Java

PEARSON Education

ADDISON-WESLEY

MASTER CLASS

Karsten Samaschke

Java

Einstieg für Anspruchsvolle

ADDISON-WESLEY [in Kooperation mit] PEARSON Studium

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.ddb.de* abrufbar.

10 9 8 7 6 5 4 3 2 1

08 07 06 05

ISBN 3-8273-2180-8

ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany

www.pearson-studium.de
Lektorat: Frank Eller, feller@pearson.de
Einbandgestaltung: Marco Lindenbeck, webwo GmbH (mlindenbeck@webwo.de)
Herstellung: Elisabeth Prümm, epruemm@pearson.de
Satz: mediaService, Siegen (www.media-service.tv)
Druck und Verarbeitung: Kösel, Krugzell (www.KoeselBuch.de)

Printed in Germany

Inhaltsverzeichnis

1

Loslegen ...

Java hin, Java her. Java ist ein richtiges Hype-Thema geworden. Es ist offensichtlich ein Thema, das die Entwickler- und Entscheider-Seelen berührt, denn in den Diskussions-Foren wird ein regelrechter Glaubens-Krieg zwischen Java und seiner direkten Konkurrenz-Technologie .NET ausgetragen.

Wir werden uns in diesem Buch nicht an den Grabenkämpfen der verschiedenen Fraktionen beteiligen. Stattdessen werden wir versuchen, einen wirklich praxisorientierten Zugang zu Java zu bekommen. Aus diesem Grund werden wir uns den wichtigsten Themen der Java-Desktop- und -Netzwerk-Programmierung widmen und uns so Schritt für Schritt das Wissen aufbauen, das wir benötigen, falls wir doch einmal an dem Glaubenskrieg der Microsoft- und Sun-Jünger teilnehmen wollen oder – was durchaus auch im Bereich des Möglichen liegt – Programme mit Java entwickeln müssen.

Nebenbei haben wir das Wissen und die Fähigkeiten erworben, um mit Java produktiv arbeiten zu können. Und möglicherweise können wir mit unserem neuen Java Know-how später auch gutes Geld verdienen ...

Der Autor drückt Ihnen in jedem Fall die Daumen und möchte Ihnen die Sprache Java so nahe bringen, wie er sie auch gelernt hat: Weniger theoretisch, mehr praktisch. Aber immer mit den Informationen, die man benötigt, um effektiv mit Java arbeiten zu können.

1.1 Geschichte von Java

Das darf in keinem Buch über Java fehlen: Die mehr oder weniger ausgeschmückte Geschichte über einen Programmierer bei Sun, der partout nicht mehr mit C und C++ arbeiten wollte und deshalb seine eigene Programmiersprache geschrieben hat.

Also gut, hier ist sie.

James Gosling, Programmierer bei Sun und gerade mit der Arbeit am Projekt „Green“ – einer Software zur Steuerung von im Netzwerk verteilten Geräten – beschäftigt, hatte eines Tages genug von C/C++ und beschloss, für dieses Projekt (und einen Geräte-Prototypen namens Star 7) eine neue Programmier-Sprache zu entwickeln. Diese Sprache sollte mit den Nachteilen von C++ aufräumen und vor allem sehr netzwerkorientiert arbeiten. Also schloss er sich in seinem Büro ein und entwickelte Java.

Aufgrund des Anforderungsprofils von „Project Green“ musste Java klein, portabel und zuverlässig sein. Die Sprache brauchte zunächst keine Rücksicht auf Ausgaben oder Benutzer-Schnittstellen zu nehmen, war sie doch auf den Einsatz in Embedded Devices ausgerichtet.

Dumm nur, dass sich Java und das „Project Green“ nicht durchsetzen konnten. Also sah es sehr ungut für Java aus und die Sache wäre heute sicherlich weitestgehend vergessen, hätte es nicht eine andere Entwicklung gegeben, die genau die gleichen Eigenschaften von Applikationen erwartete, wie dies auch bei Project Green der Fall war: das Internet. Java schien wie geschaffen für das Internet, denn mit Java entwickelte Applikationen sind klein, sicher und portabel.

Sun schaltete schnell und veröffentlichte ein Java Development Kit sowie einen Java-basierenden Browser (HotJava), der Applets ausführen konnte. Dies beeindruckte Netscape und auch Microsoft, die daraufhin Java für ihre Browser lizensierten. Damit hatte der Siegeszug von Java begonnen, denn plötzlich konnten interaktive Elemente über das Internet ausgeführt werden.

Der Rest ist Geschichte: Millionen von Downloads und Hunderttausende von Entwicklern, die in Java programmieren, sorgten für den Durchbruch der Technologie. Und mit dem Aufkommen von Microsofts .NET-Technologie setzte ein Wettlauf ein, der für beide Technologien von Vorteil ist, denn Konkurrenz belebt das Geschäft. Die Zukunft ist rosig für Java – trotz oder gerade wegen der Konkurrenz durch die technologisch sehr ähnliche .NET-Technologie!

1.2 Was ist Java?

Java ist eine Programmiersprache, ähnlich wie C, C++, Basic oder C#. Ursprünglich mit einem starken Fokus auf Web-Entwicklung (Java Applets), haben sich die Schwerpunkte von Java enorm verändert und decken heute einen riesigen Bereich ab. Java wird heute auf verschiedensten Plattformen und in verschiedensten Szenarien eingesetzt:

- Java Applets, die im Browser laufen
- relationale und objektorientierte Datenbank-Systeme
- Mobil-Telefone
- Intelligente Smartcards
- PDAs
- Mainframe-Systeme

- Automobil-Technik
- Fernseher und Set-Top-Boxen
- Webserver
- Desktops

Alle diese Geräte und Technologien haben eines gemeinsam: Für sie gibt es eine so genannte `Java Virtual Machine`, die es erlaubt, dass Java auf ihnen läuft.

Java selbst ist eine objektorientierte Programmiersprache, deren neueste Version `Java 5` heißt. Java-Programme sind plattformunabhängig und können überall dort ausgeführt werden, wo es eine passende `Java Virtual Machine` gibt. Zu diesem Zweck werden Java-Programme durch einen Compiler in den so genannten `Byte-Code` kompiliert.

Dieser `Byte-Code` ist ein Zwischen-Code zwischen dem Text, der eine Java-Applikation definiert, und reinem Maschinen-Code, der plattformspezifisch ist. `Byte-Code` erlaubt es, ein Java-Programm unter Mac OS zu schreiben und ohne Änderungen auf Linux-, Solaris- oder Windows-Systemen ablaufen zu lassen. Diese Ausführung findet durch einen Interpreter statt, der den `Byte-Code` in den plattformspezifischen Maschinen-Code übersetzt und ausführen lässt.

Java bietet ein paar Features, die die Sprache sicherer und besser nutzbar machen, als dies bei anderen Sprachen der Fall ist:

- Java kümmert sich selbst um die Allozierung und Freigabe von Speicher.
- Java kapselt den Speicher vom Programmierer (keine Pointer etc.).
- Java verfügt über ein simples Vererbungs-Modell, das keine Mehrfachvererbung von Klassen zuläßt.
- Java kann leicht um weitere Features erweitert werden.
- Java sorgt sich sehr um Sicherheit – Applets, die auf dem Rechner eines Users laufen, können prinzipiell keinen Schaden anrichten, da sie von Java in ihren Fähigkeiten beschränkt werden.

Java gibt es in drei Editionen, die sich in ihrem Funktionsumfang unterscheiden:

- Java 2 Standard Edition (J2SE), Version 5.0: Die Haupt-Ausgabe von Java, die vor allem für Desktop-Rechner geeignet ist
- Java 2 Enterprise Edition (J2EE): Erweiterte Version von Java für Unternehmenssoftware, Server und andere Projekte, die besondere Anforderungen an Skalierbarkeit und Robustheit der Software stellen. Bietet Möglichkeiten, auf propriertäre Komponenten und Geräte (Hosts, Mainframes etc.) zuzugreifen und mit diesen zu kommunizieren.
- Java 2 Micro Edition (J2ME): Besonders kleine und ressourcenschonende Version von Java, die besonders für Mobiltelefone, PDAs oder anderen Umgebungen mit beschränkten Ressourcen laufen soll

Wir werden im weiteren Verlauf dieses Buches mit der Standard-Version von Java arbeiten.

1.3 Java 2 SE (J2SE) Version 5

Ein großer Sprung: Von `Java 2 SE, Version 1.4` geht es hoch auf `Java 2 Standard Edition, Version 5.0` (so der offizielle Name). Allerdings ist dieser Sprung mehr marketingtechnisch begründet. Java 5.0 (so der gebräuchliche Name) ist eigentlich Java 1.5 (so der richtigere Name) und wurde bis zur ersten Beta auch noch so bezeichnet. Sogar in der API-Dokumentation taucht die Versions-Nummer 1.5 immer noch auf. Und manchmal findet sich sogar noch der Verweis auf Tiger (das war der Code-Name).

Lassen wir also die Kirche im Dorf und betrachten Java 5.0 als das, was es auch ist: eine gründliche und interessante Weiterentwicklung von Java 1.4.

Das ist neu bei Java 5.0:

- Unterstützung für Generics (generische Definition von Klassen und Interfaces)
- vereinfachte Konvertierung zwischen primitiven Typen und den korrespondierenden Objekt-Typen (Boxing/Unboxing)
- Meta-Daten für Klassen, Methoden und Interfaces
- neue for-each-Schleife
- Enumerations (Aufzählungen)
- formatierte Ein- und Ausgaben
- variable Argument-Listen
- Concurrency Utilities
- verbesserter Compiler und besseres Laufzeitverhalten
- neues Swing Look & Feel („Ocean“)

Außerdem gibt es die üblichen Versprechungen: Java 5 soll skalierbarer, performanter und sicherer sein als seine Vorgänger. Überprüfen lassen sich derartige Aussagen noch nicht – zum Zeitpunkt der Drucklegung dieses Buches war Java 5.0 nur als Beta 2 verfügbar.

Wir werden im Rahmen dieses Buches nicht mit allen neuen Features arbeiten können – dazu reicht der Platz nicht aus. Aber zum Einsatz kommen in jedem Fall Generics, das bessere Boxing, die neue for-each-Schleife, Aufzählungen, variable Argument-Listen, das neue Swing Look & Feel und die neuen Ausgabe-Formatierungsmöglichkeiten. Freuen Sie sich darauf, denn diese Features erleichtern die Arbeit mit Java gewaltig!

1.4 Download und Installation

Um Java-Programme zu erstellen, benötigen wir neben dem `Java-Runtime-Environment` (`JRE` – die Laufzeit-Umgebung, in der Java-Programme ausgeführt werden) das `Java 5.0 Development Kit` (`JDK` – die Entwicklungsumgebung, mit der Programme entwickelt werden können).

Das Java 5.0 Development Kit (JDK) kann man sich kostenlos von Suns Java-Homepage unter *http://java.sun.com/j2se/5.0* herunterladen. Es beinhaltet sowohl JRE als auch eine umfassende Dokumentation und die für das Erstellen von Applikationen benötigten Compiler und Debugger. Achten Sie aber darauf, dass Sie wirklich das JDK und nicht nur die JRE herunterladen.

Bei den folgenden Aussagen beschränken wir uns auf Microsoft Windows 2000 oder höher. Die Installation für andere Systeme und Plattformen ist auf der Java-Homepage unter der Adresse *http://java.sun.com/j2se/1.5.0/install.html* ausführlich beschrieben.

Nach dem Download können wir die Installation durchführen lassen. Bei Windows beschränkt sich die Installation darauf, den Installer auszuführen und dort die Standard-Einstellungen zuzulassen. Sie sollten sich allerdings den Installations-Ordner des JDKs merken, denn den benötigen Sie später zwingend.

PATH-Variable konfigurieren

Nach der Installation müssen Sie das JDK möglicherweise noch konfigurieren. Passen Sie zunächst die PATH-Umgebungsvariable an. Unter Windows 2000/XP/2003 finden Sie diese Variable unter SYSTEMSTEUERUNG > SYSTEM > ERWEITERT > UMGEBUNGSVARIABLEN. Sollte sich dort weder unter den Benutzer- noch unter den System-Variablen ein PATH-Eintrag finden lassen, dann fügen Sie den Pfad

<JDK-Installationsordner>\bin

als PATH-Umgebungsvariable hinzu:

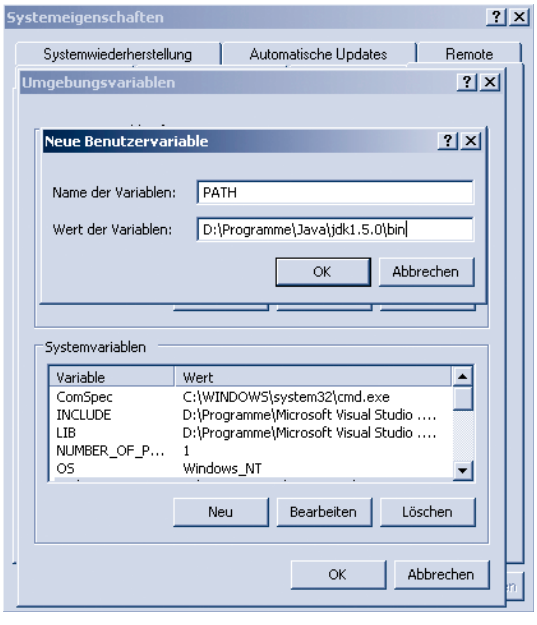


Abbildung 1.1
Setzen der Umgebungsvariable PATH mit dem Pfad zum JDK\bin-Ordner

Wenn eine PATH-Variable bereits existiert, fügen Sie den Pfad zum *bin*-Ordner des JDKs am Ende, getrennt durch ein Semikolon, ein.

CLASSPATH-Variable konfigurieren

Aus Bequemlichkeitsgründen sollten Sie ebenfalls eine CLASSPATH-Variable konfigurieren. Diese CLASSPATH-Variable sollte so aussehen:

.;<JDK-Installationsordner>\lib\tools.jar

Den Platzhalter für den JDK-Installationsordner sollten Sie natürlich durch den tatsächlichen Wert ersetzen. Das Vorgehen ist analog zum Hinzufügen oder Ergänzen der PATH-Variable.

Die CLASSPATH-Umgebungsvariable definiert Verzeichnisse, JAR- und ZIP-Archive, in denen nach Java-Klassen gesucht wird, die in Applikationen verwendet und referenziert werden.

1.5 Entwicklungsumgebung

Prinzipiell können Sie jeden Text-Editor als Entwicklungsumgebung einsetzen – selbst das Windows-Notepad reicht aus. Sie werden allerdings nicht freiwillig etwas wie das Notepad zur Entwicklung einsetzen, schließlich gibt es leistungsfähige und teilweise auch kostenlose Editoren oder gar komplette Entwicklungsumgebungen. Einige dieser Tools sollen hier kurz vorgestellt werden.

1.5.1 IntelliJ IDEA

IntelliJ IDEA in der Version 4.5 unterstützt Java 5 nativ. Der Download installiert auf Wunsch sogar das komplette JDK gleich mit und setzt die benötigen Klassenpfade. IntelliJ IDEA ist eine kostenpflichtige Entwicklungsumgebung, die aber für 30 Tage getestet werden kann. Sie unterstützt die Entwicklungstätigkeit durch AutoCompletion (Auto-Vervollständigung), IntelliSense (Klassen-Member und die Dokumentation einer Klasse werden angezeigt) und anpassbares Syntax-Highlighting.

Die IDE ist weitestgehend konfigurierbar und erlaubt es, Projekte für jede Java-Version anzulegen. Dies bedeutet für den Entwickler, dass auch nur Features und Schlüsselworte erlaubt sind, die für die jeweilige Java-Version zulässig sind. IntelliJ IDEA ermöglicht ein komfortables Debuggen von Applikationen und unterstützt den Entwickler beim Verpacken und Weitergeben seiner Applikation. Unterstützung für die gängigen Hilfstools wie JUnit, Ant oder das CVS ist selbstverständlich vorhanden und sorgt zusammen mit dem leistungsfähigen Refactoring für ein sehr angenehmes Entwickeln.

Der Autor hat den größten Teil der Listings in diesem Buch unter Verwendung von IntelliJ IDEA 4.1 und 4.5 geschrieben und kann die Entwicklungsumgebung für den professionellen Entwickler nur empfehlen.

IntelliJ IDEA kann unter der Adresse *http://www.jetbrains.com/idea/index.html* heruntergeladen werden und für 30 Tage getestet werden. Die Vollversion kostet 499,-- Dollar.

1.5.2 CodeGuide

CodeGuide steht IDEA kaum nach, was die Features und die Leistungsfähigkeit anbelangt. Die IDE unterstützt den Einsatz von Java 1.5 ebenfalls nativ und kann Java 1.5 Code kompilieren. Ebenso kennt sie alle neuen Sprach-Features. CodeGuide ist – ebenso wie IDEA – eine kostenpflichtige Entwicklungsumgebung und kann für 20 Tage kostenfrei getestet werden.

CodeGuide kann unter der Adresse *http://www.omnicore.com/codeguide.htm* heruntergeladen werden. Die Vollversion der Software kostet für einen einzelnen Entwickler 299,-- Dollar.

Ob Sie CodeGuide oder IDEA bevorzugen, ist eine persönliche Entscheidung – IDEA ist flexibler, CodeGuide unterstützt dafür mehr im Hintergrund.

1.5.3 Eclipse

Eclipse ist die ultimative Open-Source-Entwicklungsumgebung für Java. Leider unterstützt Eclipse derzeit nur die Entwicklung von Applikationen bis J2SE 1.4 – eine Unterstützung von J2SE 5 ist erst für die Eclipse-Version 3.1 vorgesehen, die 2005 erscheinen soll.

Allerdings gibt es jetzt schon ein Plug-In für Eclipse, das zumindest die Fehlermeldungen beim Schreiben von Java 5-Code unter Java 1.4 beseitigt und ein Kompilieren der Sourcen mit dem Compiler des JDK für Java 5 erlaubt. Sie können dieses Plug-In unter der Adresse *http://www.genady.net/forum/viewforum.php?f=6* herunterladen, nachdem Sie Eclipse und Java 5 installiert haben. Lesen Sie aber die Anleitung im Forum genau und befolgen Sie alle dort geschilderten Schritte – sonst wird das Plug-In nicht funktionieren.

Eclipse ist kostenlos und kann unter der Adresse *http://www.eclipse.org* heruntergeladen werden. Mit Hilfe verschiedenster Plug-Ins können Sie jede Menge zusätzlicher Features nutzen. Eine Sammlung sinnvoller Plug-Ins für Eclipse finden Sie unter der Adresse *http://www.eclipse-plugins.info*.

1.5.4 Text-Editoren

Wenn Ihnen eine Entwicklungsumgebung zu aufwändig oder zu unübersichtlich erscheint, können Sie Java-Applikationen mit jedem Text-Editor entwickeln. Damit die Arbeit mit dem Editor schnell und effektiv vonstatten geht, sollten Sie aber darauf achten, dass der Editor über Syntax-Highlighting für Java verfügt. Am besten bietet der Editor auch noch die Möglichkeit, eigene Word-Files nachzuladen, damit Sie immer auf dem aktuellen Stand der möglichen Schlüsselworte bleiben. Beliebte Editoren für Java sind vi, emacs, UltraEdit oder jEdit.

1.6 Java-Tools

Java ist kein monolithisches System. Java ist auch nicht nur eine einzelne ausführbare Datei. Zu Java gehören `java`, `javac`, `javadoc`, `appletviewer`, `jar` und `jdb`. Sie werden mit allen diesen Tools zu tun haben, wenn Sie mit Java arbeiten – eventuell wird Ihnen diese Beschäftigung von Ihrer Entwicklungsumgebung abgenommen, dennoch sollten Sie zumindest wissen, welche Tools Sie einsetzen.

