darauf achten, dass Sie die JEE-Version von Eclipse erwischen.

Alle verfügbaren Eclipse-Pakete werden unter der URL `www.eclipse.org/downloads/packages` aufgeführt. Dort ist das Paket mit dem Titel JEE interessant und Sie müssen Ihre Betriebssystemversion angeben. Sie werden zum Download weitergeleitet, der eine ZIP-Datei auf Ihren PC schafft.



Sie könnten Eclipse auch erweitern. Das Zauberwort heißt WTP, das für *Webtool Platform* steht. Unter Linux können Sie einfach alle Pakete, die

eclipse und *wtp* im Namen enthalten, suchen und nachinstallieren.

Sie können auch nachrüsten, indem Sie in Eclipse den Menüpunkt Help|Install New Software aufrufen. Unter Work With klappen Sie die Klappliste auf und wählen die zu Ihrem Eclipse gehörende Update-Seite. Wenn Sie nun einen Moment innehalten, entfaltet sich eine Liste und dort können Sie den Punkt Web, XML, Java EE Und OSGI Enterprise Development auswählen. Daraufhin wird der Button Next frei, dem Sie folgen, um die Software zu installieren.

Am einfachsten ist es allerdings, Sie beschaffen sich gleich ein Eclipse, das für JEE vorbereitet ist.

Application Server

Für den Betrieb der Server-Software wird ein Application Server benötigt. Es gibt mehrere Kandidaten, die als Application Server eingesetzt werden können:

- Sehr verbreitet ist Tomcat, der sehr schlank und als Webcontainer sehr beliebt ist, aber eben nicht den vollen Umfang der JEE zur Verfügung stellt.
- Glassfish stellt die Open-Source-Referenz für JEE dar.
- Wildfly ist der neue Name des JBoss. Auch er ist ein Open-Source-Produkt.
- Oracle bietet als kommerzielles Produkt WebLogic an.
- IBM hat als kommerzielles Produkt WebSphere im Portfolio.

Installation Glassfish

Klara entscheidet sich für Glassfish. Als Referenzimplementation sollte er wenig Überraschung bieten. Glassfish lässt sich leicht in die IDE Eclipse einbinden. Darüber hinaus hat Glassfish alle weiteren Vorzüge freier Software.

Der Start von Glassfish setzt voraus, dass auf dem Computer ein aktuelles Java Development Kit installiert ist. Beispielsweise ist bei Glassfish 4 auf jeden Fall das JDK 8 erforderlich. Für neuere Versionen finden Sie die Versionshinweise auf der Homepage von Glassfish.

Download

An den Glassfish-Server gelangen Sie unter dem Tarnnamen Java EE SDK auf der Website von Oracle. Reisen Sie mit dem Browser Ihrer Wahl zu folgender URL:

`www.oracle.com/technetwork/java/javaee/downloads`

Dort laden Sie Java EE SDK herunter. Zunächst werden Sie aufgefordert, die Lizenzbedingungen zu akzeptieren, bevor Sie endlich eine ZIP-Datei namens *java_ee_sdk-version.zip* herunterladen dürfen. In diesem Paket finden Sie ein Verzeichnis namens *glassfish*.

Installation durch unzip

Die Installation beginnt damit, die ZIP-Datei zu entpacken. Aber wohin? Wenn Sie für die Entwicklung von Software Glassfish heruntergeladen haben, sollten Sie das Paket so installieren, dass der Programmierer Schreibrechte darauf hat.

- Unter Windows entpackt man es in alter Tradition auf dem Wurzelverzeichnis irgendeines Laufwerks. Dort können die Dateien später von allen Interessierten frei verändert werden.
- Unter Linux oder Mac ist ein schöner Ort das Benutzerverzeichnis des Entwicklers.
- Soll Glassfish auf einem Linux-Server laufen, ist es eine gute Idee, einen Benutzer namens `glassfish` anzulegen und das Paket in dessen Heimatverzeichnis zu entpacken.

Wo immer Sie Glassfish hingeschafft haben: Versuchen Sie sich später zu erinnern, wo es geblieben ist. Sie werden diese Information noch brauchen!

Einbindung in Eclipse

Den Glassfish-Server kann man von Eclipse aus steuern. Das macht das Entwickeln und Testen von Anwendungen deutlich einfacher. Zunächst legen Sie für Glassfish einen Server-Eintrag an. Für die Steuerung des Servers durch Eclipse benötigen Sie die Glassfish-Tools. Mit ein wenig Glück sind diese bei Ihrer Eclipse-Version vorhanden. Vermutlich müssen Sie sie allerdings erst installieren:

1. Klicken Sie im Menü auf Help|Eclipse Marketplace.
2. In dem Eingabefeld Find geben Sie das Stichwort »glassfish« ein und drücken die Taste Return.
3. Nach kurzem Warten erscheint GlassFish Tools und daneben ein Button Install, auf den Sie bitte klicken.
4. Der nächste Bildschirm zeigt Ihnen den Installationsumfang, den Sie mit Confirm bestätigen.
5. Im letzten Bildschirm müssen Sie die Lizenzbedingungen akzeptieren, bevor Sie den Button Finish anklicken können.
6. Eclipse installiert ein paar Runden vor sich hin und erklärt Ihnen dann, dass die Änderungen erst nach einem Neustart von Eclipse scharf werden und ob Sie darum einen Neustart wünschen.
7. Eclipse verschwindet, kommt aber von selbst wieder.

Damit sind die Glassfish-Tools bereit, um Ihren Glassfish zu steuern. Vorher müssen Sie aber Eclipse darüber informieren, wo Glassfish ist und wie die

Administrationskennung aussieht.

1. Aus dem Eclipse-Menü wählen SiEfile|New|Other.
2. In dem Dialog wählen Sie unter der Rubrik Server den Eintrag Server und klicken dann auf Next.
3. In nächsten Dialog finden Sie unter der Rubrik Glassfish den Eintrag Glassfish. Mit Next kommen Sie weiter.
4. Hier können zwei verschiedene Situationen entstehen:
    - Der Glassfish-Server fragt nur nach dem Domain-Pfad. Dann wählen Sie am besten über den Browse-Button rechts neben dem Eingabefeld den Unterpfad *glassfish/domains/domain1*.
    - Andernfalls wird nach der Glassfish-Installation und der zugehörigen Java-Umgebung gefragt. Hier sollen Sie den Pfad für Glassfish angeben und darunter den Pfad der Java-Umgebung, die sich um Glassfish kümmern soll. Zur Unterstützung finden Sie ebenfalls jeweils die Browse-Buttons rechts neben dem Eingabefeld. In meiner Umgebung sind die richtigen Einträge:
        - Glassfish location: */home/arnold/glassfish4/glassfish*
        - Java location: */usr/lib/jvm/jdk1.8.0_121*

    Sie können Next wählen.
5. Auf der nächsten Seite wird Administrationsbenutzer und Passwort eingestellt. Standardmäßig heißt der Administrator `admin` und hat kein Passwort.
6. Mit Finish beenden Sie die Einrichtung von Glassfish. Danach schließen Sie mit OK den Dialog der Preferences.

Sollten Sie keinen Eintrag für Glassfish unter den Servern finden, müssen Sie die Einstellungen von Eclipse korrigieren.

1. Unter Eclipse rufen Sie Window|Preferences auf.
2. Im Dialog wählen Sie in der linken Spalte Server|Runtime Environment aus.
3. Auf der rechten Seite ist die Liste der Umgebungen leer, darum klicken Sie auf Add. Jetzt sollte Ihr Eclipse ähnlich aussehen wie in Abbildung 30.1.

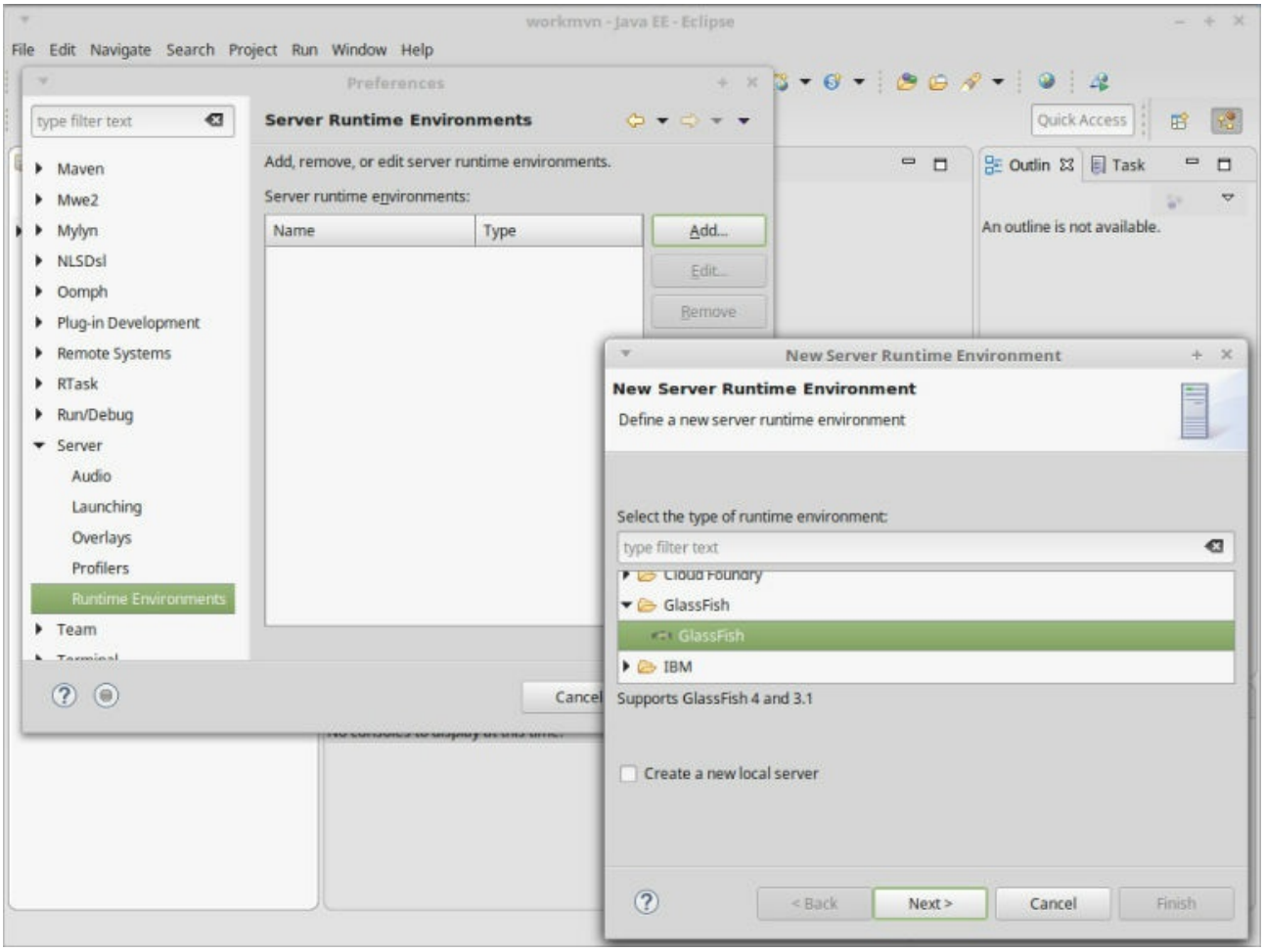


Abbildung 30.1: Glassfish-Einrichtung in Eclipse

4. In der Liste öffnen Sie Glassfish. Darunter finden Sie einen weiteren Eintrag Glassfish, den Sie selektieren. Hier müssen Sie das Verzeichnis *glassfish* unter dem Hauptverzeichnis *glassfish4* auswählen. Dann klicken Sie auf Next.
5. Nun kennt Eclipse Ihren Glassfish und Sie können den Pfad für Glassfish und darunter den Pfad der Java-Umgebung angeben. Anschließend folgt Administratorbenutzer und dessen Kennwort, wie oben beschrieben.
6. Mit Finish beenden Sie die Einrichtung von Glassfish. Danach schließen Sie mit OK den Dialog der Preferences.

Zum Test starten und stoppen wir den Glassfish von Eclipse aus. Das erreichen Sie über das Teilfenster rechts unten. Dieses besitzt eine Lasche mit der Beschriftung Server. Dieses Teilfenster können Sie jederzeit über den Menüpunkt Window|Show View|Servers nach vorn bringen. Hier tauchen alle Server auf und Sie können sie von hier aus starten oder stoppen.

1. Klicken Sie den Eintrag für Glassfish mit der rechten Maustaste an.
2. Wählen Sie Start aus dem Kontextmenü.
3. Eclipse schaltet auf die Console und zeigt Ihnen die Startmeldungen von Glassfish. Anschließend schaltet Eclipse wieder auf das Teilfenster Server und meldet dort:

Glassfish 4 [Domain1] [Started]

4. Wenn Sie die Zeile mit der rechten Maustaste anklicken, finden Sie Menüpunkte zum Stoppen des Servers.

Damit steht Glassfish auf dem lokalen Arbeitsplatzrechner für die Entwicklungsumgebung Eclipse bereit.


Eclipse, Glassfish und Glassfish-Tools müssen zusammenpassen. Beispielsweise gab es bei Erscheinen der Glassfish-Version 5 noch kein passendes Glassfish-Tools für Eclipse, das diese Version unterstützte. Da musste man so lange für die Entwicklung vorübergehend eine ältere Version von Glassfish verwenden.

Installation eines eigenständigen Glassfish-Servers

Eines Tages werden die Programme fertig sein. Das ist natürlich nur so eine Floskel der Marketingabteilung, über die Software-Entwickler immer wieder herzlich lachen können. Sollte es doch zum Äußersten kommen und die Software ist wider Erwarten fertig, soll sie auf einem extra dafür vorgesehenen Computer laufen und nicht nur in der Entwicklungsumgebung von Eclipse.

Dazu benötigen Sie einen Server-PC. Meist greift man da aus guten Grund zu Linux und installiert darauf zunächst eine Java-SDK und dann Glassfish.

1. Sie installieren das Paket, das Sie zuvor von `glassfish.java.net/download.html` heruntergeladen haben, durch den Befehl `unzip`.
2. Schaffen Sie die entpackten Dateien an einen passenden Zielort:
    - Unter Linux bietet sich das Verzeichnis */opt* an. Sie können aber auch einen eigenen Benutzer für Glassfish verwenden und dessen Verzeichnis unter */home* verwenden.
    - Bei Windows empfielt sich ein Verzeichnis auf Laufwerk *C:*.
3. Starten Sie den Glassfish-Server. Dazu benötigen Sie eine Konsole (Terminal, Eingabeaufforderung). Wechseln Sie in das Verzeichnis, in dem Glassfish liegt. Rufen Sie dort den folgenden Befehl auf:
   `glassfish4/bin/asadmin start-domain`

Nachdem der Server gestartet wurde, können Sie die Konfiguration über die folgende Adresse durchführen, die Sie in Ihrem Lieblingsbrowser aufrufen:

`localhost:4848`

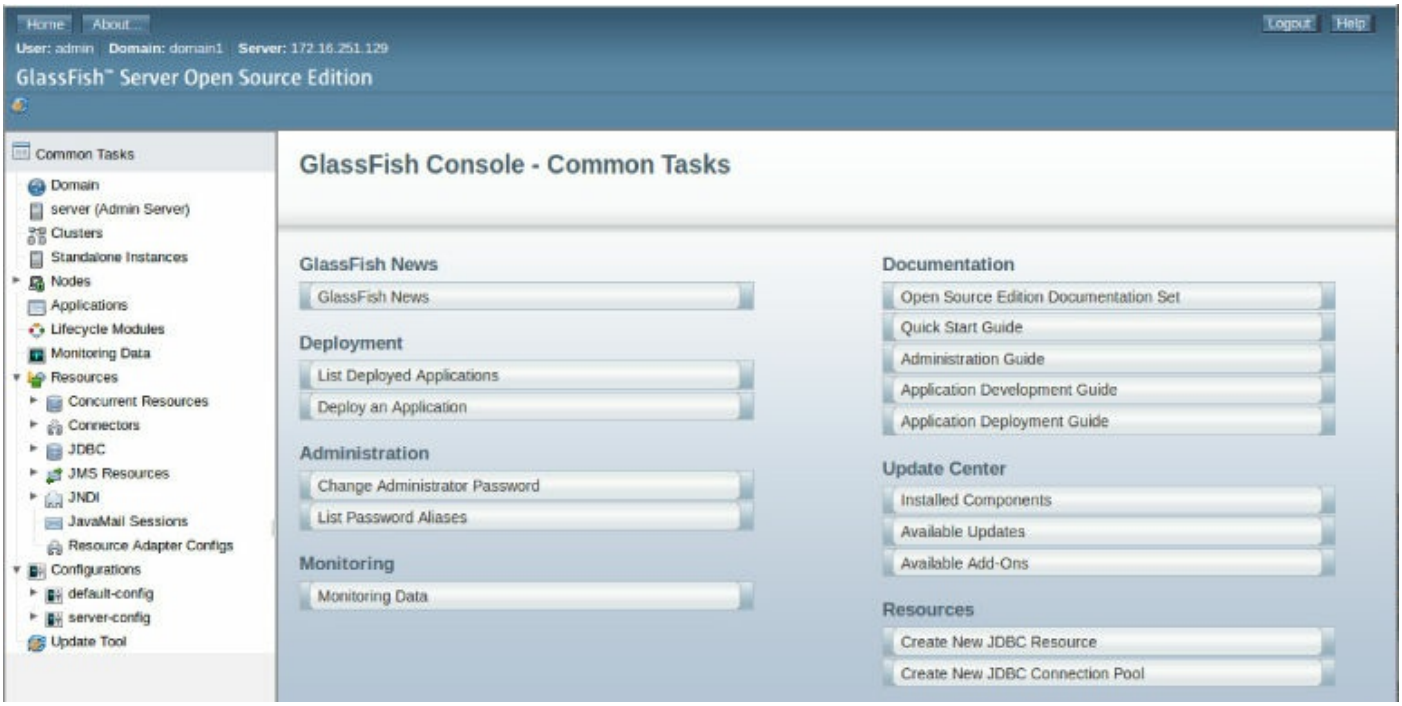


Abbildung 30.2: Glassfish-Administration per Browser

Der Hostname `localhost` bedeutet, dass Sie die Wartung auf dem Server selbst durchführen. Falls Sie für Fernwartung schwärmen, werden Sie vermutlich auf die Meldung »Configuration Error Secure Admin must be enabled to access the DAS remotely« stoßen. Diese vermeiden Sie dadurch, dass Sie folgende Schritte durchführen:

1. Von außen dürfen Sie Glassfish nur administrieren, wenn Sie ein Passwort verwenden. Standardmäßig hat der Standardbenutzer `admin` kein Passwort. Also legen Sie eines mit dem folgenden Befehl an:
   ```
   asadmin change-admin-password
   ```
2. Schalten Sie die Administration auf sicher. Zunächst wird Benutzer und das neue Passwort erbeten.
   ```
   asadmin enable-secure-admin
   ```
3. Die Umstellung auf den sicheren Zugang erfordert einen Neustart von Glassfish.
   ```
   asadmin restart-domain
   ```
4. Nun sollten Sie sich erneut per Browser auf Port 4848 anmelden. Dabei wird Ihr Browser vermutlich furchtbar husten, weil er auf ein nicht-authorisiertes Zertifikat stößt. Da müssen Sie wohl eine Ausnahme erlauben.

Application Server Tomcat

Es könnte sein, dass Sie gute Gründe haben, nicht mit Glassfish arbeiten zu wollen. Beispielsweise könnte Ihr Chef Tomcat bevorzugen. Und wer wird schon auf die Annehmlichkeiten eines monatlichen Gehalts verzichten, nur um einem Fisch den Vorzug vor einem Kater zu geben?

1. Der Webcontainer Tomcat wird auf der Website `tomcat.apache.org` angeboten.
2. Dort klicken Sie auf den Link für die aktuellste Version.
3. Auf der Seite finden Sie eine »Core«-Version.
4. Wenn Sie noch Windows verwenden, wählen Sie zwischen der 32- und der 64-Bit-Version. Alle anderen können die einfache ZIP- oder Tar.gz-Version verwenden.
5. Entpacken Sie die Datei in Ihrem Benutzerverzeichnis!

Tomcat in Eclipse

Klicken Sie auf Window|Show View|Servers, wird im unteren Teilfenster die Lasche Servers hervorgehoben. Zu Anfang ist die leer. Da soll aber gern der Tomcat auftauchen, damit Sie ihn aus der IDE heraus starten und stoppen können.

In dem leeren Feld ist immerhin eine Schrift enthalten, die sich bei genauerer Betrachtung als Link erweist. Der Link bewirkt das Gleiche wie der Aufruf über das Menü File|New|Other und die Auswahl Server|Server im folgenden Dialog.

1. Im ersten Dialog steht, welche Tomcat-Version Sie verwenden wollen. Sie müssen dazu vielleicht erst das Feld Apache in der Liste öffnen, da Tomcat zu Apache gehört. Mit Next erreichen Sie den Folgedialog.
2. Nun sollen Sie das Verzeichnis angeben, in dem Sie Tomcat installiert haben. Dazu verwenden Sie den Button Browse. Sollte Tomcat von Linux über das System installiert worden sein, liegt Tomcat typischerweise in */usr/share/tomcat*.
3. Sie können noch die passende JRE bestimmen, falls Sie eine ungewöhnliche Konstellation verwenden.

Probleme, Probleme, Probleme

Die Programmierung von Client-Server-Architekturen ist naturgemäß etwas komplexer. Der Client muss zum Server passen. Das Netzwerk sollte tatsächlich eine Verbindung darstellen. Die Entwicklungsumgebung startet mehrere Programme gleichzeitig und kontrolliert dabei auch noch einen Application Server. Und alle diese Komponenten können haarsträubende Katastrophen auslösen. Damit Sie Ihre Nerven behalten, ein paar Tipps:

- ✔ Installieren Sie Ihre Arbeitsumgebung sorgfältig! Wenn dort etwas klemmt, werden Sie Schwierigkeiten haben, ein ordentlich laufendes System zu erstellen.
- ✔ Prüfen Sie, ob alle Komponenten zusammenpassen. Gerade bei Versions-Updates kann es passieren, dass eine ältere Komponente mit der neueren nicht

arbeiten will.

- Wenn ein Programm partout nicht arbeiten will, sollten Sie auf jeden Fall den Server einmal neu starten.
- Auch der Neustart von Eclipse kann verklemmte Prozesse wieder zum Laufen bringen.
- Manchmal hilft es, für Eclipse einen neuen Workspace anzulegen.
- Wenn gar nichts hilft, ist es gut, wenn man einen zweiten Computer mit eigenständiger Entwicklungsumgebung hat, auf dem man testen kann, ob es an der Entwicklungsumgebung liegt.
- Lassen Sie sich nicht die Kommunikation von der Windows Firewall kaputt machen.

Glassfish spielt Indianer und Servlet

Der Glassfish kann wie sein roter Blutsbruder Apache dazu genutzt werden, als HTTP-Server zu arbeiten. Er kann aber nicht nur statische Seiten darstellen, sondern auch in den Dialog mit dem Client treten. Darum nennt Eclipse ein Projekt für den Application Server auch *Dynamisches Web-Projekt*.

Im ersten Schritt werden wir eine HTML-Seite im Glassfish veröffentlichen. Dazu starten wir Eclipse und erzeugen ein dynamisches Web-Projekt.

1. Klicken Sie im Menü den Punkt File|New|Project und dann Web|Dynamic Web Project an. Sie kommen weiter mit Next.
2. Nun will Eclipse einen Projektnamen, beispielsweise `Html`. Als Target Runtime sollte der Glassfish erwähnt sein. Finish erstellt das Projekt.
3. Klicken Sie mit der rechten Maustaste das Projekt an und klicken Sie auf New|Other. In dem erscheinenden Dialog wählen Sie nun Web|HTML File und dann Next.
4. Auch hier freut sich Eclipse über einen Namen. Verwenden Sie *index.html* und bestätigen Sie mit Finish. Es entsteht der Rahmen einer HTML-Seite.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

</body>
</html>
```

In die leere Zeile inmitten der `body`-Tags schreiben Sie ein paar gewichtige Worte!

5. Klicken Sie das Projekt mit der rechten Maustaste an und wählen Sie Run AS|Run ON Server. Eclipse wird Sie fragen, ob es genehm ist, den Glassfish zu benutzen, und diesen anschließend starten. Nach einer kurzen Zeit der Aktivität werden Sie einen Mini-Browser in Eclipse sehen, der ihre gewichtigen Worte anzeigt. Als URL verwendet er `localhost:8080/Html/`. Sie können diese auch Ihrem Lieblingsbrowser zuführen und erhalten das gleiche Ergebnis.

Wenn Sie Ihr Projekt entblättern, werden Sie die Datei *index.html* im Projektverzeichnis *WebContent* finden. Offensichtlich handelt es sich nicht um Java. Leider werden Sie nicht völlig umhin können, mit HTML zu spielen. Keine Sorge, es ist nicht schwer und so tief dringen wir in das Thema nicht ein. Schauen wir genauer hin:

- ✔ Tags stecken in spitzen Klammern.
- ✔ Eine HTML-Seite beginnt mit `<!DOCTYPE html>`, damit der Browser weiß, dass alles, was nun kommt, von ihm zu interpretieren ist. Dies werden wir noch benötigen.
- ✔ Ein Tag wird meist durch das gleichen Tag geschlossen, das dann allerdings einen Schrägstrich voranstellt.
- ✔ Zwischen `<head>` und `</head>` stehen Meta-Informationen wie hier die Zeichencodierung in UTF-8.
- ✔ Zwischen `<body>` und `</body` steht der sichtbare Inhalt einer HTML-Seite.

Derartige Seiten holt sich ein Browser wie Firefox von einem Webserver wie Apache. Dabei wird immer wieder der HTTP-Befehl GET verwendet. Enthält die Webseite ein Bild, wird dieses in einem weiteren Aufruf geholt und auch dort wird wieder GET ausgelöst und als Parameter der Dateiname des Bildes übergeben.

Ein HTML-Formular

Wir gehen einen Schritt weiter und erstellen ein Formular, das der Benutzer ausfüllen kann und dessen Inhalt an den Webserver gesendet wird. Der Apache holt sich in solchen Fällen Hilfe von Plug-ins oder CGI-Skriptsprachen.

Glassfish ist da cool. Er ist ein Application Server, der Java-Programme verwaltet und solche Benutzereingaben verarbeiten kann. Probieren wir es aus. Zunächst müssen wir das HTML-Dokument so erweitern, dass es ein Formular bekommt. Wir bauen eines mit zwei Eingabefeldern und einem Submit-Button:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
</head>
<body>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
</head>
<body>
  <form action="GetForm" method="post">
    <table>
      <tr>
        <td>Name</td>
        <td><input type=text name=name value="" size=30 /></td>
      </tr>
      <tr>
        <td>E-Mail</td>
        <td><input type=text name=email value="" size=30 /></td>
      </tr>
      <tr>
        <td><input type=submit value="Einfügen" /></td>
        <td></td>
      </tr>
     </table>
  </form>
</body>
</html></body>
</html>
```

Listing 30.1: HTML-Seite mit Formular

Dieses HTML-Listing besitzt ein paar neue Elemente, die eine Eingabemaske möglich machen. Es treten an:

- `form`: Dieses Tag leitet ein Formular ein. Alle Eingaben müssen von irgendwem bearbeitet werden. Wer das ist, steht im Attribut `action`. Das Attribut `method` gibt das HTTP-Kommando an. Außer `post` gibt es noch `get`.

  Im Formular enthalten sind die Eingabeelemente, von denen hier zwei benötigt werden:

  - `input type=text`: Damit entspannt sich ein Eingabefeld mit so vielen Zeichen Länge, wie im Attribut `size` angegeben ist. Sehr viel wichtiger ist das Attribut `name`, weil auf den Inhalt des Eingabeelements über diesen Namen zugegriffen werden kann. Was im Attribut `value` hinterlegt ist, wird dem Anwender als Vorgabe eingeblendet.
  - `input type=submit`: Dieses Eingabeelement ist ein Button, der dazu führt, dass die Seite beendet, die Seite des Formularattributs `action` aufgerufen und der Inhalt der Eingabemaske übergeben wird.
- `table`: Mit diesem Tag wird eine Tabelle angelegt. Damit strukturieren Sie die Eingabefelder so, dass sie nicht einfach nur untereinander auf dem Bildschirm herumliegen.

  Technisch ist das nicht erforderlich. Aber die Seite muss ja nicht immer so aussehen, als hätte sie ein Programmierer gestaltet.

  - `tr`: Innerhalb der Tabelle trennt das Tag `tr` die Zeilen.
  - `td`: Innerhalb der Zeilen trennt das Tag `td` die Spalten.

Für den Submit-Button ist die Aktion `GetForm` hinterlegt, die durch ein Servlet hinterlegt werden kann. Derzeit gibt es das Servlet noch nicht und der Server würde die bekannte HTTP-Fehlermeldung 404 ausgeben. Um aber wenigstens den Submit-Button sichtbar zu machen, müssen Sie zunächst die HTML-Seite einmal starten.

1. Klicken Sie neben Ihrem Projekt das kleine Dreieck so an, dass es nach unten zeigt. Darunter werden verschiedene Verzeichnisse sichtbar.
2. Sie sehen nun das Verzeichnis *WebContent*. Klicken Sie auch hier auf das Dreieck, damit es nach unten zeigt.
3. Hier finden Sie die angelegte HTML-Datei.
4. Klicken Sie die HTML-Datei mit der rechten Maustaste an. Es erscheint ein Menü.
5. Klicken Sie den Punkt Run AS|Run ON Server an.
6. In der Dialogbox wird der Server angezeigt. Klicken Sie auf Finish.
7. Eventuell erscheint der Hinweis, dass der Server neu gestartet werden muss. Das ist kein Drama.

8. Es erscheint die Webseite aus Abbildung 30.3.

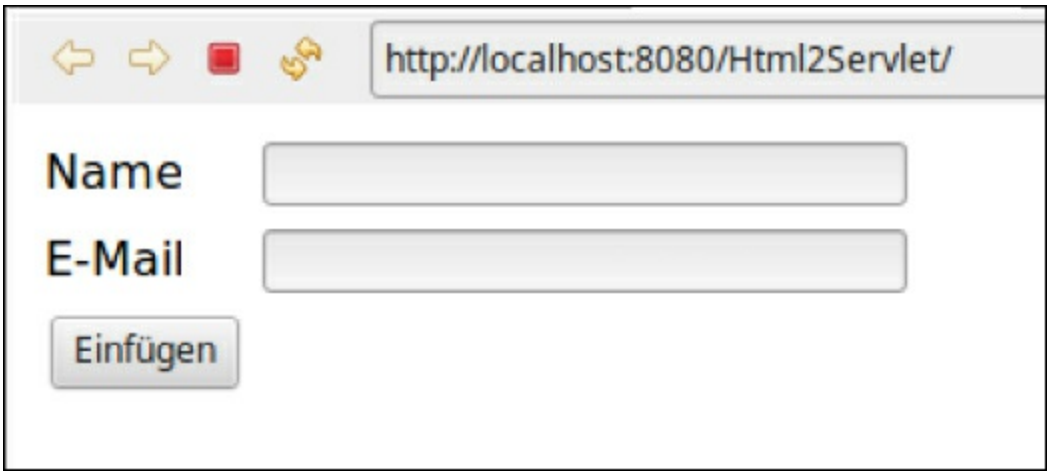


Abbildung 30.3: Das Eingabeformular

Ein Servlet verarbeitet Formulare

Nun kommt der Aspekt, warum ein dynamisches Web-Protokoll so dynamisch ist. Wir erstellen ein Servlet, das über das `action`-Attribut des HTML-Formulars aufgerufen wird.

Beim Bau eines Servlets in einem dynamischen Web-Projekt ist Eclipse außergewöhnlich hilfsbereit.

1. Klicken Sie das Projekt mit der rechten Maustaste an und wählen Sie unter dem Menüpunkt New|Other und im Folgedialog den Punkt Web|Servlet. Mit Next kommen Sie weiter.
2. Das neue Servlet muss einen Class-Namen erhalten. Wenn Sie es `GetForm` nennen, wird es auf die `action` des Formulars passen.
3. Nach einem Next sehen Sie, wie Sie die URL des Servlets anpassen können.
4. Nach einem weiteren Klick auf Next landen Sie in einem Dialog, der anzeigt, welche Methoden überschrieben werden. Darunter befinden sich die HTTP-Kommandos `doGet`, `doPost`, `doPut` und `doDelete`.
5. Sie schließen den Dialog mit Finish ab!

Freundlicherweise erstellt Eclipse einen Programmrahmen für das Servlet. Sie müssen es nur noch Ihren Wünschen gemäß ergänzen.

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/GetForm")
public class GetForm extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public GetForm() {
        super();

    }

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
            throws ServletException, IOException {

    }

    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
            throws ServletException, IOException {
        doGet(request, response);
    }
}
```

Listing 30.2: Ein Servlet

Mit der Annotation `@WebServlet` wird die URL festgelegt, unter der das Servlet von außen erreichbar ist.

Die Methode `doPost` tut nichts anderes, als die Aktion auf `doGet` umzuleiten. Wenn Sie `doGet` also mit Leben füllen, reagieren Sie auf jeden Fall auf die Aktion des Formulars.

Die Methoden `doGet` und `doPost` haben jeweils zwei Parameter:

- `HttpServletRequest request`: Über diese Variable kann die Methode auf Informationen der Anfrage zurückgreifen. Auf diese Weise greift das Servlet auf den Inhalt der Eingabeelemente des Formulars zu. Speziell dazu gibt es die folgende Methoden:

- `getParameter`: Als Parameter wird der Name des Formularelements übergeben. Der Rückgabewert liefert dessen Inhalt.
- `getParameterValues`: Diese Methode hilft, wenn ein Formularelement mehrere Werte hat, wie beispielsweise Checkboxen.
- `getParameterNames`: Mit diesem Aufruf erhält man eine Liste der Formularelemente in Form einer `Enumeration` von Strings.

✔ `HttpServletResponse response`: Dies ist die Referenz auf die Antwortstruktur.

- Über `getWriter` kann die Ausgabe des Servlets an den Aufrufer, hier den Browser, gesendet werden.
- Mit `sendRedirect` leiten Sie zu einer anderen Seite um.

Die Ausgabe des Servlets erfolgt über die Variable `response`. Sie enthält einen `Writer`, der über `println` Text ausgeben kann. Dieser Text geht zurück an den Browser des Aufrufers und ist darum idealerweise HTML-Code.

Anzeige der Formularinhalte

In Listing 30.3 erzeugt die veränderte Methode `doGet` als Ausgabe HTML-Code, der in einer Tabelle alle Eingabeelemente des Formulars mit seinen Inhalten anzeigt.

```
protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

    response.getWriter().println("<!DOCTYPE html>\n<html>" +
            "<body><table>");
    // Hole alle Parameterbezeichner
    Enumeration<String> controls = request.getParameterNames();
    while (controls.hasMoreElements()) {
        String controlName = controls.nextElement();
        // Zeige den Parameterbezeichner
        response.getWriter().println("<tr><td>" + controlName);
        // Zeige den eingegeben Wert
        response.getWriter().println("</td><td>"
                + request.getParameter(controlName)
                + "</td></tr>");
    }
    response.getWriter().println("</table>");
}
```

Listing 30.3: Die Servlet-Methode zeigt alle Parameter.

Direkter Zugriff auf die Eingabeelemente

Wenn Sie aber schon wissen, wie die Eingabeelemente heißen, können Sie mit der Methode `getParameter` auch deren Inhalte erreichen. Im folgenden Beispiel wird auf den Inhalt des Texteingabefelds mit dem Namen `name` zugegriffen.

```
protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

    response.getWriter().println("<!DOCTYPE html>\n<html><body>" +
            "Der Name ist: " + request.getParameter("name") +
            "<p/><a href=\"index.html\">zur&uuml;ck</a>" +
            "</body></html>");
}
```

Listing 30.4: Das Servlet liest den Inhalt des Eingabeelements.

In der Ausgabe entdecken Sie vielleicht noch das Tag `a`. In Kombination mit dem Attribut `href` können Sie damit einen Link basteln. Dieser führt direkt zu der Startdatei *index.html* zurück und damit schließt sich der Kreis.

Um auf eine andere Adresse umzuleiten, können Sie allerdings die Methode `sendRedirect` verwenden, die über den Response-Parameter aufgerufen wird. Als Parameter übergeben Sie die Ziel-URL.

```
response.setRedirect("index.html");
```



Wollen Sie eine dynamische Webseite bauen, die Benutzereingaben entgegennimmt, benötigen Sie Glassfish und ein paar Servlets, die von den HTML-Formularen per `action` aufgerufen werden. Die Servlets sprechen mit dem Client-Browser, indem sie HTML-Code durch die `response`-Variable schicken.

Vom Servlet aus können Sie die Daten in Datenbanken packen oder andere Programmteile aufrufen. Sie sitzen also mitten im Server und haben die komplette Kontrolle.

Heidi ist schon recht angetan von den Servlets. Sie kann für einfachere Dinge eine HTML-Seite stricken und die zentrale Arbeit auf dem Server über Servlets erledigen. Und letztlich spricht nichts dagegen, von einem Java-Client über eine `HttpURLConnection` den Kontakt zum Server aufzunehmen. Auf diese Weise lassen sich auch Smartphones anbinden.

Sie schlägt vor, das Almhüttenprojekt komplett auf der Basis von Servlets zu bauen. Klara bestätigt, dass das möglich wäre, empfielt aber, zunächst noch einen Blick auf die weiteren Möglichkeiten der JEE zu werfen.

Die Struktur des dynamischen Web-Projekts

Schauen wir noch einmal genauer auf die Inhalte des dynamischen Web-Projekts:

- *src:* Hier finden Sie wie bei einem normalen Java-Projekt die Quelltexte Ihrer Servlets, schön in Packages gegliedert.
- *WebContent:* In diesem Verzeichnis finden Sie all das, was sich später in der WAR-Datei wiederfinden wird. Wenn Sie eigene Bibliotheken in Ihr Projekt einbringen wollen, sollten Sie sich in diesem Verzeichnis befinden.

  Hier werden auch die HTML-Dateien abgelegt. Sie finden darum auch *index.html* an dieser Stelle.

  - META-INF: Hier befindet sich beispielsweise die Datei *MANIFEST.MF*.
  - WEB-INF: Hier ist ein Ablageort für diverse Konfigurationsdateien.

Ein Java-Client sendet Formularinhalte

Bisher wurde für die Datenerfassung ein HTML-Formular eingesetzt. Das ist durchaus praktisch, weil damit schnell eine Oberfläche zu bauen ist. Aber die HTTP-Schnittstelle steht nicht nur Browsern offen. Auch ein normales Java-Programm kann heucheln, ein Browser zu sein, wenn es die Klasse `HttpURLConnection` ausgiebig benutzt. Die Puritaner unter uns können natürlich auch das komplette HTTP mit einem Socket-Client abbilden.

```
package client;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.net.HttpURLConnection;
import java.net.URL;

public class HttpGastClient {

    public static HttpURLConnection connectGast(String query)
            throws IOException {
        String url = "http://localhost:8080/Gast/";
        String charset = "UTF-8";
        HttpURLConnection connection = null;
        connection = (HttpURLConnection)
                new URL(url + query).openConnection();
        connection.setRequestProperty("Accept-Charset", charset);
        connection.setDoOutput(true);
        connection.setDoInput(true);
        connection.setUseCaches(false);
        return connection;
    }

    public static void createGast(model.Gast gast) throws IOException
 {
        // Gast in einen String umwandeln
        String str = "";
        str += "name=" + gast.getName();
        str += "&adresse=" + gast.getAdresse();
        str += "&ort=" + gast.getOrt();
        str += "&email=" + gast.getEmail();
        str += "&telefon=" + gast.getTelefon();
        String query = "ctrlNeuGast";
        HttpURLConnection connection = connectGast(query);
        connection.setRequestMethod("POST");
        OutputStream os = connection.getOutputStream();
        OutputStreamWriter osw = new OutputStreamWriter(os, "UTF-8");
        osw.write(str);
        osw.flush();
        osw.close();
        // Hole die Antwort ab
        StringBuilder stringBuilder = new StringBuilder();
        if (connection.getResponseCode()==HttpURLConnection.HTTP_OK){
            InputStreamReader streamReader = new
                InputStreamReader(connection.getInputStream());
```

```
            BufferedReader bufferedReader = new
                BufferedReader(streamReader);
            String response = null;
            while ((response=bufferedReader.readLine())!=null) {
                stringBuilder.append(response + "\n");
            }
            bufferedReader.close();
        }
    }

    public static void main(String[] args) throws IOException {
        model.Gast gast = new model.Gast();
        gast.setName("Http Client");
        gast.setAdresse("Heimweg 12");
        gast.setOrt("Bad Humbug");
        gast.setEmail("gast@heidi.alm");
        gast.setTelefon("06172-007");
        createGast(gast);
    }
}
```

Listing 30.5: Java-Anwendung für einen HTTP-Zugriff

Das Programm sendet eine POST-Anfrage an den Application Server und gibt als Parameter eine Liste aus, wie sie auch die Eingabeelemente eines HTML-Forms erzeugen würde. Das Servlet kann also gar nicht unterscheiden, ob es sich um einen Browser oder ein Java-Programm handelt. Es sendet eine Antwort nach dem HTTP, das von der Anwendung ausgewertet werden kann.

Das Senden von Befehlen ist nicht weiter schwierig. Komplizierter würde eine Auswertung, weil das Java-Programm die Daten aus der HTML-Tabelle herauspulen müsste. Keine schöne Arbeit. Für den reinen Datentransfer ist HTML also nicht ideal. Aber XML ist recht ähnlich und könnte doch statt HTML zum Senden und Empfangen der Nachrichten verwendet werden. Diesen Gedanken greift SOAP auf.

Java Server Pages

Heidi hat schon einmal angefangen, einen Prototyp aus HTML und Servlets zu bauen. »Es ist total beknackt, den HTML-Code in Java-Strings zu verpacken!«, beklagt sie sich bei Klara. Klara bestätigt, dass das schnell unübersichtlich wird, wenn viel HTML zu codieren ist. Als Lösung schlägt sie Java Server Pages vor. Da läuft es andersherum.



Java Server Pages bestehen aus HTML-Code, in den Java-Code eingebettet

wird. Die Einbettung beginnt mit `<%` und endet mit `%>`. Damit die Seite Kontakt zum Application Server bekommt, hat sie Zugriff auf die Parametervariablen eines Servlets.

Versuchen wir, mit JSP die Forumularanfrage aus Listing 30.1 auszuwerten. Dazu wird im Listing die `action` auf `JspForm.jsp` umgestellt. Anschließend lassen wir Eclipse eine JSP-Datei diesen Namens erzeugen.

1. Sie klicken das Projekt mit der rechten Maustaste an und wählen New|Other. Unter Web wählen Sie JSP File und klicken dann auf Next.
2. Geben Sie als Dateinamen `JspForm.jsp` ein und klicken Sie auf Finish.

Wieder spendiert Ihnen Eclipse einen Programmrahmen, der sich aber kaum von einem normalen HTML-Dokument unterscheidet. Die neu entstandene Datei wird im Projekt neben die HTML-Seiten im Verzeichnis *WebContent* abgelegt.

Bereits die erste Zeile der JSP-Datei deutet darauf hin, dass die Datei etwas mit Java zu tun hat. Und auch die Einbettungszeichen sind zu finden. Das @-Zeichen zeigt an, dass es sich um eine Direktive handelt.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
```

Nun wird der HTML-Code, der vom Servlet in Listing 30.4 per Ausgabe an den Empfänger gesendet wird, eingebaut und durch eine kleine Tabelle etwas hübscher gemacht.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
    charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
    <%
        String name = request.getParameter("name");
        String email = request.getParameter("email");
    %>
    <table>
        <tr>
            <td>Name:</td>
            <td>
                <% out.print(name); %>
            </td>
        </tr>
        <tr>
            <td>E-Mail:</td>
            <td>
                <% out.print(email); %>
            </td>
        </tr>
    </table>
</body>
</html>
```

Listing 30.6: Einfache Java Server Page zur Ausgabe der Parameter

Am Anfang des Listings sehen Sie ein Java-Schnipsel, das über die Variable `request` die Methode `getParameter` aufruft. Die Variable `request` entspricht wie die Variable `response` dem Methodenparameter eines Servlets. Beide stehen der JSP an jeder Stelle zur Verfügung. Das Ergebnis wird in einer Java-Variablen abgelegt, die auch im nächsten Java-Schnipsel zur Verfügung steht.

Im Folgenden wird eine HTML-Tabelle aufgebaut. Der Inhalt der Java-Variablen wird über die Variable `out` und deren Methode `print` ausgegeben. Die Variable `out` ist vom Typ `Writer` und vordefiniert.


Um ganz ehrlich zu sein: Java Server Pages sind eigentlich Servlets. Wird

eine JSP-Datei auf einem Application Server geladen, fängt dieser sofort an, diese in ein Servlet mit ganz vielen `println`-Zeilen zu übersetzen.

Ein Eclipse-Projekt für Gäste

Heidi findet die Java Server Pages ganz toll. Sie baut sofort eine HTML-Seite mit einem Eingabeformular für die Feriengäste und eine Java Server Page, die die Daten der Gäste wieder zurückgibt.

Dazu bläst sie die HTML-Datei *index.html* ein wenig auf, um auch die anderen Attribute eines Gastes zu erfassen. Die wichtigsten Informationen sind Name, Adresse, Ort, E-Mail und Telefonnummer.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Gast eingeben</title>
</head>
<body>
    <h1>Eingabe für einen Gast</h1>
    <form action="gastneu.jsp" method="post">
        <table>
            <tr>
                <td>Name</td>
                <td><input type="text" name="name"></td>
            </tr>
            <tr>
                <td>Adresse</td>
                <td><input type="text" name="adresse"></td>
            </tr>
            <tr>
                <td>Ort</td>
                <td><input type="text" name="ort"></td>
            </tr>
            <tr>
                <td>E-Mail</td>
                <td><input type="text" name="email"></td>
            </tr>
            <tr>
                <td>Telefon</td>
                <td><input type="text" name="telefon"></td>
            </tr>
            <tr><td></td>
                <td><input type="submit"></td>
            </tr>
        </table>
    </form>
</body>
</html>
```

Listing 30.7: HTML-Formular zum Erfassen eines Gastes

Das Formular verwendet für die Action die JSP-Datei *gastneu.jsp*.



Im Formular muss bei Aufruf von Java Server Pages als `action` immer die Dateiendung *.jsp* angefügt werden.



Abbildung 30.4: Eingabe der Gastdaten

Für die Erstellung der JSP-Datei lässt sich Heidi von Eclipse helfen. Eclipse erstellt den Rahmen für die Java Server Page. Mit wenigen Handgriffen hat Heidi eine kleine Anwendung daraus gebaut.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
    <h2>Daten erfasst!</h2>
    <%
    model.Gast gast = new model.Gast();
    gast.setName(request.getParameter("name"));
    gast.setAdresse(request.getParameter("adresse"));
    gast.setOrt(request.getParameter("ort"));
    gast.setEmail(request.getParameter("email"));
    gast.setTelefon(request.getParameter("telefon"));
    %>
    <p/>
    <a href="index.html">zur&uuml;ck</a>
</body>
</html>
```

Listing 30.8: Java Server Page erfasst den Gast.

Die Daten werden in einem Objekt der Klasse `Gast` abgelegt. Die Klasse benötigt eine eigene Datei, die Sie in gewohnter Weise über New|Class erzeugen können. Bei dynamischen Web-Projekten ist es allerdings besonders sinnvoll, Packages zu definieren. Hier wurde das Package `model` genannt, weil es sich um das Datenmodell für den Gast handelt.

```
package model;

public class Gast {
    private String name;
    private String adresse;
    private String ort;
    private String email;
    private String telefon;
}
```

Listing 30.9: Eine Klasse für den Gast

Anschließend lassen Sie Eclipse über einen rechten Mausklick mit Source|Generate Getters And Setters den Rest des Codes ergänzen.

Ausgabe aller Kontrollelemente mit Inhalt

Mit einem Servlet haben wir in Listing 30.3 bereits alle Kontrollelemente mit dem Inhalt des aufrufenden Formulars angezeigt. Das geht mit Java Server Pages natürlich auch. Dazu holen Sie sich über die Methode `getParameterNames` alle Eingabeelementbezeichner. Damit es ordentlich aussieht, bauen Sie mit `table` eine kleine HTML-Tabelle.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ page import="java.util.Enumeration" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
      charset=ISO-8859-1">
</head>
<body>
<table>
<%
Enumeration<String> controls = request.getParameterNames();
while (controls.hasMoreElements()) {
    String controlName = controls.nextElement();
%>
    <tr><td>
    <% out.println(controlName); %>
    </td><td>
    <% out.println(request.getParameter(controlName)); %>
    </td></tr>
<%
}
%>
</table>
</body>
</html>
```

Listing 30.10: Auflistung aller Parameter durch Java Server Page

In der zweiten Zeile wird der Import für die `Enumeration`-Klasse ausgeführt. Dazu verwendet JSP eine `page`-Direktive `import`. Sie haben die `page`-Direktive vermutlich schon jeweils in jeder ersten Zeile der JSP-Dateien entdeckt. Damit können Optionen für die JSP-Datei gesetzt werden. Zu den relevanten Optionen gehören:

- ✔ `<%@ page language="Java" %>`
- ✔ `<%@ page contentType="text/html" %>`
- ✔ `<%@ page pageEncoding="UTF-8" %>`

- `<%@ page import="java.util.Enumeration" %>`

Wiederkehrende JSP-Dateien einbinden

Sehr praktisch ist die `page`-Direktive `include`, um gleiche Inhalte, wie Kopf- und Fußzeilen oder Menüs, in mehrere Seiten einzubinden.

```
<%@ page include="menue.jspf" %>
```

Hier wird die Datei *menue.jspf* in den Text eingebettet. Eingebettete Dateien sollten die Endung *.jspf* tragen.

Model View Controller

JSP-Dateien eignen sich ideal, um die Visualisierung zu übernehmen, dabei aber immer noch Java-Fragmente im HTML-Code unterzubringen. Das reine Servlet bleibt jedoch übersichtlicher, wenn es in erster Linie Java-Code enthält, sich also nicht in eingebetteten HTML-Code ergießt.

In Projekten wird gern eine Model-View-Controller-Architektur verwendet.

- View: Die Darstellung gegenüber dem Anwender kann gut in HTML oder JSP erfolgen. Die View wird deswegen gern separat gehalten, weil es immer wieder gute Gründe gibt, sie anzupassen. So wird die Ansicht im Browser per HTML oder JSP eben ganz anders realisiert als in einer Android-App oder auf einem Laptop-Client, der vielleicht mit Swing (siehe Kapitel 16) oder doch mit JavaFX (siehe Kapitel 20) arbeitet.
- Model: Eine Applikation steht auf Daten, die es zu verwalten gibt. Idealerweise sollte beim Austausch der Datenquelle das restliche Programm nicht berührt werden. Mal wird auf Daten über das Netzwerk zugegriffen oder doch aus einer lokalen Datei oder einer Datenbank. Eine Änderung sollte sich auf diese Schicht begrenzen.
- Controller: Die Daten und die Ansicht müssen irgendwie verbunden werden. Der Controller übernimmt die Funktion in beide Richtungen. Manchmal sind die Controller nicht explizit in eigene Module separiert.

In das Projekt soll ein Servlet als Controller zwischen der Maske und der JSP eingebunden werden. Dieser Controller überprüft in diesem Fall die Daten. Er könnte aber auch Kontakt zum Model aufnehmen und dafür sorgen, dass der eingegebene Gast gespeichert wird.

1. Legen Sie ein neues Servlet `ctrlNeuGast` an, indem Sie das Projekt mit der rechten Maustaste anklicken und New|Servlet auswählen.
2. Verbiegen Sie in der Datei *index.html* den Eintrag der Action auf das neue Servlet, indem Sie `action="ctrlNeuGast"` im Formular hinterlegen.

Ändern Sie die Methode `doPost` des neuen Servlets durch folgenden Inhalt:

```
protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
    String name = request.getParameter("name");
    if (!name.isEmpty()) {
        // Eingabe gut! Weiterverarbeiten!
        RequestDispatcher dispatch =
                request.getRequestDispatcher("gastneu.jsp");
        dispatch.forward(request, response);
    } else {
        request.getRequestDispatcher("index.html")
                .forward(request, response);
    }
}
```

Listing 30.11: Änderungen an der Methode doPost

Es ändert sich scheinbar nichts. Tatsächlich aber wird das Servlet als Zwischenschritt verwendet und leitet bei fehlendem Namen auf das Eingabeformular zurück.

Java Server Faces

Klara erwähnt, dass es als Weiterentwicklung der Java Server Pages inzwischen die ähnlich klingenden Java Server Faces gibt.

Die Java Server Faces trennen HTML und Java entschiedener. Während Java Server Pages als in HTML eingebettete Java-Schnipsel auftreten, gibt es das Facelet, das in XHTML für die Darstellung zuständig ist, und die Backing Bean in Java, auf die vom Facelet über die Attribute und Methoden direkt zugegriffen werden kann.

Ein Kleinstrechner als Beispiel

Um diese Kommunikation zwischen Facelet und Backing Bean zu demonstrieren, schreiben wir ein Programm, das zwei Zahlen miteinander multipliziert.

1. Legen Sie ein Dynamic Web Project an, indem Sie File|New|Dynamic Web Project anklicken.
2. Auf der ersten Seite des Dialogs klicken Sie unter Configuration den Button Modify an.
3. In dem Dialog können Sie die Unterstützung für Java Server Faces anwählen und dann mit OK bestätigen.

4. Das Projekt benötigt natürlich einen Namen. Anschließend kann es mit Finish erzeugt werden.

Die Backing Bean

Die Backing Bean wird als neue Klasse `CalcBackBean` hinzugefügt. Die Bean stellt die Variablen `a`, `b` und `c` zur Verfügung. Hinzu kommt eine Methode `calc`, die die Inhalte von `a` und `b` summiert und das Ergebnis in `c` ablegt.

```
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;
@Named
@RequestScoped
public class CalcBackBean {
    private int a;
    private int b;
    private int summe;
    public void setA(int a) {
        this.a = a;
    }
    public int getA() {
        return a;
    }
    public void setB(int b) {
        this.b = b;
    }
    public int getB() {
        return b;
    }
    public int getSumme() {
        return summe;
    }

    public String calc() {
        summe = a + b;
        return "/index.xhtml";
    }
}
```

Sie finden in der Klasse einige Annotationen. Mit der Annotation `@Named` wird gekennzeichnet, dass die Backed Bean über ihren Namen per CDI (Contexts and Dependency Injection) an das Facelet gebunden wird.

Für die Gültigkeit der Backing Bean gibt es die folgenden Annotationen:

- ✔ `@RequestScoped`: Für eine Anfrage
- ✔ `@SessionScoped`: Für eine Sitzung
- ✔ `@ApplicationScoped`: Über die Lebensdauer der Anwendung

Das Facelet kann die Attribute der Bean lesen und schreiben, sofern die entsprechenden Getter und Setter vorhanden sind. Die Bindung erfolgt zur Laufzeit per CDI, also über den Namen.

- ✔ Attribute der Backing Bean müssen `private` definiert sein.
- ✔ Für das Anzeigen durch das Facelet muss eine `public` Getter-Methode angelegt werden.
- ✔ Für ein Eingabeelement eines Facelets muss sowohl eine Getter- als auch eine Setter-Methode vorhanden sein.

Die Methode der Bean wird durch einen Button aufgerufen. Die Methode gibt einen String zurück, der für die Navigation zur Folgeseite verwendet wird. Die Methode hat keine Parameter.

Das Facelet

Ein Facelet ist ein XHTML-Dokument und hat darum die Dateierweiterung *.xhtml*. Es wird aber innerhalb des Projekts über die Erweiterung *.jsf* aufgerufen.

Anlegen eines Facelets

Das Facelet wird in Eclipse als XHTML-Datei erstellt:

1. Klicken Sie das Projekt mit der rechten Maustaste an und wählen Sie New|HTML-File.
2. Vergeben Sie einen Namen, beispielsweise *index.xhtml*. Die nächste Seite erscheint durch einen Klick auf Next.
3. Aus den Templates wählen Sie New XHTML File (Transitional).
4. Schließen Sie mit Finish.

Eclipse legt unter *WebContent* einen Dokumentrahmen vor, der aber noch bearbeitet werden muss:

1. Das `html`-Tag erhält ein weiteres Attribut:
   ```
   xmlns:h="http://xmlns.jcp.org/jsf/html"
   ```
   Durch diese Referenz werden die Tags der Art < h:xxxx > definiert.
2. Der gesamte Bereich von Head und Body wird entfernt.

Nun bauen Sie den Inhalt auf. Wir benötigen eine kleine Tabelle, in der die

Ausgangswerte und später die Summe steht, und einen Button, der die Kalkulation auslöst. Als Tabellenrahmen wird ein Panelgrid verwendet.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head>
    <title><h:outputText value="Wir addieren" /></title>
</h:head>
<h:body>
    <h:form>
        <h:panelGrid columns="2">
            <h:outputText value="A = " />
            <h:inputText value="#{calcBackBean.a}">
            </h:inputText>
            <h:outputText value="B = " />
            <h:inputText value="#{calcBackBean.b}">
            </h:inputText>
            <h:outputText value="Summe = " />
            <h:outputText value="#{calcBackBean.summe}" />
            <h:outputText value="Los geht's" />
            <h:commandButton value="OK"
                    action="#{calcBackBean.calc}" />
        </h:panelGrid>
    </h:form>
</h:body>
</html>
```

Listing 30.12: Facelet zum Summieren zweier Zahlen

Um das Facelet zu starten, klicken Sie es mit der rechten Maustaste an und wählen Run AS|Run ON Server. Den Dialog bestätigen Sie mit Finish.

Die Facelet-Elemente

Die bekannten HTML-Tags werden bei Facelets durch eigene Definitionen ersetzt. Auf diese Weise lässt sich die Backing Bean leichter einbinden. Anhand des Listings 30.12 haben Sie die Unterschiede vermutlich bereits festmachen können.

- `<h:head>` bis `</h:head>`: Bildet den Head-Bereich der Seite.
- `<h:body>` bis `</h:body>`: Bildet den Body der Seite.
- `<h:form>` bis `</h:form>`: Definiert ein Fomular in JSF. Im Gegensatz zu HTML-Formularen wird hier weder die Aktion noch die Methode hinterlegt.

- `<h:panelGrid>` bis `</h:panelGrid>`: Definiert eine Ausrichtungstabelle (HTML table). Das Attribut `columns` legt die Anzahl der Spalten fest. Die Inhalte werden dann hintereinander eingefügt und auf die Spalten verteilt.
- `<h:commandLink action="antwort.xhtml" value="Antwort" />`: Definiert einenLink auf *antwort.xhtml*, der mit Antwort beschriftet ist.

Innerhalb der Elemente erfolgt der Zugriff auf die Elemente der Backing Bean.

1. Die Referenz wird mit einem Doppelkreuz (`#`) eingeleitet, dem eine linke geschweifte Klammer (`{`) folgt.
2. Es folgt der Name der Backing Bean. Dabei wird der erste Buchstabe kleingeschrieben.
3. Durch einen Punkt abgetrennt folgt der Name des Attributs oder der Methode.
4. Zum Schluss wird die geschweifte Klammer (`}`) geschlossen.

Die wichtigsten Kontrollelemente des Facelets:

- `<h:inputText value="#{bean.attribut}" />`: Text-Eingabefeld, das seinen Inhalt an das Attribut der Backing Bean übergibt
- `<h:inputTextArea value="#{bean.attribut}" />`: Mehrzeiliges Text-Eingabfeld
- `<h:outputText value="#{bean.attribut}" />`: Anzeigeelement, das lesend auf das Attribut der Backing Bean zurückgreift
- `<h:commandButton action="#{bean.methode}" />`: Button, der bei Auslösung die Methode der Backing Bean aufrufen wird
- `<h:selectBooleanCheckbox value="#{bean.attribut}" />`: Checkbox
- `<h:dataTable value="#{bean.attribut}" />`: Tabelle mit Daten
- `<h:selectOneMenu value="#{bean.attribut}" />`: Dropbox, also eine aufklappbare Liste

Daten-Beans

Bei komplexeren Daten werden Sie Klassen bilden wollen. Diese Daten-Beans werden in die Backing Beans per Injection eingebettet. Als Beispiel wird eine Backing Bean verwendet, die `KundenCtrl` heißt. Sie besitzt ein Datenobjekt der Klasse `Kunde`.

```
@Named
@RequestScoped
public class KundenCtrl implements Serializable {

    @Inject
    private Kunde kunde;

    public Kunde getKunde() {
        return this.kunde;
    }
    public void setKunde(Kunde kunde) {
        this.kunde = kunde;
    }

    public String speicher() {
        System.out.println(kunde.getName());
        return "/antwort.xhtml";
    }
}
```

Listing 30.13: Injection einer Daten-Bean

Die injizierte Bean besitzt ebenfalls einen Scope, also einen Gültigkeitsbereich, der idealerweise dem der Backing Bean entsprechen sollte. Um dies zu gewährleisten, gibt es die Annotation `@Dependent`. Sie übernimmt den Scope der Backing Bean.

```
import javax.enterprise.context.Dependent;

@Dependent
public class Kunde {
    private String name;
    private String email;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
}
```

Listing 30.14: Die injizierte Daten-Bean

Der Zugriff auf die Attribute der Daten-Bean aus dem Facelet erfolgt über den Namen der Backing Bean. Sowohl der erste Buchstabe der Backing Bean als auch der der Daten-Bean werden dabei durch den passenden Kleinbuchstaben ersetzt.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
<h:head>
</h:head>
<h:body>
    <h:form>
        <h:panelGrid columns="2">
            <h:outputText value="Name: " />
            <h:inputText value="#{kundenCtrl.kunde.name}">
            </h:inputText>
            <h:outputText value="Los geht's" />
            <h:commandButton value="OK"
                    action="#{kundenCtrl.speicher}" />
        </h:panelGrid>
    </h:form>
</h:body>
</html>
```

Listing 30.15: Zugriff auf das Feld der Daten-Bean aus dem Facelet

Der Gast und die Datenbank

Klara stoppt Heidi in ihrer Begeisterung für Eingabemasken. »Das Eingeben von Gästen und Buchungen macht ja Spaß. Es ist aber reichlich blöd, wenn das Programm sich nett bedankt, aber die Daten nicht speichert.«

Ganz klar: Die Gäste und die Buchungen müssen in die Datenbank. Dafür muss eine Datenbank gewählt und die Tabellenstruktur festgelegt werden.

Heidi und Klara diskutieren eine Weile darüber, welche Datenbank man einsetzen kann. Zuerst nannte Heidi MySQL oder das alpengerecht benannte Gegenstück MariaDB. Klara fand wenig Gefallen daran und schlug das professionelle Oracle vor. Heidi erklärte, dass der Preis so hoch ist, dass sie den Erlös aus der Vermietung dann direkt an Oracle abführen könnte. Ziegen-Peter schlug PostgreSQL vor, das leistungsstark und lizenzkostenfrei sei. Darauf konnte man sich verständigen und es geht an die Installation.

Installation von PostgreSQL

Die Datenbank PostgreSQL finden Sie kostenlos im Internet für alle möglichen Plattformen.

1. Zum Download wenden Sie sich vertrauensvoll an die URL `www.postgresql.org/download`.
2. Wählen Sie Ihre Zielplattform aus und folgen Sie den plattformabhängigen Installationsvorschlägen. Für Mac und Windows werden Installer-Programme angeboten. Die sorgen dafür, dass für PostgreSQL ein Eintrag im Programmmenü erfolgt.
3. Im Zuge der Installation werden Sie aufgefordert, ein Passwort für den Datenbankadministrator `postgres` festzulegen. Merken Sie es sich gut. Wenn Sie sich keine Passwörter merken können, schreiben Sie es mit dem Filzstift auf den Monitor.