1.6.1 java

Der Java-Interpreter `java` interpretiert den `Byte-Code`, den der Java-Compiler erzeugt, übersetzt ihn in nativen Maschinen-Code und führt ihn aus. Er wird von der Kommandozeile ausgeführt und erwartet als Parameter den Namen der auszuführenden Datei:

```
java HelloWorld
```

Java-Klassen-Dateien enden stets mit der Datei-Endung `.class`. Diese Endung kann beim Einsatz des Interpreters weggelassen werden.

Wenn die Klassen-Datei zu einem Paket gehört (mehr zu Paketen in *Kapitel 3*), dann müssen Sie beim Aufruf von `java` den kompletten Paketnamen mit übergeben. Sollte die `HelloWorld`-Klasse Bestandteil des Paketes `masterclass.java` sein, dann sollte der Aufruf also so aussehen:

```
java masterclass.java.HelloWorld
```

Der Java-Interpreter kann mit verschiedenen Kommandozeilen-Parametern aufgerufen werden. Der wichtigste Parameter ist `-cp` oder auch `-classpath`, der eine Liste von Verzeichnissen, `JAR`- und `ZIP`-Archiven, in denen nach benötigten `class`-Dateien gesucht wird, angibt. Er ergänzt die Angaben der Umgebungsvariablen `CLASSPATH`.

1.6.2 javac

Der Java-Compiler `javac` kompiliert die Java-Source-Dateien mit der Endung `.java` zu `Byte-Code`. Er wird von der Kommandozeile ausgeführt und nimmt als Parameter den Namen der Quell-Datei entgegen:

```
javac HelloWorld.java
```

Sie können ebenfalls mehrere Quelldateien auf einmal kompilieren lassen. Trennen Sie dazu die einzelnen Datei-Angaben durch Leerzeichen:

```
javac HelloWorld.java SayHello.java
```

Ebenfalls ist es möglich, mit Platzhaltern und Wildcards zu arbeiten. Um beispielsweise alle `.java`-Dateien eines Ordners auf einen Rutsch zu kompilieren, können Sie folgenden Aufruf verwenden:

```
javac *.java
```

Der Kompiler `javac` verfügt über einige Kommandozeilen-Optionen. Hier die wichtigsten im Überblick:

- `-nowarn`: Gibt keine Warnungen aus
- `-verbose`: Gibt Informationen darüber aus, was der Compiler gerade tut
- `-deprecation`: Gibt Informationen darüber aus, was und warum der Compiler für veraltet erklärt
- `-cp <Klassenpfad>`: Gibt an, welche Verzeichnisse, `JAR`- und `ZIP`-Archive nach referenzierten Klassen durchsucht werden
- `-classpath <Klassenpfad>`: Wie `-cp`
- `-sourcepath <Quellpfad>`: Gibt den Pfad des Verzeichnisses an, in dem sich die Sourcen befinden
- `-d <Pfad>`: Gibt an, wo die generierten *.java*-Dateien abgelegt werden sollen
- `-target <Version>`: Gibt an, für welche Zielversion kompiliert werden soll. Selbstverständlich muss der zugrunde liegende Java-Code den Sprachspezifikationen der jeweiligen Version entsprechen – Sie können also nicht Java-5-spezifischen Code mit der Angabe `-target 1.1` für Java 1.1 kompilieren und hoffen, dass der Compiler dies schluckt.

Wenn Sie häufig viele Optionen übergeben müssen, empfiehlt es sich, diese Optionen samt des `javac`-Aufrufs in einer Batch-Datei zu speichern und diese dann auszuführen.

1.6.3 Appletviewer

`Appletviewer` dient der Darstellung von Java-Applets. Als Parameter nimmt das Tool die Adresse einer HTML-Datei, die das Tag `<applet>` enthält, entgegen:

```
appletviewer c:\applets\test.html
```

Ebenfalls ist es möglich, eine Internet-Adresse anzugeben – `appletviewer` lädt die referenzierte Datei herunter und zeigt das dort deklarierte Applet an:

```
appletviewer http://www.bbox.ch/default.asp?m=15
```

Dies würde das VisualRoute-Applet von der Seite des Schweizer Providers herunterladen und lokal anzeigen. Sie könnten dann grafisch den Weg von Ihrem lokalen Rechner zu einem entfernten Internet-Server verfolgen.

1.6.4 javadoc

Das Dokumentationstool `javadoc` erstellt eine detaillierte HTML-Dokumentation aus einer .java-Datei. Zu diesem Zweck müssen in den Quelltext-Dateien spezielle Kommentare vorhanden sein, die dann ausgewertet werden können. Das Kommandozeilen-Tool `javadoc` erlaubt die Angabe verschiedener Parameter, die sein Verhalten steuern:

- `-public`: Nur öffentliche Elemente anzeigen
- `-protected`: Öffentliche und als protected gekennzeichnete Elemente anzeigen

- `-private`: Alle Elemente anzeigen
- `-sourcepath <Pfadliste>`: Liste der Pfade, in denen die *.java*-Dateien liegen, die kommentiert werden sollen
- `-classpath <Pfadliste>`: Liste der Pfade, `JAR`-Archive und `ZIP`-Archive, in denen referenzierte Klassen gesucht werden sollen
- `-d <Verzeichnis>`: Ausgabe-Verzeichnis der generierten Dokumentation
- `-use`: „Use"-Seiten generieren
- `-version`: Versions-Information mit ausgeben
- `-author`: Autor-Information mit ausgeben

Um eine `javadoc`-Dokumentation der Datei *HelloWorld.java* im Ordner *c:\docs* mit Angabe von Versions- und Autor-Informationen zu erzeugen, können Sie diesen Aufruf verwenden:

```
javadoc -author -version -d c:\docs\ HelloWorld.java
```

1.6.5 jar

Um Java-Klassen oder -Programme zu verteilen, empfiehlt sich der Einsatz eines Java-Archivs, in dem alle nötigen Klassen, Referenzen, Bilder und Ressourcen enthalten sind. Anderenfalls wird es spätestens bei größeren Projekten nahezu unmöglich, den Überblick darüber zu behalten, welche Elemente weitergegeben werden müssen und ob diese alle wirklich weitergegeben worden sind.

Java bietet uns zu diesem Zweck das Kommandozeilen-Tool `jar`, das `JAR`-Archive erzeugt. Diese `JAR`-Archive entsprechen `ZIP`-Dateien und können ebenso wie diese komprimiert sein.

Das Tool `jar` erwartet einige Parameter, um korrekt zu funktionieren:

- `-c`: neues Archiv erstellen
- `-u`: vorhandenes Archiv aktualisieren
- `-v`: ausführliche Informationen ausgeben
- `-f`: Namen der Archivdatei angeben
- `-m`: Manifestinformationen einbeziehen
- `-0`: nur speichern und keine ZIP-Komprimierung verwenden
- `-M`: keine Manifest-Datei für die Einträge erstellen
- `-C`: ins angegebene Verzeichnis wechseln und folgende Datei einbeziehen

Nach der Angabe der Optionen geben Sie den Dateinamen des Archivs und die Quellen an. Um beispielsweise die beiden *.class*-Dateien *HelloWorld.class* und *WorldHello.class* in einem Archiv *hello.jar* abzulegen, können Sie diesen Aufruf verwenden:

```
jar cvf hello.jar HelloWorld.class WorldHello.class
```

Um alle Dateien des Verzeichnisses `c:\javafiles` in einem Archiv `javafiles.jar` abzulegen, können Sie folgenden Aufruf verwenden:

```
jar cvf javafiles.jar -C c:\javafiles *.*
```

Dabei werden auch untergeordnete Verzeichnisse samt deren .class-Dateien einbezogen.

1.6.6 jdb

Der Debugger `jdb` hilft Ihnen, Java-Programme zeilenweise zu durchlaufen und dabei festzustellen, wo Fehler auftreten können. Er wird ohne Parameter (oder dem Namen der zu debuggenden `.class`-Datei als Parameter) aufgerufen und präsentiert danach eine eigene Kommandozeile, auf der Sie arbeiten können. Der Einsatz dieses Debuggers sprengt allerdings den Rahmen dieses Kapitels, aber nach dem Starten von `jdb` können Sie eine Online-Hilfe erhalten, indem Sie den Befehl `help` eingeben.

Da die Arbeit mit einem Kommandozeilen-Debugger in der Regel kein besonderes Vergnügen bereitet, bieten die weiter oben angegebenen Entwicklungsumgebungen allesamt integrierte Debugger. Alternativ empfiehlt der Autor `JSwat`, einen grafischen Debugger für Java-Applikationen, der (ebenso wie alle anderen integrierten Debugger) auf `jdb` aufsetzt. `JSwat` ist kostenlos und Open Source und kann unter der Adresse *http://www.bluemarsh.com/java/jswat/* bezogen werden.

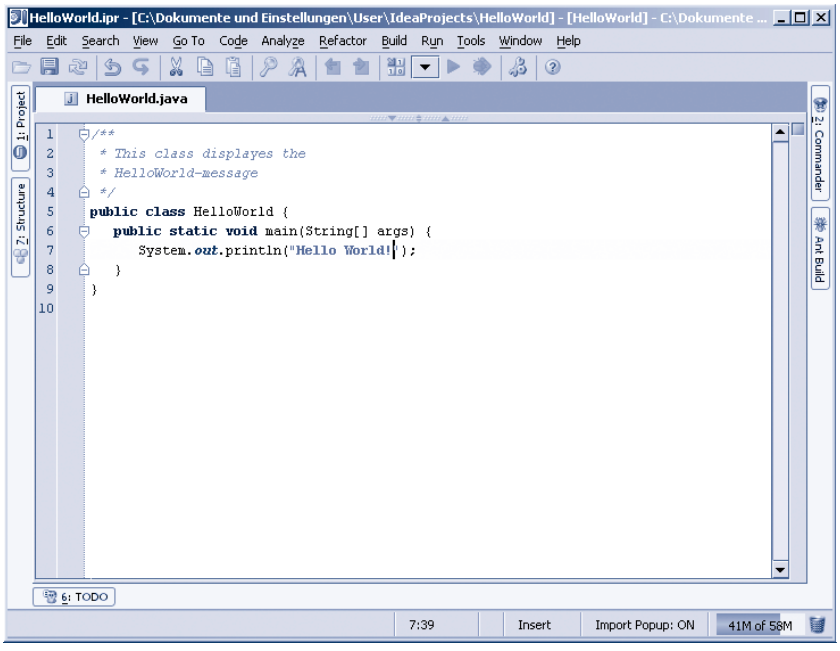
1.7 Jetzt aber los!

Nachdem wir nun alles zusammen haben, um mit Java loslegen zu können, wollen wir ein erstes kleines Beispiel ausführen: das obligatorische `HelloWorld`-Programm.

Öffnen Sie also Ihre bevorzugte Entwicklungs-Umgebung (oder den Text-Editor Ihrer Wahl), und geben Sie folgenden Code ein:

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

Abbildung 1.2
Eingabe einer Java-Datei in der Entwicklungs-Umgebung IntelliJ IDEA



Speichern Sie die Datei unter dem Namen `HelloWorld.java` (achten Sie unbedingt auf die Groß- und Kleinschreibung). Übrigens: In der Datei haben Sie eine Klasse `HelloWorld` (mehr dazu in *Kapitel 3*) deklariert, die eine statische Methode `main()` (mehr dazu im *Kapitel 8*) besitzt und über die Standard-Ausgabe die Nachricht „Hello World“ ausgibt. Das sind alles noch böhmische Dörfer für Sie? Grämen Sie sich nicht: Am Ende des Buches werden Sie über diesen Code nur noch müde lächeln können.

Öffnen Sie nun eine Kommandozeile und wechseln Sie in das Verzeichnis, in das Sie die Datei `HelloWorld.java` gespeichert haben. Geben Sie hier nun folgendes Kommando ein:

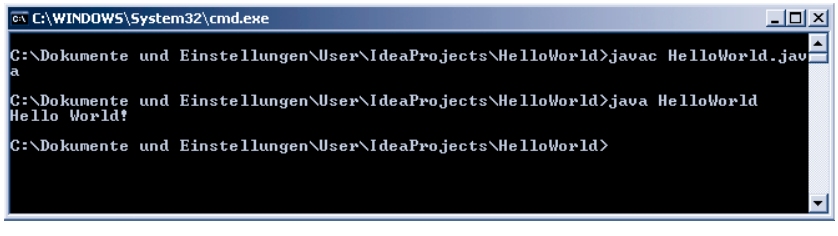
```
javac HelloWorld.java
```

Nun sollte der Java-Compiler die Datei in `Byte-Code` überführen. Ein anschließendes

```
java HelloWorld
```

wird Ihre erste Java-Applikation ausführen:

Abbildung 1.3
Ausgabe einer Java-Applikation



1.8 Fazit

Dies war der Einstieg in Java! Primär ging es darum, sich das JDK für Java 2 Standard Edition, Version 5.0 zu besorgen, zu installieren und zu konfigurieren. Das haben Sie erfolgreich erledigt – sonst hätten Sie nicht am Ende des Kapitels Ihre erste Java-Applikation schreiben und ausführen können.

Ebenso haben Sie einen Blick hinter die Kulissen der zuvor weitestgehend unbekannten Welt „Java“ geworfen. Sie wissen nun, dass Java aus verschiedenen Dienstprogrammen besteht und dass Sie Java-Programme mit Hilfes des primitivsten Text-Editors oder unter Verwendung ausgewachsener Entwicklungs-Umgebungen schreiben können. Und Sie wissen, wo Java herkommt, was es ist und wofür es eingesetzt werden kann.

Zeit für uns, endlich wirklich in die Sprache einzusteigen!

2

Java? Java!

Wie jede andere Programmiersprache auch, verfügt Java über bestimmte Strukturen, Anweisungen und Datentypen, die zum Kernumfang gehören und die Grundlagen der Sprache bilden. Wir werden uns in diesem Kapitel mit diesen Basics befassen – so dass Sie hinterher in der Lage sind, erste Schritte mit der Sprache zu unternehmen.

2.1 Anweisungen

Programme, egal in welcher Sprache geschrieben, sind grundsätzlich dumm. Sie machen nichts – es sei denn, Sie teilen dem Programm mit, was es zu tun hat. Nun reicht es leider nicht aus, mit einem Text-Editor zu schreiben, dass Java bitte eine Textverarbeitung schreiben möchte. Stattdessen müssen Sie selbst alle Anweisungen erfassen, die Java in die Lage versetzen, als Textverarbeitung zu fungieren.

Jede dieser Anweisungen ist ein einzelner Befehl, der dafür sorgen soll, dass etwas geschieht. Anweisungen müssen nicht aus nur einem Wort bestehen, sondern sind in der Regel aus mehreren Schlüsselwörtern und Variablen zusammengesetzt.

Am Ende einer Anweisung steht stets ein Semikolon. Anders als in anderen Sprachen können Sie bei Java mehrere Anweisungen in eine Zeile schreiben. Außerdem müssen Anweisungen nicht zwingend auf eine Zeile beschränkt sein, sondern können – etwa aus Lesbarkeitsgründen – auf mehrere Zeilen verteilt sein. Dies alles ist kein Problem, solange Sie die Anweisung am Ende wieder mit einem Semikolon abschließen.

Die folgenden Beispiele sind alle gültige Anweisungen:

```
int age = 28;
```

```
String aQuestion = String.format(
   "I am %s years old. How old are you?", age);
```

```
System.out.println(aQuestion); System.out.println();
```

Anweisungen können auch einen Wert erzeugen oder ein Ergebnis zurückgeben.

Anweisungen, die als Ergebnis einen Wert produzieren, werden Ausdrücke genannt. Diese Werte können im Programm gespeichert, manipuliert, ausgewertet oder auch ignoriert werden. Die von Anweisungen erzeugten Werte werden Rückgabe-Werte genannt.

Anweisungen können (und sollen!) in Blöcken gekapselt werden. Diese Blöcke sind in geschweifte Klammern eingefasst – der Beginn des Blocks wird durch eine nach rechts geöffnete geschweifte Klammer ({) repräsentiert, das Ende durch die nach links geöffnete Klammer (}):

```
if(i == 0) {
   System.out.println("i is greater than Zero!");
}
```

Blöcke kennzeichnen immer eine Gruppe von Anweisungen, die in einem Kontext zusammengehören.

2.2 Variablen

Variablen nehmen Werte auf und speichern Informationen. Sie heißen deshalb Variablen, weil ihr Inhalt variabel ist und zur Entwicklungszeit nicht unbedingt bekannt sein muss. Der Wert einer Variablen kann im Programm oder im aktuellen Block geändert werden – dies ist vom Gültigkeitsbereich der Variablen abhängig. Dieser Gültigkeitsbereich definiert die Art der Variablen:

- Klassenvariablen: Diese Variablen werden auf Applikations-Ebene definiert und sind über Instanzen hinweg gültig. Sie sind in der Regel innerhalb aller Methoden aller Instanzen einer Klasse erreichbar.
- Instanzvariablen: Werden auf Ebene einer Klassen-Instanz definiert. Sie sind in der Regel innerhalb aller Methoden einer Klasse erreichbar. Sie sind immer innerhalb einer Instanz der Klasse oder Applikation gültig.
- Lokale Variablen: Werden auf Ebene einer Methode oder gar eines Blocks definiert und sind auch nur von dort oder von untergeordneten Blöcken erreichbar. Beim Verlassen des Blocks, in dem die Variable deklariert worden ist, verliert sie ihre Gültigkeit und ihren Wert.

Wenn Sie von einer anderen Sprache zu Java gekommen sind, werden Sie möglicherweise globale Variablen vermissen. Wenn wir uns aber überlegen, was globale Variablen normalerweise ausmacht – nämlich die Möglichkeit, überall in einer Applikation verfügbar zu sein –, müssen wir deren Abwesenheit keine Träne nachweinen: Klassen- und Instanz-Variablen erledigen diese Aufgabe genauso gut.

2.2.1 Variablen deklarieren

Variablen repräsentieren Werte. Java muss aber wissen, welche Art von Werten repräsentiert werden soll. Sie müssen diese Informationen zusammen mit einem im aktuellen Kontext oder Namensraum eindeutigen Namen der Variablen bei der Deklaration der Variablen angeben:

```
<Typ> <Name>;
```

Die Deklaration einer Variablen geschieht, indem Sie zunächst den Typ der Variablen und danach deren Namen angeben. Beispiele für die Deklaration von Variablen wären etwa:

```
int age;

String name;

boolean checked;

double Double;
```

2.2.2 Konventionen

Sie können Variablen fast beliebig benennen, allerdings gilt es dann doch, einige Regeln einzuhalten:

- Der Name der Variablen darf keinem Schlüsselwort oder Datentyp-Bezeichner entsprechen, wobei Groß- und Kleinschreibung berücksichtigt werden. Wir dürfen also eine Variable vom Typ `double` mit Namen *Double* deklarieren – ein `String` namens *String* wäre aber nicht zulässig.
- Der Name der Variablen muss mit einem Buchstaben beginnen.
- Erlaubte Zeichen im Variablennamen sind alle Zeichen des Unicode-Zeichensatzes. Die ersten 256 Zeichen dieses Zeichensatzes sind identisch mit der ASCII-Variante Latin-1.

Per Konvention werden die Namen von Variablen immer kleingeschrieben. Sollte der Name aus mehreren Begriffen oder Worten zusammengesetzt sein, wird der Anfang des Variablennamens kleingeschrieben und die ersten Zeichen der folgenden Begriffe groß:

```
String thisIsTheNameOfAVariable;
int thisIsAnInteger;
```

Wir werden im weiteren Verlauf dieses Kapitels noch genauer auf die möglichen Datentypen eingehen.