Nach der Installation starten Sie das Programm `psql`, das oft auch unter dem Pseudonym PostgreSQL Shell in diversen Menüs zu finden ist. Unter Windows und Mac werden Sie bei jedem Start nach den folgenden Parametern gefragt:

- ✔ Server (localhost)
- ✔ Database (postgres)
- ✔ Port (5432)
- ✔ Username (postgres)
- ✔ Passwort für Benutzer `postgres` (Ihr Passwort)

Linux geht einen Sonderweg, weil es dort möglich ist, auch einen Datenbankserver ganz ohne grafische Oberfläche einzurichten, und da es auf den Mehrbenutzerbetrieb besser vorbereitet ist.



Linux und Mac richten bei der Installation einen eigenen Benutzer namens `postgres` ein. Dieser hat keinen direkten Login-Zugang, sondern betreibt nach dem Systemstart die Datenbank. Auf diese Weise erhalten normale Benutzer keine Datenbank-Administrationsrechte. Da nicht der Systemadministrator für den Betrieb der Datenbank eingesetzt werden muss, erhält die Datenbank keine Administrationsrechte über das System.

Wenn Sie sich doch als Datenbankadministrator `postgres` anmelden wollen, müssen Sie sich zunächst als Systemadministrator anmelden und dann mit dem Befehl `su - postgres` zum Benutzer postgres werden. Danach können Sie `psql` ohne weitere Parameter aufrufen.

```
sudo -s
su - postgres
psql
```

Auf dem Bildschirm erkennen Sie am Prompt, dass Sie nun im PostgreSQL sind und Datenbankbefehle eingeben können. Zunächst werden wir eine eigene

Datenbank namens `fewo` für die Ferienwohnungen in den Almhütten anlegen. Für die Verwaltung der Ferienwohnungsdatenbank `fewo` legen wir einen eigenen Benutzer namens `fewoadmin` an und übertragen diesem alle Rechte für diese Datenbank.

```
create user fewoadmin password 'geheim' login;
create database fewo;
grant all on database fewo to fewoadmin;
\q
```

Der letzte Befehl `\q` sorgt dafür, dass Sie die Shell wieder verlassen. Nun melden Sie sich mit dem neuen Benutzer an der Datenbank an: Unter Mac und Windows werden Sie beim Klick auf PostgreSQL Shell nach den Parametern gefragt:

- ✔ Server (localhost)
- ✔ Database (fewo)
- ✔ Port (5432)
- ✔ Username (fewoadmin)
- ✔ Passwort für Benutzer `fewoadmin` (geheim)

Unter Linux können Sie den folgenden Befehl absetzen, um sich als `fewoadmin` bei der Datenbank `fewo` anzumelden:

```
psql -d fewo -U fewoadmin -W
```

Tabellen anlegen

Nun meldet sich der Prompt mit `fewo` und Sie können eine Tabelle für den Gast anlegen.

```
create table gast (
id bigserial not null primary key,
name varchar, adresse varchar, ort varchar,
email varchar, telefon varchar);
\q
```



Wenn Sie in der Datenbankumgebung SQL-Befehle absetzen, werden diese erst ausgeführt, wenn Sie das abschließende Semikolon eingeben. Vergessen Sie dieses, wird nichts passieren, bis Sie das erste Mal ein Semikolon eingeben. Und dann wird das Datenbanksystem Ihre Befehle vermutlich nicht verstehen.

Schauen wir genauer auf das Anlegen der Tabelle. Eine Datenbanktabelle hat gewisse Ähnlichkeiten mit einer Klasse, die nur Attribute besitzt. Bei SQL werden zuerst die Namen und dann die Typen genannt. Und es gibt Unterschiede in den Typenbezeichnungen.

- ✔ `bigserial`: Bei diesem Feld handelt es sich typischerweise um einen

numerischen Schlüssel, der von der Datenbank beim Neuanlegen eines Eintrags automatisch erhöht wird. Leider haben die verschiedenen Datenbanken dafür unterschiedliche Namen. Bei MySQL heißt das Schlüsselwort beispielsweise `auto_increment`.

- `not null`: Dieses Feld darf keinesfalls leer sein.
- `primary key`: Dies ist das Schlüsselfeld der Tabelle. Die Zeile wird durch den Wert dieses Feldes eindeutig bestimmt.
- `varchar`: Dieser Wert ist eine Zeichenkette und entspricht in etwa einem String in Java. Achtung: Die Anführungszeichen in Java sind Hochkommata in SQL!

Das Feld `id` ist das Schlüsselfeld der Tabelle `Gast`. Es kann verwendet werden, wenn ein spezieller Gast angesprochen werden soll. Der Zugriff über den primären Schlüssel wird vom Datenbanksystem extrem beschleunigt.

Heidi fragt sich, wie denn eine Buchung erfolgen soll. Gäste sind ja schön, aber die müssen ja irgendwo wohnen. Klara schlägt eine Tabelle `Buchung` vor und legt sie gleich an.

```
create table buchung (
id bigserial not null primary key,
von date, bis date,
gastid integer references gast(id),
wohnung varchar);
```

Auch die Buchung erhält einen Primärschlüssel. Aber die Tabelle enthält auch einen sekundären Schlüssel, der auf den Primärschlüssel der Tabelle `Gast` verweist. Diese Verstrickung von Gast und Buchung hat folgende Konsequenzen:

- Eine Buchung kann nur angelegt werden, wenn es einen Gast für diese Buchung gibt.
- Ein Gast kann in mehreren Buchungen auftreten.
- Ein Gast kann nicht gelöscht werden, wenn er noch in einer Buchung genannt wird.

Die Tabelle `Buchung` verwendet einige weitere Datentypen, die Sie noch nicht in der Tabelle `Gast` gesehen haben:

- `date`: Dies ist ein Datum. In diesem Fall das Datum der Anreise und das der Abreise.
- `integer`: Hier handelt es sich um eine ganze Zahl, die in Java als `int` bekannt ist.

Mit diesen zwei Tabellen können Sie bereits eine ganz ordentliche Verwaltung hinbekommen. Der erfahrene Datenbänker würde einwerfen, dass man eine eigene

Tabelle für die Wohnungen anlegen sollte. Das ist nicht völlig verkehrt. Dort könnte man beispielsweise Informationen hinterlegen wie die Anzahl der Betten oder Preise.

Eine eigene Tabelle für die Wohnungen würde ganz ähnlich funktionieren wie die Tabelle `Gast`. Jede Wohnung erhielte einen primären Schlüssel, der als sekundärer Schlüssel in die Buchungstabelle eingetragen würde. Letztlich bringt das nicht viel Neues und würde hier höchstens zur Verwirrung beitragen. Also überlassen wir dies als Übung für den interessierten Leser.

Application Server trifft Datenbank

Java verfügt mit JDBC über einen Standard, um SQL-Befehle an eine Datenbank zu senden. Die Datenbank hat die Tabellen `Gast` und `Buchung` und es liegt einigermaßen nahe, in Java analoge Klassen `Gast` und `Buchung` zu verwenden. Um die Klassen auf die Tabellen abzubilden, brauchen wir ein Stück Software, das sich um die Besonderheiten von SQL und JDBC kümmern soll.

Zugriff per JDBC

Für den Zugriff auf eine Datenbank benötigt ein Java-Programm den JDBC-Treiber. Das gilt natürlich auch für den Application Server. Sie hatten davon sicher schon im Datenbank-Kapitel gelesen.

Für die hier verwendete Datenbank PostgreSQL können Sie den JDBC-Treiber unter folgender URL aus dem Internet herunterladen:

`http://jdbc.postgresql.org/download.html`

Schaffen Sie diese JAR-Datei in Ihr Projekt. Einen schönen molligen Platz findet er unter *WebContent/WEB-INF/lib*. Sie können ihn einfach per Drag & Drop in das Eclipse-Projekt verlegen.

DAO – Data Access Object

Es bietet sich an, für die Tabellen `Gast` und `Buchung` Gegenstücke auf Java-Seite zu erfinden. Und naheliegenderweise wären dies Klassen mit möglichst gleichlautenden Attributen. Damit verbleibt die Aufgabe, aus Java-Objekten der Klasse `Gast` Zeilen in der Datenbanktabelle `Gast` zu machen und umgekehrt. Um diesen Zugriff auf die Datenbank soll sich ein Datenzugriffsmodul kümmern. Im englischen Sprachgebrauch spricht man gern von einem *Data Access Object* oder auch *DAO*. Wir erstellen für den Gast eine neue Java-Klasse und nennen sie `GastDao`.

Die Parameter der Datenbanktabelle stehen als private Attribute zu Anfang der Klasse. Der JDBC-Treiber wird im Konstruktor der Klasse abgelegt.

Wie Sie sehen, ist der Konstruktur privat. `GastDao` ist also als Singleton konzipiert. Es wird nur eine Instanz geben, auf die jederzeit über `GastDao.getInstance()`

zugegriffen werden kann.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class GastDao {

    private String host = "localhost";
    private String dbName = "fewo";
    private String tableName = "gast";
    private String user = "fewoadmin";
    private String password = "geheim";

    static private GastDao instance = null;

    static public GastDao getInstance() {
        if (instance == null) {
            instance = new GastDao();
        }
        return instance;
    }

    private GastDao() {
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException e) {
            System.out.println("Kein Treiber");
            e.printStackTrace();
        }
    }

    private Connection connect() throws SQLException {
        // Verbindung mit automatischer Erzeugung der DB
        Connection dbConnect = DriverManager.getConnection
                ("jdbc:postgresql://" + host + "/"
                + dbName, user, password);
        return dbConnect;
    }
    // ...
}
```

Listing 30.16: Gast Data Access Object

Im ersten Schritt wird ein Objekt der Klasse `Gast` in der Tabelle `Gast` gespeichert. Das erledigt treu und brav der SQL-Befehl `INSERT`, den wir per JDBC an PostgreSQL schicken. Dazu schreiben wir eine Methode `addGast`.

```
public long addGast(Gast gast) {
    long id = -1L;
    try {
        Connection connection = this.connect();
        String[] returnIds = { "id" };
        PreparedStatement statement = connection.prepareStatement(
                "insert into " + tableName +
                " (name, adresse, ort, email, telefon) " +
                "VALUES (?,?,?,?,?)", returnIds);
        statement.setString(1, gast.getName());
        statement.setString(2, gast.getAdresse());
        statement.setString(3, gast.getOrt());
        statement.setString(4, gast.getEmail());
        statement.setString(5, gast.getTelefon());
        statement.executeUpdate();
        // Die ID des eingefügten Satzes zurückholen
        ResultSet generatedKeys = statement.getGeneratedKeys();
        if (generatedKeys.next()) {
            id = generatedKeys.getLong(1);
            gast.setId(id);
        }
        statement.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return id;
}
```

Listing 30.17: Methode addGast im GastDao

Die Methode `addGast` soll dem Aufrufer den Schlüssel zurückgeben, den der Gast beim Anlegen automatisch von PostgreSQL zugewiesen bekommt. Dazu wird nach dem Ausführen des SQL-Statements der erzeugte Schlüssel mit der Methode `getGenerateKeys` abgeholt. Damit dies gelingt, muss zuvor ein String-Array mit den Key-Bezeichnern gebildet und an die Methode `prepareStatement` übergeben werden.

In der Methode `addGast` wurden die Klasse `Gast` und ihre Methoden bereits ausgiebig verwendet. Dabei wurde die Klasse noch gar nicht definiert. Das wollen wir schleunigst nachholen. Wir orientieren uns einfach an der Tabelle `Gast`. Erstellen Sie eine neue Klasse und schreiben Sie folgende Zeilen:

```
package model;
public class Gast {
    private long id=0;
    private String name, adresse, ort, email, telefon;
}
```

Listing 30.18: Ein ausbaufähiger Rahmen für die Klasse `Gast`

Sie könnten die Getter und Setter-Methoden selbst schreiben. Oder Sie überlassen die Drecksarbeit anderen, die nicht so gut bezahlt werden, beispielsweise Eclipse:

- Klicken Sie die Datei *Gast.java* mit der rechten Maustaste an.
- Wählen Sie Source|Generate Getters And Setters.
- Im Dialog klicken Sie Select All und OK.

Damit ist die Klasse `Gast` vollständig.

Datenbank mit dem Servlet verbinden

Sollte also ein Servlet, eine JSP oder die Backing Bean eines Facelets einen Gast in die Datenbank schreiben wollen, geht das folgendermaßen vor sich:

1. Sie erzeugen aus den Daten der Eingabemaske ein vollständig bestücktes Objekt der Klasse `Gast`.
2. Sie holen sich die Instanz von `GastDao` über deren statische Methode `getInstance`.
3. Darüber rufen Sie die Methode `addGast` und übergeben das Objekt der Klasse `Gast`.

Listing 30.19 zeigt dies anhand einer Methode `doPost` eines Servlets.

```
protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
    String name = request.getParameter("name");
    if (!name.isEmpty()) {
        model.Gast gast = new model.Gast();
        gast.setName(name);
        gast.setAdresse(request.getParameter("adresse"));
        gast.setOrt(request.getParameter("ort"));
        gast.setEmail(request.getParameter("email"));
        gast.setTelefon(request.getParameter("telefon"));
        model.GastDao.getInstance().addGast(gast);
        RequestDispatcher dispatch =
                request.getRequestDispatcher("gastneu.jsp");
        dispatch.forward(request, response);
    } else {
        request.getRequestDispatcher("index.html")
                .forward(request, response);
    }
}
```

Listing 30.19: Servlet ruft DAO.

Wenn alles geklappt hat, verzweigt das Servlet zum JSP *gastneu.jsp*, um dort noch einmal anzuzeigen, was nun in die Datenbank geschrieben wurde.

Einen speziellen Gast abholen

Mit der Methode `addGast` wurde der Gast in die Datenbank gestellt. Aber es gibt doch immer wieder den einen oder anderen Neugierspinsel, der wissen will, ob der Gast wirklich in der Datenbank steht. Also kann er den Schlüssel verwenden und den Gast aus der Datenbank herauslesen lassen. Dazu bauen wir eine Methode `getGast` in die Klasse `GastDao` ein.

```
public Gast getGast(long id) {
    Gast gast = null;
    try {
        Connection connection = this.connect();
        Statement statement = connection.createStatement();
        String select = "SELECT "
            + "name, adresse, ort, email, telefon "
            + "FROM " + tableName + " WHERE id=" + id;
        ResultSet ergebnis = statement.executeQuery(select);
        while (ergebnis.next()) {
            gast = new Gast();
            gast.setId(id);
            gast.setName(ergebnis.getString("name"));
            gast.setAdresse(ergebnis.getString("adresse"));
            gast.setOrt(ergebnis.getString("ort"));
            gast.setEmail(ergebnis.getString("email"));
            gast.setTelefon(ergebnis.getString("telefon"));
        }
        ergebnis.close(); // Ergebnismenge schließen
        statement.close();
    } catch (SQLException e) {
    }
    return gast;
}
```

Listing 30.20: Suche einen Gast anhand der ID.

Die Methode erstellt ein Objekt der Klasse `Gast` und füllt es mit den Werten, die aus der `SELECT`-Anfrage herauskommen. Da `id` eindeutig ist, kann nur ein Gast gefunden werden oder keiner. Im letzteren Fall bleibt es bei der Rückgabe von `null`.

Gästeliste

In vielen Fällen will man eine Liste von Gästen. Oder man sucht nach Gästen nicht anhand des eindeutigen Schlüssels, sondern nach dem Namen, und der ist alles andere als eindeutig. In der Regel bekommt man eine Liste heraus.

Sie können die Klasse `GastDao` erneut um eine Methode `getGast` bereichern, die eine Liste von Gästen zurückliefert. Der gleiche Name ist erlaubt, wenn sich der Parameter unterscheidet. Und zum Glück ist der Schlüssel eine ganze Zahl, während der Name ein String ist.

```
public ArrayList<Gast> getGast(String name) {
    ArrayList<Gast> liste = new ArrayList<>();
    try {
        Connection connection = this.connect();
        Statement statement = connection.createStatement();
        String select =
                "SELECT id, name, adresse, ort, email, telefon"
                + " FROM " + tableName;
        if (!name.isEmpty()) {
            select += " WHERE name='"+name+"'";
        }
        ResultSet ergebnis = statement.executeQuery(select);
        while (ergebnis.next()) {
            Gast gast = new Gast();
            gast.setId(ergebnis.getLong("id"));
            gast.setName(ergebnis.getString("name"));
            gast.setAdresse(ergebnis.getString("adresse"));
            gast.setOrt(ergebnis.getString("ort"));
            gast.setEmail(ergebnis.getString("email"));
            gast.setTelefon(ergebnis.getString("telefon"));
            liste.add(gast);
        }
        ergebnis.close(); // Ergebnismenge schließen
        statement.close();
    } catch (SQLException e) {
    }
    return liste;
}
```

Listing 30.21: Abfrage einer Gästeliste in GastDao

Um die Gästeliste zurückzugeben, müssen wir ein Array bilden oder etwas besser eine `ArrayList`, wie Sie sie in Kapitel 15 kennengelernt haben. Denn wie umfangreich die Liste sein wird, können wir nicht vorhersagen.

Eine Gästeliste mit JSP abrufen

Als Beispiel verwenden wir dieses Mal eine Java Server Page, die den Inhalt der Datenbanktabelle `Gast` auf dem Browser ausgibt.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ page import="java.util.ArrayList" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
        charset=UTF-8">
</head>
<body>
    <h2>G&auml;steliste</h2>
    <table>
        <tr>
            <td>Name</td>
            <td>Adresse:</td>
            <td>PLZ Ort:</td>
            <td>E-Mail:</td>
            <td>Telefon:</td>
        </tr>
    <%
    ArrayList<model.Gast>liste
            = model.GastDao.getInstance().getGast("");
    for (model.Gast gast:liste) {
                %>
        <tr>
            <td> <% out.print(gast.getName()); %> </td>
            <td> <% out.print(gast.getAdresse()); %> </td>
            <td> <% out.print(gast.getOrt()); %> </td>
            <td> <% out.print(gast.getEmail()); %> </td>
            <td> <% out.println(gast.getTelefon()); %> </td>
        </tr>
        <% } %>
    </table>
    <p/>
    <a href="index.html">zur&uuml;ck</a>
</body>
</html>
```

Listing 30.22: JSP zur Anzeige der Gästeliste

Gast entfernen

Natürlich will man immer mehr Gäste, aber manchmal möchte man auch gern den einen oder anderen loswerden. In der Realität können Sie es mit laut gedrehter bayerischer Volksmusik aus dem Nachbarzimmer versuchen. Aber das ist nicht

zielsicher. Es könnte sein, dass der Gast taub ist, bayerische Volksmusik mag oder dass alle anderen Gäste ausziehen. Da ist der `DELETE`-Befehl einer Datenbank verlässlicher.

Zum Löschen von Gästen erstellen wir in der Klasse `GastDao` eine zusätzliche Methode `deleteGast`. Sie erstellt einen SQL-Befehl `DELETE`, in dessen `WHERE`-Klausel `id` abgefragt wird. Wie schon beim `INSERT` wird der Parameter über ein Fragezeichen eingefügt. Da `id` ein Zahlenwert ist, wird hier `setLong` verwendet, um das Statement zu ergänzen.

```
public void deleteGast(String strid) {
    try {
        Connection connection = this.connect();
        PreparedStatement statement = connection.prepareStatement(
                "delete from " + tableName +
                " where id=?");
        long id = java.lang.Long.parseLong(strid);
        statement.setLong(1, id);
        statement.executeUpdate();
        statement.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Listing 30.23: Löschen des Gastes in der Datenbank

Der Benutzer wird nicht viel Spaß daran haben, den primären Schlüssel des Gastes einzutippen, um diesen zu entfernen. Besser ist es, ihm eine Auswahl von Gästen zu servieren, von denen er einen abservieren kann. Darum ist die Gästeliste ein guter Ausgangspunkt für das Löschen von Gästen. Sie bauen um die Gästeliste herum ein Formular mit dem HTML-Tag `FORM`. Vor jeden Gast setzen Sie eine Checkbox als Input-Element und am Ende erstellen Sie einen Button mit der Beschriftung Löschen. Sie ergänzen also die Gästeliste aus Listing 30.22 durch folgende Zeilen:

```
<form method="post" action="ctrlListGast">
<table>
<!-- ... ->
for (model.Gast gast:liste) {
     %>
    <tr><td>
    <td>
    <input type="checkbox" name="<% out.print(gast.getId()); %>">
    </td>
    <td><% out.print(gast.getName()); %></td>
<!-- ... ->
</table>
<input type="submit" value="L&ouml;schen">
</form>
```

Listing 30.24: Die Gästeliste wird zum Auswahlformular.

Die Checkbox erhält als Name die ID des Gastes aus der Datenbank. Auf diese Weise muss das Servlet `ctrlListGast` nur ermitteln, welche Box angehakt ist, und sich deren Namen geben lassen. Das Auslesen aller Input-Elemente haben Sie oben schon einmal gesehen. Hier wird es angewandt. Da Eclipse den Rahmen des Servlets eh selbst baut, interessiert an dieser Stelle nur die Methode `doPost`.

```
protected void doPost(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException {
    Enumeration<String> controls = request.getParameterNames();
    while (controls.hasMoreElements()) {
        String controlName = controls.nextElement();
        if (request.getParameter(controlName).equals("on")) {
            GastDao.getInstance().deleteGast(controlName);
        }
    }
    request.getRequestDispatcher("gaesteliste.jsp")
            .forward(request, response);
}
```

Listing 30.25: Das Servlet wertet die Checkboxen aus.

Die Schleife klappert alle Eingabeelemente ab und schaut, welches davon einen Haken hat, also `on` als `Value` enthält. Dessen Name ist die ID, die man anschließend benutzt, um den Gast zu löschen. Das erfolgt in `GastDao` über die Methode `deleteGast`. Zum Schluss wird wieder die Gästeliste aufgerufen, die zeigt, dass der Gast verschwunden ist.

Eine DAO für die Buchung

Die Gäste sind erfasst. Nun geht es darum, eine Wohnung oder eine Hütte zu

buchen. Wir benötigen also eine Klasse `Buchung` und eine Klasse, die dafür sorgt, dass die Daten zwischen Klasse und Tabelle ausgetauscht werden, also die DAO.

Eigentlich ist es langweilig, noch eine DAO durchzukauen. Aber die Buchung ist anders.

- Eine Buchung erfolgt immer über einen Zeitraum. Also benötigen Sie dazu ein Datum. Java kennt von Haus aus zwei Typen `Date`. Einer stammt aus `java.util`. Den verwenden Sie standardmäßig. Und dann gibt es ein anderes `Date` aus dem Package `java.sql`. Dieser findet seinen Einsatz in der JDBC-Klasse.
- Die zweite Besonderheit liegt darin, dass eine Buchung immer durch einen Gast erfolgt. Niemand will aber den gesamten Gast in die Buchung kopieren. Stattdessen legen Sie die ID des Gastes in eine Spalte der Buchung und können darüber immer nachschauen, wie der Gast heißt, der die Wohnung bucht, und wie seine Kontaktdaten aussehen. Diese Art der Referenz bezeichnet man als *Foreign Key* oder Fremdschlüssel.

Die Bean Buchung

Zunächst erstellen Sie die Klasse `Buchung`, auf der die DAO aufsetzt. Sie besteht aus den Attributen, die auch bei der Tabellenerzeugung verwendet wurden. Natürlich sind alle Attribute privat und Sie erstellen noch die Setter und Getter dazu.

```
package buchung;
import java.io.Serializable;
import java.util.Date;

public class Buchung implements Serializable {
    private long id = 0, gastid;
    private Date von, bis;
    private String wohnung;
}
```

Listing 30.26: Die Klasse `Buchung` ist die Basis für XML und Datenbanktabelle.

Die Tabelle wird auf Java-Seite durch die Klasse `Buchung` abgebildet. Der Datumstyp `Date` muss der Standardtyp aus `java.util` sein, damit er später bei der XML-Umwandlung keinen Ärger macht.

Die Datenbank-Klasse BuchungDao

Für die Klasse `BuchungDao` benötigen Sie vor allem eine Methode für das Anlegen einer Buchung, die wir `addBuchung` nennen, und eine Methode, um die Buchungsliste zu holen. Die soll `getBuchung` heißen. Natürlich können Sie noch

eine Reihe weiterer Methoden erstellen, aber das soll Ihnen als Übungsaufgabe für später bleiben.



Als Datumstyp wird `java.sql.Date` verwendet, da dies von den JDBC-Methoden so erwartet wird. Daten, die von außen kommen oder nach außen gehen, müssen umgewandelt werden. Leider besteht REST nämlich auf dem Typ `java.util.Date`.

Ansonsten ähnelt die Klasse `BuchungDao` der Klasse `GastDao`. Die Referenz auf den buchenden Gast versteckt sich im Attribut der Buchung und muss an dieser Stelle nicht berücksichtigt werden.

```
package buchung;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.sql.Date;

public class BuchungDao {

    private String host = "localhost";
    private String dbName = "fewo";
    private String tableName = "buchung";
    private String user = "fewoadmin";
    private String password = "geheim";

    static private BuchungDao instance = null;

    static public BuchungDao getInstance() {
        if (instance == null) {
            instance = new BuchungDao();
        }
        return instance;
    }

    private BuchungDao() {
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException e) {
            System.out.println("Kein Treiber");
            e.printStackTrace();
        }
    }

    private Connection connect() throws SQLException {
        Connection dbConnect = DriverManager.getConnection(
                "jdbc:postgresql://" + host + "/" + dbName,
                user, password);
        return dbConnect;
    }

    public long addBuchung(Buchung buchung) {
        long id = -1L;
        try {
```

```
            Connection connection = this.connect();
            String[] returnIds = { "id" };
            PreparedStatement statement
                    = connection.prepareStatement(
                    "insert into " + tableName +
                    " (von, bis, gastid, wohnung) " +
                    "VALUES (?,?,?,?)", returnIds);
            statement.setDate(1,
                    new Date(buchung.getVon().getTime()));
            statement.setDate(2,
                    new Date(buchung.getBis().getTime()));
            statement.setLong(3, buchung.getGastid());
            statement.setString(4, buchung.getWohnung());
            statement.executeUpdate();
            // Die ID des eingefügten Satzes zurückholen
            ResultSet generatedKeys = statement.getGeneratedKeys();
            if (generatedKeys.next()) {
                id = generatedKeys.getLong(1);
                buchung.setId(id);
            }
            statement.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return id;
    }

    public ArrayList<Buchung> getBuchung() {
        ArrayList<Buchung> liste = new ArrayList<>();
        try {
            Connection connection = this.connect();
            Statement statement = connection.createStatement();
            String select = "SELECT id, von, bis, gastid, wohnung"
                    + " FROM " + tableName;
            ResultSet ergebnis = statement.executeQuery(select);
            while (ergebnis.next()) {
                Buchung buchung = new Buchung();
                buchung.setId(ergebnis.getLong("id"));
                buchung.setVon(ergebnis.getDate("von"));
                buchung.setBis(ergebnis.getDate("bis"));
                buchung.setGastid(ergebnis.getLong("gastid"));
                buchung.setWohnung(ergebnis.getString("wohnung"));
                liste.add(buchung);
            }
            ergebnis.close(); // Ergebnismenge schließen
            statement.close();
        } catch (SQLException e) {
```

```
        }
        return liste;
    }
}
```

Listing 30.27: Datenbankzugriffe per BuchungDao

Besonders auffallend ist die Umwandlung von `Date` nach `Date` in der Methode `addBuchung`:

```
statement.setDate(1, new Date(buchung.getVon().getTime()));
```

Das `Date` in der Klasse `Buchung` stammt aus `java.util`, weil REST dies erfordert. Dagegen wünscht sich JDBC das `Date` aus `java.sql`.

Datenbankanwendung per JPA

In Kapitel 24 wird JPA als Zugriff für Datenbanken bereits vorgestellt. JPA stammt eigentlich aus dem JEE-Umfeld und so ist es kein Wunder, dass es hier auch besser unterstützt wird, insbesondere vom Glassfish.

Einrichtung

Wenn Sie ein dynamisches Web-Projekt anlegen, können Sie die JPA-Unterstützung anfordern.

1. Klicken Sie unter Configuration den Button Modify an.
2. In der Liste der Facets finden Sie auch JPA. Dieses müssen Sie anwählen. Anschließend können Sie den Dialog mit OK schließen.
3. Sie benötigen einen Projektnamen. Dann kommen Sie mit Next weiter.
4. Für die weiteren Einstellungen der JPA müssen Sie ein weiteres Mal Next anklicken.
5. Als Platform müsste `EclipseLink` ausgewählt sein.
6. Als JPA Implementation müsste `GlassFish System Library` ausgewählt sein.
7. Unter Connection können Sie eine bereits angelegte Datenbank-Verbindung auswählen oder über den Link Add Connection eine neue erstellen.

Sie können einem Projekt auch nachträglich zu einer JPA-Unterstützung verhelfen.

1. Klicken Sie das Projekt mit der rechten Maustaste an und wählen Sie Configure|Convert TO JPA-Project.
2. Es erscheint ein Dialog. Auf der linken Seite muss JPA angewählt sein, auf der rechten Seite unter Runtimes Ihr Server Glassfish. Klicken Sie auf Next!

3. Unter Connection wird nun die Verbindung zur Datenbank aufgetan. Add Connection.
4. Im neuen Dialog wird das Verbindungs-Profil angewählt.

Weiterfahren mit Add Connection:

1. Der erste Dialog wählt das Profil anhand des Datenbank-Management-Systems aus. Hier wählen Sie PostgreSQL aus und vergeben einen Namen, anhand dessen Sie die Verbindung wiedererkennen, beispielsweise `PostgresFewo`. Klicken Sie auf Next.
2. Auf der nächsten Seite geben Sie die Verbindungsparameter ein.
    - Database: Den Namen der Datenbank, also `fewo`.
    - URL: Hier wird der JDBC-Verbindungsstring der Datenbank eingegeben. Bei PostgreSQL lautet er `jdbc:postgresql://localhost:5432/fewo`

      Hier müssen Sie die Vorgabe `database` am Ende durch `fewo` korrigieren.
    - User Name: Hier kommt der Benutzer der Datenbank, also `fewoadmin`.
    - Password: Das Passwort ist bekanntlich geheim.
    - Sie können in einer Checkbox anwählen, dass das Passwort gespeichert wird.

    Sind die Verbindungsparameter korrekt, können Sie den Button Test Connection anklicken. Gibt es noch einen Fehler in den Verbindungsparametern, können Sie sich in der Dialogbox die Fehlermeldung ansehen.
3. Unter Next sehen Sie noch einmal die Verbindungsparameter, können aber auch gleich Finish anklicken.

Im Projekt finden Sie ein Verzeichnis mit dem Namen *JPA Content*, in dem sich die Datei *persistence.xml* befindet. In dieser finden Sie die Konfiguration der Datenbank.

Eclipse stellt für die Datei *persistence.xml* einen Assistenten zur Verfügung, der die Bearbeitung ermöglicht, ohne die XML-Strukturen selbst zu editieren. Wenn Sie also auf die Datei doppelt klicken, sehen Sie diesen Assistenten.

1. Wählen Sie die Lasche Connection aus. Sie befindet sich unterhalb des Assistenten.
2. Stellen Sie Transaction Type auf Resource Local. Dadurch wird der Link Populate From Connection weiter unten aktivierbar.
3. Klicken Sie auf diesen Link. Es erscheint ein Dialog, aus dem Sie Ihre

Connection auswählen können. Bestätigen Sie mit OK.

Erstellen der Entities

Nun sollen die Klassen erstellt werden, die die Datenbanktabellen repräsentieren.

1. Klicken Sie das Projekt mit der rechten Maustaste an und wählen Sie JPA Tools|Generate Entities From Tables.
2. Der Folgedialog zeigt Ihnen alle Tabellen Ihrer Datenbankverbindung. Wählen Sie `gast` und `buchung` an. Klicken Sie auf Next.
3. Auf der Folgeseite ist ein Diagramm der Tabellenassoziationen zu sehen. Dort sehen Sie eine 1:n-Beniehung zwischen `gast` und `buchung`. Klicken Sie auf Next.
4. Im Folgedialog wählen Sie als Key Generator `identity` aus.
5. Sie können mit Finish schließen.

Es entstehen zwei Java-Beans, die mit ein paar Annotationen gefüllt sind, um die Verbindung zu der Datenbanktabelle zu beschreiben. Ich habe mir erlaubt, die Klasse von ein paar Leerzeilen, Kommentaren und Gettern und Settern zu befreien.

```java
package model;

import java.io.Serializable;
import javax.persistence.*;
import java.util.List;

@Entity
@NamedQuery(name="Gast.findAll", query="SELECT g FROM Gast g")
public class Gast implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String adresse;
    private String email;
    private String name;
    private String ort;
    private String telefon;

    //bi-directional many-to-one association to Buchung
    @OneToMany(mappedBy="gast")
    private List<Buchung> buchungs;

    // ... hier waren Getter und Setter ...

    public List<Buchung> getBuchungs() {
        return this.buchungs;
    }

    public void setBuchungs(List<Buchung> buchungs) {
        this.buchungs = buchungs;
    }

    public Buchung addBuchung(Buchung buchung) {
        getBuchungs().add(buchung);
        buchung.setGast(this);
        return buchung;
    }
    public Buchung removeBuchung(Buchung buchung) {
        getBuchungs().remove(buchung);
        buchung.setGast(null);
        return buchung;
    }
```

```
}
```

Listing 30.28: Die Klasse `Gast`

Interessant ist, dass die Klasse `Gast` die 1:n-Beziehung in einer `List` aus dem Java Collection Framework (siehe Kapitel 15) abbildet.

Das Gegenstück bildet die Klasse `Buchung`. Auch hier habe ich den Standardkonstruktor und die langweiligen Getter und Setter entfernt.

```java
package model;

import java.io.Serializable;
import javax.persistence.*;
import java.util.Date;

@Entity
@NamedQuery(name="Buchung.findAll",
            query="SELECT b FROM Buchung b")
public class Buchung implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    @Temporal(TemporalType.DATE)
    private Date bis;

    @Temporal(TemporalType.DATE)
    private Date von;

    private String wohnung;

    //bi-directional many-to-one association to Gast
    @ManyToOne
    @JoinColumn(name="gastid")
    private Gast gast;

    public Date getBis() {
        return this.bis;
    }

    public void setBis(Date bis) {
        this.bis = bis;
    }

    public Date getVon() {
        return this.von;
    }

    public void setVon(Date von) {
        this.von = von;
    }

    public Gast getGast() {
        return this.gast;
```

```
    }

    public void setGast(Gast gast) {
        this.gast = gast;
    }
}
```

Listing 30.29: Die Klasse Buchung

Da eine Buchung nur von einem Gast wahrgenommen werden kann, wird keine Liste benötigt. Aber dafür wird der Verweis auf die andere Spalte mit `@JoinColumn` festgelegt.

HTML-Formular zur Buchung

Zum Test werden wir eine Buchungsanfrage per JPA in die Datenbank zaubern und hoffentlich wohlbehalten wieder hervorholen. Der Einfachheit halber werden wir dazu ein HTML-Formular stricken und die Ergebnisse mit einem Servlet auslesen.

Das HTML-Formular sendet per POST, um die Eingabemaske an das Servlet `GetBuchung` zu senden. Das zweite Formular besteht aus einem einfachen Button, der per GET das Servlet veranlassen wird, eine Buchungsliste darzustellen.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html;
        charset=UTF-8" />
</head>
<body>

<h2>Buchung</h2>
```

```
  <form action="GetBuchung" method="post">
    <h3>Gast</h3>
    <table>
    <tr><td>Name</td>
       <td><input type="text" name="name"/></td></tr>
    <tr><td>Adresse</td>
       <td><input type="text" name="adresse"/></td></tr>
    <tr><td>Ort</td>
       <td><input type="text" name="ort"/></td></tr>
    <tr><td>E-Mail</td>
       <td><input type="text" name="email"/></td></tr>
    <tr><td>Telefon</td>
       <td><input type="text" name="telefon"/></td></tr>
    </table>
    <h3>Wohnung</h3>
    <table>
        <tr><td>Wohnung</td>
       <td><input type="text" name="wohnung"/></td></tr>
    <tr><td>von (yyyy-MM-dd)</td>
       <td><input type="text" name="von"/></td></tr>
    <tr><td>bis (yyyy-MM-dd)</td>
       <td><input type="text" name="bis"/></td></tr>
  </table>
  <input type="submit" value="Einfügen">
  </form>
  <h2>Buchungsliste</h2>
  <form action="GetBuchung" method="get">
    <input type="submit" value="Liste anzeigen">
  </form>
</body>
</html>
```

Listing 30.30: Das Formular zum Anston der Buchung

HTML besitzt leider kein Standardelement für die Eingabe eines Datums. Stattdessen wird hier ein normales Texteingabefeld verwendet. Wir hoffen, dass der Benutzer kooperativ ist. Ansonsten wird es Exceptions hageln. Für die endgültige Version wird das natürlich abgesichert. Der letzte Satz ist übrigens so eine Standardfloskel.

Der EntityManager im Servlet

JPA arbeitet mit einem `EntityManager`, der die Entities bei der Hand nimmt und

ihnen sagt, ob sie gelesen oder geschrieben werden. Wer sich mit Datenbanken auskennt, wird sofort bemerken, dass ich mich um einen Join gedrückt habe. Das stimmt. Ich wollte nicht allzuweit abschweifen.

```
import java.io.IOException;
import java.time.LocalDate;
import java.time.ZoneId;
import java.time.format.DateTimeFormatter;
import java.util.Arrays;
import java.util.Date;
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import model.Gast;

@WebServlet("/GetBuchung")
public class GetBuchung extends HttpServlet {
    private static final long serialVersionUID = 1L;

    static final String persUnit = "JpaBuchung";

    public GetBuchung() {
        super();
    }
```

Listing 30.31: Das Servlet GetBuchung

Damit ist der Rahmen nicht ungewöhnlich. Allerdings finden Sie hier bereits eine reiche Fülle von Importen, die Sie noch brauchen werden. Die Variable `persUnit` enthält den Namen der Persistence Unit, die in der Datei *persistence.xml* festgelegt ist und standardmäßig aus dem Namen des Projekts erstellt wird.

Zunächst werfen Sie bitte einen Blick auf die Methode `doPost`, die eine Buchung in die Datenbank schafft. Dabei wird zunächst ein Objekt der Klasse `Gast`, dann der Klasse `Buchung` aus den Eingaben in die Formularelemente erstellt. Anschließend

wird der `EntityManager` aus der `EntityManagerFactory` erzeugt, die den Namen der `PersistenceUnit` trägt. Innerhalb der Transaktion sorgen zwei Aufrufe von `persist` dafür, dass die Objekte in die Datenbank gelangen.

```
protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
                throws ServletException, IOException {
    // Auslesen der Anfrage und Verpackung in Objekte
    model.Gast gast = new model.Gast();
    gast.setName(request.getParameter("name"));
    gast.setAdresse(request.getParameter("adresse"));
    gast.setOrt(request.getParameter("ort"));
    gast.setEmail(request.getParameter("email"));
    gast.setTelefon(request.getParameter("telefon"));
    model.Buchung buchung = new model.Buchung();
    buchung.setGast(gast);
    buchung.setWohnung(request.getParameter("wohnung"));
    DateTimeFormatter dtf = DateTimeFormatter.
            ofPattern("yyyy-MM-dd");
    LocalDate dt;
    dt = LocalDate.parse(request.getParameter("von"), dtf);
    Date von = Date.from(dt.atStartOfDay(
            ZoneId.systemDefault()).toInstant());
    buchung.setVon(von);
    dt = LocalDate.parse(request.getParameter("bis"), dtf);
    Date bis = Date.from(dt.atStartOfDay(
            ZoneId.systemDefault()).toInstant());
    buchung.setBis(bis);
    // EntityManager starten und Objekte speichern
    EntityManagerFactory factory =
            Persistence.createEntityManagerFactory(persUnit);
    EntityManager manager = factory.createEntityManager();
    manager.getTransaction().begin();
    manager.persist(gast);
    manager.persist(buchung);
    manager.getTransaction().commit();
    manager.close();
}
```

Listing 30.32: Die Methode `doPost` speichert eine Buchung.

Ein bisschen gewaltig wirkt die Umwandlung der Eingabe in die Variablen vom Typ `Date`, den die JPA erwartet.

Das Auslesen der Datenbank würde ein Datenbankspezialist unbedingt mit einem Join erledigen. Da wir nur die Buchungen mit dem jeweils buchenden Gast umsetzen, verschachtelt die Methode `doGet` zwei Query-Aufrufe. Damit die Liste lesbar wird, wird sie in HTML eingebettet.

```
protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
                throws ServletException, IOException {
    EntityManagerFactory factory =
            Persistence.createEntityManagerFactory(persUnit);
    EntityManager manager = factory.createEntityManager();
    // Daten auslesen und anzeigen
    Query query = manager.createQuery(
            "select b from Buchung b");
    List<model.Buchung>buchliste = query.getResultList();
    response.getWriter().println(
            "<!DOCTYPE html>\n<html><body><table>");
    for (model.Buchung buch : buchliste) {
        Query qgast = manager.createQuery(
                "select g from Gast g where g.id=:id");
           qgast.setParameter("id", buch.getGast().getId());
        qgast.getSingleResult();
        Gast gast = (Gast) qgast.getSingleResult();
        response.getWriter().print("<tr><td>"+
                gast.getName()+"</td><td>");
        response.getWriter().print(buch.getWohnung()+
                "</td></tr>");
    }
    response.getWriter().println("</table><p>"+
            "<a href=\"index.html\">zurück</a>"+
            "</body></html>");
    manager.close();
}
```

Listing 30.33: Die Methode `doGet` liest die Buchungsliste.

Jersey und der ganze REST

Der Einsatz von Java Server Pages und Java Server Faces ist prima, wenn man vor allem mit Browsern arbeiten will. Will man selbstständige Client-Anwendungen schreiben, sind JSP und JSF nicht ideal.

Heidi hat schon einmal etwas von REST gelesen. Da heißt es, dass REST Datenressourcen verwaltet und an fast beliebige Clients liefert. Beispielsweise

könnte ein REST-Service für Gäste und einer für die Buchungen aufgesetzt werden. Die Verknüpfung könnte zur Not der Client übernehmen.

REST und HTTP

Ein REST-Service (Representational State Transfer) bietet über HTTP eine Ressource statuslos zur Verfügung. Jede Ressource wird durch die URL bestimmt. Die Unterverzeichnisse der Ressource werden als Parameter verwendet. Die HTTP-Methoden werden als Aktionen auf der Ressource angesehen. Der Server kann verschiedene MIME-Formate wie HTML, XML oder JSON anbieten. Der Client legt das von ihm präferierte Format bei der Anfrage fest. Da sich ein REST-Service komplett aus Web-Standards zusammensetzt, ist es möglich, ihn von beinahe allen Programmiersprachen und Plattformen aus zu nutzen.

- HTTP stellt die Befehle `GET`, `POST`, `PUT` und `DELETE` zur Verfügung. Der REST-Service (Representational State Transfer) nutzt diese HTTP-Methoden, um Aktionen auf einer Ressource zu definieren.
  - `GET` holt ein oder mehrere Elemente.
  - `PUT` wird verwendet, um ein neues Element anzulegen.
  - `DELETE` entfernt ein Element.
  - `POST` kann die Daten eines Elements ändern, aber auch ein neues anlegen, wenn noch keines für den angegebenen Schlüssel existiert.
- Ein Webserver verwendet als Datenformat HTML. REST kann das auch, aber REST kann auch andere Formate nutzen, wie sie in MIME standardisiert sind.
  - **text/plain**: Hier werden rohe Textdaten übertragen. Das dürfte aber nur für besondere Fälle wirklich interessant sein.
  - **application/xml**: Mit XML (siehe Kapitel 25) kann man alle möglichen Datenstrukturen abbilden. Es gibt kaum eine Programmiersprache, die nicht über eine XML-Bibliothek verfügt. Für den Austausch von Daten ist das also eine gute Basis.
  - **application/json**: JSON stammt aus dem JavaScript-Umfeld und ist etwas einfacher gestrickt, aber kann so auch von leistungsärmeren Sprachen verwendet werden.
  - **text/html**: Der REST-Service kann auch HTML-Daten an einen Browser senden.
  - **application/x-www-form-urlencoded**: Damit lassen sich auch Daten, die aus einem HTML-Form stammen, an einen REST-Service senden.

  Der Client erklärt, in welchem Datenformat er bedient werden möchte. Der REST-Server ruft dann diejenige Methode, die dieses Format beherrscht, oder gibt die Fehlermeldung 415 zurück, wenn nichts Passendes implementiert

wurde.

Die vier HTTP-Kommandos erlauben das Anlegen, Ändern, Löschen und Auslesen der REST-Ressource. Mit diesen Operationen lassen sich auch Datenbanktabellen pflegen. Insofern ist es naheliegend, dass man die HTTP-Kommandos den SQL-Gegenstücken zuordnet. Einen solchen REST-Service, der die HTTP-Methoden als Datenbankoperationen durchreicht, bezeichnet man als *CRUD* (Create, Read, Update, Delete) oder *RUDI* (Read, Update, Delete, Insert).

HTTP	SQL
GET	SELECT
PUT	CREATE
POST	UDATE
DELETE	DELETE

Tabelle 30.1: Entsprechung der HTTP- und SQL-Befehle

Tatsächlich ist der REST-Service `gast` auch nichts anderes als ein CRUD für die Tabelle `Gast`. Das scheint zunächst sinnlos, schließlich kann man Datenbanken ja auch gleich per JDBC quer über das Netzwerk zugreifen. Der REST-Service hat aber entscheidende Vorteile:

- JDBC funktioniert nur für Java, nicht aber für andere Programmiersprachen.
- Einen REST-Server können Sie per Internet rund um die Welt ansprechen, wohingegen JDBC eher für lokale Netzwerke gedacht ist.

Sie merken schon, dass Sie viel programmieren müssen. Immerhin hilft Ihnen die Bibliothek Jersey dabei. Sie implementiert die Java API for RESTful Web Services (JAX-RS).

Projekterstellung

Unter Eclipse legen Sie ein neues dynamisches Web-Projekt an:

1. File|New|Other. In der Dialogbox wählen Sie Dynamic Web Project.
2. Als Name verwenden Sie `HeidiRest`.
3. Unter Configuration klicken Sie auf den Button Modify.
4. In dem erscheinenden Dialog finden Sie den Punkt JAX-RS (REST Web Services). Wenn die Version mindestens 2 ist, dann kreuzen Sie diesen Punkt an. Ansonsten binden Sie später die Jersey-JAR-Dateien von Hand ein, wie unten beschrieben.
5. Schließen Sie den Dialog mit OK.

6. Schließen Sie die Projekterstellung mit Finish.

Jersey-JARS einbinden

Sollte Ihre Eclipse-Version noch nicht die REST-Version 2 unterstützen, müssen Sie sich die passenden JAR-Dateien aus dem Internet herunterladen.

1. Surfen Sie zur URL `jersey.java.net/download.html`. Hier können Sie das Jersey JAX-RS-Bundle herunterladen. Es handelt sich um eine ZIP-Datei mit dem Namen *jaxrs-ri-XXXX.zip*. Dabei steht XXXX für die aktuelle Version. Sie enthält ein Verzeichnis *jaxrs-ri*, in dem sich, über mehrere Verzeichnisse (*api*, *lib* und *ext*) verteilt, diverse JAR-Dateien befinden.
2. Kopieren Sie die JAR-Dateien in das Verzeichnis *WebContent/WEB-INF/lib* Ihres Projekts.

Und wenn Sie sowieso gerade das Bibliotheksverzeichnis am Wickel haben, ist es kein dummer Gedanke, auch den JDBC-Treiber von PostgreSQL nach *WebContent/WEB-INF/lib* zu kopieren. Denn über kurz oder lang wird der REST-Service natürlich auf die Datenbank zugreifen wollen.

Grundgerüst für den Gast

Die Umgebung steht. Nun benötigen Sie eine Klasse, die sich um die Anfragen kümmert.

1. Klicken Sie das Projekt mit der rechten Maustaste an und wählen Sie New|Other.
2. Im Dialog können Sie unter Glassfish den Punkt Restful Web Service From Pattern anwählen. Klicken Sie auf Next.
3. Geben Sie als Package `gast` ein.
4. Geben Sie als Klassennamen `GastServer` ein.
5. Ändern Sie unter Path den Eintrag von `generic` auf `gast`.
6. Unter MIME-Type ist Application/xml vorgewählt.
7. Schließen Sie die Erstellung mit Finish ab.

Eclipse baut Ihnen einen REST-Service-Rahmen, der Anfragen der HTTP-Kommandos `GET` und `PUT` akzeptiert und als Datenformat XML verarbeitet. Das folgende Listing wird erzeugt. Ich habe allerdings die Kommentare der Übersichtlichkeit halber entfernt.

```
package gast;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;

@Path("gast")
public class GastServer {
    @SuppressWarnings("unused")
    @Context
    private UriInfo context;

    public GastServer() {
    }

    @GET
    @Produces("application/xml")
    public String getXml() {
        throw new UnsupportedOperationException();
    }

    @PUT
    @Consumes("application/xml")
    public void putXml(String content) {
    }
}
```

Listing 30.34: Eclipse-REST-Service-Vorgabe

Das Listing enthält neben dem Konstruktor eine Methode, die auf `GET` reagiert und eine, die auf `PUT` anspricht. Beide Methoden arbeiten mit XML. XML ist schon deshalb interessant, weil auf diesem Wege beliebige Datenstrukturen zwischen Client und Server ausgetauscht werden können.



Wenn bei Ihnen die Importe rot werden, hat es wohl nicht so recht mit dem Einbinden der Jersey-JAR-Dateien geklappt.

Die Gast-Application

Nun erzeugen Sie eine weitere Klasse, die Sie bitte `GastApplication` nennen.

Diese Klasse benötigen wir vor allem, um die Annotation `@ApplicationPath` unterzubringen. Deren Inhalt wird bei der URL zwischen Projektname und Ressourcenname eingebaut.

```
package gast;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("fewo")
public class GastApplication extends Application {
}
```

Listing 30.35: Die Gast-Application

Vorbereitung für den Datentransfer

Mit XML ist es möglich, komplette Datenobjekte zu versenden. Diese Möglichkeit werden wir für die Klasse `Gast` nutzen. Darum ist es sinnvoll, nun die Klassen `Gast` (Listing 30.18) und `GastDao` (Listing 30.16) in das Package `gast` zu übernehmen.

HTML als Testhilfe

So toll XML auch ist, wir wollen erst einmal schauen, ob alles klappt. Um den Server möglichst einfach mit dem Browser zu testen, ergänzen Sie eine `GET`-Methode, die HTML produziert, und fügen diese in der Klasse `GastServer` ein.

```
@GET
@Produces("text/html")
public String getHtml() {
    return "<html><body><h1>HTML-REST</h1></body></html>";
}
```

Test über den Browser

Zum Start des REST prüfen wir, ob er auf eine HTML-Anfrage reagiert.

1. Klicken Sie das Projekt mit der rechten Maustaste an.
2. Wählen Sie Run AS|Run ON Server.
3. Falls der Server nicht schon läuft, erscheint ein Dialog, der Ihnen die Serverumgebung anzeigt. Klicken Sie auf Next.
4. Der Folgedialog zeigt Ihnen, welche Projekte automatisch gestartet werden. Hier können Sie bei Bedarf alte Projekte herausnehmen.
5. Klicken Sie auf Finish.

Nach dem Start wird Eclipse automatisch einen Minibrowser öffnen, aber sofort

einen Fehler bringen, weil die Adresse vermutlich nicht korrekt ist. Korrigieren Sie das Adressfeld zu folgender URL:

`http://localhost:8080/HeidiRest/fewo/gast`

Der Minibrowser sollte nun wie in Abbildung 30.5 aussehen.



Abbildung 30.5: Der REST-Service hat reagiert.

Die URL ist nach folgendem Muster aufgebaut:

- **localhost**: ist der Name oder die IP-Adresse des Computers im Netzwerk, der den Application Server mit dem Dienst verwaltet. Der lokale Rechner heißt immer `localhost`.
- **8080**: Das ist der Port, auf dem der Dienst angeboten wird. Da 80 oft bereits durch einen Webserver belegt ist, verwenden die Application Server gern 8080.
- **HeidiRest**: Dieser Teil des Namens ergibt sich aus dem Projekt.
- **fewo**: Hierin erkennen Sie den `@ApplicationPath` Ihrer `GastApplication`. Sie können den `@ApplicationPath` auch mit einem leeren String versehen, um den Pfad um ein Element zu verkürzen.
- **gast**: Der letzte Teil der Adresse ergibt sich aus der Annotation `@Path` in der Serverklasse.

Gast-Daten per HTML

Nun versuchen wir, Daten zum REST-Service zu schaffen. Das funktioniert mit einem HTML-Formular etwas einfacher als die Erstellung eines XML-Clients. Darum erstellen wir eine kleine HTML-Seite, auf der wir zwei Formulare vereinigen. Das eine verwendet die `GET`-Methode, um Daten anzufordern, und das andere Formular soll die Daten eines Gastes erfassen, um es per `POST` an den Server zu senden.

Klicken Sie das Projekt mit der rechten Maustaste an und wählen Sie New|HTML-File. Geben Sie als Namen *index.html* an. Eclipse erstellt wieder einen Rahmen, den Sie zwischen den `body`-Tags ergänzen:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta content="text/html; charset=UTF-8" />
</head>
<body>
<h1>Gästeliste</h1>
  <form action="fewo/gast" method="get">
    <table>
    <tr><td>Name</td>
    <td><input type="text" name="name"/></td></tr>
    </table>
  <input type="submit" value="Anzeigen">
  </form>
<h1>Gast anlegen</h1>
  <form action="fewo/gast" method="post">
    <table>
    <tr><td>Name</td>
       <td><input type="text" name="name"/></td></tr>
    <tr><td>Adresse</td>
       <td><input type="text" name="adresse"/></td></tr>
    <tr><td>Ort</td>
       <td><input type="text" name="ort"/></td></tr>
    <tr><td>E-Mail</td>
       <td><input type="text" name="email"/></td></tr>
    <tr><td>Telefon</td>
       <td><input type="text" name="telefon"/></td></tr>
    </table>
  <input type="submit" value="Einfügen">
  </form>
</body>
</html>
```

Listing 30.36: HTML-Seite zum Testen des REST

Die Datei befindet sich im Projektverzeichnis *WebContent*. Klicken Sie sie mit der rechten Maustaste an und wählen Sie Run AS|Run ON Server. Es sollte nun eine Eingabemaske für eine Gästeliste und für das Anlegen eines Gastes erscheinen.

Wenn Sie den oberen Button Anzeigen anklicken, erhalten Sie die Meldung aus Abbildung 30.5. Wenn Sie den zweiten Button Einfügen anklicken, erhalten Sie eine Fehlermeldung mit dem HTTP-Status 405, der besagt, dass diese Methode für diesen Server nicht erlaubt sei. Das ist richtig, denn eine `POST`-Methode, die auf

HTML reagiert, hat unser Server noch nicht.

Der Service bedient HTML-POST-Anfragen

Die Methode `postHtml` übernehmen Sie bitte in die Klasse `GastServer`.

```
@POST
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
public Response postHtml(
        @FormParam("name") String name,
        @FormParam("adresse") String adresse,
        @FormParam("ort") String ort,
        @FormParam("email") String email,
        @FormParam("telefon") String telefon) {
    if (name == null) {
        throw new WebApplicationException(
                Response.status(400).entity
                ("Could not be created.").build());
    }
    Gast gast = new Gast();
    gast.setName(name);
    gast.setAdresse(adresse);
    gast.setOrt(ort);
    gast.setEmail(email);
    gast.setTelefon(telefon);
    long pos = -1;
    pos = GastDao.getInstance().addGast(gast);
    URI location = context.getAbsolutePathBuilder()
            .path("" + pos).build();
    return Response.created(location).build();
}
```

Listing 30.37: POST-Methode für HTML-Formulare

Nach der Annotation `@POST` wird mit `@Consumes` der MIME-Typ für HTML-Formulare festgelegt. Dadurch ist es möglich, über `@FormParam` die Inhalte der Formular-Kontrollelemente in Parametervariablen zu überführen.

Es werden Ihnen ein paar Importe fehlen, die Sie mit der Tastenkombination Strg + ⇧ + O automatisch ergänzen können. Die Importliste sollte schließlich folgendermaßen aussehen:

```
import java.net.URI;
import javax.ws.rs.Consumes;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;
```



Sollten Sie jetzt noch Fehler bei Verwendung der Klassen `Gast` und `GastDao` vorfinden, haben Sie diese noch nicht in das Package `gast` übernommen.

Wenn Sie nun die Datei *index.html* mit Run AS|Run ON Server starten, sollte es möglich sein, Daten zu erfassen, die dann in der Datenbank hinterlegt werden. Wenn keine Fehlermeldung auftaucht, ist das ein gutes Zeichen. Aber im nächsten Schritt werden wir mal nachschauen, ob sie auch wirklich angekommen sind.

Die Parameter der HTML-GET-Methode

Die HTML-GET-Methode wirft uns bisher nur einen kurzen Gruß vor die Füße. Aber sie sollte lieber Daten liefern. Eine Gästeliste wäre nicht schlecht. `GastDao` ist bereits darauf vorbereitet, Gäste nach dem Namen zu filtern und das HTML-Formular bietet für den Namen ein Eingabefeld.

Ein `GET`-Aufruf verwendet die URL, um Parameter zu übergeben. Sie werden durch ein kaufmännisches Und (&) von der eigentlichen URL getrennt. Darum wird statt `@FormParam` eine andere Annotation für das Auslesen der Parameter gewählt, nämlich `@QueryParam`.

```
localhost:8080/HeidiRest/fewo/gast?name=Meier
```

Die Methode `getNameHtml` holt sich eine Liste von Gästen aus der `GastDao` und erzeugt daraus eine HTML-Tabelle, die als String an den Aufrufer zurückgegeben wird. Sie ersetzt die bisherige Methode `getHtml`, die konsequenterweise entfernt werden muss.

```
import java.util.List;
import javax.ws.rs.QueryParam;
// ...
@GET
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
public String getNameHtml(@QueryParam("name") String name) {
    String str = "<html><body><h1>G&auml;steliste</h1>";
    str += "<table br=1>";
    if (name == null) {
        name = "";
    }
    List<Gast> liste = GastDao.getInstance().getGast(name);
    for (Gast gast : liste) {
        str += "<tr><td>" + gast.getId();
        str += "</td><td>" + gast.getName();
        str += "</td><td>" + gast.getAdresse();
        str += "</td><td>" + gast.getOrt();
        str += "</td><td>" + gast.getEmail();
        str += "</td><td>" + gast.getTelefon();
        str += "</td></tr>";
    }
    str += "</table></body></html>";
    return str;
}
```

Listing 30.38: GET holt den Parameter per `@QueryParam`.

Wenn Sie nun wieder die Datei *index.html* über die rechte Maustaste mit Run AS|Run ON Server starten, können Sie Anzeigen anklicken und erhalten eine Liste aller Gäste, die sich in der Datenbank tummeln.

Einzelabfrage per Pfad

Ein HTTP-Parameter kann aber auch durch künstliche Verlängerung des URL-Pfades übergeben werden. Dazu wird hängt man als Pseudoverzeichnis zunächst den Variablennamen und dann in einem weiteren Pseudoverzeichnis den gewünschten Inhalt an. Beispielsweise würde die folgende URL den Gast mit der `id` 4711 ansprechen:

```
localhost:8080/HeidiRest/fewo/gast/id/4711
```

Aufgefangen wird dieser Zugriff durch die Methode `getOneHtml`. Dazu erhält sie eine Annotation `@Path`, die ihr mitteilt, dass dem Pfadnamen *id* ein Parameter folgt. Dieser Parameter kann dann über die Annotation `@PathParam` und den Namen `id` schließlich zum Methodenparameter `id` vom Typ `long` gemacht werden.

Da die Methode über die Parameter anders angesteuert wird, kann sie parallel zu `getNameHtml` im `GastServer` eingebaut werden.

Die Methode holt sich über `GastDao` den Gast und gibt ihn als HTML-Seite an den Aufrufer zurück. Ausprobieren können Sie das, indem Sie einfach die URL von oben in die Adressleiste des Browsers eintippen und statt 4711 eine Nummer verwenden, die in Ihrer Datenbank vorkommt.

```
import javax.ws.rs.NotFoundException;
import javax.ws.rs.PathParam;
// ...
@GET
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
@Produces("text/html")
@Path("id/{id}")
public String getOneHtml(@PathParam("id") long id) {
    Gast gast = GastDao.getInstance().getGast(id);
    if (gast == null)
        throw new NotFoundException();
    String str = "<html><body><h1>Gast</h1>";
    str += " " + gast.getId();
    str += " " + gast.getName();
    str += " " + gast.getAdresse();
    str += " " + gast.getOrt();
    str += " " + gast.getEmail();
    str += " " + gast.getTelefon();
    str += "</body></html>";
    return str;
}
```

Listing 30.39: `GET` holt den Parameter über den Pfad.

Wenn Sie sich bei Ausgabe der Gästeliste eine Nummer merken, können Sie im Browser das Adressfeld entsprechend ändern und so die Methode testen.

Gäste löschen

Auf die gleiche Art und Weise kann der Parameter übergeben werden, um einen Gast zu löschen. Allerdings können wir das mit dem Browser nicht so einfach ansprechen. Das Testen verschieben wir also auf später.

```
import javax.ws.rs.DELETE;
// ...
@DELETE
@Path("id/{id}")
public Response deleteGast(@PathParam("id") Long id) {
    if (GastDao.getInstance().deleteGast(id)) {
        return Response.accepted().build();
    }
    throw new NotFoundException();
}
```

Listing 30.40: Die `DELETE`-Methode

Datenübertragung per XML

Heidi kann schon einmal Gäste über den Browser anlegen und speichern. Aber eigentlich wollte sie ja mit Java-Clients auf die Gäste losgehen. Diese sollen dann gern mit Objekten hantieren und nicht die Daten aus jeder Menge HTML herausfummeln.

REST bietet dazu XML und JSON an. Die Vorgehensweise unterscheidet sich nicht erheblich. Wir bauen den Rahmen aus, den uns Eclipse in Listing 30.34 erzeugt hat. Wir ersetzen die vorgegebenen Methoden `getXml` und `putXml` durch eigene. Darum ist es am sinnvollsten, die automatisch erzeugten zunächst zu löschen. Stattdessen bauen wir folgende Methoden ein:

- Die Methode `getSomeXml` liefert eine Liste, die durch den Namen gefiltert werden kann. Hier wird */name/Schmidt* an die URL gehängt, um alle Gäste mit Namen Schmidt zu erhalten.
- Die Methode `getOneXml` liefert genau einen Gast, der über seine ID identifiziert wird. Die Parameterübergabe funktioniert über den Pfad. Um den zweiten Eintrag zu bekommen, wird */id/2* an die URL angehängt.

Die Methoden verwenden die bereits bei den HTML-Methoden eingesetzten Methoden `getGast` der Klasse `GastDao`. Allerdings werden bei den XML einfach Objekte der Klasse `Gast` als Parameter und Rückgabewerte verwendet. Mit einem kleinen Kniff wird der Appication Server die Umwandlung zwischen Objekt und XML für uns übernehmen. Sie bauen nun folgende Methoden ein:

```
@GET
@Produces(MediaType.APPLICATION_XML)
@Path("name/{name}")
public List<Gast> getSomeXml(@PathParam("name") String name) {
    List<Gast> liste = GastDao.getInstance().getGast(name);
    if (liste.isEmpty())
        throw new NotFoundException();
    return liste;
}

@GET
@Produces(MediaType.APPLICATION_XML)
@Path("id/{id}")
public Gast getOneXml(@PathParam("id") long id) {
    Gast gast = GastDao.getInstance().getGast(id);
    if (gast == null)
        throw new NotFoundException();
    return gast;
}
```

Listing 30.41: GET-Methoden für XML-Anfragen

Die Methode `putGast` reagiert auf `PUT` und legt einen Gast an. Die eigentliche Arbeit wird wieder `GastDao` überlassen. Auch zum Anlegen des Gastes wird das Objekt einfach als Parameter übernommen. Nur die Annotationen deuten darauf hin, dass mit XML gearbeitet wird.