Generell kann eine Variable an einer beliebigen Stelle innerhalb des Codes deklariert werden. Dies muss allerdings in jedem Fall vor ihrer Verwendung geschehen.

Achtung

Achten Sie auf die korrekte Groß- und Kleinschreibung, wenn Sie mit Variablen arbeiten! Eine Variable `one` ist etwas anderes als eine Variable `oNe` oder eine Variable `ONe`. Dies sind jeweils eigenständige Variablen, die möglicherweise nicht deklariert sind.

2.2.3 Mehrere Variablen vom gleichen Typ deklarieren

Mehrere Variablen vom gleichen Typ können zwar auch nacheinander deklariert werden – viel einfacher und mit weniger Schreibaufwand verbunden ist es jedoch, sie in einer Zeile nur doch Kommata voneinander getrennt zu deklarieren. Statt

```
String one;
String two;
String three;
```

können wir auch

```
String one, two, three;
```

schreiben.

2.2.4 Zuweisen von initialen Werten

Es ist eine gute Idee, Variablen bei ihrer Deklaration einen Wert zuzuweisen, da wir sie so in einen definierten Zustand versetzen:

```
int age = 25;
String one = "one", two = "two", three = "three";
boolean checked = false;
```

Wie wir sehen, funktioniert diese Zuweisung auch bei mehrfachen Variablen-Deklarationen.

Variablen, denen bei der Deklaration kein Wert zugewiesen worden ist, erhalten abhängig vom Datentyp immer einen impliziten Initialisierungswert:

- Numerische Variablen: 0
- Zeichen / Strings: '\0' (Null-String)
- boolesche Variablen: false
- Objekte: null

2.2.5 Lebensdauer und Sichtbarkeit von Variablen

Lokale Variablen sind grundsätzlich immer nur innerhalb eines Bereiches von der Stelle ihrer Deklaration bis zum Ende der Methode sichtbar. Die Deklaration muss nicht am Anfang einer Methode geschehen, sondern kann an beliebiger Stelle zwischen Anfang und Ende einer Methode erfolgen. Falls eine Variable innerhalb eines Blocks angelegt wird, endet ihre Lebensdauer am Ende des Blocks – allerdings darf in der Methode, in der sich der Block befindet, keine andere lokale Variable gleichen Namens deklariert werden. Was aber zulässig ist: lokale Variablen definieren, die gleichnamige Klassen- oder Instanzvariablen verdecken (d.h. die Klassen- oder Instanzvariablen verlieren nicht ihren Wert, sondern können innerhalb der Methode nicht mehr über ihren zuvor definierten Bezeichner angesprochen werden).

Instanzvariablen beginnen ihren Lebenszyklus, sobald eine Instanz ihrer Klasse erzeugt worden ist. Sie werden dann mit ihrem Standard- oder dem bei der Deklaration festgelegten Vorgabewert initialisiert. Sie sind in der ganzen Klasse sichtbar, können aber von lokalen Variablen verdeckt werden. Sollte dennoch auf die Instanz-Variable zugegriffen werden, geschieht dies mit Hilfe des Schlüsselwortes `this`: Statt `name` schreiben wir dann `this.name` und greifen auf die Instanz-Variable zu. Ohne das Schlüsselwort `this` wird dagegen auf die verdeckende lokale Variable zugegriffen. Sobald die Klassen-Instanz terminiert und zerstört worden ist, endet auch der Lebenszyklus der Instanz-Variablen.

Klassen-Variablen leben während der kompletten Lebensphase eines Programms. Bei ihnen gelten die gleichen Sichtbarkeits-Regeln wie bei Instanzvariablen: Sie sind also überall in der Klasse sichtbar, können aber durch lokale Variablen verdeckt und bei Bedarf mit Hilfe des Schlüsselwortes `this` wieder sichtbar gemacht werden.

2.3 Datentypen

Java unterscheidet verschiedene Datentypen, die Daten halten können. Folgende Datentypen stehen uns zur Verfügung:

- `byte`: 8-Bit-Ganzzahl zwischen -128 und 127
- `short`: 16-Bit-Ganzzahl zwischen -32.768 und 32.767
- `int`: 32-Bit-Ganzzahl zwischen -2.147.483.648 und 2.147.483.647
- `long`: 64-Bit-Ganzzahl zwischen -9.223.372.036.854.775.808 und 9.223.372.036.854.775.807
- `float`: 32-Bit-Gleitkomma-Zahl, die einen Wertebereich zwischen 1.4E-45 und 3.4E+38 verarbeiten kann
- `double`: 64-Bit-Gleitkomma-Zahl für einen Wertebereich zwischen 4.9E-324 und 1.7E+308
- `char`: Ein einzelnes Zeichen
- `boolean`: Ein Datentyp, der als Wert entweder *true* oder *false* annehmen kann

Neben diesen Datentypen existieren noch zwei Objekt-Typen, die so elementar sind, dass man sie mit Datentypen durchaus gleichsetzen kann: `Object` und `String`. `Object` ist die Basis-Klasse aller in Java definierten Klassen und wird immer dann eingesetzt, wenn man nicht weiß oder nicht festlegen kann, was für ein Datentyp erwartet wird. `String` repräsentiert eine Zeichenkette beliebiger Länge.

2.3.1 Datentyp-Objekte

Die acht Datentypen werden gerne auch als primitive Datentypen bezeichnet, da sie die einzigen Elemente von Java sind, die keine Objekte darstellen. Allerdings existieren zu diesen primitiven Datentypen sogenannte Wrapper-Klassen, die einen objektorientierten Umgang mit diesen Elementen erlauben:

```
public Byte(byte value)
public Byte(String s)
   throws NumberFormatException

public Short(short value)
public Short(String s)
   throws NumberFormatException

public Integer(int value)
public Integer(String s)
   throws NumberFormatException

public Long(long value)
public Long(String s)
   throws NumberFormatException

public Float(float value)
public Float(double value)
public Float(String s)
   throws NumberFormatException

public Double(double value)
public Double(String s)
   throws NumberFormatException

public Character(char value)

public Boolean(boolean value)
public Boolean(String s)
```

Jeder der Konstruktoren der Klassen nimmt ein Element des repräsentierten primitiven Typs als Parameter entgegen. Alternativ kann in den meisten Fällen auch eine `String`-Instanz übergeben werden, die den entsprechenden Wert repräsentiert. Intern versuchen die Klassen dann, die übergebene `String`-Instanz in den primitiven Typ umzuwandeln – klappt dies nicht, wird eine `NumberFormatException` geworfen.

2.3.2 valueOf-Zuweisungen

Neben den Konstruktoren verfügen die Datentyp-Objekte allesamt über statische Methoden, die das Erzeugen von Instanzen erlauben, ohne den Konstruktor bemühen zu müssen. Die Verwendung dieser Methoden ist empfohlen, da sie in der Regel minimal schneller funktionieren als die Konstruktoren:

```
public static Byte valueOf(String s)
   throws NumberFormatException
public static Byte valueOf(byte b)

public static Short valueOf(String s)
   throws NumberFormatException
public static Short valueOf(short s)

public static Integer valueOf(String s)
   throws NumberFormatException
public static Integer valueOf(int i)
```

```
public static Long valueOf(String s)
   throws NumberFormatException
public static Long valueOf(long l)

public static Float valueOf(String s)
   throws NumberFormatException
public static Float valueOf(float f)

public static Double valueOf(String s)
   throws NumberFormatException
public static Double valueOf(double d)

public static Character valueOf(char c)

public static Boolean valueOf(boolean b)
public static Boolean valueOf(String s)
```

Folgende Code-Fragmente sind allesamt gültig und geben Instanzen der Objekte zurück, die den zugehörigen primitiven Datentyp repräsentieren:

```
Byte byt = Byte.valueOf("125");

Short s = Short.valueOf(12345);

int anInt = 100000;
Integer integer = Integer.valueOf(anInt);

Float f = new Float("123.88");

Boolean b = new Boolean("true");
```

2.3.3 Typ-Konvertierung

Vor Java 5 war die implizite Umwandlung der primitiven Typen in ihre objektorientierten Geschwister immer damit verbunden, dass der Typ gecastet (also umgewandelt) werden musste. Dies wurde spätestens dann lästig, wenn wieder in einen primitiven Typ zurückgecastet werden sollte. Zu diesem Zweck mussten immer die entsprechenden Umwandlungsmethoden der Objekt-Instanzen aufgerufen werden.

Mit Java 5 ist das nun anders. Hier kann die Umwandlung der primitiven Datentypen in ihre objektorientierten Geschwister einfach durch Zuweisung stattfinden. Ein Konstrukt wie

```
int ten = 10;
Integer tenObj = ten;
int otherTen = tenObj;
```

wäre früher nicht möglich gewesen – nun funktioniert es, denn es ist ein neues Feature bei Java 5. Diese Art der Umwandlung von Werten nennt man übrigens implizite Typ-Konvertierung.

2.4 Konstanten

Konstanten sind das Gegenteil zu Variablen - sie halten definierte Werte, die (so wie der Name es bereits ausdrückt) konstant sind. In Java gibt es allerdings keine Konstanten - wohl aber die Möglichkeit, Variablen zu deklarieren, deren Wert sich nicht ändern kann.

Um eine unveränderliche Variable zu erzeugen, stellen Sie der Variablen-Deklaration das Schlüsselwort `final` voran und legen den Wert der konstanten Variable fest:

```
final <Typ> <Name> = <Wert>
```

Die Konvention besagt, dass die Namen von konstanten Variablen stets in Großbuchstaben geschrieben werden sollen.

Hier ein paar Beispiele für die Definition von konstanten Variablen:

```
final int LEFT = 1;
final static float PI = 3.141592;
final static String DEFAULT_MESSAGE = "Hello";
```

Hilfreich sind konstante Variablen auch, um Werten sprechende Namen zu geben. Statt

```
this.align = 1;
```

können wir auch

```
this.align = LEFT;
```

schreiben. Letzteres ist übersichtlicher und besser lesbar.

Hinsichtlich des Laufzeitverhaltens und der Performance unterscheiden sich konstante Variablen nicht von anderen Variablen. Dies begründet sich vor allem daraus, das konstante Variablen gewöhnliche Variable sind, die nur gegen Überschreiben geschützt werden. Echte Konstanten würden dagegen vom Compiler aus dem Code entfernt und durch ihren repräsentierten Wert ersetzt werden – eine Handhabung wie bei Variablen (hinsichtlich Speicher-Allozierung und Zugriffsschutz) entfiele, und die Performance könnte zumindest theoretisch steigen.

In Java sind Konstanten also keine Konstanten im üblichen Sinne. Da der Begriff jedoch eingängiger und besser lesbar als „konstante Variablen“ ist, soll er im Folgenden dennoch verwendet werden, um konstante Variablen zu bezeichnen.

2.5 Kommentare

Kommentare helfen uns, Programme, Klassen oder Code-Fragmente besser zu strukturieren und zu dokumentieren. Kommentare werden von Compiler und Interpreter ignoriert und helfen nur dem menschlichen Betrachter.

Bei Java unterscheiden wir drei Arten von Kommentaren:

- Einzeilige Kommentare: Diese beginnen mit zwei Schrägstrichen (//) und enden am Ende der Zeile
- Mehrzeilige Kommentare: Diese beginnen mit /* und enden mit */. Sie können sich über mehrere Zeilen erstrecken (müssen es aber nicht).
- Dokumentations-Kommentare: Diese Kommentare dienen dazu, automatisiert eine Dokumentation zu erstellen, und beschreiben Methoden oder Klassen. Sie können nur vor Methoden- oder Klassen-Deklarationen stehen. Sie werden mit /** eingeleitet und enden mit */.

Die Dokumentationskommentare folgen einem festen Schema und werden mit Hilfe des Tools `javadoc` extrahiert und in HTML-Dokumente umgewandelt. Definitions-Kommentare bestehen aus Beschreibung und zusätzlichen Elementen.

Diese Elemente werden mit Hilfe des Zeichens @ eingeleitet und geben Auskunft über den Autor, die Version, die Parameter oder den Rückgabe-Wert der Methode oder Klasse. Folgende Elemente werden unter anderem unterstützt:

- @author: Autor
- @version: Version
- @param: Parameter-Info
- @return: Rückgabe-Info
- @exception: Von der Methode geworfene Exceptions

Ein Dokumentations-Kommentar könnte beispielsweise so aussehen:

```
/**
 * Example documentation comment
 * @param obj An object which is passed to the method
 * @return The result of the operation in the method
 */
public String performAction(Object obj) {
   // ...
}
```

Mehr über Dokumentations-Kommentare finden Sie unter der Adresse *http://java.sun.com/j2se/javadoc/*.

2.6 Ausdrücke und Operatoren

Ausdrücke sind Anweisungen, die einen Wert erzeugen. Sie können einen booleschen Wert formulieren oder weisen Variablen Ergebnisse von Berechnungen zu. Ausdrücke bestehen stets aus einem Operator (beispielsweise dem Gleichheitszeichen, das eine Zuweisung kennzeichnet, oder dem Plus, das für die Addition steht) und können darüber hinaus über Operanden verfügen, auf die der Operator angewendet wird. Jeder Ausdruck hat einen Rückgabe-Wert, der sich aus der Anwendung des Operators auf die Operanden ergibt.

Java unterscheidet verschiedene Operatoren. Auf die wichtigsten dieser Operatoren wollen wir im Folgenden kurz eingehen.

2.6.1 Arithmetische Operatoren

Java unterscheidet folgende Operationen für die Arithmetik:

Tabelle 2.1 Übersicht arithmetischer Operatoren

Operator	Bezeichnung	Bedeutung
+	Addition	5 + 2
-	Subtraktion	5 – 2
*	Multiplikation	5 * 2
/	Division	5 / 2
%	Modulo	Gibt den ganzzahligen Rest einer Division zurück. `5 % 2` ergibt als Modulo 1.
++	Prä-Inkrement	`++a` erhöht die Variable um 1 und gibt deren neuen Wert als Ergebnis zurück.
++	Post-Inkrement	`a++` gibt als Rückgabe den Wert der Variablen zurück und erhöht diese anschließend um 1.
--	Prä-Dekrement	`--a` verringert den Wert der Variablen um 1 und gibt den neuen Wert zurück.
--	Post-Dekrement	`a--` gibt den Wert der Variablen zurück und verringert deren Wert anschließend um 1.

Diese Operatoren benötigen zwei Operanden, je einen auf jeder Seite des Operators. Die Art der Rückgabe bei allen Operatoren ergibt sich daraus, welche Datentypen verwendet werden – bei einer `int`-Zahl und einem `double`-Wert ergibt sich immer eine Rückgabe vom Typ `double`.

Weisen Sie diese Rückgabe jedoch einer `int`-Zahl zu, erfolgt bei einer Division dennoch eine Rundung der Rückgabe auf einen ganzzahligen Wert.

Achtung

Achten Sie bei der Division stets darauf, dass der zweite Operand ungleich 0 ist, da eine Division durch 0 nicht definiert ist und zu einer Exception führt.

2.6.2 Vergleichsoperatoren

Diese Art von Operatoren vergleicht Ausdrücke miteinander und gibt einen logischen Wert (*true* oder *false*) zurück, der das Ergebnis dieses Vergleichs repräsentiert. Relationale Operatoren arbeiten grundsätzlich auf allen numerischen Werten und im Falle der Gleichheits- und Ungleichheits-Operatoren auch auf Objekten:

Tabelle 2.2
Relationale Operatoren

Operator	Bezeichnung	Bedeutung
==	Gleich	`a == b` ergibt `true`, wenn beide Variablen den gleichen Wert haben. Sind `a` und `b` Variablen, die Objektreferenzen halten, so müssen beide auf das gleiche Objekt verweisen.
!=	Ungleich	`a != b` ergibt `true`, wenn beide unterschiedliche Werte haben. Sind `a` und `b` Variablen, die Objektreferenzen halten, so müssen sie auf unterschiedliche Objekt-Instanzen zeigen.
<	Kleiner	`a < b` ergibt `true`, wenn der Wert von `a` kleiner als der numerische Wert von `b` ist
<=	Kleiner gleich	`a <= b` ergibt `true`, wenn der Wert von `a` kleiner oder gleich `b` ist
>	Größer	`a > b` ergibt `true`, wenn der Wert von `a` größer als `b` ist.
>=	Größer gleich	`a >= b` ergibt `true`, wenn der Wert von `a` größer oder gleich `b` ist.

2.6.3 Logische Operatoren

Die Ergebnisse von Vergleichen sind boolesche Werte. Diese Werte können wir miteinander kombinieren und dadurch weiterverarbeiten. Folgende logische Operatoren werden unterschieden:

Tabelle 2.3
Logische Operatoren

Operator	Bezeichnung	Bedeutung
!	Logisches NOT	Der boolesche Wert von `a` wird negiert - ist `a true`, ergibt `!a false`. Ist `a false`, ergibt `!a true`.
&&	Logisches AND	`a && b` ergibt `true`, wenn `a` und `b` wahr sind. Ist `a false`, wird `b` nicht ausgewertet.
&	Logisches AND	`a & b` ergibt `true`, wenn `a` und `b` wahr sind. Ist `a` falsch, wird `b` dennoch ausgewertet.
||	Logisches OR	`a || b` ergibt `true`, wenn mindestens einer der beiden Ausdrücke wahr ist. Sollte `a` bereits wahr sein, wird `b` nicht mehr ausgewertet.
|	Logisches OR	`a | b` ergibt `true`, wenn mindestens einer der beiden Ausdrücke wahr ist. Ist `a` bereits wahr, wird `b` dennoch ausgewertet.
^	Logisch XOR	`a ^ b` ergibt `true`, wenn `a` und `b` unterschiedliche Wahrheitswerte haben.

Logische Operatoren wirken bei der ersten Verwendung nicht unbedingt logisch, sondern eher irritierend. Lassen Sie sich davon nicht abschrecken – wir verwenden sie über das gesamte Buch hinweg, so dass ihre Benutzung bald selbstverständlich sein wird.

2.6.4 Weitere Operatoren

Neben den bisher aufgeführten Operatoren gibt es weitere Operatoren, die aber in keines der anderen Schemata passen:

Tabelle 2.4 Weitere Operatoren

Operator	Bezeichnung	Bedeutung
`instanceof`	Instanz von	`a instanceof b` gibt `true` zurück, wenn `a` vom selben Typ wie `b` oder `a` vom selben Typ wie eine Ableitung von `b` ist.
`<< >>`	Bit-Shifting	
`(<Typ>)`	Typ-Konvertierung	`a = (<Typ von a>)b` castet den Typ von `b` explizit nach `a`.
`new <Typ>`	Instanzierung	`<Typ> a = new <Typ>()` erzeugt eine neue Instanz vom angegebenen Typ.
`? :`	Fragezeichen-Operator	Kurzform für if-else-Konstrukte. `a ? b : c` prüft, ob `a` wahr ist, und gibt dann `b` zurück. Wenn `a` falsch ist, dann wird `c` zurückgegeben.

2.6.5 Vorrangregeln

Um in einem Ausdruck mehr als einen Operator verwenden zu können, verwendet Java intern Regeln, die festlegen, welche Operatoren Vorrang vor anderen haben. Daraus ergibt sich eine Reihenfolge, in der die Operatoren ausgewertet werden. Die Regeln, nach denen Java dabei vorgeht, sind aber keine Geheimnisse, sondern werden hier gezeigt. Auch die Reihenfolge, nach der die Regeln angewendet werden, unterliegt einer bestimmten Regel: Sie geschieht von oben nach unten:

Tabelle 2.5 Operator-Reihenfolge

Operatoren	Anmerkung
`()`	Klammern gruppieren Ausdrücke. Diese Information wird zuerst ausgewertet.
`++ -- ! ~ instanceof`	
`new`	
`* / %`	Multiplikation, Division und Modulo
`+ -`	Addition, Division
`<< >>`	Shifting
`< <= > =>`	Relationale Vergleiche
`== !=`	Gleichheit, Ungleichheit
`&`	AND
`^`	XOR

Operatoren	Anmerkung
|	OR
&&	AND
||	OR
? :	Fragezeichen-Operator
=	Zuweisung

2.7 Bedingungen und Verzweigungen

Bedingungen und Verzweigungen setzen die Logik einer Applikation um. Sie sorgen dafür, dass bestimmte Teile der Applikation nur beim Eintreten bestimmter Bedingungen, die erst zur Laufzeit bestimmt werden können, ausgeführt werden. Bedingungen und Verzweigungen sind somit das Salz in der Suppe der Programmierung und sorgen dafür, dass Sie Ihre Ideen umsetzen können.

2.7.1 if-Anweisung

Die `if`-Anweisung ist eine der elementarsten Anweisungen jeder Programmiersprache. Sie führt die ihr nachfolgende Anweisung (oder die nachfolgenden Anweisungen, soweit diese in einem Block gekapselt sind) aus, wenn die gestellte Bedingung zutrifft und den Wert `true` zurückgibt. Die Syntax einer `if`-Anweisung sieht so aus:

```
if(<Bedingung>) {
   <Anweisungen>
}
```

Die Bedingung einer `if`-Schleife muss stets einen Wert zurückgeben, der `true` oder `false` entspricht. Wenn gewünscht ist, dass auch etwas geschieht, falls die Bedingung den Wert `false` zurückgibt, dann verwenden Sie das optionale Schlüsselwort `else`:

```
if(<Bedingung>) {
   <Anweisungen>
} else {
   <Anweisung>
}
```

Sie können sogar mehrere `if`-Anweisungen verschachteln. Dies bedeutet, dass der `else`-Zweig weitere `if`-Anweisungen enthält:

```
if(<Bedingung>) {
   <Anweisungen>
} else if(<Bedingung>) {
   <Anweisungen>
} else {
   <Anweisungen>
}
```

Sehen wir uns einige Beispiele für `if`-Anweisungen an:

```
if(a == 10) {
   System.out.println("a is 10");
}

if(name == "Hans") {
   System.out.println("Hello Hans!");
} else {
   System.out.println("You are not Hans!");
}

if(time >= 11 && time <= 14) {
   System.out.println("It's time for meal!");
} else if(time >= 14 && time <= 17) {
   System.out.println("It's time for lunch!");
} else if(time > 6 && time <= 11) {
   System.out.println("What about some breakfast or lunch?");
} else {
   System.out.println("Nothing to eat right now!");
}
```

Wie Sie gerade am letzten Beispiel gesehen haben, kann es bei vielen unterschiedlichen Werten, auf die wir prüfen wollen, sehr schnell recht monoton und unübersichtlich werden. Aus diesem Grund bietet uns Java das `switch`-Statement, mit dem wir Tests und Aktionen gruppieren können.

2.7.2 Switch-Anweisung

Das `switch`-Statement erlaubt es, einen Test durchzuführen und in Abhängigkeit vom Testergebnis Anweisungen auszuführen. Es erlaubt es weiterhin, eine Standard-Aktion festzulegen, die ausgeführt werden soll, wenn keine Übereinstimmung der Testbedingung mit den Ergebnis-Werten gefunden werden sollte:

```
switch(<Ausdruck>) {
   case <Wert>:
      <Anweisungen>
      [break;]
   default:
      <Anweisungen>
}
```

Die zu prüfenden Werte eines `switch`-Statements müssen primitive Datentypen sein. Leider sind dabei nicht alle primitiven Datentypen zulässig. Erlaubt sind ausschließlich:

- byte
- char
- short
- int

Die einzelnen `case`-Zweige testen ausschließlich auf Gleichheit. Sollte eine Übereinstimmung auftreten, wird der im `case`-Zweig eingeschlossene Code (soweit vorhanden, denn im `case`-Zweig muss nicht zwingend Code stehen)

ausgeführt. Diese Ausführung beschränkt sich nicht nur auf den aktuellen Zweig, sondern läuft weiter, bis alle folgenden `case`- und `default`-Zweige durchlaufen sind und die `switch`-Anweisung endet. Wenn Sie dies verhindern wollen, sollten Sie am Ende eines `case`-Zweiges eine `break`-Anweisung einfügen, da dann an dieser Stelle die Ausführung beendet wird.

Sehen wir uns dies kurz an einem Beispiel an:

```
switch(digit) {
   case 2:
   case 4:
   case 6:
      System.out.println("Value is either 2 or 4 or 6");
      break;
   case 7:
      System.out.println("Value is 7");
   case 8:
      System.out.println("Value is 8");
      break;
   default:
      System.out.println("Value is something different!");
      break;
}
```

Sollte die Variable `digit` den Wert 2, 4 oder 6 enthalten, wird die Ausgabe

```
Value is either 2 or 4 or 6
```

lauten. Repräsentiert `digit` den Wert 7, wird

```
Value is 7
Value is 8
```

ausgegeben, denn im `case`-Zweig für den Wert 7 ist kein `break`-Statement enthalten und deshalb wird der folgende Code ausgeführt, bis ein `break`-Statement oder das Ende der `switch`-Anweisung folgt.

Enthält `digit` den Wert 8, wird genau dies bestätigt:

```
Value is 8
```

Für jeden anderen Wert wird die Ausgabe „*Value is something different*“ lauten.

2.8 Schleifen

Schleifen durchlaufen den enthaltenen Code so lange, bis entweder eine Abbruchbedingung zutrifft oder auf eine `break`-Anweisung getroffen wird.

Java unterscheidet vier verschiedene Schleifentypen:

2.8.1 for-Schleife

Die for-Schleife ist eine der mächtigsten und flexibelsten Schleifenformen überhaupt. In ihrem Kopf beinhaltet sie drei Ausdrücke, die allesamt optional sind:

```
for(Init; Test; Update) {
   <Anweisungen>
}
```

Die einzelnen Ausdrücke haben folgende Bedeutung:

- Der Init-Ausdruck wird beim Start der Schleife aufgerufen und initialisiert in der Regel einen Zähler oder eine Prüf-Variable. Variablen, die im Init-Bereich deklariert werden, sind nur innerhalb der Schleife und ihres Rumpfes gültig – dies bedeutet für uns, dass wir Zählvariablen deklarieren können und diese Variablen in derselben Methode in anderen for-Schleifen auch wieder gleich benannt sein dürfen.
- Der Test-Ausdruck kann die Schleife abbrechen, wenn er den Wert `false` zurückgibt. In der Regel wird hier die im Init deklarierte Zählvariable überprüft, ob sie einen Maximal-Wert bereits erreicht hat. Das Ergebnis des Test-Ausdrucks muss ein boolescher Wert (`true` oder `false`) sein. Alternativ kann der Test-Ausdruck auch komplett leer sein, dann erfolgt keine Prüfung.
- Der Update-Ausdruck dient dazu, den im Init deklarierten und im Test geprüften Schleifenzähler zu erhöhen. Er wird – ebenso wie der Test-Ausdruck – bei jedem Durchlauf der Schleife ausgeführt. Ist der Update-Ausdruck leer, wird er ignoriert.

For-Schleifen sind – wie bereits erwähnt – sehr flexibel. Sehen wir uns deshalb einige Beispiele an, die diese Schleifenart verwenden:

```
boolean run = true;
for(;;) {
   // ...
   if(!run) {
      break;
   }
}
```

Dieses Beispiel implementiert eine Endlosschleife – keine Start-Bedingung, keine Abbruchbedingung, kein Test. Aus diesem Grund muss innerhalb des Schleifenkörpers an irgendeiner Stelle ein Abbruch per `break` möglich sein.

```
for(int i=1; i<10; i++) {
   // ...
}
```

Diese for-Schleife deklariert im Init-Bereich eine Zählvariable `i`. Im Test-Bereich wird überprüft, ob der Wert von `i` kleiner als 10 ist. Innerhalb des Update-Bereichs wird `i` um 1 erhöht, so dass die Schleife insgesamt neunmal durchlaufen wird.

Sie werden bei der Lektüre dieses Buches häufig auf `for`-Schleifen stoßen. Sie sind eine der wichtigsten Schleifenarten überhaupt und werden aufgrund ihrer Flexibilität gerne eingesetzt.

2.8.2 for-each-Schleife

Die `for-each`-Schleife ist neu bei Java 5. Sie bietet uns die Möglichkeit, alle Elemente eines Arrays oder einer Auflistung zu durchlaufen:

```
for(<Element-Typ> <Element> : <Liste>) {
   <Anweisungen>
}
```

Der Kopf der Schleife beinhaltet die Anweisungen zum Auslesen der Daten. Hier wird zunächst eine lokale Variable eines bestimmten Element-Typs deklariert. Durch einen Doppelpunkt getrennt folgt das Array oder die Liste, aus der Daten abgerufen werden sollen.

Die `for-each`-Schleife durchläuft grundsätzlich alle Elemente einer Liste. Sie kann nur mit Hilfe des Schlüsselwortes `break` abgebrochen werden.

Beispiele für `for-each`-Schleifen werden Sie im Rahmen dieses Buches mehrfach finden. Beispielsweise hier:

```
String[] items = new String[] {"One", "Two", "Three"};
for(String item : items) {
   System.out.println(item);
}
```

Die Ausgabe dieses Fragments wäre

```
One
Two
Three
```

Beachten Sie, dass der Typ des Elements, dem die einzelnen Listenelemente zugewiesen werden, immer dem Listentyp entsprechen muss. Zwar findet unter Umständen eine implizite Typ-Konvertierung statt, jedoch funktioniert das in keinem Fall mehr bei komplexen Objekten.

2.8.3 while-Schleife

Die `while`-Schleife wird so lange durchlaufen, wie ihre Laufbedingung zutrifft:

```
while(<Laufbedingung>) {
   <Anweisungen>
}
```

Vor dem ersten Durchlaufen des Schleifenkörpers wird die im Kopf definierte Laufbedingung überprüft. Trifft diese Bedingung zu, wird der Schleifenkörper durchlaufen. Dieser Vorgang wiederholt sich anschließend – und zwar so lange, bis die Laufbedingung nicht mehr zutrifft.

Ein Beispiel für eine `while`-Schleife könnte so aussehen:

```
int current = 0;
while(current < 4) {
   current++;
   System.out.println(current);
}
```

Die Ausgabe ist:

```
1
2
3
4
```

Die Zahl 5 wird nicht mehr ausgegeben, denn nach Ausgabe der Zahl 4 wird die Zählvariable `current` um 1 erhöht und ihre Laufbedingung (`current` kleiner als 5) trifft nicht mehr zu.

2.8.4 do-while-Schleife

Die `do-while`-Schleife funktioniert analog zur `while`-Schleife, es gibt aber einen ganz wesentlichen Unterschied: Sie prüft auf die Gültigkeit ihrer Laufbedingung immer erst am Ende eines Schleifendurchlaufs:

```
do {
   <Anweisungen>
} while (<Laufbedingung>);
```

Dieses Konstrukt sorgt dafür, dass der Schleifenkörper in jedem Fall einmal durchlaufen wird. Sehen wir uns dies anhand eines kleinen Beispiels an:

```
int current = 0;
do {
   current++;
   System.out.println(current);
} while (current < 4);
```

Die Ausgabe wird in diesem Fall

```
1
2
3
4
```

sein.

2.8.5 break und continue

Wir haben das Schlüsselwort `break` bereits mehrfach kennen gelernt. `Break` macht genau das, was der Name aussagt: Es beendet die Ausführung der Schleife oder des `switch`-Statements und sorgt dafür, dass die Programm-Ausführung im äußeren Code-Block fortgesetzt wird. Wenn Sie Schleifen ineinander verschachteln, dann wird die Programm-Ausführung in der nächsten äußeren Schleife fortgesetzt.

Ein Beispiel für den Einsatz von `break` könnte so aussehen:

```
int count = 0;
while(count < array.length) {
   if(count > 4) {
      break;
   }

   System.out.println(
      String.format("%d = %s",
         count, array[count]));
   count++;
}
```

Hier haben wir eine `while`-Schleife, die alle Elemente eines Arrays ausgibt. Zu diesem Zweck wird eine Zählvariable `count` eingesetzt, die die Position im Array bezeichnet. Da nur die ersten fünf Elemente des Arrays ausgegeben werden sollen, wird innerhalb der Zählschleife überprüft, ob die Zählvariable einen Wert größer als 4 hat – die Zählung bei Arrays beginnt immer am Element 0, so dass hier die Elemente 0 bis 4 ausgegeben werden. Sollte die Abbruchbedingung erreicht sein, wird die Schleife mit Hilfe von `break` verlassen.

Das Schlüsselwort `continue` ist nicht – wie man vielleicht vermuten könnte – das genaue Gegenstück zum Schlüsselwort `break`. Wenn Sie das Schlüsselwort `continue` innerhalb einer Schleife verwenden, wird die Ausführung des folgenden Schleifenkörpers übersprungen – die Schleife wird also gleich wieder am Kopf fortgesetzt.

Sehen wir uns an, wie der Einsatz von `continue` aussehen könnte:

```
int count = 0;
while(count < array.length) {
   count++;
   if(count > 1 && count < 4) {
      continue;
   }

   System.out.println(
      String.format("%d = %s",
         count, array[count - 1]));
}
```

Innerhalb der `while`-Schleife, die Elemente eines Arrays ausgeben soll, verwenden wir wieder die Zählvariable `count`, die die Position innerhalb des Arrays bezeichnet. Die Elemente an den Positionen 2 und 3 sollen nicht ausgegeben werden – die `if`-Schleife führt diese Überprüfung für uns durch und sorgt für den erneuten Sprung an den Anfang der Schleife, falls die Zählvariable `count` einen Wert zwischen 2 und 3 hat.

2.9 Methoden-Deklaration

Methoden strukturieren Code und erlauben es, bestimmte Teile des Codes zu kapseln und wieder zu verwenden. Java unterscheidet – anders als beispielsweise BASIC – nicht explizit zwischen Funktionen (also Methoden, die eine Rückgabe haben) und Prozeduren (Methoden ohne Rückgabe). Dies geschieht implizit durch Angabe eines Rückgabe-Typs oder des Schlüsselwortes `void`:

```
<Rückgabe-Typ> <Methoden-Name>(<Parameter>) {
   <Anweisungen>
   [return <Rückgabe>;]
}
```

Methoden können Parameter erwarten. Wir deklarieren dies, indem wir eine durch Kommata getrennte Liste von Parametern angeben, die jeweils den Typ und den Namen des Parameters definieren. Der Rumpf der Methode wird in geschweifte Klammern (`{}`) eingefasst.

Um eine Methode ohne Parameter und ohne Rückgabe-Wert zu definieren, bedienen wir uns des Schlüsselwortes `void` für den Rückgabe-Typ und können folgenden Code verwenden:

```
void method() {
   // Anweisungen
}
```

Wollen wir, dass die Methode einen `String`-Parameter mit Namen `param` entgegennimmt, können wir diesen Code verwenden:

```
void method(String param) {
   // Anweisungen
}
```

Wenn wir einen weiteren Parameter (in diesem Fall den `int`-Parameter `age`) übergeben, könnte dies so aussehen:

```
void method(String param, int age) {
   // Anweisungen
}
```

Wenn die Methode eine Rückgabe zurückgeben soll, geben wir den Typ des Rückgabe-Wertes statt des Schlüsselworts `void` an. In diesem Fall muss aber auch tatsächlich eine Rückgabe generiert und mit Hilfe des Schlüsselwortes `return` zurückgegeben werden:

```
String method(String param, int age) {
   String result = String.format(
      "You passed the Parameters %s and %d to the method.",
         param, age);

   return result;
}
```

Um Methoden zu nutzen, rufen wir sie mit Hilfe ihres Namens und ihrer Parameter auf:

```
method("Karsten Samaschke", 28);
```

Sollte die Methode über eine Rückgabe verfügen, können wir diese Rückgabe in einer Variable gleichen Typs entgegennehmen:

```
String result = method("Karsten Samaschke", 28);
```

Sollte sich die Methode in einem Objekt befinden, müssen wir den Namen der Objektinstanz voranstellen und diesen durch den Punkt (.) vom eigentlichen Methoden-Namen trennen. Dieses Vorgehen nennt sich „Qualifizierung“:

```
String result = instance.method("Karsten Samaschke", 28);
```

2.9.1 Parameterübergabe

Parameter werden in Java stets per `call-by-value` übergeben. Dies bedeutet, das die Methode eine Kopie des Wertes der originalen Variablen erhält und diesen manipulieren kann, ohne das der originale Wert der Variablen geändert wird.

Eine Besonderheit ergibt sich aus diesem Verhalten bei der Übergabe von Objekten: Übergeben wir Objekte als Parameter, erhält die aufgerufene Methode zwar eine Kopie des Wertes der originalen Variablen. Da diese aber lediglich eine Referenz auf das Objekt hält, und eine Kopie dieser Referenz als Parameter-Wert übergeben wird, zeigen sowohl originale Variable als auch die Methoden-Variable auf das gleiche Objekt. Änderungen an Objekten innerhalb einer Methode wirken sich deshalb immer auch auf das originale Objekt aus.

Eine Übergabe von Werten als `call-by-reference` (also nicht die Übergabe einer Kopie, sondern einer Referenz auf die originale Variable) gibt es bei Java im Unterschied zu anderen Programmiersprachen nicht. Sie würde bei Objekten auch keinen Sinn machen, denn aufgrund der eben geschilderten Übergabe-Mechanismen wird keine Objekt-Instanz, sondern immer nur eine Referenz auf das entsprechende Objekt übergeben und muss deshalb nicht explizit als `call-by-reference` angefordert werden.

Sollen primitive Datentypen so übergeben werden, das sich Änderungen an ihren Werten auch auf den Wert der originalen Variable auswirken, so könnten die weiter oben beschriebenen Wrapper-Klassen zum Einsatz kommen.

Achtung

Änderungen an Objekten innerhalb von Methoden wirken sich auch auf die originalen Objekte aus. Änderungen an primitiven Datentypen innerhalb von Methoden wirken sich nicht auf die originalen Variablen aus.

2.10 Fazit

Der Einstieg in Java ist nicht schwer – jedenfalls was die Sprach-Strukturen, Anweisungen und Datentypen angeht. Die Definition von Variablen und Konstanten ist ebenso simpel wie die Arbeit mit Operatoren, die sich zwar komplex liest, aber in der Praxis einfach und intuitiv erfolgen kann. Die Übersichtlichkeit der Kontroll-Strukturen (`if` und `switch`) sowie die geringe Anzahl der verfügbaren Schleifen halten die Schwelle zum Einstieg niedrig.

Sie werden sich nun sicherlich fragen, warum solch ein Wirbel um Java veranstaltet wird, denn schließlich ist die Sprache offensichtlich recht simpel und ohne großen Aufwand zu erlernen? Genau aus diesem Grund wird Java so gerne eingesetzt: Die Sprache ist schnell zu erlernen und relativ intuitiv einzusetzen (soweit man das von einer Programmiersprache behaupten kann).

Dennoch ist Java sehr mächtig und leistungsfähig, wie wir im weiteren Verlauf dieses Buches noch sehen werden. Insofern hilft es uns, dass der Einstieg so leicht erfolgen kann.

3

Objektorientierte Programmierung

In diesem Kapitel werden wir eines der wichtigsten Themen bei der Entwicklung mit Java und auch anderen modernen Hochsprachen besprechen: Die Objektorientierung.