```
@PUT
@Consumes(MediaType.APPLICATION_XML)
@Produces(MediaType.APPLICATION_XML)
public Response putGast(Gast gast) {
    long pos = GastDao.getInstance().addGast(gast);
    if (pos < 0) {
        throw new WebApplicationException(Response.status(400)
                .entity("Not created.").build());
    }
    gast.setId(pos); // uebernehme die erzeugte ID
    URI uri = context.getAbsolutePathBuilder()
            .path(Long.toString(pos)).build();
    return Response.created(uri).entity(gast).build();
}
```

Die Konvertierung von XML zum Objekt und umgekehrt müssen Sie nicht explizit programmieren, sondern sie wird automatisch übernommen, wenn Sie die Klasse `Gast` ein wenig darauf vorbereiten.

```
package gast;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Gast {
    private long id=0;
    private String name, adresse, ort, email, telefon;
    // ...
}
```

Zugriff per REST-Client

Für einen Client setzen Sie ein neues Java-Projekt auf. Dort werden wir einen kleinen HTTP-Client per `HttpURLConnection` und einen Jersey-Client implementieren.

1. Über das Hauptmenü wählen Sie File|New|Project.
2. Im Dialog finden Sie Java Project und klicken auf Next.
3. Als Projektnamen wählen Sie `HeidiRestClient` und bestätigen mit Finish.
4. Die Anfrage, ob Sie die Perspektive ändern wollen, beantworten Sie mit NO.

REST-Client per HttpUrl

Prinzipiell ist ein REST-Service so gearbeitet, dass alle mit ihm spielen können, die Web-Standards wie HTTP und XML beherrschen. Um das zu demonstrieren, verwenden Sie ein normales Java-Programm. Obwohl es sogar möglich wäre, mit Sockets auf REST loszugehen, verwenden wir doch die Http-Bibliotheken. Es geht hier ja nicht um Selbstzüchtigung, sondern darum, zu zeigen, dass es funktioniert.

1. Klicken Sie das Projekt mit der rechten Maustaste an und wählen Sie New|Class.
2. Als Klassenname verwenden Sie `RestClientHttpUrl`.
3. Schließen Sie mit Finish.

Das folgende Programm läuft allein mit der Standardbibliothek der Java Standard Edition. Allerdings wertet es die XML-Daten nicht aus. Dazu müssten Sie Kapitel 25 bemühen.

In Kapitel 29 wird der Aufbau einer HTTP-Verbindung ausführlicher beschrieben. Als Beispiel werden hier zwei unterschiedliche `GET`-Anfragen und eine `DELETE`-Anfrage ausgeführt.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class RestClientHttpUrl {

    public static String get(String url) {
        String rueckgabe="";
        try {
            HttpURLConnection verbind = (HttpURLConnection)
                    new URL(url).openConnection();
            verbind.setDoInput(true);
            verbind.setRequestProperty("Content-Type",
                    "application/xml; charset=UTF-8");
            verbind.setRequestMethod("GET");
            InputStream inStream = verbind.getInputStream();
            BufferedReader in = new BufferedReader(
                    new InputStreamReader(inStream));
            String empfangeneZeile;
            StringBuffer antwort = new StringBuffer();

            while ((empfangeneZeile = in.readLine()) != null) {
                antwort.append(empfangeneZeile + "\n");
            }
            in.close();
            rueckgabe = antwort.toString();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return rueckgabe;
    }
    static boolean delete(String url, long id) {
        try {
            HttpURLConnection verbind = (HttpURLConnection)
                    new URL(url+"/id/"+id).openConnection();
            verbind.setRequestMethod("DELETE");
            verbind.setDoOutput(true);
            verbind.setDoInput(true);
            return verbind.getResponseCode()
                    == HttpURLConnection.HTTP_ACCEPTED;
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
        return false;
    }

    public static void main(String[] args) {
        String urlStr;
        urlStr = "http://localhost:8080/HeidiRest/fewo/gast";
        System.out.println(get(urlStr+"/id/2"));
        System.out.println(get(urlStr+"/name/Schmidt"));
        urlStr = "http://localhost:8080/HeidiRest/fewo/gast";
        delete(urlStr, 103);
    }
}
```

Listing 30.42: Ein einfacher REST-Client ohne Jersey

Die Methode `main` liest zunächst den Gast mit der ID 2 und dann alle Gäste mit dem Namen Schmidt aus. Wenn in Ihrem Datenbestand keine 2 und kein Schmidt ist, erhalten Sie eine entsprechende Exception. Ansonsten erscheinen auf dem Bildschirm die XML-Strings. Zuletzt wird der Gast mit der Nummer 103 gelöscht. Auch hier müssen Sie die Nummer auf Ihre Datenbanksituation anpassen.

REST per Jersey-Client

Wenn Sie Jersey in Ihren Client einbinden, haben Sie den Vorteil, dass das Programm kürzer ist und die XML-Umwandlung gleich an Bord ist. Dafür müssen Sie all die Jersey-JARs einbinden, so wie Sie es beim REST-Server bereits getan haben. Allerdings müssen Sie bei einem Standard-Java-Projekt die JAR-Dateien auch noch in den Class-Builder-Path setzen.

1. Legen Sie mit File|New|Project und Java Project ein normales Java-Projekt an.
2. Weiter geht es mit Next.
3. Als Namen verwenden Sie `HeidiJerseyClient`.
4. Mit Finish wird das Projekt erstellt.
5. Klicken Sie das Projekt mit der rechten Maustaste an und wählen Sie New|Folder.
6. Als Folder Name geben Sie *lib* ein und bestätigen mit Finish.
7. Klicken Sie auf das Projektverzeichnis *lib*, damit es sich öffnet.
8. Selektieren Sie alle JAR-Dateien. Dann klicken Sie eine davon mit der rechten Maustaste an und wählen Build Path|Add TO Build Path. Nun erscheinen die JAR-Dateien auch unter *Referenced Libraries*.
9. Erzeugen Sie eine neue Klasse, indem Sie das Projekt mit der rechten Maustaste anklicken und New|Class wählen.

10. Als Name geben Sie `RestClientJersey` an und bestätigen mit Finish.

```
import java.util.List;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.GenericType;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.UriBuilder;

import org.glassfish.jersey.client.ClientConfig;

public class RestClientJersey {
    static final String HOST = "http://localhost:8080";
    static final String SERVICE = "HeidiRest";
    static final String APPLICATION = "fewo";
    static final String SERVER = "gast";

    static private Gast getOneXml(WebTarget target, long id) {
        String server = SERVER + "/id/" + id;
        System.out.println(server);
        Gast antwort = target.path(APPLICATION).path(server)
                .request().accept(MediaType.APPLICATION_XML)
                .get(Gast.class);
        return antwort;
    }

    static private List<Gast> getSomeXml(WebTarget target,
 String name) {
        String server = SERVER;
        if (!name.isEmpty()) {
            server += "/name/"+name;
        }
        System.out.println(server);
        List<Gast> antwort = target.path(APPLICATION).path(server)
                .request().accept(MediaType.APPLICATION_XML)
                .get(new GenericType<List<Gast>>() {});
        return antwort;
    }

    public static void main(String[] args) {
        ClientConfig config = new ClientConfig();
        Client client = ClientBuilder.newClient(config);
        WebTarget target = client.target(
                UriBuilder.fromUri(HOST+"/"+SERVICE).build());

        Gast einer = getOneXml(target, 2);
           System.out.println(einer.getEmail());
```

```
        List<Gast> einige = getSomeXml(target, "Arnold");
        System.out.println("some");
        for (Gast gast : einige) {
            System.out.println(gast.getEmail());
        }
    }
}
```

Listing 30.43: Ein REST-Client

Kopieren Sie die Datei *Gast.java* aus dem Projekt *HeidiRest*. Eventuell müssen Sie das Package anpassen. Sofern Ihr REST-Service noch läuft, können Sie den Client starten. Da er Jersey benutzt, braucht der Client sich nicht um XML zu kümmern, sondern kann einfach die Objekte in Empfang nehmen.

Ein neuer Gast

Für den Aufruf mit der `PUT`-Methode schreiben Sie im Jersey-Client eine neue Methode `putXml`. Sie funktioniert analog zu den `GET`-Methoden, nur dass diesmal die Daten hineingehen statt als Rückgabewert zurückgehen.

```
static private long putXml(WebTarget target, Gast gast) {
    Response response = target.path(APPLICATION).path(SERVER)
            .request(MediaType.APPLICATION_XML)
            .put(Entity.entity(gast, MediaType.APPLICATION_XML),
            Response.class);
    return response.getStatus();
}
```

In der `main`-Methode erzeugen Sie zum Test ein Objekt von `Gast` und übergeben es vertrauensvoll an `putXml`.

```
Gast neugast = new Gast();
neugast.setName("Jersey Client");
neugast.setOrt("Jersey Ort");
long status = putXml(target, neugast);
System.out.println("Status post: "+status);
```

Die REST-Klasse BuchServer

Die REST-Server-Klasse `BuchServer` enthält nur noch Methoden, die XML bedienen. Eine von beiden ermöglicht das Auslesen der Liste aller Buchungen. Die andere erstellt eine neue Buchung. Dabei bedient sie sich der Methode `addBuchung` aus `BuchungDao`.

Die `PUT`-Methode verwendet sowohl die Annotation `@Produces` als auch `@Consumes`, da sie das gespeicherte Buchungsobjekt wieder zurückliefert. Das ist vor allem für die neu angelegte ID für den Aufrufer interessant.

```java
package buchung;

import java.net.URI;
import java.util.List;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;

@Path("buchung")
public class BuchServer {

    @GET
    @Path("all")
    @Produces(MediaType.APPLICATION_XML)
    public List<Buchung> getAllXml() {
        System.out.println("getAll");
        List<Buchung> list = BuchungDao.getInstance().getBuchung();
        return list;
    }

    @Context
    UriInfo context;

    @PUT
    @Consumes(MediaType.APPLICATION_XML)
    @Produces(MediaType.APPLICATION_XML)
    public Response putBuchung(Buchung buchung) {
        Long pos = BuchungDao.getInstance().addBuchung(buchung);
        if (pos < 0) {
            throw new WebApplicationException(Response.status(400)
                    .entity("Not created.").build());
        }
        buchung.setId(pos); // übernehme die erzeugte ID
        URI uri = context.getAbsolutePathBuilder()
                        .path(Long.toString(pos)).build();
        return Response.created(uri).entity(buchung).build();
    }
```

```
}
```

Listing 30.44: BuchServer definiert die möglichen Zugriffe.

BuchApplication

Die Application-Klasse ist wieder kurz und schmerzlos. Sie dient uns zur Festlegung des Pfades in der URL.

```
package buchung;
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("buch")
public class BuchApplication extends Application {
}
```

Listing 30.45: Die Application-Klasse definiert den Pfad.

Buchung als XmlRoot

Damit die Umwandlung zwischen Klasse und XML durch den Application Server übernommen wird, muss die Klasse `Buchung` noch als `@XmlRoot` gekennzeichnet werden.

```
import java.io.Serializable;
import java.util.Date;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Buchung implements Serializable {
    private long id = 0;
    private long gastid;
    private Date von, bis;
    private String wohnung;
    // ...
}
```

Listing 30.46: Buchung mit `@XmlRoot`

Der Client bucht

Als Client sehen Sie hier noch einmal die Jersey-Variante, da sie einfach kürzer und etwas übersichtlicher ist. Wie schon beim Gast ist es natürlich eine reine Fleißaufgabe, den Client auf `HttpURLConnection` umzustellen, falls der Client in einer Umgebung laufen soll, in der Jersey nicht verfügbar ist.

Analog zum Gast-Client wird für den Client ein neues einfaches Java-Projekt angelegt. Da Sie hier Jersey verwenden, müssen Sie auch dessen JAR-Dateien in

das Projekt einbinden und in den Build-Path übernehmen, wie bereits beim Gast-Client gezeigt.

Der Client simuliert eine normale Buchung. Dazu wird zunächst der Gast angelegt. Dessen ID wird dann verwendet, um eine neue Buchung anzulegen und auch diese in die Datenbank zu schaffen. Zu guter Letzt werden alle Buchungen angezeigt.

```
import java.sql.Date;
import java.util.List;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.GenericType;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriBuilder;
import org.glassfish.jersey.client.ClientConfig;

public class RestBuchClient {
    static final String HOST = "http://localhost:8080";
    static final String SERVICE = "HeidiRest";
    static final String BUCHAPP = "buch";
    static final String GASTAPP = "fewo";
    static final String BUCHSRV = "buchung";
    static final String GASTSRV = "gast";

    static private List<Buchung> getAllBuchung(WebTarget target) {
        String server = BUCHSRV + "/all";
        List<Buchung> antwort = target.path(BUCHAPP)
                .path(server).request()
                .accept(MediaType.APPLICATION_XML)
                .get(new GenericType<List<Buchung>>() {
                });
        return antwort;
    }

    static private long putXmlGast(WebTarget target, Gast gast) {
        Response response = target.path(GASTAPP).path(GASTSRV)
                .request(MediaType.APPLICATION_XML)
```

```
                .put(Entity.entity(gast,
                MediaType.APPLICATION_XML));
        Gast neugast = response.readEntity(Gast.class);
        return neugast.getId();
    }

    static private long putXmlBuch(WebTarget target,
                                   Buchung buchung) {
        Response response = target.path(BUCHAPP).path(BUCHSRV)
                .request(MediaType.APPLICATION_XML)
                .put(Entity.entity(buchung,
                MediaType.APPLICATION_XML));
        Buchung buch = response.readEntity(Buchung.class);
        return buch.getId();
    }

    public static void main(String[] args) {
        ClientConfig config = new ClientConfig();
        Client client = ClientBuilder.newClient(config);
        WebTarget target = client.target(UriBuilder.fromUri(
                HOST + "/" + SERVICE).build());

        // Ein neuer Gast will die Grunzer Alm buchen
        Gast gast = new Gast();
        gast.setName("Erna Buch");
        gast.setEmail("erna.buch@gast.edu");
        long gastid = putXmlGast(target, gast);
        gast.setId(gastid);

        Buchung buchung = new Buchung();
        buchung.setGastid(gast.getId());
        buchung.setVon(Date.valueOf("2017-08-12"));
        buchung.setBis(Date.valueOf("2017-08-19"));
        buchung.setWohnung("Grunzer Alm DG");
        long buchstatus = putXmlBuch(target, buchung);

        System.out.println("Status post: " + buchstatus);
        System.out.println("Liste aller Buchungen");
        List<Buchung> antwort = getAllBuchung(target);
        for (Buchung buch : antwort) {
            System.out.println(buch.getWohnung()+" "
                    +buch.getVon()+" "+buch.getBis());
        }
    }
}
```

Listing 30.47: Der Jersey-Client legt einen Gast und dessen Buchung an.

Was denn noch?

Klara ist stolz auf das Geschaffte. Nun noch einen Client mit grafischer Oberfläche bauen und ihre Arbeit ist erledigt. Das denkt sie zumindest. Allerdings gibt Heidi zu bedenken, dass eine Buchung in mehreren Schritten erfolgt. Der Gast fragt an, dann bucht er und erst wenn er bestätigt, ist die Buchung vollzogen.

Dann fehlt der Preis. Der sollte sich vielleicht automatisch berechnen, indem man den Preis der Wohnung ermittelt. Also brauchen wir eine Tabelle für die Wohnung, die den Preis für Haupt- und Nebensaison enthält. Ach ja, die Buchung sollte auch notieren können, was bereits bezahlt ist.

Bei den Gästen benötigt man ein Bemerkungsfeld, damit man notieren kann, ob sie sich danebenbenommen haben. Und Heidi erzählt, dass sie dringend einen Belegungskalender und eine Kollisionserkennung bei Buchungen benötigt.

Klara sieht schon, hier kommt sie so schnell nicht weg. Und sie beginnt, Frankfurt zu vermissen, diesen ruhigen Ort, wo das Leben noch gemütlich ist.

SOAP: Ferngespräch in XML

Klara möchte Heidi auf jeden Fall noch einmal SOAP vorstellen. SOAP realisiert den Fernaufruf von Funktionen (Remote Procedure Call) typischerweise auf der Basis des Protokolls HTTP. Aber statt HTML-Texte zu senden und zu empfangen, verwendet man bei SOAP XML. SOAP hat schließlich nicht die Absicht, Browser ins Spiel zu bringen. Das Ziel ist es vielmehr, dass ein Service-Provider seine Klasse anbietet, deren Methoden per SOAP-Request über das Netzwerk von einem Service-Consumer aufgerufen werden. Natürlich spielen dabei Daten als Parameter und Rückgabewerte eine Rolle.

Heidi meint, das erinnere etwas an RMI. Klara bestätigt, dass es eben auch eine Art des Remote Procedure Calls sei. Während RMI aber für das lokale Netzwerk gedacht sei und voraussetze, dass Client und Server Java sprechen, ist SOAP auf der Basis von HTTP und XML sehr viel offener. Selbst das Interface zwischen Client und Server wird in XML hinterlegt.

Um den Charakter als Fernaufruf einer Methode deutlich zu machen, erstellt Klara einen Server, der einen Dienst anbietet, zwei Zahlen miteinander zu addieren. Heidi versucht, beeindruckt auszusehen.

Der Standard für SOAP heißt JAX-WS und wird aus Server-Seite durch den Application Server (Glassfish) oder den Webcontainer (Tomcat) implementiert. Der Client muss entsprechende Bibliotheken einbinden. Glücklicherweise können Sie dies alles Eclipse überlassen.

WSDL: Web Service Description Language

Consumer und Producer müssen sich darüber einigen, wie die Schnittstelle genau aussieht, deren Methoden aufgerufen werden können. Diese wird in WSDL (Web Service Description Language) beschrieben. Auch hier haben Sie das Glück, dass Eclipse eine solche Datei aus einer Klasse generieren kann. Ihre Aufgabe besteht also zunächst darin, eine solche Klasse zu schreiben.

Der Service-Provider

Freundlicherweise unterstützt Eclipse die Erstellung eines SOAP-Service-Providers. Mit folgenden Schritten gelangen Sie zu Ihrem Provider:

1. Erzeugen Sie ein Dynamic Web Project und nennen Sie es beispielsweise `Soap1`.
2. Der Web-Service-Generator wird unwirsch, wenn es kein Package für die Klasse gibt. Also sollten Sie zunächst eines anlegen.
3. Erzeugen Sie anschließend eine einfache Java-Klasse, die die Schnittstelle des Servers bilden soll. Im Beispiel heißt diese `Calc`:

```
package rechne;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
public class Calc {

    @WebMethod
    public int addiere(int a, int b) {
        return a + b;
    }
}
```

   Diese Klasse bildet die Server-Schnittstelle. Darum wurde die Klasse mit der Annotation `@WebService` und die Methode mit `@WebMethod` gekennzeichnet.
4. Im nächsten Schritt erzeugen wir aus dieser Klasse einen Webservice. Dazu klicken Sie mit der rechten Maustaste auf das Projekt und wählen New|Other. Es erscheint ein Dialog. Dort finden Sie unter Web Services den Punkt Web Service. Mit Next kommen Sie weiter.
5. Abbildung 30.6 zeigt den Folgedialog, in dem Sie als Service Implementation Ihre Klasse `Calc` eintragen. So finden Sie die korrekte Bezeichnung am einfachsten:
   a. Klicken Sie auf den Button Browse.

b. Es erscheint eine lange Liste von Kandidaten.
c. Geben Sie den Namen der Klasse, also `Calc`, in das Suchfeld ein.
d. Wählen Sie Ihre Klasse und klicken Sie auf OK.

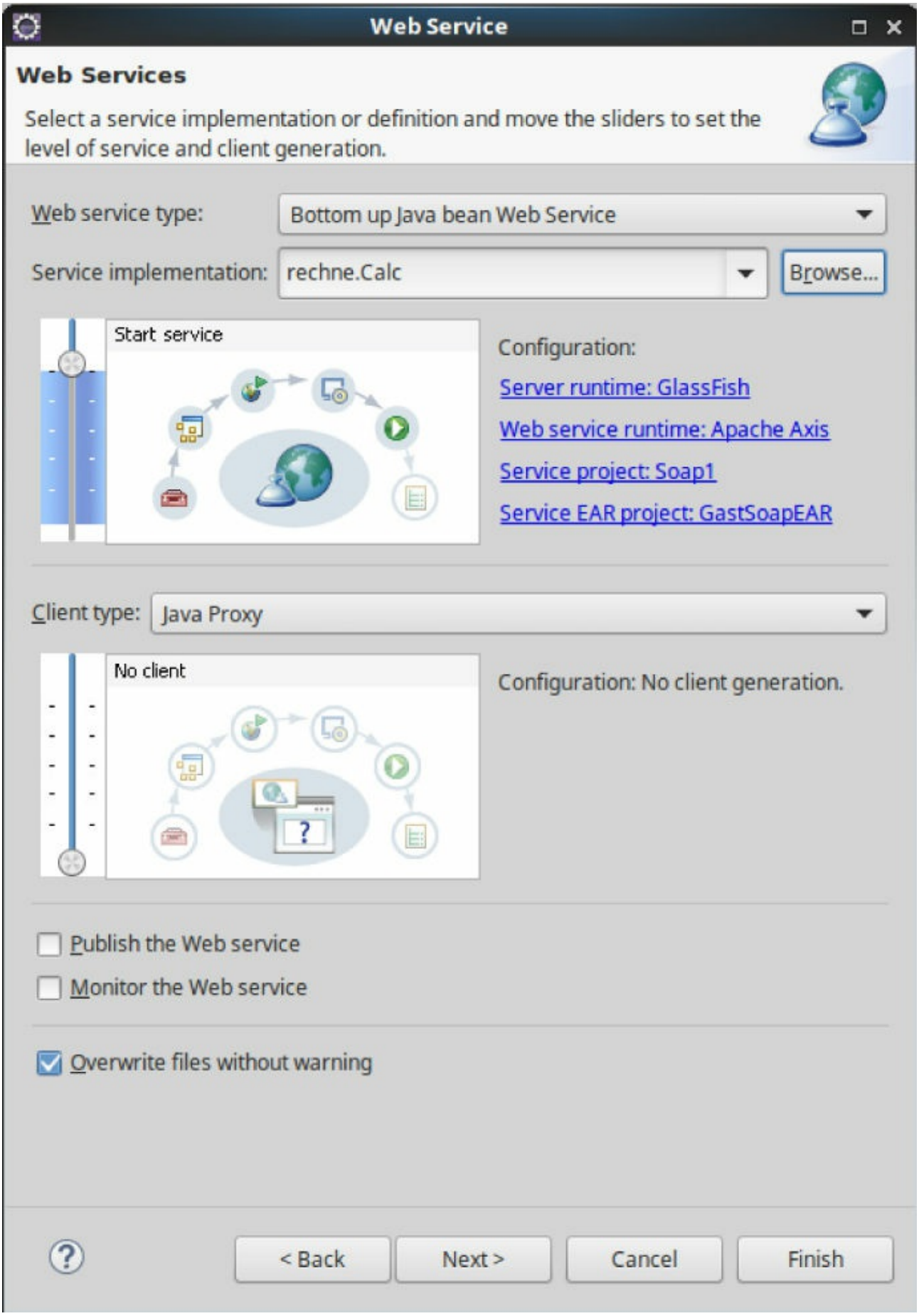


Abbildung 30.6: Einstellen des Web Service

6. Klicken Sie auf Finish und Eclipse amüsiert sich ein wenig ohne Sie.

Eclipse hat aus der Klasse die Datei *Calc.wsdl* erzeugt, die Sie im Projektpfad

WebContent/wsdl finden. Diese Datei ist quasi der Vertrag, den der Provider dem Consumer anbietet. Im nächsten Abschnitt werden Sie auf der Basis dieser Datei Ihren Client bauen.

Der Service-Consumer

Der Server steht, nun soll ein Java-Client erstellt werden. Ein solcher Service-Consumer benötigt natürlich keinen Application Server, sondern er ist nichts anderes als ein normales Java-Programm. Allerdings muss er, damit er sich mit dem SOAP-Provider verbinden kann, an dessen WSDL-Vertrag binden.

Um einen solchen Client zu erzeugen, erstellen Sie ein einfaches Java-Projekt. Eclipse unterstützt Sie darin, dass es innerhalb des Projekts den Rahmen für einen Web-Service-Client erzeugen kann. Auf diese Weise ersparen Sie sich den Umgang mit dem HTTP und der lästigen XML-Analyse. Stattdessen rufen Sie den Server auf die gleiche Weise auf, wie Sie eine gewöhnliche Methode aufrufen würden.

1. Sie legen ein gewöhnliches Java-Projekt über File|New Java-Project an. Sie werden nach einem Namen gefragt, wie beispielsweise `Soap1Client`.
2. Schließen Sie das Projekt mit Finish ab.
3. Nun erstellen Sie den Quelltextrahmen für den Web-Service-Client. Dazu klicken Sie das Projekt mit der rechten Maustaste an und wählen New|Other. Dort finden Sie unter Web Services den Punkt Web Service Client. Wählen Sie ihn an und klicken Sie auf Next. Der Dialog sieht sehr ähnlich aus wie in Abbildung 30.6, enhält aber dieses Mal nur den Client-Typ.
4. Unter Service Definition muss der Server eingetragen werden, auf den Sie zugreifen wollen. Dazu will Eclipse die WSDL-Datei lesen. Um die WSDL-Datei zu finden, klicken Sie auf Browse.
5. Es erscheint ein Dialog, der Ihnen die Webservice-Projekte der Arbeitsumgebung von Eclipse anzeigt. Sie klappen Ihren Server – hier `Soap1` – auf und wählen unter WebContent|Wsdl die Datei *Calc.wsdl* an, die die Informationen für die Serverklasse `Calc` enthält.
6. Sie bestätigen sowohl die Wahl mit OK als auch den vorherigen Dialog mit OK und landen wieder im Dialog Web Service Client.
7. Schließen Sie den Dialog mit Finish. Daraufhin erstellt Eclipse eine Schnittstelle zum Webservice. Sie finden die entsprechenden Java-Klassen im Ordner *src*.
8. Für das Rahmenprogramm des eigentlichen Clients erstellen Sie eine normale Java-Klasse, indem Sie das Projekt mit der rechten Maustaste anklicken und New|Class anwählen. Geben Sie hier als Namen beispielsweise `MeinClient` an.

```
package rechne;

import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;

public class MeinClient {

    public static void main(String[] args)
            throws ServiceException, RemoteException {
        CalcServiceLocator locator = new CalcServiceLocator();
        Calc calc = locator.getCalc();
        int c = calc.addiere(4, 5);
        System.out.println(c);
    }
}
```

9. Vergewissern Sie sich, dass der Server für Soap1 gestartet ist.
10. Klicken Sie die Datei *MeinClient.java* im Projekt `Soap1Client` mit der rechten Maustaste an.
11. Wählen Sie Run AS|Java Application.
12. In der Konsole erscheit eine 9.

Nun haben Sie zwei Zahlen über das Netzwerk an einen Server gesandt, der beide miteinander verrechnet und das Ergebnis wieder zurück an den Client geschickt hat. Dass das Ergebnis auch noch stimmt, ist also fast Nebensache.



In der Klasse `CalcServiceLocator` finden Sie übrigens die URL Ihres Webservice in der Variablen `Calc_address`. Wenn Sie den Server über das Netzwerk ansprechen wollen, muss diese Adresse natürlich so angepasst werden, dass sie nicht mehr `localhost`, sondern den Namen des Servers enthält.

Objekte als Parameter

Einfach nur zwei Zahlen zu addieren, ist weniger spannend als die Frage, wie man komplexe Objekte über das Netzwerk schicken kann. SOAP ist ein prozeduraler Dienst. Sein Zweck ist die Ausführung von Aktionen auf einem fremden Rechner. Aber als Parameter oder als Rückgabewert sollen natürlich auch komplexe Daten übertragen werden können, beispielsweise ein Gast. Nun wird Heidi wieder wach.

Der nächste SOAP-Server soll einem Gast beim Umzug helfen. Dazu ändert er einfach dessen Ort. Wenn das gelingt, kann man den Service später ausbauen. Erstellen Sie ein dynamisches Web und legen Sie darin die Klasse `Umzug` in einem

Package `gast` an. Und dann klauen Sie sich aus einem der anderen Projekte eine Klasse `Gast`.

```
package gast;

public class Gast {
    private long id = 0;
    private String name, adresse, ort, email, telefon;
    // ...
```

Listing 30.48: Die Klasse `Gast`

In der neuen Klasse `Umzug` lassen wir den Gast umziehen.

```
package gast;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
public class Umzug {

    @WebMethod
    public Gast ziehe(Gast gast, String ort) {
        gast.setOrt(ort);
        return gast;
    }
}
```

Listing 30.49: Die Klasse `Umzug`

Auch hier erzeugen wir über New|Other|Web Services|Web Service eine WSDL für die Klasse `Umzug`. Zum Test erzeugen wir wieder einen SOAP-Client. Auch hier erstellen Sie eine Klasse `MeinClient`. Diese wird im Package `gast` angelegt werden. Dort finden Sie eine Klasse `Umzug`. Sie ist bei der Übernahme der WSDL automatisch übernommen worden.

```
package gast;

import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;

public class MeinClient {

    public static void main(String[] args)
            throws RemoteException, ServiceException {
        UmzugServiceLocator locator = new UmzugServiceLocator();
        Umzug umzug = locator.getUmzug();
        Gast gast = new Gast();
        gast.setOrt("Gintoft");
        Gast neugast = umzug.ziehe(gast, "Munkbrarup");
        System.out.println(gast.getOrt());
        System.out.println(neugast.getOrt());
    }
}
```

Listing 30.50: Der Client

SOAP und JPA

Im dritten Schritt soll das Gast-Objekt in die Datenbank geschoben werden. Ein neues Projekt muss her. Das muss sowohl SOAP entsprechen als auch JPA können.

1. Über File|New|Dynamic Web Project erzeugen Sie ein Projekt.
2. Also Name verwenden Sie `Soap3`.
3. Über Modify fügen Sie JPA hinzu.
4. Mit Next und Next kommen Sie zu der Seite, auf der Sie die JPA Facet konfigurieren. Wählen Sie unter Connection eine Verbindung zu der PostgreSQL-Datenbank `fewo` oder erstellen Sie die Verbindung neu.
5. Erstellen Sie die Entity `Gast` aus der Tabelle, indem Sie das Projekt mit der rechten Maustaste anklicken und JPA Tools|Generate Entities From Tables wählen. Wählen Sie dort `gast` aus. Klicken Sie auf Next und noch einmal auf Next.
6. Wählen Sie als Key Generator `identity` und schließen Sie mit Finish.
7. Klicken Sie auf die Datei *persistence.xml* im Projektverzeichnis *JPA Content* doppelt.
8. Unter Connection wählen Sie als Transaction Type `Resource Local`. Nun können Sie Populate From Connection anwählen und mit Ihrer Datenbankverbindung füllen.

9. Kopieren Sie den PostgreSQL-JDBC-Treiber in das Projektverzeichnis *WebContent/WEB-INF/lib*.
10. Erstellen Sie eine Klasse über New|Class namens `SoapGast`.

```
package model;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

@WebService
public class SoapGast {

    @WebMethod
    public void speichere(Gast gast) {
        EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("Soap3");
        EntityManager manager = factory.createEntityManager();
        manager.getTransaction().begin();
        manager.persist(gast);
        manager.getTransaction().commit();
        manager.close();

    }
}
```

Listing 30.51: Die Klasse SoapGast fügt einen Gast in die Datenbank per JPA ein.