Objektorientierte Programmierung bedeutet einen Wandel in der Herangehensweise gegenüber traditionellen Verfahren – statt in Abläufen denkt man nun in Objekten. Eine Lösung – egal, ob eigenständiges Programm oder kleine Komponente – besteht dabei immer aus Objekten, die bestimmte Strukturen repräsentieren.

Das Geheimnis objektorientierter Programmierung besteht darin, zusammengehörige Daten und Methoden in einer eigenen Struktur zu kapseln. Dadurch werden gleich mehrere Ziele erreicht:

- Der Quellcode wird übersichtlicher, weil besser strukturiert
- Der Quellcode wird modularer, weil in einzelne kleine Bestandteile zerlegt
- Der Quellcode wird wiederverwendbarer, da einzelne Elemente in sich abgeschlossen sein können
- Der Quellcode wird sicherer, denn in kleineren Elementen und Strukturen kann man Fehler besser erkennen
- Einzelne oder alle Komponenten einer Applikation können auf Wunsch in anderen Applikationen weiter verwendet werden

Wie Sie diese Ansprüche verwirklichen können, werden wir im Folgenden betrachten.

3.1 Klassen und Objektinstanzen

„Objektorientierung“ ist eines der sogenannten Buzz-Words der 90er Jahre gewesen: Alles wurde auf einmal objektorientiert, viele Sprachen wurden mit objektorientierten Ansätzen versehen – meist nicht unbedingt zu ihrem Besten, denn eine nachträglich aufgepfropfte Objektorientierung leidet oftmals unter

Einschränkungen und Limitierungen, die die alltägliche Arbeit unnötig verkomplizieren.

Die gute Nachricht: Bei Java ist das anders. Java ist von vornherein als objektorientierte Sprache entwickelt worden. Bei Java ist Objektorientierung nicht nachträglich aufgesetzt, sondern integraler Bestandteil der Sprache. Bis auf die primitiven Datentypen ist bei Java alles ein Objekt.

Die schlechte Nachricht: Dies verhindert nicht, daß wir uns intensiver mit dieser Materie auseinandersetzen müssen. Oftmals haben Ein- und Umsteiger ein Denkproblem, wenn sie plötzlich objektorientiert arbeiten sollen.

3.1.1 Abstraktion

Die wichtigste Idee der objektorientierten Programmierung ist die Trennung von *Konzept* und *Umsetzung*. Dies bedeutet, das ein fundamentaler Unterschied zwischen einer *Klasse* und einem *Objekt* existiert: Eine Klasse beschreibt eines oder mehrerer Objekte, während das Objekt eine konkrete Manifestierung einer Klasse darstellt.

Nehmen wir ein Beispiel. Denken Sie generell an Autos – nicht an ein bestimmtes Fabrikat oder ein bestimmtes Modell, sondern an das, was ein Auto ausmacht: Autos bestehen aus Reifen, Türen und einem Motor. Sie verfügen über eine Farbe und haben einen Kilometerstand, der angibt, wie weit das Fahrzeug schon bewegt worden ist. Autos können fahren und dabei eine bestimmte Anzahl an Personen transportieren. All dies sind Eigenschaften und Fähigkeiten aller Autos – nicht eines konkreten Fahrzeuges. Derartige Eigenschaften und Fähigkeiten (wir sprechen hier allerdings eher von Methoden) können in einer Klasse definiert werden.

Klasse

Eine *Klasse* definiert Eigenschaften und Methoden. Eine Klasse ist etwas Abstraktes – sie erlaubt es uns, etwas zu beschreiben und ähnelt dabei sehr stark einem Bauplan, einem Konzept oder einem Rezept.

Klassen beschreiben konkrete Elemente als Summe ihrer *Eigenschaften*. Eigenschaften repräsentieren Zustände und Werte, führen aber selbst keine Verarbeitung durch. Eine Klasse, die Autos beschreiben sollte, könnte demnach etwa über die Eigenschaften „Farbe“, „Alter“ oder „Kilometerstand“ verfügen. Eigenschaften repräsentieren somit Zustände eines Objekts.

Daneben verfügt eine Klasse meist auch über *Methoden*. Diese Methoden können auf die Eigenschaften der Klasse wirken oder Aktionen durchführen, die andere Klasse beeinflussen. Eine Auto-Klasse könnte etwa über die Methoden „Altere“, „Erhöhe die Anzahl der Beulen“, „Verdrecke die Sitze“ oder „Drehe den Tacho zurück“ verfügen. Methoden repräsentieren also ein Verhalten eines Objekts.

Zuletzt gibt uns eine Klasse auch immer Aufschluß darüber, wie wir ein konkretes Objekt erzeugen können.

Eine Klasse stellt uns also folgende Dinge zur Verfügung:

- Eigenschaften, die das konkrete Objekt beschreiben
- Methoden, mit denen diese Eigenschaften manipuliert werden können
- Informationen darüber, wie konkrete Objekte erzeugt oder hergestellt werden können

Objekt

Mit Hilfe der von einer Klasse definierten Eigenschaften, Methoden und Informationen ist jedes konkrete *Objekt* beschreibbar. Wir sind somit in der Lage, eine beliebige Anzahl an konkreten Objekten zu erzeugen. Jedes dieser Objekte steht für sich, beschreibt sich selbst und ist dadurch von anderen Objekten unterscheidbar. Ein konkretes Objekt ist dabei jedoch immer eine Instanz der Klasse, nach der es modelliert worden ist.

Nehmen wir eine Klasse „Auto“: Diese Klasse stellt die Eigenschaften bereit, die wir benötigen, um jedes derzeit auf einer Autobahn befindliche Fahrzeug hinlänglich zu beschreiben. Wir sind darüber hinaus in der Lage, neue Fahrzeuge zu erstellen (oder zu bauen) und können mit Hilfe der in der Klasse definierten Methoden die Eigenschaften von Fahrzeugen manipulieren..

Sie werden oftmals feststellen, daß die Begriffe „Instanz“ und „Objekt“ synonym gebraucht werden. Tatsächlich werden Sie das hier in diesem Buch auch feststellen können. Natürlich bestehen im Sinne einer Definition Unterschiede zwischen Instanzen und Objekten. Praktisch spielt dies aber keine Rolle und die Begriffe werden so eingesetzt, wie es besser klingt. Und genau so halten wir es in diesem Buch auch.

Durch die Unterscheidung von Klassen und Objekten können wir uns auf das Wesentliche konzentrieren und bei der Definition einer Klasse die unwichtigen Details außen vor lassen. Wir können somit das Wesen eines Problems leichter erfassen und reduzieren im Idealfall den Aufwand zur Problemlösung enorm, da wir Methoden und Eigenschaften einmal definieren und mehrfach wiederverwenden können.

3.1.2 Kapselung

Klassen definieren Eigenschaften und Methoden. Die Eigenschaften werden durch Variablen repräsentiert, die für jede Instanz neu angelegt und instanziert werden. Diese Variablen werden auch gerne als *Attribute*, *Member-* oder *Instanzvariablen* bezeichnet und repräsentieren den Zustand des beschriebenen Objekts.

Methoden repräsentieren dagegen das Verhalten eines Objekts. Sie erlauben das Manipulieren der Zustände der aktuellen Objektinstanz oder anderer Instanzen.

Klassen fassen Methoden und Eigenschaften zusammen, was als *Kapselung* bezeichnet wird. Die Kapselung dient dem Zweck, die Komplexität bei der Bedienung eines Objekts zu verringern – wir müssen nicht speziell wissen, wie ein Auto im Inneren aufgebaut ist, um es auf einer Autobahn fahren zu können. Die Kapselung sorgt dafür, das wir von diesen Informationen ferngehalten werden. Angenehmer Nebeneffekt: Wir können bei der Arbeit mit einem konkreten Objekt meist nicht all zu viel Schaden anrichten, denn wir erhalten aufgrund der Kapselung keinen Zugriff auf die Membervariablen und die Hilfsmethoden des Objekts.

3.1.3 Wiederverwendbarkeit

Aufgrund der Kapselung von Code steigt die *Wiederverwendbarkeit* dieses Codes. Dies bedeutet für uns, das wir die in einer Klasse definierten Methoden auch in den konkreten Objekten verwenden können, ohne sie neu implementieren zu müssen. Ebenso ist es möglich, in Klassen definierten Code aus anderen Methoden oder Objekten heraus zu verwenden. Dies alles ermöglicht es, Code wieder zu verwenden, ohne ihn stets neu schreiben zu müssen. Dadurch steigt die Effizienz bei der Entwicklung, insbesondere deshalb, weil die Fehleranfälligkeit schon beim Schreiben von Applikationen sinkt.

3.2 Beziehungen zwischen Klassen

Klassen können auf unterschiedliche Art und Weise miteinander in Beziehung stehen. Bei der objektorientierten Programmierung unterscheiden wir drei Beziehungstypen:

- Spezialisierung
- Komposition
- Assoziation

3.2.1 Spezialisierung

Die *Spezialisierung* von Klassen soll an einem einfachen Beispiel erläutert werden: Autos, Lastwagen, Lokomotiven, Straßenbahnen, Busse, Motorräder und Fahrräder haben alle eines gemeinsam: Sie sind Fahrzeuge. Wenn wir die Gemeinsamkeiten aller dieser Fahrzeuge untersuchen, läßt sich feststellen, das sie sich in einigen Punkten ähnlich sind:

- Sie verfügen über die Fähigkeit, sich zu bewegen
- Sie verfügen über Räder
- Sie verfügen über ein Steuer-Instrument (Lenker, Lenkrad, Joystick oder ähnliches)
- Sie verbrauchen Energie
- Sie haben eine wie auch immer geartete Art von Karosserie (und sei es nur der Fahrrad-Rahmen)
- Sie haben eine Farbe

Wenn wir alle diese Fahrzeuge beschreiben sollen, würden wir sinnvollerweise eine Klasse `Fahrzeug` schreiben, die alle oben genannten Eigenschaften und Methoden beinhaltet.

Alle oben genannten Fahrzeuge verfügen über die Eigenschaften und Methoden der Klasse `Fahrzeug` und fügen dieser eigene Methoden und Eigenschaften hinzu – ein Auto würde beispielsweise die Art des Karosserie-Aufbaus definieren, wogegen eine Fahrrad vielleicht eher darüber definiert wird, ob es ein Sportfahrrad oder ein Mountainbike ist. Eine Lokomotive benötigt Informationen darüber, ob sie einen Elektro-, Diesel- oder Dampfantrieb hat.

Allgemeiner gesagt: Die Spezialisierung beruht darauf, das eine Klasse B Eigenschaften einer Klasse A besitzt und dieser eigene Eigenschaften oder Methoden hinzufügen kann. Diese Art der Beziehung wird bei Java durch *Ableitung* oder *Vererbung* ausgedrückt. Diese Vererbung kann dabei mehrstufig sein – eine Klasse kann von einer anderen Klasse erben, die ihrerseits auch von einer Klasse geerbt hat. Dadurch entsteht eine Vererbungshierarchie, die für Fahrzeuge beispielsweise so aussehen könnte:

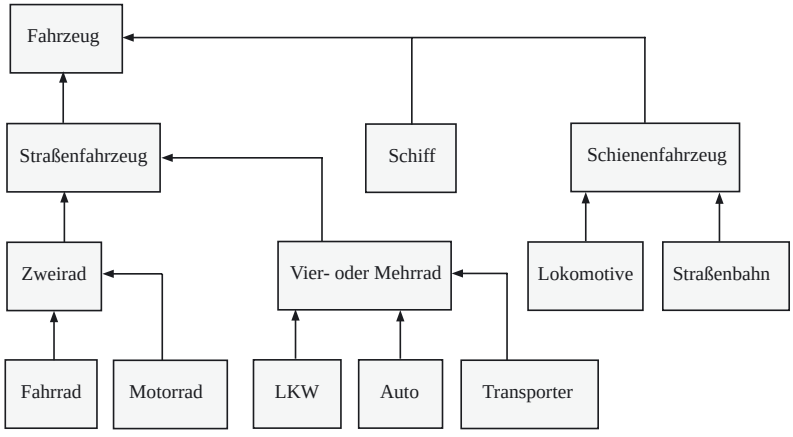


Abbildung 3.1
Vererbungshierarchie

Im Vergleich zu anderen Programmiersprachen beschränkt Java die Möglichkeiten der Vererbung: Klassen können in Java nur von einer Super-Klasse erben – ein paralleles Erben von mehreren Super-Klassen ist nicht möglich, kann aber mit Hilfe von Interfaces nachgestellt werden.

3.2.2 Komposition und Aggregation

Eine andere Art von Beziehung besteht zwischen einem Auto und seinen Komponenten: So verfügt ein Auto in der Regel über vier oder fünf Reifen, die ihrerseits sinnvollerweise durch eine Klasse `Tyre` repräsentiert werden könnten. Das Auto verfügt dann beispielsweise über ein Array aus `Tyre`-Instanzen. Ebenfalls könnte ein Auto über ein Lenkrad verfügen, das durch eine `Wheel`-Instanz repräsentiert werden würde. Diese Zusammensetzung eines Objekts aus anderen Objekten wird als *Komposition* bezeichnet.

Ebenfalls im Zusammenhang mit Komposition ist oftmals von *Aggregation* die Rede. Dieser Begriff bezeichnet eine nicht-existenzielle Beziehung zwischen zwei Objekten. Ein Beispiel für die Aggregation wäre das Hinzufügen von Koffern zum Kofferraum eines Fahrzeuges – das Fahrzeug ist auch funktionsfähig, wenn keine Koffer eingeladen worden sind. Anders ist dies bei Rädern der Fall – ohne Räder funktioniert ein Fahrzeug in der Regel nicht.

Diese Art von Unterscheidung zwischen Aggregation und Komposition ist mehr inhaltlicher denn praktischer Natur – in der Implementierung gibt es in der Regel keinen Unterschied, da beide Beziehungen über Instanz-Variablen abgebildet werden. Es obligt der Klasse selber, die Abhängigkeiten zu prüfen und gegebenenfalls entsprechende Warnungen oder Fehler zu produzieren.

3.2.3 Assoziation

Die *Assoziation* ist die allgemeinste Art von Beziehungen zwischen Klassen. Sie beruht darauf, das eine Klasse eine Instanz einer anderen Klasse samt deren Methoden nutzt. Ein Objekt A nutzt also Methoden eines Objekts B für seine Zwecke aus. Ein typischer Anwendungsfall sind Hilfsklassen, die Methoden beinhalten, die von anderen Klassen gleichermaßen verwendet werden sollen.

Ebenfalls eine Verwendung von Assoziationen ist der Einsatz von Instanzvariablen, die Referenzen auf andere Objekttypen halten oder die Verwendung von Objekten als Methoden-Argumenten.

3.3 Polymorphie

Der Begriff *Polymorphie* bezeichnet die Fähigkeit von Objektvariablen, Objekte unterschiedlicher Typen aufzunehmen. Dies ist allerdings nicht unbegrenzt möglich, sondern ist auf Objekte eines Typs und davon abgeleitete Objekte beschränkt. Ein Beispiel: Eine Variable vom Typ `Fahrzeug` kann sowohl `Fahrzeug`-, als auch `Boot`-, `Zweirad`- oder `Auto`-Objekte halten. Dagegen kann eine Variable vom Typ `Zweirad` nur `Zweirad`-, `Fahrrad`- und `Motorrad`-Objekte halten – bei Zuweisung eines `Auto`-Objekts würde ein Fehler auftreten. Die Polymorphie funktioniert also nur innerhalb einer Vererbungshierarchie (und dort auch nur abwärts) und spiegelt dabei diese Hierarchie wieder.

Spannend wird Polymorphie dann, wenn mit *überlagerten* Methoden gearbeitet wird. Das Prinzip dahinter: In einer übergeordneten Klasse A wird eine Methode b definiert. Diese kann nun von einer ableitenden Klasse C überschrieben werden. Wird nun eine Instanz der Klasse C mit der überschriebenen Methode b einer Variablen vom Typ A zugewiesen – was dank Polymorphie funktioniert, da C von A erbt und somit in der Hierarchie unterhalb von A liegt – wird erst zur Laufzeit bestimmt, welche Ausprägung von b tatsächlich eingebunden wird: Die von A definierte Version, oder die von C definierte. Sollte eine Variable vom Typ C zugewiesen sein, wird die von C definierte Version von b eingebunden.

Nehmen wir nun an, eine Klasse D erbt ebenfalls von A und definiert keine eigene Version der Methode b. Was wird nun geschehen? Wird ein Fehler generiert? Nein. Zur Laufzeit wird die von der nächsthöheren Hierarchie-Ebene definierte Version der Methode b eingebunden. Hat diese ebenfalls keine eigene Definition der Methode, wird die Methode b von deren Basis-Klasse eingebunden.

Ein Beispiel: In der Klasse `Fahrzeug` definieren wir eine Methode `fahre()`, die dafür sorgt, das sich das Fahrzeug (virtuell gesehen) bewegt. Dabei wird der Kilometerstand erhöht und der Tankinhalt verringert. Die Klasse `Schienenfahrzeug`, die als Basis-Klasse für `Lokomotive` und `Straßenbahn` fungiert, muss diese Methode anders implementieren – Loks und Straßenbahnen verfügen in der Regel über einen Elektro-Antrieb und leeren deshalb keinen Tank. Die Klasse `Straßenfahrzeug` dagegen muss diese Methode nicht anders implementieren und kann die ursprüngliche Version verwenden.

Weisen wir nun einer Variablen vom Typ `Fahrzeug` eine `Schienenfahrzeug`-Instanz zu und rufen deren Methode `fahre()` auf, wird zur Laufzeit der Applikation die `fahre()`-Implementierung der `Schienenfahrzeug`-Klasse verwendet. Selbiges gilt für Instanzen der `Lokomotive`- und `Straßenbahn`-Klassen, soweit diese nicht selbst wieder über eine eigene Implementierung (etwa für eine dieselbetriebene Lok) verfügen. Wird nun aber eine Instanz der `Auto`-Klasse zugewiesen, die keine eigene Überlagerung von `fahre()` hat, überprüft der Java-Interpreter auch die Super-Klasse `Vier- oder Mehrrad` auf das Vorhandensein einer `fahre()`-Überlagerung. Dieser Prozess wiederholt sich für die jeweilige Super-Klasse solange, bis eine fahre()-Implementierung gefunden worden ist. Sollte dies nicht geschehen – etwa weil es keine weitere Überlagerung der Methode gibt – wird die Implementierung der Basis-Klasse `Fahrzeug` eingebunden werden.

Dieses Verhalten führt in der Praxis zu sehr eleganten Lösungen, da Entwickler so nicht gezwungen sind, stets alle Methoden einer Super-Klasse zu implementieren. Stattdessen können sie sich in Ableitungen auf die Methoden beschränken, die sie tatsächlich anders oder neu implementieren müssen, um die gewünschte Funktionalität sicher zu stellen.

3.4 Definition von Klassen

Klassen zu definieren ist nicht schwer. Alles, was wir benötigen, ist das Schlüsselwort *class*, gefolgt von dem Klassen-Namen. Der Aufbau einer Klassen-Deklaration sieht also immer so aus:

```
[<Zugriffmodifier>] class <Klassen-Name> {
   ...
}
```

Beginn und Ende der Klasse werden dabei durch geschweifte Klammern gekennzeichnet. Wollten wir also eine Klasse `SimpleCar` deklarieren, könnte ein einfacher Ansatz so aussehen:

```
public class SimpleCar {}
```

Achtung

Java hat eine goldene Regel: Jede öffentliche Klasse gehört in eine eigene gleichnamige **.java*-Datei. Dies bedeutet, daß eine öffentliche Klasse *SimpleCar* in einer Datei *SimpleCar.java* gespeichert wird. Weitere Klassen darf diese Datei nur aufnehmen, wenn diese innerhalb von *SimpleCar* geschachtelt werden oder nicht als öffentlich deklariert sind, anderenfalls müssen die entsprechenden Klassen auch wieder innerhalb einer eigenen Datei definiert werden.