Über diese Klasse bilden Sie nun eine WSDL-Definition, indem Sie New|Other|Web Service|Web Service auswählen. Als Basis wählen Sie die Klasse `model.SoapGast`.

Der passende Client wird wiederum als Java-Projekt angelegt. Der Dienst wird über New|Other|Web Service|Web Service Client angelegt und er erhält eine Klasse `MeinClient`, die einen Gast per SOAP in die Datenbank schafft.

```
package model;

import java.rmi.RemoteException;

import javax.xml.rpc.ServiceException;

public class MeinClient {
    public static void main(String[] args)
            throws RemoteException, ServiceException {
        SoapGastServiceLocator locator =
                new SoapGastServiceLocator();
        SoapGast soap = locator.getSoapGast();
        Gast gast = new Gast();
        gast.setName("Willy Schmidt");
        gast.setOrt("Gintoft");
        soap.speichere(gast);
    }
}
```

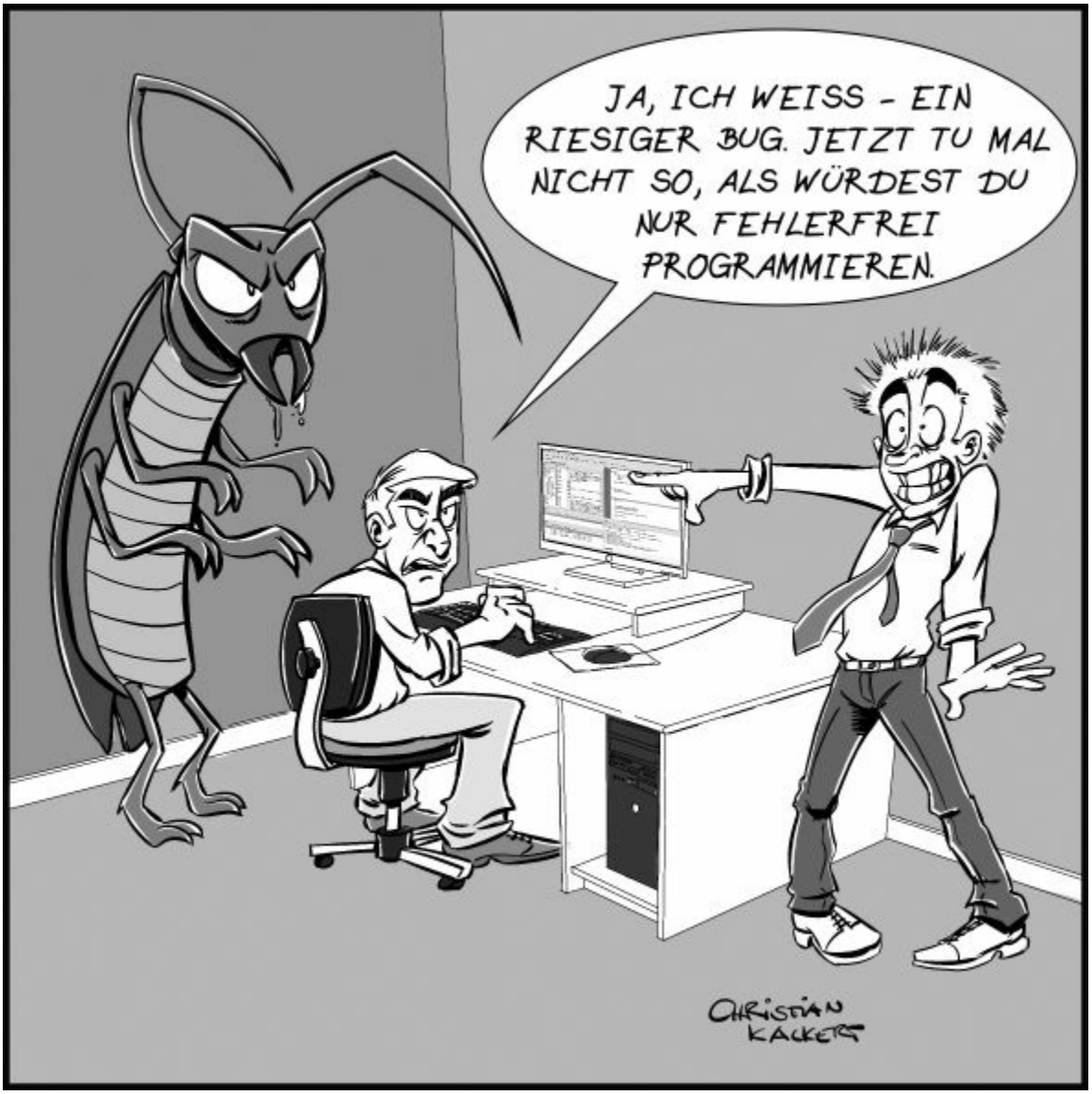
Listing 30.52: Der Client übersendet einen Gast per SOAP.

Nun schaut Heidi in der Datenbank nach und stellt fest, dass der Eintrag angekommen ist. »Schön! Aber was ist nun der Unterschied zu REST?« Klara versucht zu erläutern, dass die Ebene der Client-Server-Kommunikation angehoben wird, weil der Server nun nicht nur Ressourcen liefert, sondern auch komplexere Anfragen ausführen kann.

Heidi versteht es besser am Beispiel. Wenn der Gast umbuchen will und eine andere Wohnung zu einem anderen Zeitpunkt haben möchte, kann SOAP nicht nur die Buchung ändern, sondern auch in der Preistabelle nachsehen, ob für diesen Zeitpunkt Hauptsaison-Preise gelten und gleich die Rechnung korrigieren. Die Aktion wird vom Client auf den Server verlagert. Dafür ist REST interessanter, wenn es um den reinen Ressourcen-Zugriff geht. Denn die Schnittstelle für den Client ist bei REST sehr viel einfacher zu programmieren.

Teil VIII

Werkzeuge



Kapitel 31

Dem Fehler auf der Spur

IN DIESEM KAPITEL

Aus Fehlern lernen und Code reparieren

Schrittweise Fehler aufspüren

Testautomaten für vorgefertigte Fehlersituationen

Immer diese Fehler! Ich hatte mal behauptet, dass Programmieren den Charakter schult, weil man lernen muss, die eigene Unzulänglichkeit zu akzeptieren. Ich musste dann allerdings feststellen, dass es viele Kollegen gibt, die das Schwarze-Peter-Spiel perfekt beherrschen. Schuld ist – je nach persönlicher Präferenz – Apple, Windows, Linux, C, Java, Google oder Oracle. Die schlechte Nachricht für alle Programmierer: Nein, du bist schuld!

Alles so schön rot hier

Wenn Eclipse einen Fehler in Ihren Programmen findet, dann reagiert die IDE gleich dreifach:

- Der fehlerhafte Code wird in alter Paukermanier mit einem roten Kringelstrich unterstrichen.
- Links am Rand wird ein großer runder Punkt gesetzt.
- Auf der rechten Seite finden Sie eine kleine rote Markierung, die vor allem bei längeren Listings anzeigt, wo etwa der Fehler im Listig steckt. Klicken Sie darauf, landen Sie direkt bei dem Fehler.

Im Gegensatz zum Pauker aus der Schule müssen Sie die Entdeckung Ihrer Fehler nicht fürchten. Im Gegenteil: Wenn Eclipse den Fehler anzeigt, dann brauchen Sie ihn nicht zu suchen.

Fehlermeldung

Fahren Sie den Mauszeiger über den rot unterkringelten Fehler und halten Sie ein wenig still. Dann erscheint ein Tooltipp mit der Fehlermeldung, leider auf Englisch. Und selbst wenn Sie ein Wörterbuch oder ein Übersetzungs-Tool verwenden, kann es sein, dass Ihnen die Meldung kryptisch vorkommt. Compiler haben eben eine

besondere Art, sich auszudrücken, und wie in einer Ehe dauert es eine Weile, bis Sie Ihren Partner verstehen.

Unter der Fehlermeldung erscheint eine Liste mit Vorschlägen zur Behebung der Situation. Auch diese ist englisch, aber oft einfacher zu verstehen. Vor allem können Sie vielleicht anhand der Reparaturvorschläge erkennen, was dem Compiler nicht gefällt.

Das Tollste ist, dass Sie auf eine der Lösungen doppelklicken können, und Eclipse behebt den Fehler selbst. Mit etwas Geschick und Erfahrung können Sie Eclipse auf diese Weise dazu bringen, einen großen Teil der Programme selbst zu generieren.


Auch wenn Sie den roten Punkt auf der linken Seite der Zeile anklicken, hilft Ihnen Eclipse weiter.

Gelbe Warnungen

Manchmal malt Eclipse nicht in Rot, sondern in Gelb. Gelbe Markierungen sind Warnungen. Das heißt, dass das Programm ausführbar ist, aber irgendwas gefällt Eclipse nicht sonderlich. So bemerkt es Variablen, die nie benutzt werden. Tatsächlich kann sich dahinter ein echter Fehler verbergen. Vielleicht haben Sie sich vertippt und an entscheidender Stelle einen falschen Variablennamen verwendet.

Es lohnt sich also, den Warnungen nachzugehen. Auf diese Weise finden Sie manchmal Fehler in Ihrer Logik, die wesentlich schwieriger zu finden sind als die syntaktischen Fehler, die Ihnen der Compiler rot anmalt.

Der Eclipse-Debugger

Ein Fehler wird im Englischen als *Bug* (Käfer) bezeichnet. Der Legende nach soll ein Käfer in einem der ersten Computer verstorben sein und durch seinen Körper einen Kurzschluss zwischen zwei Leiterbahnen verursacht haben. So entstand ein Fehlverhalten, das bis zum Auffinden der Leiche unerklärbar war.

Ein Debugger ist ein Werkzeug, das es Ihnen erlaubt, durch Ihren Quellcode Schritt für Schritt zu gehen, um herauszufinden, was Ihr Programm wirklich macht. Dazu müssen Sie zunächst einen Haltepunkt setzen, an dem der Debugger nach dem Programmstart stoppt. Einen solchen Haltepunkt bezeichnet man als *Breakpoint*.

Ein Breakpoint erscheint als grüner Punkt links neben einer Code-Zeile. Wäre er rot, würden Sie ihn glatt für einen Fehler halten. Sie können einen Breakpoint auf verschiedene Weise setzen:

- ✔ Doppelklicken Sie links neben die Zeilennummer.

- ✔ Klicken Sie mit der rechten Maustaste links neben die Zeilennummer und wählen Sie aus dem Kontextmenü Toggle Breakpoint.
- ✔ Verwenden Sie die Tastenkombination Strg + ⇧ + B.

Sie können mehrere Breakpoints setzen und sich so von Breakpoint zu Breakpoint hangeln wie Tarzan von einem Baum zum nächsten.

Das Programm im Debugger starten

Eclipse kennt mehrere Arten, das Programm im Debugger zu starten:

- ✔ Klicken Sie auf den grünem Käfer in der Symbolleiste.
- ✔ Wählen Sie aus dem Menü Run|Debug.
- ✔ Drücken Sie die Taste F11.

Eclipse wechselt in die Debug-Perspektive, wie in Abbildung 31.1 zu sehen.

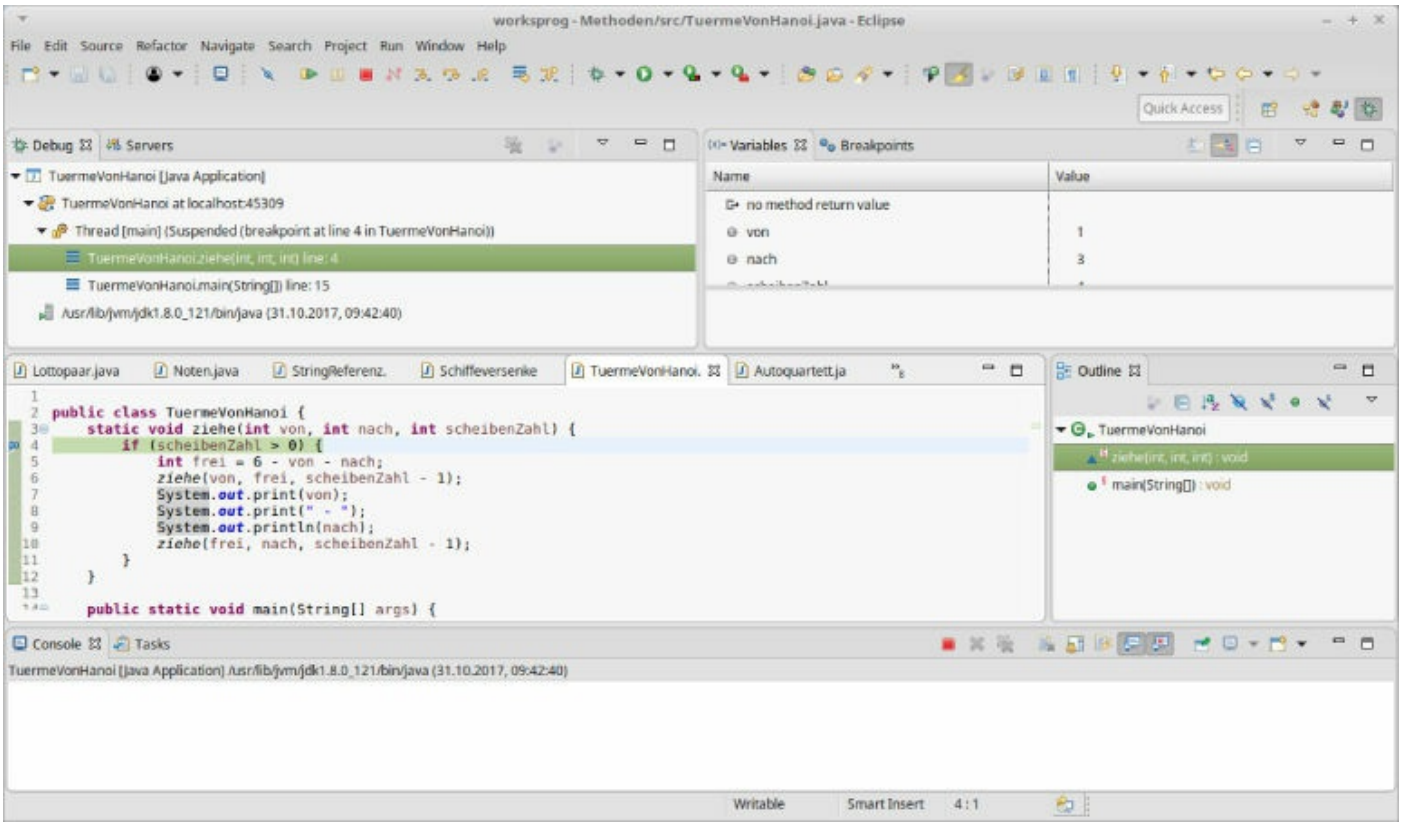


Abbildung 31.1: Eclipse in der Debug-Perspektive

- ✔ Im Zentrum links steht das Listing. Die Zeile, in der der Debugger aktuell steht, ist grün hinterlegt.
- ✔ Oben links sehen Sie die Aufrufhierarchie der durchlaufenen Methoden.
- ✔ Oben rechts sehen Sie die Variablen und deren Inhalt. Dort finden Sie hinter einer Lasche verborgen eine Liste aller gesetzten Breakpoints.
- ✔ Unten sehen Sie wie üblich die Console mit den Programmausgaben.



In die normale Java-Umgebung gelangen Sie über das Menü Window|Perspective|Open Perspective. Dahinter sind alle verfügbaren Perspektiven aufgeführt.

Ganz oben in der Symbolleiste tauchen wichtige Symbole auf, die Ihnen beim Durchlaufen des Programms hilfreich zur Seite stehen.



Abbildung 31.2: Eclipse und seine Debug-Symbole

Mit diesen Symbolen schreiten Sie durch den Quellcode. Die Bedeutung der Symbole erfahren Sie auch, wenn Sie den Mauszeiger ganz still über dem Symbol halten, durch einen Tooltipp. Aber an dieser Stelle verrate ich noch einen Satz mehr zu jedem Symbol.



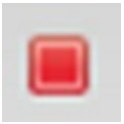
Der grüne Käfer startet das Programm im Debugger. Alternativ funktioniert auch die Taste F11 oder der Aufruf des Menüpunkts Run|Debug.



Wenn Sie dieses Symbol anklicken, läuft das Programm weiter, bis es zum nächsten Breakpoint gelangt. Alternativ können Sie die Taste F8 verwenden.



Mit diesem Symbol können Sie das Programm im Lauf stoppen. Sie führen also quasi einen fliegenden Breakpoint aus. So erkennen Sie beispielsweise, wo das Programm in einer Endlosschleife festhängt.



Sie haben genug gesehen. Das Programm kann nun sterben. Die Tastenkombination Strg + F2 bewirkt dasselbe.



Wenn der Debugger gerade steht, können Sie im Quellcode um einen Schritt weitergehen. Stehen Sie gerade auf einem Methodenaufruf, folgt der Debugger der Methode in ihr Inneres. Alternativ können Sie die Taste F5 drücken.



Auch mit diesem Symbol gelangen Sie zur nächsten Anweisung. Allerdings wird der Debugger nicht in eine Methode hineinspringen, sondern auf die nächste Anweisung gehen, die hinter dem Aufruf steht. Die alternative Taste ist F6.



Wenn Sie sich innerhalb einer Methode befinden, können Sie mit diesem Symbol die Methode nicht weiter schrittweise durchlaufen. Stattdessen stoppt der Debugger bei der ersten Anweisung nach dem Methodenaufruf.

Kapitel 32

Automatisierte Tests: JUnit

IN DIESEM KAPITEL

- Testziele definieren
- Korrekte Bestände gegen Verschlimmbesserung schützen
- IDE-Unterstützung durch Eclipse und NetBeans

Fehlerfreie Software ist der Wunschtraum jedes Computeranwenders. Und Programmierer wissen, warum diese Hoffnung eine Utopie ist. Es ist ja gar keine böse Absicht, sondern vielfach der Zeitdruck, dass die Zeit zum Testen fehlt. Und natürlich ist es langweilig, die Software wieder und wieder zu starten, um zu schauen, ob das Programm auch unter verschiedenen Randbedingungen keine Fehler macht. Die einfachste Lösung ist natürlich, man schickt die ungetestete Software zum Kunden. Der wird die Fehler bestimmt finden. Durch Nachbesserungen wird sie immer besser. Darum spricht man von Bananen-Software, weil sie beim Kunden reift.

Als erstes Hilfsmittel im Kampf gegen die Schlamperei steht der Compiler in der ersten Reihe. Java achtet sehr streng darauf, dass Datentypenvorgaben eingehalten werden. Wenn der Programmierer auf die Warnungen achtet, können leichtfertige Unsauberkeiten auffallen. Aber leider garantiert ein korrekt kompliliertes Programm noch lange nicht die Korrektheit.

Damit wären wir wieder bei der langweiligen Testerei. Aber für langweilige Arbeiten hat man den Computer erfunden. Also könnte man die Aufgabe auf den Computer abschieben und den die Tests machen lassen. In Java unterstützt JUnit den Tester. Damit ist es möglich, für die Methoden einer Klasse Testmethoden zu schreiben und deren Ergebnisse festzulegen. Diese Szenarien können bei Änderungen am Quellcode immer wieder aufgerufen werden, um zu prüfen, ob die Korrektur eines Fehlers nicht längst behobene Fehler wieder eingebaut hat.



»Durch Testen kann man stets nur die Anwesenheit, nie aber die Abwesenheit von Fehlern beweisen.« (Edsger Wybe Dijkstra)

Konfiguration von Eclipse

JUnit bekommt erst richtig Farbe, wenn es in der Entwicklungsumgebung integriert ist. Das ist glücklicherweise bei den wichtigsten IDEs der Fall. Bevor Sie JUnit unter Eclipse verwenden können, müssen Sie es als Library hinzufügen. Dazu gehen Sie folgendermaßen vor:

1. Klicken Sie mit der rechten Maustaste auf das Projekt und wählen Sie aus dem Kontextmenü den Punkt Properties.
2. Im Dialog klicken Sie in der linken Spalte auf Java Build Path.
3. Auf der rechten Seite klicken Sie auf den Reiter Libraries.
4. Klicken Sie auf den Button Add Library auf der rechten Seite.
5. In dem neuen Dialog wählen Sie Junit aus und klicken auf Next.
6. Im nächsten Dialog können Sie die Version von JUnit wählen und dann mit Finish bestätigen.
7. In der Liste der Libraries taucht nun auch JUnit auf. Sie können den Dialog mit OK beenden.
8. Im Projekt können Sie sehen, dass neben der JRE System Library auch JUnit als Library eingebunden ist.

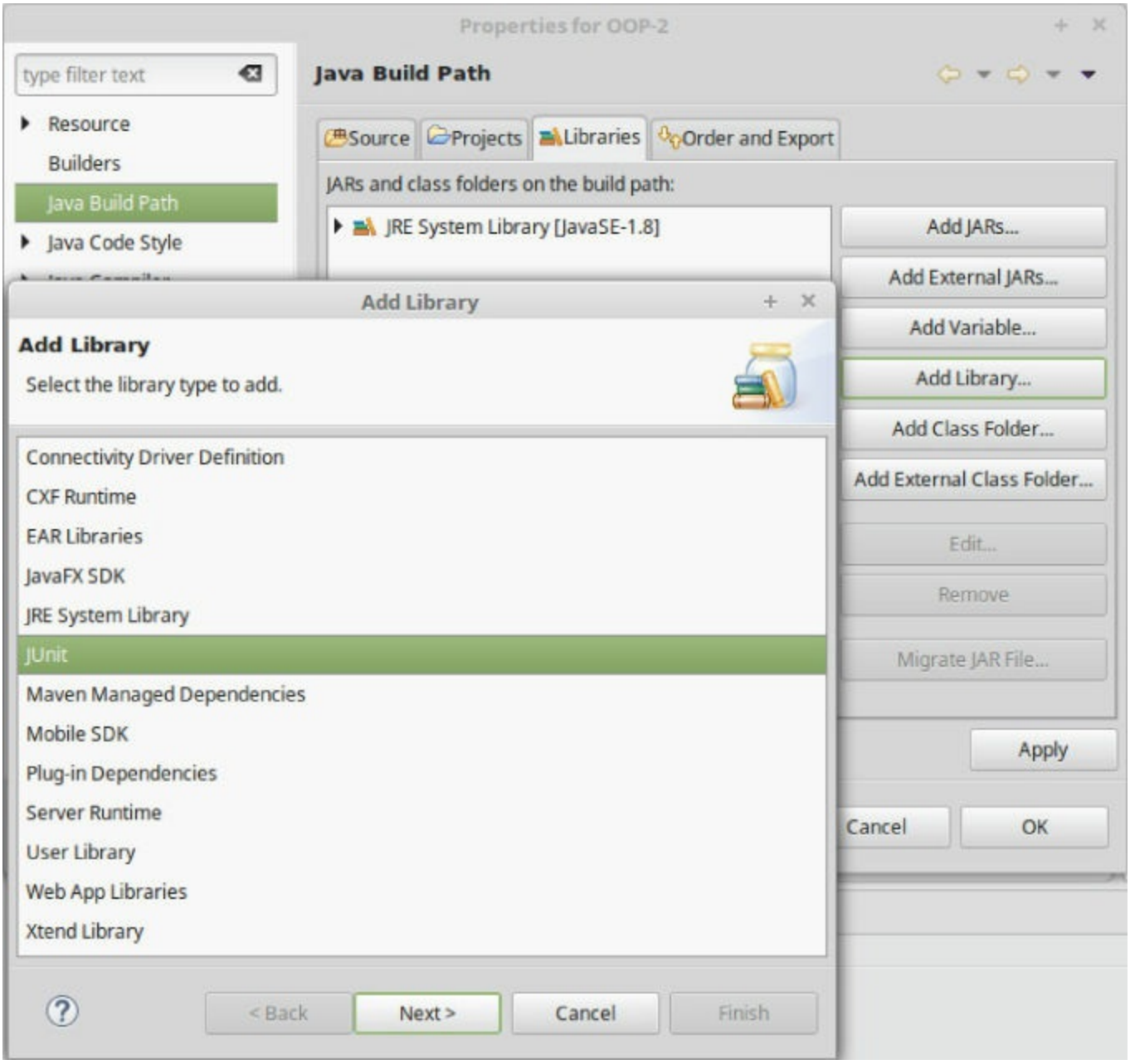


Abbildung 32.1: Einbinden der JUnit Library

Erstellen eines TestCase

Als Opfer für einen JUnit-Test ziehen wir eine Klasse `Bruch` heran, die Zähler und Nenner als Attribute enthält, aber vor allem Methoden für die Grundrechenarten anbietet. Um diese Methoden zu prüfen, wenden Sie folgende Schritte an.

1. Klicken Sie die Klasse im Projektfenster mit der rechten Maustaste an und wählen Sie New|Junit Test Case.
2. Sie können nun einen Namen für den Test Case vergeben. Standardmäßig wird der Klassenname gewählt und »Test« angehängt.
3. Wenn Sie den Button Next anklicken, können Sie Eclipse bitten, die Testfälle für Ihre gesamte Klasse und jede ihrer Methoden zu erstellen, indem Sie den Klassennamen anwählen. Sie können sogar für die Methoden der Klassen

Testfälle erstellen lassen, die Ihre Klasse erweitert. Darum wird in jedem Fall die Klasse `Object` angeboten.

4. Klicken Sie auf Finish. Eine neue Klasse bevölkert Ihr Projekt.

Für jede Methode der Klasse wird nun eine Methode angelegt, die mit »test« beginnt. Sie enthält zur Zeit nur den Aufruf von `fail` mit dem Kommentar, dass der Test erst noch implementiert werden muss.

```
public class BruchTest {
    // ...
    @Test
    public void testDiv() {
        fail("Not yet implemented");
    }
    // ...
}
```

Listing 32.1: Ausschnitt aus dem Test Case

Um den Test zu starten, klicken Sie die Testklasse mit der rechten Maustaste an und wählen Run AS|Junit Test. Es erscheint im Projekt-Fenster eine Aufstellung, welche Tests positive Ergebnisse bringen. Das ist zurzeit eher deprimierend, da ja alle Tests auf die Methode `fail` stoßen.

Im ersten Schritt wollen wir einen Test der Methode `mul` ausformulieren. Dazu führen wir eine Multiplikation durch und testen anschließend mit einer der JUnit-Assert-Methoden, ob das Ergebnis mit unseren Erwartungen übereinstimmt.

```
@Test
public void testMul() {
    Bruch bruch = new Bruch(1,2);
    Bruch mulbruch = bruch.mul(new Bruch(2,3));
    assertTrue(mulbruch.equals(new Bruch(2,6)));
}
```

Listing 32.2: Test der Bruchmultiplikation

Das Ergebnis ist in Abbildung 32.2 zu sehen. Die Test-Methode für die Division `testDiv` liefert noch einen Fehler, da hier immer noch die Methode `fail` meldet, dass der Test noch nicht implementiert ist. Die Meldung erscheint im Fenster unterhalb, wenn Sie auf das Ergebnis der Testmethode klicken.

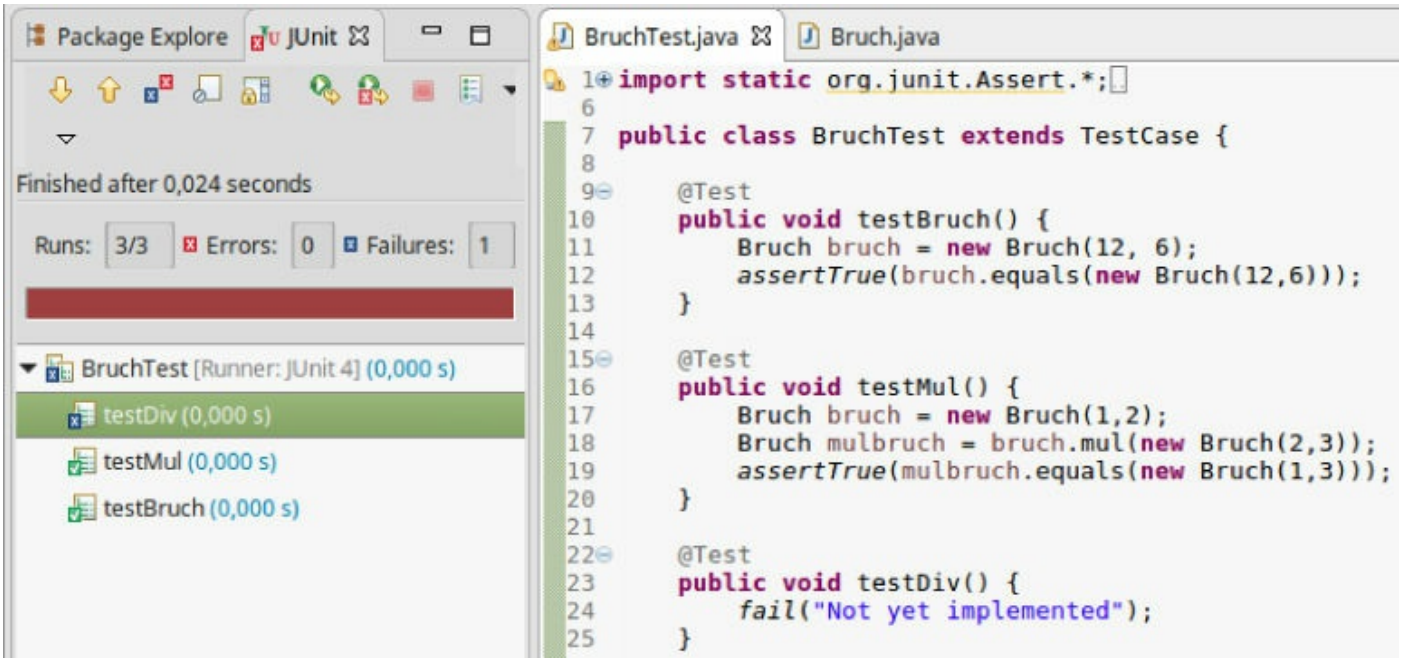


Abbildung 32.2: Test des JUnit-Test-Case

Sie können auch `assertTrue` als ersten Parameter eine Meldung mitgeben, die erscheint, wenn der Test scheitert und nähere Details angefragt werden. Neben den Methoden `fail` und `assertTrue` stehen Ihnen noch weitere Methoden zur Verfügung:

- ✔ `fail(String)`

  Löst eine Fehlermeldung aus, wenn ein Zustand erreicht wurde, der definitiv fehlerhaft ist.
- ✔ `assertTrue(wahr)`

  Die Methode erwartet, dass der Parameter `true` ist. Typischerweise werden darin die Ergebnisse der zu prüfenden Methoden abgefragt.
- ✔ `assertFalse([message,] boolean condition)`

  Diese Methode erwartet, dass das Ergebnis `false` ist.
- ✔ `assertEquals(erwartet, ergebnis)`

  Die Methode prüft, ob die beiden Parameter den gleichen Wert haben.
- ✔ `assertEquals(erwartet, ergebnis, abweichung)`

  Diese Methode prüft, ob das Ergebnis in einem gewissen Toleranzbereich des Wunschergebnisses liegt.
- ✔ `assertNull(referenz)`

  Wenn die Referenzvariable nicht `null` ist, scheitert die Methode.
- ✔ `assertNotNull(referenz)`

  An dieser Stelle sollte die Referenz ungleich `null` sein, damit die Methode nicht scheitert.
- ✔ `assertSame(erwartet, ergebnis)`

  Hier wird geprüft, ob die beiden Parameter dasselbe Objekt referenzieren.

- `assertNotSame([message,] expected, actual)`

  Hier wird geprüft, ob die beiden Parameter verschiedene Objekte referenzieren.



Wenn Sie für Vergleiche die Methode `equals` verwenden, sollte diese auch implementiert sein. Ansonsten wird auf `equals` von `Object` zurückgegriffen, was nur prüft, ob die Referenzen auf das gleiche Objekt zeigen.

Annotationen

Die Methoden sind mit der Annotation `@Test` versehen. Das bewirkt, dass sie bei einem Testlauf aufgerufen werden. Auch hier gibt es noch ein paar Varianten, die im Einzelfall hilfreich sein können. Einige der Namen wurden beim Übergang von JUnit 4 auf JUnit 5 geändert.

- `@Test`

  Die Methode wird als Testmethode markiert.
- `@Test (expected = Exception.class)`

  Als Ergebnis wird erwartet, dass eine Exception geworfen wird. Wenn diese nicht ausgelöst wird, gilt das als Fehler.
- `@Test(timeout=100)`

  Die Methode muss innerhalb von 100 Millisekungen abgeschlossen sein. Ansonsten gilt sie als fehlerhaft.
- `@Before`

  ab JUnit 5: `@BeforeEach`

  Diese Methode wird vor jedem Test zur Initialisierung der Testumgebung ausgeführt.
- `@After`

  ab JUnit 5: `@AfterEach`

  Diese Methode wird für Aufräumarbeiten der Testumgebung nach jedesm Test ausgeführt.
- `@BeforeClass`

  ab JUnit 5: `@BeforeAll`

  Diese Methode wird einmal vor dem Start aller Tests durchgeführt. Sie muss als `static` definiert sein.
- `@AfterClass`

  ab JUnit 5: `@AfterAll`

  Diese Methode wird einmal nach dem Ende aller Tests ausgeführt. Sie muss als `static` definiert sein.

- ✔ `@Ignore("Kommentar zur Ignoranz")`

  ab JUnit 5: `@Disabled`

  Diese Annotation nimmt eine Testmethode kurzzeitig aus dem Rennen. Das kann im Zuge größerer Änderungen sinnvoll sein. Die Meldung in Klammern landet im Protokoll. Die ganze Klammer kann auch weggelassen werden.

JUnit-Testaufbau bei NetBeans

Auch die Entwicklungsumgebung NetBeans unterstützt JUnit. Wenn Sie für eine Klasse eine Testumgebung haben wollen, gehen Sie folgendermaßen vor:

1. Sie klicken die zu testende Klasse mit der rechten Maustaste an und wählen Tools|Create/Update Tests.
2. NetBeans schlägt eine neue Klasse mit gleichem Klassennamen vor, allerdings um »Test« ergänzt.
3. Sie beschließen die Generierung mit OK. Die Testklasse wird in einem neuen Projektverzeichnis *Test Packages* neben dem Verzeichnis *Source Packages* abgelegt.

NetBeans generiert allerdings keine Testfälle. Dies müssen Sie noch selbst schreiben. Hier erstellen Sie eine Methode `testMul`, die die Bruch-Multiplikation prüft:

```
@Test
public void testMul() {
    Bruch bruch = new Bruch(1, 2);
    Bruch mulbruch = bruch.mul(new Bruch(2, 3));
    assertTrue(mulbruch.equals(new Bruch(1, 3)));
}
```

Listing 32.3: Eine Test-Methode für die Bruchmultiplikation

Sie klicken nun die Klasse `BruchTest` im Projekt mit der rechten Maustaste an und wählen Test File und Sie sehen anschließend im Fenster unter dem Quelltext, dass der Test zu 100 Prozent funktioniert hat.

Solange Sie allerdings noch keine mit `@Test` markierte Methode besitzen, wird der Test mit der Meldung scheitern, dass es keine lauffähigen Methoden gäbe.

Kapitel 33

Die Versionskontrolle Git

IN DIESEM KAPITEL

Rückgriff auf ältere Stände

Abzweigungen für Neuentwicklungen

Synchronisierung mehrerer Arbeitsplätze

Wenn früher alles besser war, dann fehlt Ihnen eine Versionskontrolle. Diese Erfahrung macht jeder Programmierer früher oder später. Gestern! Das war dieser schöne Tag, wo das Programm noch einwandfrei lief. Dann wollte ich nur diese kosmetische Veränderung machen und seither läuft nichts mehr.

Eine Versionskontrolle wird benötigt, um …

- alte Versionen hervorzukramen.
- gemeinsame Programmierstände zu koordinieren.
- neue Features zu entwickeln, ohne den Auslieferungsstand zu gefährden.
- schnell mal eine Arbeitsumgebung zu erzeugen.

Eine Versionskontrolle speichert ein Projekt aus mehreren Dateien inklusive seiner historischen Veränderungen. Wenn Sie also bei jeder relevanten Änderung eine Sicherung auslösen, die man als *Commit* bezeichnet, können Sie später alle Versionen bis zur ersten Sicherung zurückholen. Oder Sie können vergleichen, was es für Änderungen zwischen einst und jetzt gab.

Git arbeitet mit lokalen Repositories. Das bedeutet, dass Sie auf Ihrem lokalen Computer eine vollständige Historie Ihrer Änderungen besitzen.

Das zentrale Repository

Darüber hinaus ist es möglich, ein zentrales Git-Repository zu pflegen. Das ist immer dann sinnvoll, wenn mehrere Entwickler an einem Projekt zusammenarbeiten. Die Entwickler füttern das zentrale Repository aus ihren lokalen Repositorys durch einen *Push*.

Änderungen gehen also zunächst per *Commit* in das lokale Repository und von dort später mit einem *Push* in das zentrale Repository.

Um die Änderungen der Kollegen zu übernehmen, führt man einen *Pull* aus. Er

übernimmt deren Änderungen sowohl in das lokale Repository als auch in den lokalen Stand.

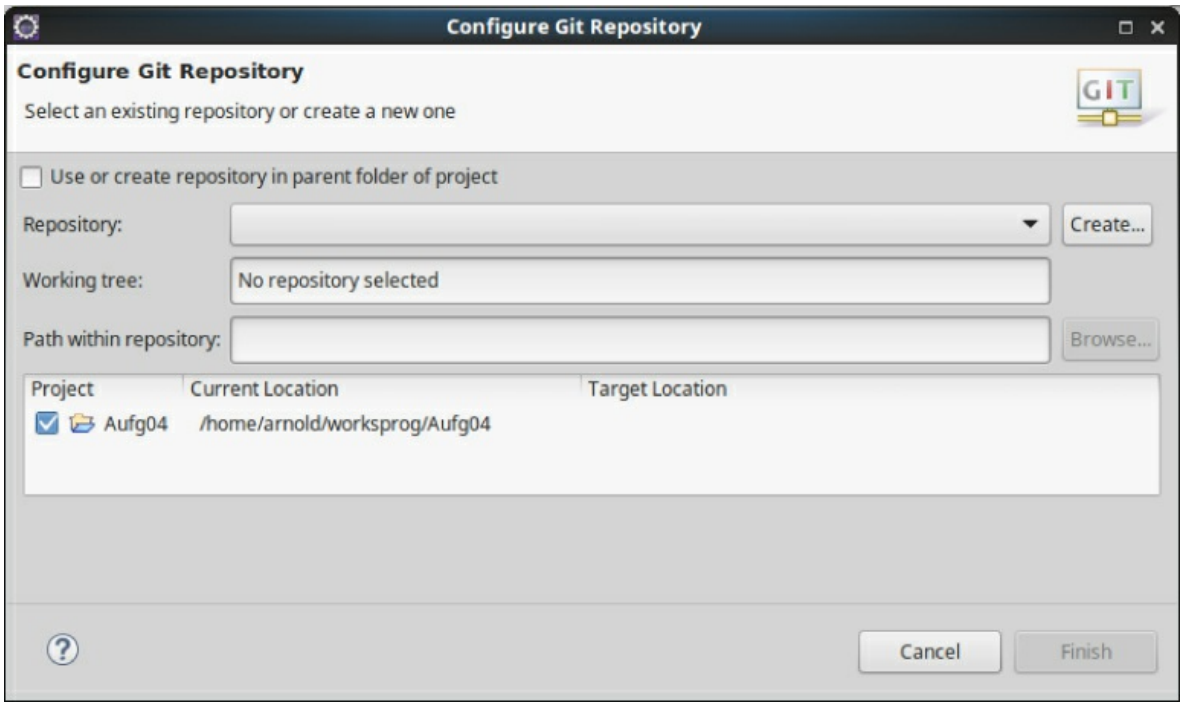
Kommt ein neuer Kollege hinzu, benötigt er einen neuen Arbeitsplatz. Dazu klont man das zentrale Repository in ein lokales Arbeitsplatz-Repository. Der Kollege kann also die gesamte Historie des Projekts nutzen.

Ein lokales Repository für Eclipse

Wenn Sie ein neueres Eclipse oder ein Eclipse für Java EE verwenden, ist eGit bereits eingerichtet. Bei älteren Versionen kann es sein, dass Sie es erst über den Marketplace installieren müssen.

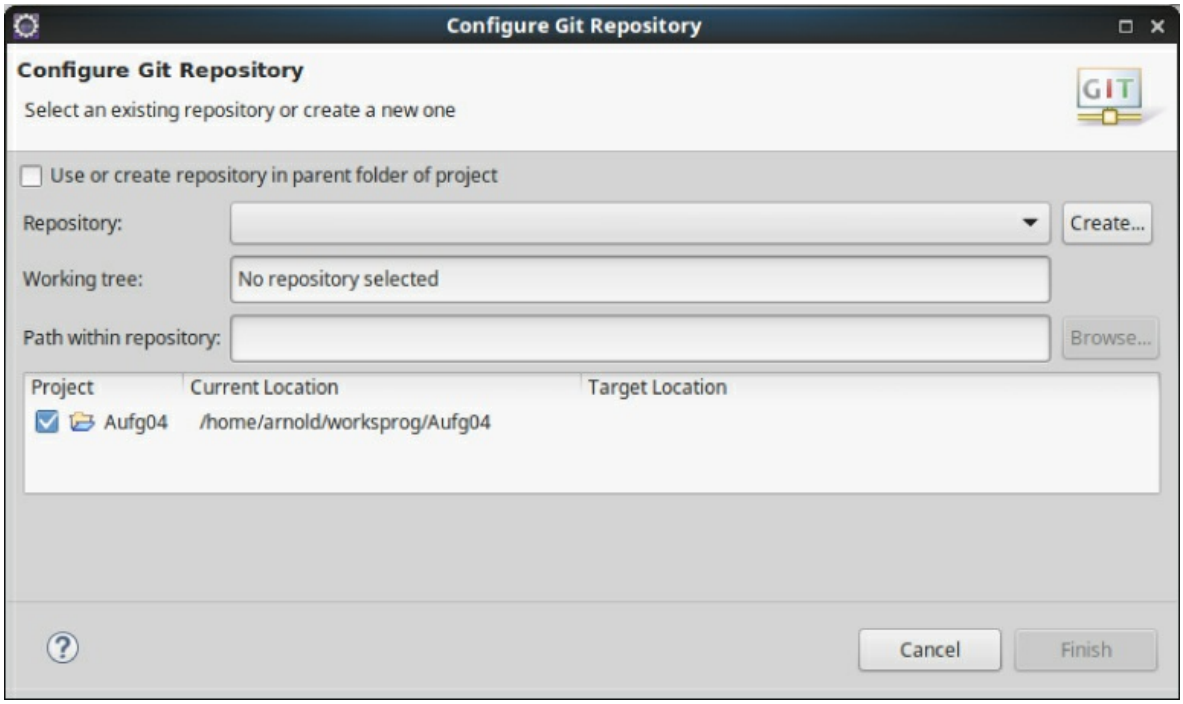
Sie können für jedes Projekt spontan ein lokales Repository anlegen, indem Sie folgende Schritte durchführen:

1. Klicken Sie das Eclipse-Projekt mit der rechten Maustaste an.
2. Wählen Sie Team|Share Project. Es erscheint ein Dialog zur Konfiguration des Git-Repositories.
3. In dieser Box wird entschieden, wo das Repository auf dem lokalen PC liegen soll.
    - Um ein eigenes Verzeichnis zu wählen, klicken Sie auf den Button Create hinter dem Eintrag Repository. Im Folgedialog können Sie den Pfad eingeben oder über den Button Browse auswählen. Mit Finish erhalten Sie den Pfad im Ursprungsdialog.



    - Im Verzeichnis des Eclipse-Arbeitsbereichs: Klicken Sie einen Haken in den Button Use OR Create Repository IN Parent Folder OF Project. Im

Folgedialog klicken Sie auf Create Repository. Schließen Sie mit Finish.



4. Mit Finish wird das lokale Repository erzeugt.

Hinter dem Projekt erscheint in eckigen Klammern der Name Ihres lokalen Repositorys und die Anmerkung `NO-HEAD`.

Der allererste Commit

Nun gibt es zwar ein Repository, aber das ist zunächst leer. Um den Stand hineinzubringen, gehen Sie folgendermaßen vor:

1. Klicken Sie das Eclipse-Projekt mit der rechten Maustaste an.
2. Wählen Sie Team|Commit.
3. Es erscheint im unteren Fenster von Eclipse ein Bereich Git Staging. Alle Dateien sind *unstaged*, stehen also nicht unter der Kontrolle von Git.
4. Klicken Sie ein Element in Unstaged Changes an und drücken Sie anschließend Strg + A, um alle Dateien zu erwischen.
5. Verschieben Sie die Dateien in das Feld Staged Changes, indem Sie eine der markierten Dateien mit der rechten Maustaste anklicken und aus dem Kontextmenü Add TO Index wählen.
6. Passen Sie die Felder Author und Committer Ihren Entwicklerangaben an.
7. Geben Sie unbedingt eine Commit Message ein, die möglichst in einer Zeile genau beschreibt, was geändert wurde. Diese Zeile erscheint später als Hauptüberschrift für die Änderung.
8. Klicken Sie auf Commit.

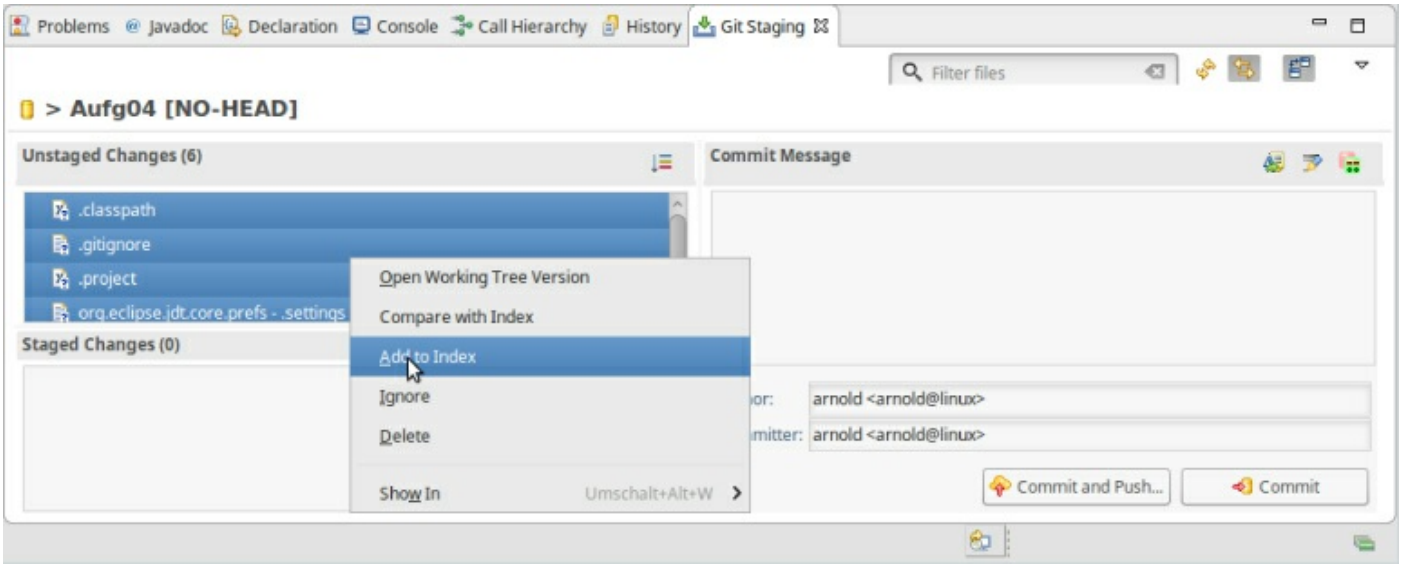


Abbildung 33.1: Der erste lokale Commit in Eclipse

Nun ist der Bereich von Unstaged Changes so lange leer, bis Sie an einer Source etwas ändern und abspeichern. Langsam füllt sich die Liste. Bevor Sie die Änderungen in das Git stellen, müssen Sie es explizit mit der rechten Maustaste anklicken und Add TO Index auswählen.

Nur die Einträge im Feld Stages Changes werden beim nächsten Commit übernommen. Ändern Sie nun weiter an dem Text, erscheint die Datei sowohl im Bereich Staged als auch im Bereich Unstaged. Beim nächsten Commit wird nur der Zustand übernommen, wie er bei der Übernahme in den Index vorlag.

Geschichtsforschung

Wenn Sie Ihr Projekt mit der rechten Maustaste anklicken, können Sie über den Menüpunkt Team|Show IN History eine Übersicht über seine Geschichte erlangen.

Es erscheint im unteren Bereich von Eclipse eine Lasche History. Im oberen Teilfenster ist die Liste der Änderungen mit der ID, dem Kommentar beim Commit und dem Autor. Sie können einen beliebigen Stand anklicken und sehen darunter auf der linken Seite Details. Auf der rechten Seite sind die Dateien aufgelistet, die geändert wurden. Klicken Sie auf eine dieser Dateien doppelt, erscheint sie in dieser Version im Editor.

Sie können sich auch die Vergangenheit einer einzelnen Datei anschauen. Dazu klicken Sie mit der rechten Maustaste an die Datei oder in den Editor und wählen Team|Show Local History. Es erscheint eine Liste von Daten und Uhrzeiten, wann diese Datei gesichert wurde. Sie können eine dieser Versionen mit der rechten Maustaste anklicken. Über den Menüpunkt Compare Current With Local erhalten Sie einen zweigeteilten Editor, in dem die beiden Stände nebeneinander stehen und die Differenzen gut zu sehen sind.

Branches

Ein Branch ist eine Abzweigung des Projekts, den man vor allem dann einsetzt, wenn umfangreichere Änderungen dazu führen, dass das Projekt längere Zeit nicht einsetzbar ist. So kann beispielsweise die grafische Umgebung umgestellt werden, ohne dass man die Möglichkeit verliert, auf dem derzeit ausgelieferten Stand Änderungen durchzuführen.

Wenn der Branch eines Tages fertig ist, können Sie ihn wieder mit dem Hauptentwicklungszweig vereinigen. Diesen Vorgang bezeichnet man als *Merge*. Dabei kann es vor allem dann zu Konflikten kommen, wenn am Hauptzweig zwischendurch gearbeitet wurde.

Für den Fall, dass sich der Branch als Fehlschlag erweist, können Sie den gesamten Branch wieder löschen.

Branches unter Eclipse

Um einen neuen Branch anzulegen, klicken Sie das Projekt mit der rechten Maustaste an und wählen Team|Switch TO|New Branch. Sie werden aufgefordert, einen Namen für diesen Branch zu vergeben. Dann können Sie ihn mit einem Klick auf Finish erzeugen.

Sie können jederzeit zwischen dem Hauptzweig (master) und anderen Branches umschalten, indem Sie das Projekt mit der rechten Maustaste anklicken und Team|Switch TO anwählen. Dahinter werden alle verfügbaren Branches aufgezählt. Sie können also kurzfristig Kundenanfragen am Release-Stand bearbeiten.

Über Team|Merge können Sie einen Branch wieder in den Hauptzweig eingliedern. Wollen Sie einen Branch löschen, müssen Sie sich zunächst mit Team|Switch TO in den Hauptzweig bringen. Anschließend können Sie sich über Team|Switch TO|Other eine Übersicht der Branches ansehen. Dort können Sie den Branch auswählen und mit dem Klick auf Delete entfernen.

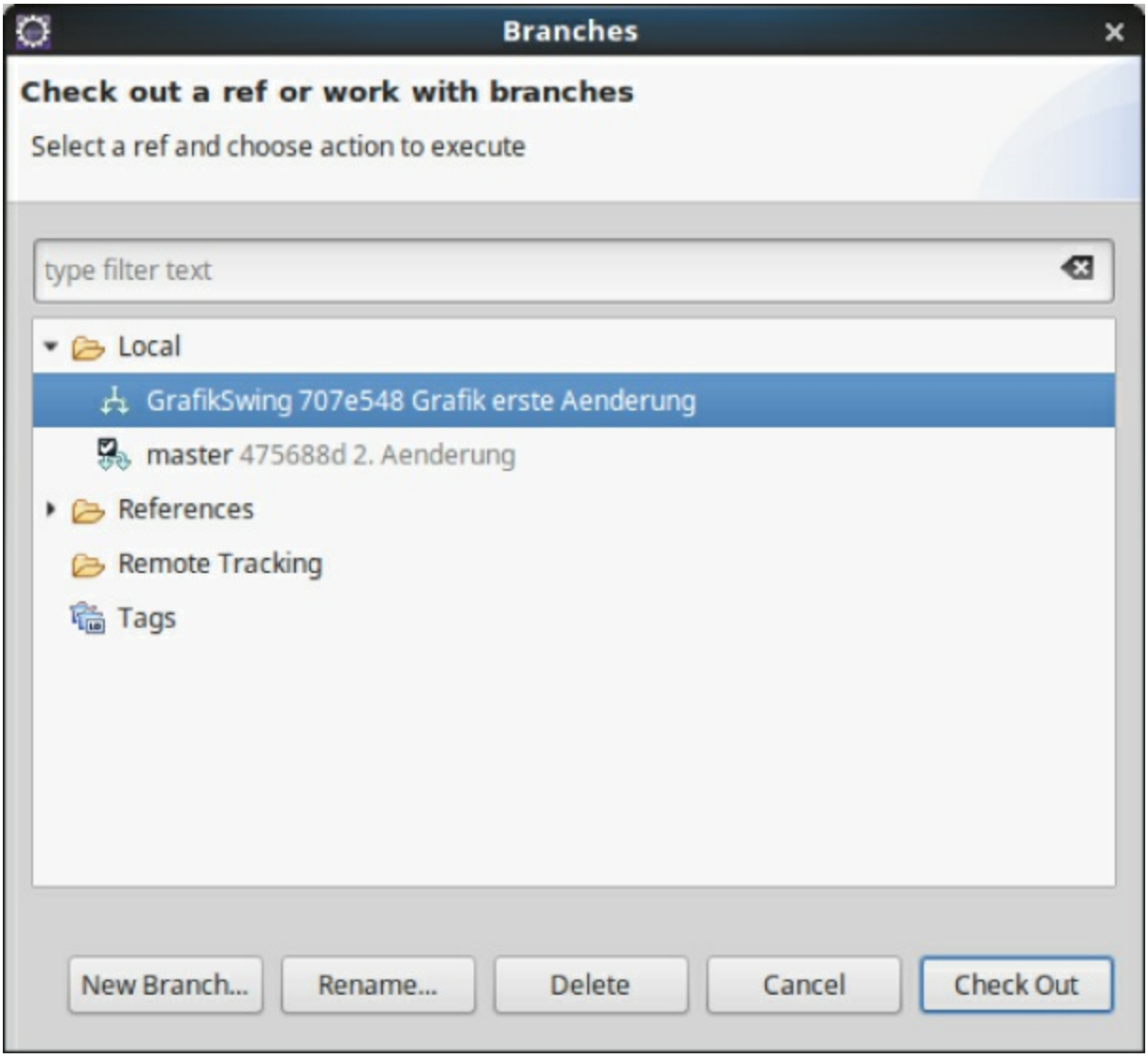


Abbildung 33.2: Eclipse-Ansicht über die Branches über Team|Switch TO|Other

Ein zentrales Repository

Besonders wertvoll wird eine Versionskontrolle, wenn sie mit einem zentralen Repository kombiniert ist. Git bietet verschiedene Möglichkeiten, ein zentrales Repository anzubieten.

- ✔ Die einfachste Variante entsteht dadurch, dass man ein Repository auf ein zentrales Netzwerklaufwerk stellt. Es ist sogar möglich, das zentrale Repository auf einem externen Laufwerk anzulegen.
- ✔ Sehr viel sicherer und professioneller ist es natürlich, einen Git-Server aufzusetzen, der über Netzwerkprotokolle ssh, ftp oder http (https) arbeitet. Einer der bekanntesten Vertreter ist sicher GitHub (`www.github.com`), das viele Open-Source-Projekte verwaltet.
- ✔ Die Vorstellung, dass die eigenen Quellcodes irgendwo im Internet auf einem amerikanischen Server liegen, wird vielen Software-Firmen nicht behagen. Einen ähnlichen Server wie GitHub im eigenen Netzwerk bietet die Open-

Source-Lösung GitLab (`www.gitlab.com`) an. Sie kann über einen Browser verwaltet werden.

- Eine sehr schlanke Lösung stellt Gitolite dar. Es ist als Open Source bei jeder Linux-Distribution vorhanden und kann ohne grafische Oberfläche eingesetzt werden. Gitolite verwaltet mehrere Benutzer und wird durch eine Konfigurationsdatei gesteuert, die selbst in einem Repository gepflegt wird. Änderungen daran werden durch einen Push gültig.

Der Zugriff auf das zentrale Repository erfolgt immer indirekt über das lokale Repository. Wenn Sie Ihre lokalen Commits auf das zentrale Repository übernehmen wollen, führen Sie einen *Push* durch. Um in den Genuss der Änderungen der Kollegen zu kommen, erfolgt ein *Pull*.

Authentifizierung

Die zentrale Versionskontrolle lässt natürlich sofort die Frage aufkommen, wie die Anwender gegenüber dem zentralen Repository ihre Berechtigung nachweisen.

- Die einfachste Lösung ist, dass sie sich gar nicht ausweisen. Das kann beispielsweise in kleinen Firmen der Fall sein, die ihr zentrales Repository einem Windows-Netzwerkdienst (SMB/SAMBA) anvertrauen. In diesem Fall wird eine file-URI verwendet.

  Diese Lösung ist auch bei großen Open-Source-Projekten üblich, allerdings ist der nicht authentifizierte Zugriff dort nur lesend möglich. Hier wird ftp oder http als Protokoll eingesetzt.
- Die Protokolle ftp und http bieten die Möglichkeit, sich über über Benutzerkennung und Passwort auszuweisen. Beispielsweise wird dies von GitHub und GitLab verwendet.
- Typischerweise wird der Zugriff aber über das Protokoll ssh ausgeführt und damit auch authentifiziert. In diesem Fall erstellt der Entwickler ein Schlüsselpaar, übergibt den öffentlichen Schlüssel an das zentrale Git und hinterlegt den privaten Schlüssel auf dem Entwicklungsrechner. Auf diese Weise ist es möglich, ohne ständige Passworteingabe mit dem zentralen Repository zu arbeiten. Diese Authentifizierung wird von Gitolite eingesetzt, aber auch GitLab und GitHub unterstützen sie.
- Um sicherzugehen, dass der Arbeitsplatz nicht von einem Fremden für die Authentifizierung in Abwesenheit des Berechtigten missbraucht wird, kann bei der Definition des Schlüssels auch eine Passphrase hinterlegt werden, die dann bei jedem Zugriff zusätzlich zum Schlüssel eingetippt werden muss.

Erstellen eines Schlüsselpaares

Für das Erstellen eines Schlüsselpaares verwendet man das Programm `ssh-keygen`.

Bei Linux gehört dies zum Standardlieferumfang von ssh dazu. Bei anderen Plattformen finden Sie es als Bestandteil der Git-Clients wie beispielsweise Git for Windows. Für die Erstellung benötigen Sie eine Kommadozeileneingabe. Bei Git for Windows ist sie extra für solche Zwecke beigefügt. Von der Kommandozeile aus rufen Sie `ssh-keygen` auf und schließen den Befehl mit der Taste Return ab.