3.4.1 Eigenschaften / Instanz-Variablen

Eigenschaften werden in der Regel über *Getter*- und *Setter*-Methoden implementiert. Innerhalb dieser Methoden wird auf Instanz-Variablen zugegriffen, die die verschiedenen Werte halten.

In unserer Auto-Klasse werden wir nun drei Instanz-Variablen definieren, die den verschiedenen Eigenschaften eines einfachen Autos entsprechen sollen:

Listing 3.1
Eine einfache Klasse

```
public class SimpleCar {

   private String manufacturer = "Unknown";
   private String color = "White";
   private double miles = 0;

}
```

Diese internen Variablen sind nicht von außen (also anderen Klassen oder auch anderen Objekt-Instanzen) sichtbar. Deklariert werden Sie genauso, wie Sie auch normale Variablen in Methoden deklarieren, also in der Form

```
[<Zugriffsmodifier>] <Typ> <Name> [= <Standardwert>];
```

Um sie erreichbar zu machen, nutzen wir Getter- und Setter-Methoden.

3.4.2 Getter und Setter

Mit Hilfe von Getter-Methoden, kann der Wert der repräsentierten Variable abgerufen werden. Eine Getter-Methode hat in der Regel diesen Aufbau:

```
[<Zugriffsmodifier>] <Typ> get<Variable>() {
   return <Variable>;
}
```

Der Zugriffsmodifizierer ist optional. Wird er nicht angegeben, geht Java davon aus, daß die Methode zwar öffentlich erreichbar sein soll, sich dies aber auf das aktuelle Package beschränkt (vgl. *public*-Modifier). Vor der Rückgabe eines Wertes kann der Setter noch interne Überprüfungen vornehmen. So kann sichergestellt werden, dass nur gültige Werte an die übergeordnete Methode zurückgegeben werden. Angenehmer Nebeneffekt: Es muss kein Zugriff auf die internen Variablen der Klasse gewährt werden – weder lesend noch schreibend.

Das Gegenstück zu Getter-Methoden sind Setter-Methoden. Sie erlauben eine Zuweisung von Werten zu den repräsentierten Variablen und nehmen stets einen Parameter entgegen. Eine Setter-Methode sieht in der Regel so aus:

```
[Zugriffsmodifier] void set<Variable>(<Typ> <Name>) {
   this.<Variable> = <Name>;
}
```

Eine Setter-Methode erlaubt es, vor dem Zuweisen des Wertes an die interne Variable noch Überprüfungen vorzunehmen. Die Klasse kapselt damit ihren internen Aufbau und stellt sicher, das nur gültige Werte zugewiesen werden können. Dadurch steigt die Sicherheit der Applikation und die Fehleranfälligkeit verringert sich.

Übertragen auf unsere Klasse ergibt sich folgender Aufbau:

Listing 3.2
Getter und Setter einer Klasse

```
public class SimpleCar {

   private String manufacturer = "Unknown";
   private String color = "White";
   private double miles = 0;

   public String getManufacturer() {
      return manufacturer;
   }

   public void setManufacturer(String manufacturer) {
      this.manufacturer = manufacturer;
   }

   public String getColor() {
      return color;
   }

   public void setColor(String color) {
      this.color = color;
   }

   public double getMiles() {
      return miles;
   }

   public void setMiles(double miles) {
      this.miles = miles;
   }
}
```

Im Grunde ist damit unsere Klasse bereits fertig – sie kapselt die einzelnen Eigenschaften eines (noch) abstrakten Autos. In der derzeitigen Form haben wir eine reine Datenklasse geschrieben – oftmals ist dies genau das Richtige, um eine Repräsentation von zusammengehörigen Daten zu erreichen.

Jedoch werden Sie oftmals keine reinen Datenklassen einsetzen – meist verfügen die Klassen auch noch über Methoden, mit deren Hilfe die enthaltenen Daten manipuliert oder bestimmte Informationen abgerufen werden können. Dinge eigentlich, die zwar meist auch von außen geschehen könnten, dann aber eben nicht Bestandteil des Objekts wären, und für jedes andere Projekt neu implementiert werden müssten, was inbesondere im Hinblick auf Wartbarkeit nicht optimal und befriedigend wäre.

Also fügen wir einige Methoden zur Klasse hinzu. Die Methode `drive()` erlaubt es uns, eine Streckenangabe in Kilometern zu übergeben und berechnet diese Entfernung in Meilen um. Mit Hilfe der Methode `getKilometers()` erhalten wir die Möglichkeit, eine Meilen-Angabe in Kilometer umzurechnen. Die Methode `getStatus()` schließlich gibt uns den Status des PKW anhand der gefahrenen Meilen zurück:

Der Unterschied zwischen *Eigenschaften* (*Properties*) und *Methoden* (*Funktionen*) ist bei Java mehr geistiger als syntaktischer Natur. Tatsächlich werden mit Properties die Methoden bezeichnet, die als Getter oder Setter für Instanz-Variablen fungieren, während Methoden eine verarbeitende Funktion innerhaben und Werte manipulieren oder Operationen durchführen. Lassen Sie sich also nicht verwirren – akzeptieren Sie die Begriffe „Eigenschaft“ oder „Property“ als Synonym für Getter und Setter. In der Praxis wird bei Java ohnehin mehr von Getter- und Setter-Methoden gesprochen, als von Objekt-Eigenschaften.

Listing 3.3
Repräsentiert ein Fahrzeug: SimpleCar.java

```
public class SimpleCar {

   private String manufacturer = "Unknown";
   private String color = "White";
   private double miles = 0;

   private final double KMTOMILE = 0.62137;
   private final double MILETOKM = 1.60934;

   public String getManufacturer() {
      return manufacturer;
   }

   public void setManufacturer(String manufacturer) {
      this.manufacturer = manufacturer;
   }

   public String getColor() {
      return color;
   }

   public void setColor(String color) {
      this.color = color;
   }

   public double getMiles() {
      return miles;
   }

   public void setMiles(double miles) {
      this.miles = miles;
   }

   public void drive(double kilometres) {
      this.miles += Math.round(
         (kilometres * KMTOMILE)* 100.) / 100.;
   }

   public double getKilometres(long miles) {
      return Math.round((miles * MILETOKM) * 100.) / 100.;
   }

   public String carStatus() {
      String status = "old and used";
      if(this.getMiles() <= 10) {
         status = "absolutely new";
      }
      else if(this.getMiles() <= 1000) {
         status = "new";
      }
      else if(this.getMiles() <= 10000) {
         status = "nearly new";
      } else if(this.getMiles() <= 20000) {
         status = "in perfect shape";
      }
      else if(this.getMiles() <= 50000) {
         status = "used car";
      }
```

```
        else if(this.getMiles() <= 150000) {
            status = "fair offer";
        }

        return status;
    }
}
```

Listing 3.3 *(Forts.)*
Repräsentiert ein Fahrzeug: SimpleCar.java

Direkt unterhalb der drei Instanz-Variablen haben wir noch die beiden konstanten Variablen `KMTOMILE` und `MILETOKM` hinterlegt, die uns die von den Methoden `drive()` und `getKilometres()` vorzunehmenden Umrechnungen erleichtern, indem sie die Umrechnungsfaktoren beinhalten.

Die Methode `drive()` simuliert eine Auto-Fahrt, deren Entfernung in Meilen angegeben wird, `getKilometres()` dient der generellen Umrechung einer Meilen-Angabe in Kilometer. Bei beiden Umrechnungen wird auf zwei Stellen nach dem Komma gerundet.

Die Methode `carStatus()` liefert eine Status-Angabe über den Zustand des Autos – zwar nur anhand der Kilometer geschätzt, aber sicherlich zutreffender als die Angaben des manches Gebrauchtwagen-Händlers.

Damit ist unsere Klasse fertig und einsatzbereit. Sehen wir uns im Folgenden an, wie wir Instanzen der Klasse erzeugen und mit diesen arbeiten können.

3.5 Erzeugen von Klassen-Instanzen

Klassen-Instanzen werden mit Hilfe des Schlüsselworts *new* erzeugt. Die Syntax dafür sieht folgendermaßen aus:

```
<Typ> <Instanz> = new <Typ>();
```

Kommt Ihnen das bekannt vor? Falls ja, haben Sie gut beobachtet: Die Syntax ist ähnlich der Syntax, mit der Sie schon bisher Variablen deklariert haben. Tatsächlich haben Sie dort teilweise auch schon Klassen-Instanzen erzeugt, nur eben nicht von selbstdefinierten Klassen, sondern von Klassen, die die Java-Runtime bereitstellt.

Übertragen wir diese Syntax auf unsere `SimpleCar`-Klasse, dann könnten wir folgenden Code schreiben:

```
public class SimpleCarInvoker
{

    public static void main(String[] args) {
        SimpleCar car = new SimpleCar();
    }
}
```

Wir haben zwar nunmehr eine neue Instanz der `SimpleCar`-Klasse erzeugt, so richtig sinnvoll mit ihr gearbeitet haben wir aber noch nicht.

Wenn Sie eine neue Instanz einer Klasse erzeugen wollen, die sich innerhalb einer anderen Klasse befindet, dann können Sie dies aus der äußeren Klasse heraus so wie gewohnt machen. Was aber, wenn Sie eine neue Instanz der inneren Klasse erzeugen wollten?

Dann müssten Sie dem Typ immer den Typ der äußeren Klasse voranstellen:

```
<äußerer Typ> <Instanz> =
    new <äußerer Typ>();
```

Bei einer äußeren Klasse `WrapperClass` und einer inneren Klasse `ContainedClass` sähe dies dann so aus:

```
WrapperClass.ContainedClass cls = new WrapperClass.ContainedClass();
```

Selbstverständlich können Sie auch die äußere Klasse unter Verwendung des `using`-Statements einbinden. Dann sähe der Aufbau so aus:

```
using WrapperClass;
...
ContainedClass myInstance
= new ContainedClass();
```

Listing 3.4
Erzeugen einer neuen Instanz der SimpleCar-Klasse (SimpleCarInvoker.java)

3.6 Qualifizierung: Zugriff auf Methoden

Der Zugriff auf Methoden und Eigenschaften einer Klasse erfolgt in Form der sogenannten *Punkt-Notation*, die auch als *Qualifizierung* bezeichnet wird. Diese trennt Instanzvariable und deren Methode durch einen Punkt:

```
<Rückgabe> = <Variable>.<Methode>;
```

Wollten wir also unserer `SimpleCar`-Klasseninstanz Werte zuweisen oder von ihr Daten abrufen, könnte dies folgendermaßen aussehen:

Listing 3.5 Zuweisen und Abrufen von Daten einer Klasse

```
public class SimpleCarInvoker
{
   public static void main(String[] args) {
      SimpleCar car = new SimpleCar();
      car.setManufacturer("BMW");
      car.setColor("Blue");
      car.drive(1000);

      System.out.println(String.format(
         "You are driving a %s with %s color which " +
         "has the following status: %s",
             car.getManufacturer(),
             car.getColor(),
             car.carStatus()));
   }
}
```

Nun haben wir eine funktionsfähige Applikation. Gestartet wird sie über Kommandozeile mit folgendem Befehl:

```
java SimpleCarInvoker
```

Wenn wir sie ausführen, werden wir folgende Ausgabe erhalten:

You are driving a BMW with Blue color which has the following status: new

Soweit ist unsere Klasse gut funktionsfähig. Gehen wir nun an die Arbeit, sie ein wenig zu verfeinern.

3.7 Konstruktoren und Destruktoren

Konstruktoren und *Destruktoren* erfüllen verschiedene und völlig gegensätzliche Zwecke – sind jedoch beides wichtige Bestandteile von Java-Klassen: Konstruktoren initialisieren eine Klasse und letztere räumen hinterher wieder auf. Der Zeitpunkt, an dem Konstruktoren wirksam werden, läßt sich ziemlich genau bestimmen: Sobald eine neue Klassen-Instanz angelegt wird, erfolgt die Einbindung des Konstruktors.

Bei Destruktoren dagegen ist der Zeitpunkt nicht so genau bestimmbar: Sie werden aufgerufen, wenn die Klassen-Instanz vom Java-eigenen *Garbage-Collector* (quasi dem Müllmann) beseitigt wird. Der genaue Zeitpunkt ist durch das *Garbage-Collector*-Konzept nicht genau bestimmbar – beruht dieser Ansatz des Aufräumens doch darauf, dass er asynchron erfolgt.

3.7.1 Konstruktor

Syntaktisch ist der Aufbau eines Konstruktors einfach gehalten – es handelt sich um eine Methode, die den gleichen Namen wie die Klasse trägt:

```
<Klassenname>(<Parameter>) { ... }
```

Konstruktoren verfügen über folgende wesentliche Eigenschaften:

- Sie heißen wie die umgebende Klasse
- Sie haben keinen Rückgabetyp
- Sie sind nicht direkt aufgerufen werden
- Sie können keine Werte zurückgeben

Im Lebenszyklus einer Klasse kommt der Konstuktor an sehr exponierter und zeitiger Stelle: Direkt nachdem Java den Speicherplatz für das neue Objekt reserviert und die internen Variablen initialisiert hat, wird der Konstruktor ausgeführt – also noch vor allen Methoden und Zuweisungen.

Wenn wir in unserer `SimpleCar`-Klasse einen Konstruktor verwenden würden, könnte dieser folgenden Aufbau haben:

Listing 3.6
Konstruktor einer Klasse

```
public class SimpleCar {

   private String manufacturer = "Unknown";
   private String color = "White";
   private double miles = 0;

   private final double KMTOMILE = 0.62137;
   private final double MILETOKM = 1.60934;

   /**
    * Constructor for the class
    */
   public SimpleCar() {
      this.setManufacturer("Porsche");
      this.setColor("Blue");
   }

   ...
}
```

Innerhalb des Konstruktors werden Hersteller und Farbe mit Hilfe der Setter-Methoden der Klasse zugewiesen. Sollten Sie nun direkt nach der Instanzierung der Klasseninstanz per

```
SimpleCar myCar = new SimpleCar();
```

die entsprechenden Werte für Hersteller und Farbe abrufen, würden Sie erfahren, dass Sie einen blauen Audi besitzen.

3.7.2 Destruktor

Wie bereits zuvor erwähnt, dienen Destruktoren dem Aufräumen, wenn eine Klasseninstanz nicht mehr benötigt wird. Sie könnten einen Destruktor nutzen, um Datenbank-Verbindungen wieder zu schließen oder andere, möglicherweise komplexere, Aufräumvorgänge durchzuführen.

Die Syntax eines Destruktors ist sehr einfach: Es handelt sich immer um die Methode `finalize`, die im Gegensatz zu einem Konstruktor übrigens auch direkt aus anderen Methoden heraus aufgerufen werden kann:

```
protected void finalize() throws Throwable { ... }
```

Innerhalb der Methode, die die von `Object` geerbte Methode `finalize` überschreibt, könnten Sie nun Ihren Code zum Aufräumen bereitstellen. Es bietet sich an der Stelle übrigens an, bei eigenen `finalize`-Implementierungen zumindest noch den Aufruf des Super-Destruktors einzufügen. Daher sollte die einfachste `finalize`-Implementierung so aussehen:

```
protected void finalize() throws Throwable {
   super.finalize();
}
```

Sie sollten übrigens nicht darauf warten, daß die Methode vom System aufgerufen wird. Das kann innerhalb kurzer Zeit passieren, muss es aber nicht – dies entscheidet der Garbage-Collektor. Aus diesem Grund empfiehlt es sich, die `finalize`-Methode bei Bedarf manuell einzubinden, um eventuell nötige Aufräumarbeiten zeitnah vornehmen zu können.

3.8 Methoden-Überladung

Haben Sie sich nicht auch schon beim Erstellen von Methoden gefragt, warum Sie die Parameter immer so starr vorgeben müssen und nicht mit einer Art von optionalen Parametern arbeiten können?

Oder anders herum: Sie möchten mehrere Methoden anbieten, die allesamt das gleiche erledigen, aber unterschiedliche Parameter besitzen? Dann hatten Sie bisher ein echtes Problem – schließlich durften Sie derartige Methoden immer wieder neu implementieren, auch wenn sich nur der Kopf unterschieden hat.

Aus diesem Grund bietet Java ein Feature, das alle oben genannten Probleme löst und Ihnen das Leben deutlich erleichtert: *Methoden-Überladung* – gerne auch Neudeutsch als *Method-Overloading* bezeichnet.

Unter Methoden-Überladung versteht man die Deklaration von Methoden innerhalb einer Klassen-Hierarchie, die zwar den gleichen Namen und den gleichen Rückgabewert haben, sich dennoch in Rückgabe-Typ und/oder Anzahl bzw. Typ der Parameter unterscheiden.

Intern geschieht beim Aufruf einer überladenen Methode folgendes: Der Java-Interpreter überprüft, ob es für den angegebenen Methodennamen eine Imple-

mentierung gibt, die genau die angegebenen Objekt-Typen (oder deren Basis-Typen) in der angegebenen Reihenfolge beinhaltet. Sollte er eine derartige Übereinstimmung finden, wird die entsprechende Implementierung der Methode ausgeführt.

Achtung

Bei der Überladung von Methoden spielt der Name des jeweiligen Parameters für den Interpreter keinerlei Rolle. Relevant sind lediglich die Typen der Parameter und deren Reihenfolge. Eine Methode

```
void doSomething(String myParam) {}
```

ist also für den Interpreter (und auch schon für den Compiler) äquivalent zu einer Methode

```
void doSomething(String myOtherParam) {}
```

In der Regel sollten Sie bereits beim Kompilieren des Quelltextes zu Java-Bytecode eine entsprechende Fehlermeldung erhalten.

Übertragen wir dieses Prinzip auf die `SimpleCar`-Klasse, bietet sich eine Stelle besonders für die Überladung an: Der Konstruktor. Schließlich kann es ja sein, daß wir Werte bereits beim Erzeugen der Klasse zuweisen und uns somit mehrere Zuweisungen sparen wollten, die wir sonst mit Hilfe der Setter-Methoden durchführen müßten.

Sinnvoll erscheint es, alle oder einige Werte bereits beim Erzeugen der Klasse zuweisen zu lassen:

Listing 3.7
Mehrfach überladene Konstruktoren

```
public class SimpleCar {

   private String manufacturer = "Unknown";
   private String color = "White";
   private double miles = 0;

   private final double KMTOMILE = 0.62137;
   private final double MILETOKM = 1.60934;

   public SimpleCar() {
      this.setManufacturer("Porsche");
      this.setColor("Blue");
   }

   public SimpleCar(String manufacturer) {
      this();
      this.setManufacturer(manufacturer);
   }

   public SimpleCar(double miles) {
      this();
      this.setMiles(miles);
   }
```

Listing 3.7 *(Forts.)* Mehrfach überladene Konstruktoren

```
    public SimpleCar(String manufacturer, double miles) {
       this(manufacturer);
       this.setMiles(miles);
    }

    public SimpleCar(String manufacturer, String color) {
       this(manufacturer);
       this.setColor(color);
    }

    public SimpleCar(String manufacturer, String color,
       double miles) {
       this(manufacturer, color);
       this.setMiles(miles);
    }

    ...
}
```

Insgesamt verfügt die `SimpleCar`-Klasse nun über fünf Konstruktoren, die die Übergabe unterschiedlicher Parameter-Kombinationen zulassen:

- Standard-Konstruktor ohne Parameter
- Konstruktor, der die Anzahl der bereits gefahrenen Meilen entgegen nimmt (double)
- Konstruktor, der Hersteller und Meilen entgegen nimmt (String, double)
- Konstruktor, der Hersteller und Farbe entgegen nimmt (String, String)
- Konstruktor, der Hersteller, Farbe und Meilen entgegen nimmt (String, String, double)

Wenn wir nunmehr eine neue Instanz der `SimpleCar`-Klasse erzeugen, können wir nunmehr dies gleich unter Angabe der uns zur Verfügung stehenden Informationen machen. Sehen wir uns an, wie dies am Beispiel der `SimpleCarInvoker`-Klasse gelöst werden kann:

Listing 3.8 Aufruf eines überladenen Konstruktors

```
public class SimpleCarInvoker
{

   public static void main(String[] args) {
      SimpleCar car = new SimpleCar("BMW", "Blue", 1000);

      System.out.println(String.format(
         "You are driving a %s with %s color which " +
         "has the following status: %s",
             car.getManufacturer(),
             car.getColor(),
             car.carStatus()));
   }
}
```

In unserem Beispiel haben wir den Konstruktor mit drei Parametern verwendet und die Werte für Automarke, Farbe und Meilenstand gesetzt.