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/arnold/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/arnold/.ssh/id_rsa.
Your public key has been saved in /home/arnold/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:DkPwvgWnthuDIsgmTPio+IEPcc2AppBNpXwNZqJdNsQ arnold@tab
```

Während der Erstellung werden Sie gefragt, ob Sie eine Passphrase anlegen wollen. Falls Sie einen passwortfreien Umgang möchten, drücken Sie nur die Taste Return. Ansonsten geben Sie die Passphrase ein und bestätigen diese anschließend.

Wenn die Erstellung abgeschlossen ist, finden Sie im Verzeichnis *.ssh* unter dem Benutzerverzeichnis die Dateien *id_rsa* als privaten Schlüssel und *id_rsa.pub* als öffentlichen Schlüssel. Letzteren können Sie beliebig verteilen. Auch dem zentralen Git übergeben Sie diesen. Der private Schlüssel dagegen wird nicht herausgegeben.

Eclipse als Client

Wenn Sie eine zentrale Git-Umgebung haben, können Sie mit Team|Push Branch ‘Master’ Kontakt zu Ihrem zentralen Git aufnehmen.

1. In dem folgenden Dialog übernehmen Sie die URL des zentralen Git-Projekts in das Feld URI. Der Dialog wird die URL sofort in ihre Bestandteile zerlegen und die anderen Eingabefelder füllen.
2. Tragen Sie Benutzerkennung und Passwort Ihres zentralen Git-Kontos ein.
3. Nach zweimaligem Next erreichen Sie den Button Finish.
4. Ein weiteres Mal dürfen Sie User und Password eingeben.
5. Klicken Sie auf Finish.
6. Und noch ein Mal dürfen Sie User und Password eingeben.
7. Es erscheint eine Meldung, die eine erfolgreiche Übertragung bestätigt. Mit OK wird der Dialog beendet.

Wenn Sie einen ssh-Schlüssel oder keine Authentifizierung eingerichtet haben, entfallen natürlich die Eingaben der Benutzerkennungen und Passwörter.

Eclipse-Projekt aus GitLab importieren

Ein Projekt aus einem zentralen Git-Repository kann direkt durch Eclipse

importiert werden. Dazu werden folgende Schritte durchlaufen:

1. Klicken Sie den Menüpunkt File|Import an.
2. Im Dialog unter Git wählen Sie den Punkt Projects From Git an. Mit Next kommen Sie zur nächsten Dialogseite.
3. Der Folgedialog bietet an, zwischen lokalen Repositorys, einer URI und eventuell einem GitHub-Zugriff auszuwählen. Für einen GitLab-Server wird URI angewählt, dann Next.
4. Im Eingabefeld URI können Sie die komplette URL des GitLab-Projekts angeben, wie Sie sie im GitLab finden. Next.
5. Es erscheint eine Liste mit den Branches, die das Projekt anbietet. Next
6. Als lokaler Pfad wird ein Pfad außerhalb des Eclipse-Projekts vorgeschlagen, meist im Verzeichnis *git* unterhalb Ihres Heimatverzeichnisses. Hier liegt dann Ihr persönliches Repository. Weiter kommen Sie mit Next.
7. Wählen Sie Import Existing Projects und dann Next.
8. Beenden Sie den Assistenten mit Finish.

Stand aus GitLab übernehmen

Um auf den aktuellen Stand im GitLab zu kommen, wird ein Pull ausgeführt. Dabei werden gegebenenfalls Konflikte gemeldet, die Sie eventuell von Hand beheben müssen.

1. Rechtsklick auf das Projekt.
2. Im Kontextmenü Team|Pull auswählen.
3. Liegen Änderungen vor, erhält man eine Übersicht über die Commits inklusive der Kommentare.

Git unter Android Studio

Auch das Android Studio ist auf Git vorbereitet. Dies muss aber zunächst aktiviert werden.

1. Wählen Sie aus dem Menü VCS|Enable Version Control Integration.
2. Öffnen Sie die Klappbox und wählen Sie Git aus.
3. Beenden Sie den Dialog.

Als Konsequenz erscheint der Punkt Git im Menü VCS und in der untersten Leiste erscheint ein Punkt Version Control. Darin findet man unter Local Changes die Unversioned Files.

1. Klicken Sie die Unversioned Files mit der rechten Maustaste an und wählen Sie Git|Add. Die Dateien stehen nun unter Default.
2. Klicken Sie Default mit der rechten Maustaste an und wählen Sie dann Git|Commit Files. Ein Dialog erscheint.
3. Das Feld Commit Message muss ausgefüllt werden.
4. Nun können Sie den Button Commit anklicken.

Damit ist das Projekt im lokalen Git abgelegt. Soll das Projekt in ein zentrales Git-Repository übernommen werden, folgen Sie diesen Schritten:

1. Wählen Sie aus dem Hauptmenü VCS|Git|Push. Es erscheint ein Dialog.
2. In dem Dialog steht im linken oberen Feld ein Link auf Define Remote. Klicken Sie ihn an.
3. Ein neuer Dialog geht auf. Als Name wird `origin` vorgeschlagen. Unter URL tragen Sie die URL des zentralen Git-Projekts ein.
4. Es erscheint ein Dialog, in dem der Benutzername und das Passwort des zentralen Git-Servers erfragt werden.
5. Der Push-Dialog hat nun das remote Repository übernommen, aber noch keinen Push ausgelöst. Dies erfolgt über den Button Push.

Teil IX

Der Top-Ten-Teil





Weitere … für Dummies-Bücher finden Sie unter www.fuer-dummies.de.

Kapitel 34

Zehn Dinge

Zehn Hilfen bei der Fehlersuche

1. Sie haben eine Fehlermeldung des Compilers? Herzlichen Glückwunsch! Das ist der einfachste Fall. Die Fehlermeldung besagt, was dem Compiler nicht gefällt. Nun legt der Compiler nicht immer den Finger auf die Wunde. Manchmal liegt er ein paar Zentimeter daneben.
2. Der Compiler schlägt unter Eclipse vor, was man tun kann. Die Vorschläge erreichen Sie, indem Sie den Mauszeiger über dem unterstrichenen Fehler stehen lassen oder indem Sie in das Fehlersymbol links der Zeile klicken. Im Zweifelsfall probieren Sie es aus. Mit der Tastenkombination Strg + Z können Sie den Vorschlag wieder rückgängig machen.
3. Eine Bedingung will hartnäckig rot bleiben? Ein immer wieder auftretender Fehler ist, dass in der Bedingung nur ein Gleichheitszeichen statt zweier verwendet wird (= statt ==).
4. Der Compiler findet alles gut, aber das Programm stürzt mit einer Exception ab? Super! Damit finden Sie in der ersten Zeile, welcher Fehler aufgetreten ist. Danach kommt die Stelle, wo das Programm gestorben ist, und in aufsteigender Reihenfolge, wer die Methode aufgerufen hat. Gehen Sie so lange weiter, bis Sie die erste Zeile finden, die zu Ihrem Programm und damit zu Ihrem Verantwortungsbereich gehört. Unter Eclipse können Sie durch Anklicken dorthin gelangen. Der Compiler meldet die Datei und die Zeilennummer, so dass Sie auch ohne Eclipse die Fehlerstelle finden können.
5. Eine `NullPointerException` deutet darauf hin, dass das Objekt nicht erzeugt wurde. Vermutlich haben Sie eine `new`-Anweisung vergessen.
6. Die `ArrayOutOfBoundsException` zeigt an, dass der Index auf das Array jenseits der Array-Größe war.
7. Eine `InputMismatchException` deutet auf eine fehlerhafte Eingabe, manchmal auch aus der Datei hin: Buchstaben, wenn Zahlen erwartet werden. Der Klassiker: Auf einem deutschen Computer erwartet beispielsweise `Double.parseDouble`, dass der Anwender ein Komma als Dezimaltrenner verwendet.
8. Läuft Ihr Programm in einer Endlosschleife? Dann sollten Sie sich die Variablen der Bedingung der betroffenen Schleife anschauen. Wird die Variable

innerhalb der Schleife auch irgendwann verändert? Läuft die Variable in die falsche Richtung?

9. Wenn Ihnen nicht klar ist, ob bestimmte Teile Ihres Programms wirklich durchlaufen werden oder ob bestimmte Variablen den korrekten Inhalt haben, spicken Sie Ihr Programm mit `System.out.println`-Anweisungen!
10. Sind Sie immer noch ratlos, setzen Sie einen Debugger ein. Setzen Sie Breakpoints und prüfen Sie, ob die Variablen wirklich die Werte enthalten, die Sie erwarten.

Zehn ganz persönliche Dinge

Ich wollte schon immer mal in einem Fachbuch ein Schlusskapitel schreiben, bei dem sich Held und Heldin in den Armen liegen und heiraten, um dann Kinder zu zeugen, die für meine Rente sorgen. Aber wenn man über objektorientierte Programmierung schreibt, ist das virtuelle Überleben der Protagonisten spätestens beim Thema Vererbung gefährdet. Es gibt also kein Happy End. Ist das nicht traurig?

Wir sind nun am Ende. Vielleicht haben Sie schon vorab hier nachgeschlagen, um herauszufinden, wie es ausgeht. Dann haben Sie eindeutig geschummelt. Aber keine Sorge: Ich petze nicht.

Sollten Sie das Buch komplett durchgearbeitet haben, möchte ich Ihnen nun ein paar kluge Worte zum Schluss sagen.

1. Ich gratuliere Ihnen dazu, all meine Bemerkungen und Kalauer durchgestanden zu haben.
2. Falls ich einen Kalauer ausgelassen haben sollte, schicken Sie ihn mir. Vielleicht will der Verlag ja tatsächlich noch eine weitere Auflage von mir haben.
3. Falls Sie keine Fehler im Buch gefunden haben, haben Sie zu flüchtig gelesen. Fangen Sie noch einmal von vorn an!
4. Sollten Sie doch Fehler gefunden haben, ärgern Sie sich nicht, sondern senden Sie sie mir. Manchmal bin ich kritikfähig und beantworte sogar Fragen. In jedem Fall hilft es mir, zukünftige Bücher besser zu machen.
5. Wenn Sie sich über die Fehler richtig ärgern, dann senden Sie diese bitte an den Verlag. Der sammelt sie, kürzt mein Honorar und fordert mich auf, das Buch bei der nächsten Auflage zu verbessern.
6. Erzählen Sie allen Ihren Freunden und Bekannten, dass dies ein ganz tolles Buch sei. Dann steigt die Auflage und das macht dann vielleicht die Honorarkürzungen aufgrund der vielen Fehler wieder wett.

7. Seien Sie nicht völlig überrascht, wenn Sie trotz der vielen Fehler etwas gelernt haben sollten. Das war der ursprüngliche Sinn dieses Buches.
8. Falls Sie erste Schritte machen wollen, selbst zum Autor zu werden, schreiben Sie doch eine Rezension zu diesem Buch. Dabei kommt es entscheidend auf Ehrlichkeit an – sofern Sie das Buch gut finden.
9. Ich weiß nicht, wie es Ihnen mit diesem Buch gegangen ist. Ich hatte meinen Spaß – Sie hoffentlich auch.
10. Wenn Sie nun noch Informationen suchen, kommen Sie zu spät: Jetzt folgt nur noch das Stichwortverzeichnis.

Arnold Willemer

Stichwortverzeichnis

A

abs 1
ActionEvent 1
ActionListener 1
actionPerformed 1
addActionListener 1
addWindowListener 1
Alert (FX) 1
Android 1
    Button 1
    Combobox 1
    EditText 1, 2
    Focus 1
    Klappbox 1
    onClick 1
    OnClickListener 1
    onCreate() 1
    Paint 1
    RadioButton 1
    RadioGroup 1
    RelativeLayout 1
    Spinner 1
Android Studio 1, 2
Android-Emulator 1
Anhang bei E-Mails 1
Annotation 1
Anonyme Klasse 1
Application 1

Application (FX) 1
Application Server 1, 2
Array 1, 2, 3, 4
    mehrdimensional 1
    new 1
    sortieren 1
    String 1
Array.asList 1
ArrayList 1
ASCII 1
Attribut 1, 2
Ausdruck 1
Ausnahmen 1
Autoboxing 1
Autowrapping 1
AVD-Manager 1
AWT 1

B

Backing Bean 1
Baum
    binär 1
Bean 1
Bermuda 1, 2
    Klassenlos 1
    Swing 1
Bermuda (FX) 1
Betrag 1
Bezeichner 1
Bild als Projekt-Ressource 1
binarySearch 1, 2

Binärer Baum 1
Block 1
BMI 1, 2
Body-Mass-Index 1
Bogenmaß 1
Boolean 1, 2, 3
BorderLayout 1
BorderPane (FX) 1–2
Box 1
BoxLayout
    Swing 1
break 1
Breakpoint 1
BufferedReader 1
BufferedWriter 1, 2
Bug 1
Button 1
Button (FX) 1
ButtonGroup 1
Byte 1, 2

C

Calendar 1
Canvas (FX) 1
CardLayout 1
case 1
Casting 1
catch 1, 2
ceil 1
char 1, 2
Character 1

charAt (String) 1
CheckBox (FX) 1
Checkbox (Swing) 1
ClassNotFoundException 1
clearRect
    Swing 1
Client 1
Clipping 1
clone 1
Collection 1
Color
    Swing 1
Combobox 1
ComboBox (FX) 1
Comparable 1, 2
Comparator 1
compareTo 1
Compiler 1
ConcurrentModification Exception 1
Connection (SQL) 1
ConstraintLayout (Android) 1
Constraints (FX) 1
contains (String) 1
continue 1
copyArea 1
cos 1
CRUD 1
CSS (FX) 1

D

Dalvik Virtual Machine 1

DAO 1
Data Access Object 1
Date 1
    Datenbank 1
Datei 1
    close 1
    delete 1
    listFiles 1
    Random-Access 1
    read 1, 2
    readLine 1
    rename 1
    Sequenziell 1
    Verzeichnis 1
    write 1, 2
Dateiauswahldialog 1
Dateisystem 1
Datenbank 1
dBase 1
De Morgan 1
Debugger 1
Derby 1
descendingIterator 1
do 1
Double 1, 2
draw3DRect 1
drawArc 1
drawImage 1
drawLine 1
drawOval 1
drawRect 1
drawRoundRect 1

drawString 1
Duration 1
DVM 1
Dynamische Bindung 1

E

E-Mail 1
    Anhang 1
EAR 1
Eclipse 1, 2
    Bilder einbinden 1
    FX 1
    JPA-Projekt 1
EditText (Android) 1
Eingabe 1, 2
    Konsole 1
Ellipse 1
else 1
ensureCapacity 1
Entity (JPA) 1
Entity-Manager (JPA) 1
enum 1
Enumeration 1
equals 1, 2, 3, 4
equals (String) 1
equalsIgnoreCase 1
Euklid 1, 2
Exception 1
    finally 1
    getMessage 1
    throws 1

toString 1
unchecked 1
Exklusives Oder 1
exp 1
Exponent 1
extends 1

F

Facelet 1
Fallunterscheidung 1
false 1
Fenster schließen 1
FIFO 1
File 1
FileChooser (FX) 1
FileFilter 1
FileInputStream 1
FileOutputStream 1
FileReader 1
FileWriter 1
fill 1
fill3DRect 1
fillArc 1
fillOval 1
fillRect 1
fillRoundRect 1
filter 1
final 1
finally 1
Fließkommazahl 1
Float 1, 2

floor 1
FlowLayout
    Swing 1
FlowLayout (Swing) 1
FlowPane (FX) 1, 2
Fokus 1
Font 1
FontMetrics 1
for 1
forEach 1
Foreign Key 1
Fragezeichen 1
FunctionalInterface 1
Funktion 1
FX
    Alert 1
    BorderPane 1–2
    Button 1
    CheckBox 1
    ComboBox 1
    Constraints 1
    FileChooser 1
    FlowPane 1, 2
    GridPane 1, 2
    Kontrollelemente 1
    Label 1
    ListView 1
    Menü 1, 2
    ObservableList 1, 2, 3
    Pane 1
    RadioButton 1
    StackPane 1

Standarddialoge 1
TableColumn 1
TableView 1, 2
TabPane 1
TextField 1
TextInputDialog 1
ToggleButton 1
ToggleGroup 1
FXML 1

G

Ganze Zahlen 1
Garbage Collection 1
Generics 1
Geschweifte Klammern 1
getColor 1
getFontMetrics 1
getImage 1
ggT 1
Glassfish 1
Glassfish-Tools 1
Google Maps 1
Grafische Oberfläche 1
Graphics 1, 2
Graphics2D 1
Area 1
Ellipse2D 1
Farbverlauf 1
Line2D 1
Point2D 1
Rectangle2D 1

GridBagConstraints 1
GridBagLayout 1
GridLayout (Swing) 1, 2
GridPane (FX) 1, 2

H

Hallo Welt 1
Hash 1
hashCode 1, 2
HashMap 1
HashSet 1
Hashtable 1
hasPrevious 1
HeadlessException 1
Host 1
HTML
    FORM 1
HTTP 1
    DELETE 1, 2
    GET 1, 2, 3, 4, 5, 6
    Kommandos 1, 2
    POST 1, 2, 3, 4
    PUT 1, 2, 3, 4
HttpServletRequest 1, 2
HttpServletResponse 1, 2
HttpURLConnection 1, 2

I

Icon 1
IDE 1
if 1

Image 1, 2
ImageIcon 1, 2
import 1
Index 1
indexOf (String) 1
IndexOutOfBoundsException 1
Inkrementieren 1
Insets 1
Instant 1
int 1
Integer 1
Interface 1, 2
Interpreter 1
Interrupt 1
InterruptedException 1
IOException 1, 2
isEmpty (String) 1
Iterator 1, 2
    descendingIterator 1

J

jar 1, 2, 3
    Einbinden 1
JAR-Datei 1
Java Build Path 1
Java Collection Framework 1
Java Development Kit 1, 2
Java Enterprise Edition 1
Java FX 1
Java Persistence API (JPA) 1
Java Runtime Environment 1, 2

Java Server Faces 1
Java Server Pages 1
Java Style Guide 1
java.sql.Date 1
java.time 1
javac 1
Javadoc 1, 2
JavaMail 1
JAX-RS 1
JAX-WS 1
JAXP 1
JButton 1
JCheckBox (Swing) 1
JComboBox 1
JDBC 1, 2, 3, 4, 5
    Treiber 1
    Verbindung 1
JDK 1, 2
JEE 1
Jersey
    JARs einbinden 1
JFileChooser 1
JFrame 1
JLabel 1
JList (Swing) 1
JMenu 1
JMenuBar 1
JMenuItem 1
join 1
JPA 1, 2
JPA-Projekt (Eclipse) 1
JPanel 1

JPasswordField 1
JRadioButton 1
JRE 1, 2
JScrollPane 1, 2
JSF 1
JSON 1
JSP 1
JSplitPane 1
JTabbedPane 1, 2
JTable (Swing) 1
JTextArea 1, 2
JTextField 1
JTree (Swing) 1
JUnit 1

K

Kapazität 1, 2
Katze 1
KeyListener 1
keyPressed 1
keyReleased 1
keyTyped 1
Klammern 1
Klammern, geschweift 1
Klasse 1–2
Klassenaufteilung 1
Kollision 1
Kommentar 1
Konfigurationsdatei 1
Konstanten 1
Konstruktor 1

Kontrollelemente (FX) 1
Kontrollelemente (Swing) 1
Kreis 1
Kritischer Bereich 1, 2

L

Label (FX) 1
Lambda 1, 2, 3
Laufzeitumgebung 1
Layout 1
    FlowLayout 1
Leichtgewichtsprozess 1
length (String) 1
LIFO 1
line separator 1
LinkedHashSet 1
LinkedList 1, 2
Linux 1
List 1
ListIterator 1
ListView (FX) 1
Literal 1, 2
LocalDate 1, 2, 3
LocalDateTime 1
LocalTime 1
log 1
Logarithmus 1
Logischer Operator 1
Long 1, 2
Lotto 1
LValue 1

M

Macintosh 1

Mail 1

main 1

Manifest 1

Map 1

- containsKey 1
- keySet 1
- Methoden 1
- put 1

Math 1

Mausereignisse 1

Mehrdimensionale Arrays 1

Mehrfachvererbung 1

Menü (FX) 1, 2

Menüleiste 1

Messagebox 1, 2

Methode 1, 2, 3, 4

MIME 1

Minimum 1

mkdir 1

Model View Controller (MVC) 1

Modifizierer 1

Modulo 1

mouseClicked 1

mouseDragged 1

mouseMoved 1

mousePressed 1

mouseReleased 1

MVC (Model View Controller) 1

N

Nassi-Schneidermann-Diagramm 1

Negieren 1

NetBeans 1

Netzwerkprogrammierung 1, 2

new

- Array 1
- Klasse 1

nextLine 1

NOT 1

notify 1

null 1, 2

NullPointerException 1

O

Object 1, 2

Object-Relational Mapping (ORM) 1

ObjectInputStream 1

ObjectOutputStream 1

Objektorientierte Programmierung 1

ObservableList (FX) 1, 2, 3

ODER-Verknüpfung 1

Operatoren 1, 2

Oval 1

P

pack 1

package 1, 2, 3

paint (Swing) 1, 2

Palindrom 1
Pane (FX) 1
Panel 1
Parameter 1
parseDouble 1, 2
parseInt 1
persistence.xml 1
PNG-Datei 1
POJO 1
Polymorphie 1
Port 1
Portabilität 1
PostgreSQL 1
Potenz 1
pow 1
PreparedStatement 1
previous 1
Primitive Typen 1
private 1, 2
Properties 1, 2
protected 1
public 1, 2

Q

Queue 1

R

RadioButton (FX) 1
Radiobutton (Swing) 1
RadioGroup (Android) 1
Random 1

Random-Access-Dateizugriff 1
RandomAccessFile 1, 2
read 1
readLine 1
readObject 1
Rechteck 1
Referenz 1, 2
Regulärer Ausdruck 1
Rekursion 1, 2
Relationales Datenmodell 1
repaint 1
replace (String) 1, 2
replaceAll (String) 1
request 1
response 1
REST 1
return 1
reverse 1
RMI 1
round 1
Rückgabewert 1, 2
RUDI 1
Runden 1
Runnable 1

S

Scanner 1, 2, 3, 4
SceneBuilder 1
Schiebebalken 1
Schleife 1
Scrollbar 1

seek 1
Semikolon 1
Serializable 1, 2, 3
Server 1
Service-Consumer 1
Servlet 1
Set 1
setBounds (Swing) 1
setClip 1
setColor 1
setDefaultCloseOperation 1
setEditable 1
setElementAt 1
setLayout (Swing) 1
setSize 1
setVisible 1
Short 1, 2
showInputDialog 1
showMessageDialog 1, 2
shuffle 1
Signatur 1
sin 1
Singleton 1
Skalieren 1
sleep 1
SMTP 1
SOAP 1
SOAP-Client 1
Socket 1, 2
sort 1, 2
Sortieren 1
Sortierung 1

Späte Bindung 1
SQL 1, 2
    DELETE 1, 2
    INSERT 1
    SELECT 1
    UPDATE 1
    WHERE 1
sqrt 1
ssh-keygen 1
Stack 1
StackPane (FX) 1
Standarddialoge (FX) 1
Standardkonstruktor 1
static 1
Stream (Collection) 1
Stream (Datei) 1
String 1
    Array 1
    charAt 1
    contains 1
    equals 1
    equalsIgnoreCase 1
    indexOf 1
    isEmpty 1
    length 1
    replace 1, 2
    replaceAll 1
    substring 1
    trim 1
    Typwandlung 1
StringBuffer 1
StringBuilder 1, 2

Stringliteral 1
Struktogramm 1
substring 1
super 1, 2
Swing 1
    BoxLayout 1
    Checkbox 1
    clearRect 1
    Color 1
    drawArc 1
    drawOval 1
    drawString 1
    Fensterbreite 1
    fillArc 1
    fillOval 1
    FlowLayout 1, 2
    FocusListener 1
    getColor 1
    GridLayout 1
    JCheckBox 1
    JList 1
    JTable 1
    JTree 1
    setBounds 1
    setColor 1
    setLayout 1
    showInputDialog 1
    showMessageDialog 1
    Timer 1
switch 1
synchronized 1, 2

T

TableColumn (FX) 1
TableView (FX) 1, 2
TabPane (FX) 1
tan 1
Tastatur 1
Texte 1
TextField (FX) 1
TextInputDialog (FX) 1
this 1
Thread 1–2
    interrupt 1
    isInterrupted 1
    join 1
Thread.sleep 1
Throwable 1
throws 1
ToggleButton (FX) 1
ToggleGroup (FX) 1
Tortendiagramm 1
toString 1, 2
Transaktionen 1
transient 1
TreeMap 1
TreeSet 1, 2
trim (String) 1
trimToSize 1
true 1
try 1
Typsicherheit 1
Typwandlung (String) 1

U

überladen 1
überschreiben 1
UML 1
UND-Verknüpfung 1
unsign 1
util 1

V

Variable 1
Variablennamen 1
Vector 1
Vererbung 1, 2
Vergleichsoperator 1
Verkettete Liste 1
Verzeichnis 1, 2
Virtuelle Maschine 1
Visitor 1
Vorgabekonstruktor 1

W

wait 1
WAR 1
Warnung 1
Web-Client 1
WebLogic 1
WebSphere 1
while 1
Wildfly 1

WindowAdapter 1
WindowListener 1
Windows 1
Winkelfunktionen 1
Wrapper 1
write 1
writeObject 1
WSDL 1

X

XML 1, 2, 3
    Document 1
    Einlesen 1
    Schreiben 1
XOR 1

Z

Zeichen 1
Zeichenketten 1
Zeilentrennzeichen 1
ZonedDateTime 1
ZoneId 1
Zufallszahlen 1
Zuweisung 1

WILEY END USER LICENSE AGREEMENT

Besuchen Sie www.wiley.com/go/eula, um Wiley's E-Book-EULA einzusehen.

Table of Contents

Titelseite 5
Impressum 6
Einleitung 22
- Über dieses Buch 22
- Konventionen in diesem Buch 23
- Törichte Annahmen über den Leser 23
- Wie dieses Buch aufgebaut ist 24
- Symbole, die in diesem Buch verwendet werden 25
- Wie es weitergeht 26

Teil I: Programmieren lernen 27
- Kapitel 1: Ein Loblied auf Java 28
  - Das Profil von Java 28
  - Programmiersprache, Interpreter und Compiler 30
  - Die virtuelle Maschine und das ultimative Wohnmobil 31
  - Vergleich mit anderen Programmiersprachen 32
- Kapitel 2: Entwicklung mit IDEen und ohne 34
  - Ein kleines Programmbeispiel als Opfer 34
  - Java-Programmierung mit dem JDK 36
  - Eclipse 40
  - NetBeans 47
  - Aufgabe: Grüßaugust selbst erstellen 51
- Kapitel 3: Aufstieg zum Programmierer 52
  - Java-Programm-Skelett 52
  - Programmerweiterungen 54
  - Variable Attribute 58
  - Kleines Rechenwerk 67
  - Zeichen 71
  - Ordnende Elemente 75
  - Aufgaben 80
  - Anmerkungen 82
- Kapitel 4: Entscheiden und Wiederholen 83
  - Nur unter einer Bedingung 83
  - Logeleien 87
  - Die Fallunterscheidung – switch case 95
  - Endlich lernen wir Schleifen zu binden! 98

Überblick behalten mit Struktogrammen 108
Aufgaben 111
Kapitel 5: Funktionierende Methoden 115
Vorstellung der Methode main 115
Methoden gliedern und sparen Arbeit 116
Salzkartoffeln kochen für Anfänger 118
Methoden mit Datenfluss 120
Überladen von Methoden 123
Rekursionen 125
Aufgaben 131
Kapitel 6: Die Strings: Viele, viele Buchstaben 132
Variablen und Texte 132
Texteingabe 133
Textmanipulationen 134
Typwandler 141
StringBuffer und StringBuilder 142
Beispiel Pfadumbau 144
Aufgaben 147
Kapitel 7: Arrays 150
Wir bauen ein Array 151
Übung: Ein Array für die Monatstage 153
Übung: Ein Kartenspiel 155
Das String-Array der Methode main 157
Aufbruch in neue Dimensionen 158
Aufgaben 160
Kapitel 8: Bermuda: Die Schiffe und das Meer 161
Spielregeln 161
Arrays für Spielfeld und Schiffe 162
Der Spielablauf 165
Das komplette Listing 169
Teil II: Objektorientiert und modular 177
Kapitel 9: Klasse Objekte 178
Ein Beispiel für objektorientierte Programmierung 178
Eine Klasse im Selbstbau 180
Selbstverweisende Klassen 190
Die Vorteile der Privatsphäre 198
Konstruktionsbedingt 204
Statische Attribute und Methoden 207

Wrapper-Klassen für Primitive 209
Aufzählungstyp mit enum definieren 210
Aufgaben 212
Kapitel 10: Klassen erweitern 216
Noch mal Klasse: Der Mensch als Objekt 216
Referenzen auf erweiterte Objekte 218
Konstruktoren in der Vererbung 219
Die Mensa der Universität Norgaardholz 220
Der Beruf wird geschützt 230
Anonyme Klassen und das Lambda 231
Aufgaben 233
Kapitel 11: Verteilung der Klassen auf Pakete 237
Packages 237
Ausführbare Dateien und Bibliotheken: JAR 243
Kapitel 12: Bermuda-Klassen 249
Das Projekt erstellen 249
Die Klassen 250
Teil III: Erweitern Sie den Horizont 266
Kapitel 13: Hoffentlich Exception-versichert 267
Ausnahmsweise Fehler 267
Eingabe einer Zahl per Dialogbox 268
Der catch-Parameter 273
Ein gelungener Wurf 276
Der Stammbaum der Exceptions 278
Die Katzen-Exception 280
Nochmal clone 286
Aufgaben 287
Kapitel 14: Fertige Bausteine zur Selbstbedienung 288
Wie der Zufall es will 288
Das Datum, der Kalender und die Neuerungen 293
Rechnerisches aus der Math-Klasse 303
Hilfsmethoden für Arrays 305
Kapitel 15: Datenelemente sammeln und anordnen 308
Das flexible Array: ArrayList 309
Ein altes flexibles Array: Der Vector 314
In Ketten: Die LinkedList 315
Sammlung der Einzigartigen: Set 324

Zugriff per Schlüssel: Der assoziative Speicher 333
Die Interfaces 338
Die Methoden des Java Collection Frameworks 341
Wie sortiert man Autos? 343
Lambda-Ausdrücke und Streams 347
Hinweis 353
Teil IV: Fenster- und Grafikprogrammierung 354
Kapitel 16: Mit Swing Fenster, Mäuse und Grafik programmieren 355
JFrame: Guck mal durchs Fenster 355
Zeichnen und Malen 357
Modernisierung: Graphics2D 384
Ereignisreichtum 390
Aufgaben 403
Kapitel 17: Erstellen eines Layouts 404
Alle bitte nebeneinander aufstellen: FlowLayout 405
Gleichmäßig vergittert: GridLayout 406
Die Mitte wird fett: BorderLayout 408
Kombination der Layouts mit Panels 410
Das Layout der Dialoge: GridBagLayout 420
Das BoxLayout und die Box 422
Null-Diät für Layout-Manager 426
Aufgaben 427
Kapitel 18: Kontrollelemente im Swing 428
Hochgradiger Simpel: Das Label 428
Das Auffangen der Ereignisse 430
Druckvolle Buttons 432
Ein BMI-Programm als Spielwiese 434
Eine doch nicht so einfache Liste 449
Der JTable und sein Modell 453
Bäume pflanzen mit JTree 462
Check den Button 466
Menüs und Dateidialoge 470
Der Fokus und der Multiliner 474
Aufgaben 476
Kapitel 19: Bermuda im Swing 479
Klassenbildung 479
Grafik aus Images 480
Ein Fensterrahmen für Bermuda 481

Das Bermuda-Panel 482
Erstellen einer Bermuda-JAR-Datei 487
Kapitel 20: Java FX, das neue Swing? 489
Mit Eclipse zur FX-Anwendung 490
Bühnendekoration 492
Mit NetBeans zur Java-FX-Anwendung 494
Ein Button als Kontrollelement 497
Mit dem Pane um Fassung ringen 499
Es kommt Leben in die Bude: EventHandler 517
Ein paar Kontrollelemente 521
Standarddialoge 550
Schickmachen mit CSS 560
Komponieren in FXML im SceneBuilder 563
Aufgaben 571
Kapitel 21: Bermuda im FX-Kleid 572
Das Hauptprogramm 572
Das Canvas für die Grafikdarstellung 574
Kapitel 22: Android 578
Alles ist ein wenig anders 578
Arbeitsumgebung schaffen 579
Gestaltung einer App 593
Java macht die Activity erst aktiv 606
Ereignisse der Kontrollelemente 612
Texte aus dem Java-Programm lesen 615
Bermuda auch auf Android 621
Und nun? 639
Teil V: Daten längerfristig speichern 641
Kapitel 23: Dateien halten die Daten frisch 642
Daten aus Textdateien lesen 642
Daten in einer Textdatei 646
Serialisierte Klassen 651
Spielstandsicherung für Bermuda 654
Properties als Konfigurationsdatei 656
Auf Strings über Strings zugreifen 657
Properties als Konfigurationsdatei 658
Random-Access-Dateizugriffe 659
Umgang mit dem Dateisystem 667

Verzeichnisse auslesen 670
Aufgaben 676
Kapitel 24: Datenbankzugriff aus Java 677
Datenbanken und ihre Tabellen 677
Eine Datenbank für die Bonbonbude 678
Wenn Datenbanken einmal groß werden 693
Transaktionen 697
Datenbankunterstützung durch Eclipse 698
Aufgaben 700
Java Persistence API: JPA 700
Kapitel 25: XML-Dateien 710
Die Strukturen von XML-Dateien 710
Einlesen einer XML-Datei 713
XML-Dateien schreiben 722
Eine beliebige XML-Datei auslesen 726
Kapitel 26: JSON 730
Der JSON-Sprachschatz 730
JSON in Java 731
Aufbau der JSON-Struktur 732
Klassen sorgen selbst für JSON 735
Frag mal Google Maps 744
Teil VI: Praktisches 747
Kapitel 27: Javabohnen als Datenmodell 748
Serialisierbarkeit 749
Attributänderungen und Meldung 749
Einspruch! 753
Kapitel 28: Parallelverarbeitung mit Threads 756
Erst einmal Schlaf nachholen 757
Leichtgewichtige Threads 757
Kollisionen in der Vasenfabrik 763
Warten auf Kartons 766
Wir warten nicht auf Godot: Time-out 769
Prozessoren rösten 771
Multitasking in der GUI 775
Aufgabe 778
Teil VII: Netzwerke und Internet 779
Kapitel 29: Netzwerkprogrammierung 780

Das Liebesgeflüster der Sockets 780
Mails versenden 785
Java als Web-Client 789
Java ruft Java: RMI 791
Time-out und Parallelität 796
Kapitel 30: Heidis Berge und die Enterprise Edition 798
Das Konzept des Application Servers 799
Vorbereitung und Installation 802
Glassfish spielt Indianer und Servlet 810
Java Server Pages 820
Java Server Faces 829
Der Gast und die Datenbank 836
Datenbankanwendung per JPA 856
Jersey und der ganze REST 866
SOAP: Ferngespräch in XML 890
Teil VIII: Werkzeuge 900
Kapitel 31: Dem Fehler auf der Spur 901
Alles so schön rot hier 901
Gelbe Warnungen 902
Der Eclipse-Debugger 902
Kapitel 32: Automatisierte Tests: JUnit 905
Konfiguration von Eclipse 905
Erstellen eines TestCase 907
Annotationen 910
JUnit-Testaufbau bei NetBeans 911
Kapitel 33: Die Versionskontrolle Git 912
Ein lokales Repository für Eclipse 913
Branches 915
Ein zentrales Repository 917
Git unter Android Studio 920
Teil IX: Der Top-Ten-Teil 922
Kapitel 34: Zehn Dinge 923
Zehn Hilfen bei der Fehlersuche 923
Zehn ganz persönliche Dinge 924
Stichwortverzeichnis 926
End User License Agreement 950