Achtung

Beachten Sie, dass Sie sich an die von der Klasse vorgegebene Reihenfolge der Parameter halten müssen – es ist für Compiler und Interpreter nicht ersichtlich, ob die erste Zeichenkette eine Automarke, eine Farbe oder eine Email-Adresse ist.

3.8.1 Konstruktor-Kaskadierung

Sehen wir uns noch einmal einen der überladenen Konstruktoren genauer an:

Listing 3.9
Konstrukor-Kaskadierung

```
public SimpleCar(double miles) {
   this();
   this.setMiles(miles);
}
```

Die Anweisung `this();` in der ersten Zeile der Methode ist interessant: Sie ruft direkt den Standard-Konstruktor der Klasse auf:

Listing 3.10
Standard-Konstruktor

```
public SimpleCar() {
   this.setManufacturer("Porsche");
   this.setColor("Blue");
}
```

Ändern wir nun in der Klasse `SimpleCarInvoker` so ab, das statt Marke, Farbe und Meilen nur die Meilen-Anzahl beim Instanzieren einer neuen `SimpleCar`-Klasse übergeben werden:

Listing 3.11
Übergabe von nur einem Parameter beim Instanzieren

```
public class SimpleCarInvoker
{

   public static void main(String[] args) {
      SimpleCar car = new SimpleCar(10000);

      System.out.println(String.format(
         "You are driving a %s with %s color which " +
         "has the following status: %s",
             car.getManufacturer(),
             car.getColor(),
             car.carStatus()));
   }
}
```

Wenn Sie das Beispiel kompilieren und ausführen, werden Sie diese Ausgabe erhalten:

You are driving a Porsche with Blue color which has the following status: nearly new

Wir besitzen also nun einen blauen Porsche mit 10.000 Meilen Gesamtfahrleistung. Sie sehen, das Konstruktor-Kaskadierung sehr sinnvoll sein kann, um bestimmte Standard-Werte in jedem Fall zu setzen. Dieses Ergebnis ist zwar vergleichbar mit dem Zuweisen von Standard-Werten zu den internen Variablen, dennoch aber um einiges flexibler, wie Sie im Verlauf dieses Kapitels noch feststellen werden.

3.8.2 Variable Argumente

Variable Argumente sind eine Neuerung, die bei Java 5 eingeführt worden ist. Eine derartige Funktionalität existierte nicht direkt in vorgehenden Versionen.

Eine Methode mit variablen Argumenten (zu der übrigens auch ein Konstruktor zählen kann), hat folgenden syntaktischen Aufbau:

```
<Rückgabetyp> <Name>(<Parametertyp> ... <Listenname>) { ... }
```

Wesentlich sind die drei Punkte zwischen dem Parametertyp und dem Namen der Liste, die die zugewiesenen Parameter aufnehmen soll.

Daraus, das hier von Liste – oder genauer von Array – die Rede ist, ergibt sich schon, dass die übergebenen Parameter durchlaufen werden müssen, da sie nicht als einzelne Variablen, sondern als Elemente der variablen Argumentsliste vorliegen.

Sehen wir uns an, wie dies in der Praxis umgesetzt werden kann. Nehmen wir dazu an, wir wollten eine Methode `define()` innerhalb der `SimpleCar`-Klasse implementieren:

Listing 3.12 Methode mit variablen Argumenten

```
/**
 * Sets some or all values for the class using variable arguments
 * @param args Arguments. Note: First String-argument is threatet
 * as Manufacturer, second one as Color. Every numeric argument is
 * treatet as mileage
 */
public void define(Object ... args) {
   boolean didSetManufacturer = false;

   for(int i=0; i<args.length; i++) {
      if(args[i] instanceof String) {
         String value = (String)args[i];
         if(!didSetManufacturer) {
            if(null != value && value.length() > 0) {
               this.setManufacturer(value);
            }
            didSetManufacturer = true;
         } else {
            if(null != value && value.length() > 0) {
               this.setColor(value);
            }
         }
      } else if(args[i] instanceof Number) {
         Number num = (Number)args[i];
         this.setMiles(num.doubleValue());
      }
   }
}
```

Gleich der Kopf der Methode zeigt an, dass wir eine variable Argumentliste verwenden werden. Der Nutzer kann hier keinen, einen oder beliebig viele Parameter übergeben. Im Methoden-Rumpf durchlaufen wir das Argument-Array.

Bei jedem Durchlauf wird überprüft, ob das aktuell zu behandelnde Argument vom Typ `String` oder `Number` ist – in ersterem Fall wird davon ausgegangen, das zunächst ein Hersteller gesetzt und danach die Farbe bestimmt werden soll, in letzterem Fall wird die übergebene Zahl als Meilen-Angabe interpretiert. Dies geschieht mit allen übergebenen Parametern.

Wenn wir nun die Methode innerhalb der `SimpleCarInvoker`-Klasse ansprechen wollten, könnten wir folgenden Code verwenden:

```
SimpleCar car = new SimpleCar();
car.define("Chrysler", "Yellow", 26483);
```

Listing 3.13
Aufruf der Methode define mit einer variablen Argumentliste

Wenn Sie das Beispiel kompilieren und ausführen, werden Sie diese Ausgabe erhalten:

You are driving a Chrysler with Yellow color which has the following status: used car

Nehmen wir nun, sie wüßten nicht, welche Farbe das durch die `SimpleCar`-Klasse definierte Auto haben soll. Dann würden Sie die Methode `define()` möglicherweise so aufrufen:

```
car.define("Chrysler", 26483);
```

Oder Sie drehen die Reihenfolge der Argumente einfach um:

```
car.define(26483, "Chrysler");
```

Die Zuweisung ist in beiden Fällen gültig und das Ergebnis in beiden Fällen gleich – Sie fahren einen blauen Chrysler, der irgendwie gebraucht wirkt:

You are driving a Chrysler with Blue color which has the following status: used car

Die Schlußfolgerung daraus muss lauten, daß Sie bei Verwendung von variablen Argumentlisten vorsichtig sein müssen. Sie sollten jeden einzelnen Parameter genau überprüfen und erst nach dieser Prüfung verwenden. Wenn Sie tatsächlich so kontrolliert vorgehen, sind Sie allerdings äußerst flexibel beim Umgang mit Parametern.

3.9 Zugriffs-Modifier

Zugriffs-Modifier regeln den Zugriff auf Klassen, Methoden oder Variablen. Mit ihrer Hilfe wird bestimmt, ob und falls ja, wie Elemente sichtbar oder unsichtbar für andere Elemente sind. Derartige Festlegungen bieten sich insbesondere im Hinblick auf private Variablen und Methoden von Klassen an – schließlich sollen diese vor aufrufenden oder abgeleiteten Elementen geschützt sein, so das niemand Elemente Ihrer internen Logik nutzen oder interne Variablen direkt modifizieren kann.

Im Sinne von objektorientierter Programmierung ist diese sogenannte *Kapselung* von Elementen ein wesentlicher Bestandteil des Programmier-Paradigmas. Durch sie wird sichergestellt, das nur die Elemente sichtbar sind, auf die explizit ein Zugriff gewünscht wird. Alle anderen Elemente werden verborgen und nur aus der betreffenden Klasse heraus angesprochen. Durch diese Trennung der Elemente in sichtbare und unsichtbare Elemente sorgen Sie für mehr Klarheit und eine bessere Wartbarkeit Ihrer Lösungen.

Zugriffs-Modifier sind Schlüsselworte, die Sie Klassen-, Methoden- oder Variablen-Deklarationen hinzufügen können. Auf Ebene der Zugriffsrechte unterscheidet Java folgende vier Modifier: Standard-Modifier, `public`, `protected` und `private`.

3.9.1 Standard-Modifier

Der Standard-Modifier wird immer dann angewendet, wenn kein anderer Modifier angegebenen worden ist. Der Standard-Modifier erlaubt einen Zugriff auf das entsprechende Element aus der gleichen Klasse und anderen Klassen des gleichen Pakets heraus.

Um einen Standard-Zugriffsmodifier zu verwenden, schreiben Sie eine Klassen-, Methoden- oder Variablen-Deklaration ohne weitere Schlüsselworte:

```
String aString;
void setSomething(Object value) { ... }
class someClass { ... }
```

3.9.2 public

Der Zugriffs-Modifier `public` gibt den Zugriff auf ein Element frei. Aus allen anderen Klassen kann nun auf das Element ohne Einschränkungen zugegriffen werden.

Sinnvoll ist der Einsatz des Zugriffsmodifiers `public` in folgenden Szenarien:

- Sie möchten eine Variable, Methode oder Klasse öffentlich zugänglich machen
- Sie möchten eine öffentliche Konstante definieren
- Sie möchten Getter- und / oder Setter-Methoden für private Member bereitstellen
- Sie möchten Methoden wiederverwendbar gestalten und von anderen Klassen aus ansprechen

Um den Zugriffsmodifier `public` einzusetzen, schreiben Sie eine Klassen-, Methoden- oder Variablendeklaration, der Sie den Zugriffsmodifier voranstellen:

```
public String aPublicString;
public final double KMTOMILE = 0.62137;
public Object getSomeValue() { ... }
public class somePublicClass { ... }
```

3.9.3 protected

Wenn Sie den Zugriff auf Elemente nur für abgeleitete Klassen und auf Klassen des gleichen Pakets zulassen wollen, empfiehlt sich der Einsatz des Zugriffsmodifiers `protected`. Dies ist insbesondere bei Business-Methoden sinnvoll, die auch abgeleiteten Klassen zur Verfügung stehen sollen, aber nicht von fremden Klassen sichtbar sein dürfen.

Um Elemente als `protected` zu deklarieren, stellen Sie einfach den Modifier der entsprechenden Deklaration voran:

```
protected String someProtectedStuff;
protected void setSomeValue(Object someValue) { ... }
protected class someProtectedClass { ... }
```

3.9.4 private

Als `private` deklarierte Klassen, Methoden, Getter, Setter oder Variablen sind von außen und von abgeleiteten Klassen nicht erreichbar. Diese Elemente sind nur vom beherbergenden Element erreichbar. Eine als `private` deklarierte Methode ist also nur von der Klasse, in der sie sich befindet, erreich- und vor allem: sichtbar. So können Sie Ihre Business-Methoden, Instanz-Variablen oder Hilfsklassen vor neugierigen Blicken schützen und sicherstellen, das nur Mitglieder der jeweiligen Klasse auf das entsprechende Element zugreifen können.

Um Elemente als `private` zu deklarieren, stellen Sie einfach den Modifier der entsprechenden Deklaration voran. Innerhalb einer einfachen Klasse könnte dies beispielsweise so aussehen:

```
public class SomeClass {

   private class SomeHelperClass {
      public String doSomething(String item) {
         // ...
      }
   }

   private String somePrivateItem = "Startvalue";

   public String getSomePrivateItem {
      return this.somePrivateItem;
   }

   public void manipulateThePrivateItem() {
      SomeHelperClass helper = new SomeHelperClass();
      this.somePrivateItem =
         helper.doSomething(this.somePrivateItem);
   }

}
```

Bei diesem Beispiel können nur die Methoden `manipulateThePrivateItem()` und `getSomePrivateItem()` von außen erreicht werden. Ein direkter Zugriff auf `somePrivateItem()` oder gar die Manipulation der Variablen ist nicht möglich. Die Hilfsklasse `SomeHelperClass` ist ebenfalls nicht von außen erreichbar.

3.10 Statische Methoden und Variablen

Auf *statische Methoden* und *statische Variablen* kann ohne Erzeugen einer Klassen-Instanz zugegriffen werden. Diese Elemente „bewegen“ sich nicht innerhalb eines bestimmten Objekts, sondern sind unabhängig vom Objekten gültig und ohne Objekt-Instanzierung einsetzbar. Sie werden in der Regel verwendet, um beispielsweise Konstanten oder Hilfsmethoden, die keine Klassen-Instanz benötigen, zu erzeugen. In dem Falle dient die Klassenzugehörigkeit meist eher als Organisationsstruktur.

Statische Methoden werden als *Klassenmethoden*, statische Variablen als *Klassenvariablen* bezeichnet. Nicht-statische Methoden und -Variablen heißen dagegen *Objekt-* oder *Instanz-Methoden* bzw. *-Variablen*.

Die Deklaration von statischen Methoden und Variablen geschieht, indem Sie das Schlüsselwort *static* voranstellen. Dies kann auch in Verbindung mit anderen Zugriffsmodifiern geschehen:

```
public static int someStaticVariable;
private static void someStaticMethod() { ... }
```

Lassen Sie uns noch einen kurzen Blick auf ein Beispiel von statischen Elementen werfen. Wir erkennen so besser, was bei deren Verwendung geschieht und wie wir uns dieses Verhalten zu Nutze machen könnten:

Listing 3.14
Mit Hilfe von statischen Membern kann die Anzahl von Klassen-Instanzen gezählt werden

```
public class SampleClass {

   private static int instanceCount = 0;
   private int localCount = 0;

   SampleClass() {
      instanceCount++;
      localCount++;
   }

   public static int getInstanceCount() {
      return instanceCount;
   }

   public int getLocalCount() {
      return localCount;
   }
}
```

Die Klasse `SampleClass` besitzt eine Klassenvariable `instanceCount`. Beim Erstellen einer neuen Klassen-Instanz wird deren Wert um eins erhöht und kann anschließend mit Hilfe der statischen Getter-Methode `getInstanceCount()` abgerufen werden.

Bei der nicht als statisch deklarierten Instanz-Variablen `localCount` ist der Wert beim Abruf immer Eins, denn ihr initialer Wert ist Null und wird durch den Default-Konstruktor erhöht. Der jeweilige Inhalt von `localCount` läßt sich mit Hilfe der Getter-Methode `getLocalCount()` auslesen.

Die Klassenvariable `instanceCount` behält im Gegensatz zu `localCount` ihren Wert über die verschiedenen Instanzen der Klasse hinweg. Bei jeder neuen Instanzierung wird dieser Wert durch den Default-Konstruktor erhöht.

Der Beweis für diese Aussage kann mit Hilfe einer Methode, die immer neue Instanzen der `SampleClass`-Klasse erzeugt, leicht erbracht werden:

```
public class Invoker
{
   public static void main(String[] args) {
      System.out.println("--- START ---");
      System.out.println(String.format(
         "Current instance count: %d",
         SampleClass.getInstanceCount()));
```

```
      for(int i=0;i<10;i++) {
         System.out.print("Creating instance... ");

         SampleClass sc = new SampleClass();
         System.out.print(String.format(
            "Current instance count: %d",
            SampleClass.getInstanceCount()));
         System.out.println(String.format(
            ". Local count: %d",
            sc.getLocalCount()));
      }

      System.out.println("--- FINISHED ---");
   }
}
```

Hier geschieht nichts weiter, als das zehn neue Klassen-Instanzen der `SampleClass`-Klasse erzeugt werden. Nach der Erzeugung wird der in `instanceCount` enthaltene Zählerstand mit Hilfe der ebenfalls statischen Getter-Methode `getInstanceCount()` ausgegeben. Im Anschluß erfolgt die Ausgabe des lokalen Zählerstandes mit Hilfe von `getLocalCount()`.

Wenn Sie die Klassen kompilieren und per

```
java Invoker
```

aufrufen, werden Sie folgende Ausgabe erhalten:

--- START ---

Current instance count: 0

Creating instance... Current instance count: 1. Local count: 1

Creating instance... Current instance count: 2. Local count: 1

Creating instance... Current instance count: 3. Local count: 1

Creating instance... Current instance count: 4. Local count: 1

Creating instance... Current instance count: 5. Local count: 1

Creating instance... Current instance count: 6. Local count: 1

Creating instance... Current instance count: 7. Local count: 1

Creating instance... Current instance count: 8. Local count: 1

Creating instance... Current instance count: 9. Local count: 1

Creating instance... Current instance count: 10. Local count: 1

--- FINISHED ---

Sie sehen, statische Variablen und Methoden eignen sich sehr gut dafür, Werte auch über Instanz-Grenzen hinweg zwischenzuspeichern. Ebenfalls sehr sinnvoll ist ihr Einsatz in Verbindung mit dem Modifier `final`, wie im Weiteren gezeugt wird.

3.11 Finale Klassen, Methoden und Variablen

Der Zugriffsmodifier `final` sagt aus, dass das betreffende Element nicht mehr verändert werden dürfen. Dabei hat er je nach Einsatzgebiet unterschiedliche Bedeutungen:

- Auf Variablen angewendet, bezeichnet er eine unveränderliche Variable
- Auf Methoden angewendet, sorgt er dafür, daß von dieser Methode keine Überschreibungen mehr vorgenommen werden können
- Im Kontext einer Klassen-Deklaration besagt er, das von der entsprechenden Klasse nicht mehr abgeleitet werden kann

Der Einsatz des `final`-Schlüsselwortes ist selbsterklärend: Die Dinge, die er beschreibt, sind endgültig. Beispiele für den Einsatz des `final`-Modifiers könnten so aussehen:

```
public final String UNCHANGEABLE = "This cannot be changed!";
public final static int ATYPE = 1;
public final void getSomeValue() { ... }
public final class FinalClass { ... }
```

Beachten Sie bitte die Schreibweise bei finalen Variablen-Deklarationen: Die Konvention besagt, daß die entsprechenden Variablen-Namen komplett in Großbuchstaben geschrieben werden.

Übrigens bietet es sich in den meisten Fällen an, finale Variablen auch als statisch zu kennzeichnen – so wie dies bei der zweiten beispielhaften Variablen-Deklaration der Fall war. Dies bietet den Vorteil, das Sie die entsprechenden Variablen auch ohne eine explizite Klassen-Instanzierung einsetzen können, was Ihnen etwas Schreib-Arbeit erspart.

3.12 Vererbung

Vererbung – oder eigentlich treffender: *Ableitung* – ist eines der wesentlichsten Elemente objektorientierter Entwicklung.

Wenn Sie eine Klasse von einer anderen Klasse ableiten, dann definieren oder spezifizieren Sie Methoden oder Verhaltensweisen der neuen Klasse, die sich von der abgeleiteten Klasse unterscheiden. Klingt abstrakt? Vielleicht hilft Ihnen dieser Ansatz weiter: Sie haben bei konsequenter Anwendung von Vererbung die Möglichkeit, den Erstellungsaufwand bei neuen Klassen soweit zu minimieren, das im Idealfall die Implementierung weniger zusätzlicher Methoden oder eines neuen Konstruktors ausreichend sind – Sie sparen sich jede Menge Schreibarbeit, da die ererbten Methoden und Eigenschaften auch bei der abgeleiteten Klasse vorhanden sind.

Wußten Sie, das Sie schon mit Vererbung gearbeitet haben? Schon in dem Moment, als Sie ihre erste Klasse erzeugt haben, nahmen Sie eine Ableitung vor, denn alle Java-Klassen erben implizit von der Basis-Klasse `Object`.

Die Ableitung einer Klasse von einer anderen Klasse erfolgt unter Verwendung des Schlüsselwortes `extends` bei der Deklaration einer Klasse:

```
[<Zugriffsmodifier>] class <Name> extends <Typ> { ... }
```

Die Subklasse erbt nun alle nicht als `private` deklarierten Methoden, Klassen und Instanz-Variablen der Super-Klasse. Ein direkter Zugriff auf die Elemente der Super-Klasse, die als `private` deklariert worden sind, ist nicht möglich.

Nehmen wir an, wir hätten eine Klasse `Animal` definiert, die ein Tier repräsentieren würde. Diese Klasse könnte etwa folgenden Aufbau haben:

Ein Wort zur Sprach-Regelung: Die Klasse, von der abgeleitet wird, bezeichnet man in der Regel als *Basis-* oder *Super-Klasse*. Wir verwenden hier beide Begriffe synonym. Die abgeleitete Klasse wird gerne als *Sub-Klasse* bezeichnet. Daneben sind durchaus auch die Begriffe *Ober-* und *Unter-Klassen* üblich.

```
/**
 * Repräsentiert ein Tier
 */
public class Animal
{

   private String name = "Lion";
   private String helloMessage = "Huuuuar!";

   public Animal() {}

   public Animal(String name, String helloMessage) {
      this.setName(name);
      this.setHelloMessage(helloMessage);
   }

   public void setName(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }

   public void setHelloMessage(String helloMessage) {
      this.helloMessage = helloMessage;
   }

   public String getHelloMessage() {
      return helloMessage;
   }
}
```

Listing 3.15
Ein beliebiges Tier wird durch die Klasse Animal repräsentiert

Der Aufbau der Klasse ist bewußt recht einfach gehalten, schließlich handelt es sich bei der `Animal`-Klasse um eine Basis-Klasse, von der eine Ableitung erst vorgenommen werden soll. Also entscheiden wir uns für den kleinsten gemeinsamen Nenner aller möglichen Ableitungen und implementieren eine Klasse, die nur den Namen des Tiers und dessen Begrüßung für Fremde verwaltet.

Ableitung sind bei dieser Basis-Klasse nicht nur denkbar, sondern sogar nötig:

```
/**
 * Ein Hund
 */
public class Dog extends Animal
{
   public Dog() {
      this.setHelloMessage("Wuff!");
      this.setName("dog");
   }
}
```

Listing 3.16
Ableitung einer Klasse von der Basis-Klasse SimpleCar

Die Ableitung erfolgt mit Hilfe des `extends`-Schlüsselwortes, das angibt, von welcher Klasse geerbt werden soll:

```
<Zugriffsmodifier> class <Name> extends <Super-Klasse>
```

Achtung

Java beherrscht keine *Mehrfachvererbung*! Dies bedeutet nicht, dass von einer Super-Klasse nicht mehrfach abgeleitet werden kann, sondern, dass eine Sub-Klasse nicht von mehr als einer Super-Klasse erben darf. Dies ist ein echter Vorteil gegenüber anderen Programmiersprachen, denn somit entfällt ein Konstrukt, das praktisch nur zu Komplikationen geführt hat. Mehrfach-Vererbung wird bei Java in Form von *Interfaces* nachempfunden. Hierzu später mehr.

Die von `Animal` abgeleitete `Dog`-Klasse verfügt neben den ererbten Methoden nur über einen eigenen Konstruktor. Dieser setzt Namen und Willkommens-Begrüßung, so daß die Klasseninstanz nur erzeugt werden muss und dann ohne weitere Arbeit verwendet werden kann. Alle übrigen Methoden bleiben unangetastet.

3.12.1 Polymorphie im Einsatz

Sehen wir uns nun kurz noch an, wie eine neue Instanz der Klasse erzeugt werden kann:

Listing 3.17 Instanzierung der abgeleiteten Klasse

```
/**
 * Erzeugen einer neuen Animal-Instanz
 */
public class AnimalInvoker
{
   public static void main(String[] args) {
      Animal animal = new Dog();

      System.out.println(String.format(
         "You own a %s which welcomes strangers with "
         + "a loud and friendly %s",
         animal.getName(), animal.getHelloMessage()));
   }
}
```

Die Instanzierung der Klasse erfolgt wie gewohnt, mit dem einzigen Unterschied, das wir einer Variablen vom Typ `Animal` eine Instanz der von `Animal` abgeleiteten Klasse `Dog` zuweisen – womit wir uns eines der Wesensmerkmale der Polymorphie zu nutze machen, um eine Sub-Klasse nutzen zu können, denn diese behält alle Methoden und Argumente der Super-Klasse.

Der genaue Typ der eingesetzten Klasse ist uns zum Zeitpunkt des Schreibens der Klasse nicht entscheidend, denn so sind wir in der Lage, den Code wiederverwendbarer zu halten.

Sollten wir eine explizite Instanz der abgeleiteten Klasse benötigen, könnten wir nach einer kurzen Prüfung immer noch in den entsprechenden Typ konvertieren:

```
if(animal instanceof Dog) {
   Dog dog = (Dog)animal;
}
```

Wenn Sie die Klassen kompilieren und von der Kommandozeile aufrufen, sollten Sie über Ihr neues Haustier und dessen Willkommens-Gruß informiert werden:

You own a dog which welcomes strangers with a loud and friendly Wuff!

Verwenden wir nun statt einer `Dog`- eine `Cat`-Klasse:

Der Einsatz von polymorphen Mechanismen und die Zuweisung von Sub-Klassen zu Variablen vom Typ der Super-Klasse bewahrt uns davor, uns ständig den tatsächlichen Typ der Sub-Klasse kennen zu müssen. In Form des *Factory-Patterns* werden wir davon noch viel intensiver Gebrauch machen.

Listing 3.18
Auch eine Cat-Klasse kann von der Animal-Klasse abgeleitet sein

```
/**
 * Eine Katze
 */
public class Cat extends Animal
{
   private String catName = "Minka";

   public Cat() {
      this.setHelloMessage("Miau!");
      this.setName("cat");
   }

   public String getCatName() {
      return this.catName;
   }

   public void setCatName(String catNameToSet) {
      if(null != catNameToSet && catNameToSet.length() > 0) {
         this.catName = catNameToSet;
      }
   }

   public String getName() {
      return super.getName() + " called "
         + this.getCatName();
   }
}
```

Die `Cat`-Klasse führt eine neue Eigenschaft ein, die es erlaubt, den Rufnamen des Tieres zu setzen. Realisiert wird diese Eigenschaft über die Instanz-Variable `catName`. Der Zugriff findet mit Hilfe der Getter- und Setter-Methoden `getCatName()` und `setCatName()` statt.

Um den Namen des Tieres auch in die Ausgabe der `AnimalInvoker`-Klasse einzufügen, ohne deren Code ändern zu müssen, `überschreiben` wir stattdessen die von der Super-Klasse geerbte Methode `getName()`. Beachten Sie dabei auch, wie auf die Methode in der Super-Klasse zugegriffen wird:

```
return super.getName() + " called " + this.getCatName();
```

Der Zugriff geschieht mit Hilfe des Schlüsselworts `super`, das immer auf die Super-Klasse verweist. Wir setzen also die neue Rückgabe aus der alten Rückgabe und dem Namen unserer Katze zusammen.

Jetzt ändern wir den Code der `AnimalInvoker`-Klasse noch so ab, daß statt eines `Dog`- nunmehr ein `Cat`-Objekt instanziert wird:

Listing 3.19 Die geänderte AnimalInvoker-Klasse instanziert nunmehr ein Cat-Objekt

```
public class AnimalInvoker
{
   public static void main(String[] args) {
      Animal animal = new Cat();

      System.out.println(String.format(
         "You own a %s which welcomes strangers with "
         + "a loud and friendly %s",
         animal.getName(), animal.getHelloMessage()));
   }
}
```

Die Ausgabe bestätigt es: Aus dem namenlosen Hund wurde Minka, eine Katze:

You own a cat called Minka which welcomes strangers with a loud and friendly Miau!

Die für uns angenehmen Vorteile der Ableitung und des Einsatzes von Polymorphie sind also:

- Die Ersparnis von Schreibarbeit
- Die Wiederverwendbarkeit bereits geschriebenen Codes
- Die Rückwärts-Kompatibilität von abgeleiteten Klassen zur Super-Klasse
- Die Möglichkeit, mit Hilfe von `super` auf die erreichbaren Methoden, Eigenschaften, Variablen und Implementierungen der Super-Klasse zugreifen zu können

Leider ist der hier gezeigte Ansatz der Erzeugung Instanzen von Sub-Klassen von Animal nicht optimal, denn jedes Mal, wenn eine andere Sub-Klassen-Instanz erzeugt werden soll, muss AnimalInvoker geändert werden.

Wir benötigen daher eine Methode, die uns entweder eine `Animal`-, `Cat`- oder `Dog`-Instanz zurückgibt. An dieser Stelle kommt das *Factory-Entwurfsmuster* (*Factory-Pattern*) ins Spiel.

3.12.2 Factory-Prinzip

Stellen Sie sich die Funktionsweise einer *Factory* genau so vor, wie Sie sich eine Fabrik aus Sicht eines Unternehmers vorstellen würden: Die Fabrik kann grundsätzlich alles bauen – man muss ihr nur sagen, was benötigt wird. Das in der Realität hier noch einige unwesentliche Faktoren wie etwa Geld, Arbeiter, Qualifizierung, Rohstoff-Situation oder Ähnliches eine Rolle spielen könnte, ignorieren wir einfach, ebenso wie den Fakt, das man Tiere nicht einfach bauen kann.

An die Fabrik ergeht nunmehr der Auftrag, ein bestimmtes Tier zu bauen, und sie tut es. Wie genau dies vor sich geht, interessiert dabei die aufrufende Methode nicht. Die Fabrik gibt am Ende das fertige Tier zurück – und dieses kann nun dressiert, abgerichtet, oder liebkost werden.

Wenn wir ein derartiges *Factory-Pattern* umsetzen wollten, könnte dies für unser Beispiel etwa so aussehen:

Listing 3.20
Factory für das Erstellen von Animal-Instanzen oder abgeleiteten Klassen

```
/**
 * Factory für die Erstellung verschiedener Animal-Instanzen
 */
public class AnimalFactory
{
   public static final int UNKNOWN = 0;
   public static final int DOG = 1;
   public static final int CAT = 2;
   public static final int COW = 3;
   public static final int DUCK = 4;

   public static Animal createAnimal() {
      return createAnimal(UNKNOWN);
   }

   /**
    * Gibt eine Animal-Instanz oder eine Sub-Klasse zurück
    * @param type Typ des Tieres
    */
   public static Animal createAnimal(int type) {
      Animal result = null;

      switch(type) {
         case AnimalFactory.DOG:
            result = new Dog();
            break;
         case AnimalFactory.CAT:
            result = new Cat();
            ((Cat)result).setCatName("Sylvester");
            break;
         case AnimalFactory.COW:
            result = new Animal();
            result.setName("Cow");
            result.setHelloMessage("Muuuuuh!");
            break;
         case AnimalFactory.DUCK:
            result = new Animal();
            result.setName("Duck");
            result.setHelloMessage("Naknaknak");
            break;
         default:
            result = new Animal();
      }

      return result;
   }
}
```

Die AnimalFactory hat einen recht einfachen Aufbau bekommen – jedenfalls für eine Klasse, die das *Factory-Pattern* implementiert: Sie definiert zunächst einige Konstanten, die bestimmte Tierarten repräsentieren. Außerdem stellt sie die überladene Methode createAnimal() zur Verfügung. Bei Aufruf der Methode ohne Parameter wird die überladene Version der Methode mit dem Wert der konstanten Variablen UNKNOWN, der für ein unbekanntes Tier steht, aufgerufen.

Bei direktem Aufruf der Methode createAnimal() unter Verwendung eines ganzzahligen Wertes, der den Wert einer der definierten Konstanten repräsentiert, wird eine neue *Animal*-Instanz, die mit den für das entsprechende Tier vorgesehenen Werten belegt ist, zurückgegeben.

Sollte ein dem Wert der konstanten Variablen DOG oder CAT entsprechender Wert übergeben worden sein, erfolgt die Rückgabe einer entsprechend initalisierten Dog- oder Cat-Instanz.

Verwenden der Factory

Innerhalb einer aufrufenden Klasse gestaltet sich der Einsatz der AnimalFactory zum Erzeugen von Klassen-Instanzen fast noch einfacher als das Erzeugen von neuen Instanzen:

Listing 3.21
Erstellen einer Instanz der Animal-Klasse oder abgeleiteter Klassen per Factory

```
public class AnimalInvoker
{
   public static void main(String[] args) {
      int animalType =
         (int)(Math.random() * (AnimalFactory.DUCK + 1));
      Animal animal = AnimalFactory.createAnimal(animalType);

      System.out.println(String.format(
         "You own a %s which welcomes strangers with "
         + "a loud and friendly %s",
         animal.getName(), animal.getHelloMessage()));
   }
}
```

Am Anfang der Methode main() wird mit Hilfe von Math.random eine Zufallszahl erzeugt, deren Wert zwischen Null und einschließlich Vier liegt. Dies entspricht dem Wertebereich, der innerhalb der AnimalFactory-Klasse definiert worden ist.

Die erzeugte Zufallszahl wird als Typ des von der AnimalFactory zu erzeugenden Instanz-Objekts angenommen und deren Methode createAnimal() als Parameter übergeben. Die von der Klasse zurückgegebene Animal-Instanz wird dann für die gewohnte Ausgabe verwendet.

Da der Typ des Tieres zufällig bestimmt worden ist, können wir an dieser Stelle natürlich nicht voraussagen, welche Ausgabe bei einem Aufruf der Klasse erzeugt wird. Wiederholen Sie den Vorgang einfach mehrmals – Sie werden feststellen, dass die Ausgabe differiert, ohne daß wir den Quellcode unserer Applikation ändern mussten.

3.13 Abstrakte Klassen und Methoden

Mit Hilfe von *abstrakten Klassen* definieren Sie Basis-Funktionalitäten, die von ableitenden Klassen überschrieben werden können oder sollen.

Achtung

Abstrakte Klassen können nie direkt instanziert werden. Sie können aber beispielsweise als Rückgabe einer Methode – etwa in Form einer *Factory* – auftreten. Ein Konstrukt wie etwa

```
abstract class AbstractClass {
   // ...
}
AbstractClass myClass = new AbstractClass();
```

kann es bei abstrakten Klassen nicht geben und würde auch folgerichtig beim Kompilieren eine Fehlermeldung provozieren.

Wenn Sie auf abstrakte Klasse oder Methoden stoßen oder selbst ein derartiges Element definieren wollen, dann begreifen Sie das Element als eine Art Platzhalter für von ihm abgeleitete Elemente, auch wenn Sie diese zu diesem Zeitpunkt nicht kennen und auch nicht kennen müssen.

Ein abstraktes Element wird immer durch den Zugriffsmodifier `abstract` definiert. Mögliche Einsatzbereiche sind auf Klassen- und Methoden-Ebene zu suchen:

```
public abstract class SomeClass { ... }
abstract void doSomething() { ... }
```

Der Einsatzbereich von abstrakten Klassen und -Methoden ist klar umrissen: Diese Elemente definieren Funktionalitäten und Verhaltensweisen, deren genaue Implementation den ableitenden Klassen überlassen bleibt oder deren Grundfunktionalität von ableitenden Klassen verwendet werden kann. Abstrakte Klassen stellen also meist auch Implementationen bereit, auf die später zurückgegriffen werden kann.

3.13.1 Deklaration einer abstrakten Klasse

Sehen wir uns nun an, wie wir eine abstrakte Klasse deklarieren können:

```
public abstract class AbstractClass {

   public int increment(int value) {
      return ++value;
   }
}
```

Listing 3.22
Abstrakte Klasse, die bereits Basis-Funktionalitäten bereit stellt

Die abstrakte Klasse `AbstractClass` verfügt über eine bereits implementierte Methode namens `increment()`. Diese Methode erhöht den Wert der übergebenen Zahl und gibt das Ergebnis zurück. Durch die bereits implementierte

Methode ist diese Klasse nicht nur als Mutter aller abgeleiteten Klassen zu verstehen, sondern stellt ebenfalls eine Basis-Funktionalität bereit, auf die von Ableitungen zurückgegriffen werden kann.

3.13.2 Implementieren der abstrakten Klasse

Da eine abstrakte Klasse nicht direkt instanziert werden kann, wird sie mehr in Form eines Platzhalters für ihre Ableitungen zum Einsatz kommen. Die einfachste mögliche Form der Implementierung einer Ableitung der `AbstractClass`-Klasse könnte dann so aussehen:

Listing 3.23
Eine simple Implementierung der abstrakten Klasse

```
public class AbstractClassImpl extends AbstractClass { }
```

Die Ableitung von der Super-Klasse geschieht exakt so, wie es auch bei einer nicht abstrakten Klasse der Fall wäre: Mit Hilfe des Schlüsselwortes `extends`.

Wissend, das in der `AbstractClass`-Klasse bereits Basis-Funktionalitäten existieren, können wir nun recht einfach mit einer konkreten Implementierung arbeiten:

Listing 3.24
Einbinden einer Implementierung der AbstractClass-Klasse

```
public class AbstractClassInvoker {

   public static void main(String args[]) {
      AbstractClass impl = new AbstractClassImpl();
      int value = (int)(Math.random() * 1000);

      System.out.println("Incrementing value of: " + value);
      System.out.println("Result is: " + impl.increment(value));
   }
}
```

In unserem Code ist es uns egal, wie genau die Implementierung arbeitet – wir verwenden eisern den Typ der Basis- bzw. abstrakten Klasse. Tatsächlich ist der Variablen vom Typ `AbstractClass` natürlich eine Instanz der `AbstractClassImpl`-Klasse zugewiesen – und die von der Super-Klasse geerbte Methode `increment()` nimmt die zuvor generierte Zufallszahl entgegen und erhöht sie.

Die Ausgabe wird bei jedem Aufruf eine andere sein, könnte aber mit ein wenig Glück (oder wenn Sie es lange genug versuchen) auch so aussehen:

Incrementing value of: 820

Result is: 821

Überschreiben und Implementieren von Methoden einer abstrakten Klasse

Natürlich werden Sie deutlich häufiger die von der abstrakten Super-Klasse bereitgestellten oder definierten Methoden überschreiben beziehungsweise implementieren.

Dies könnte in diesem Beispiel beispielsweise so aussehen:

Listing 3.25
Die Klasse DoubleIncrementor überschreibt die Methode increment der Basis-Klasse

```
public class DoubleIncrementor extends AbstractClass {

   public int increment(int value) {
      int result =super.increment(value);
      return ++result;
   }

   public static void main(String args[]) {
      AbstractClass impl = new DoubleIncrementor();
      int value = (int)(Math.random() * 1000);

      System.out.println("Incrementing value of: " + value);
      System.out.println("Result is: " + impl.increment(value));
   }
}
```

Auch hier erfolgt die Ableitung von der Super-Klasse mit Hilfe des `extends`-Schlüsselwortes.

Diesmal implementiert die Klasse eine eigene `increment()`-Version, in der mit Hilfe des `super`-Schlüsselworts auf die nunmehr von außen nicht mehr erreichbare Methode `increment()` der Basis-Klasse zugegriffen wird.

Das diese Implementierung der `AbstractClass`-Klasse übergebenen Wert um Zwei erhöht, werden Sie leicht feststellen können, denn die Ausgabe könnte so aussehen:

Incrementing value of: 263

Result is: 265

3.13.3 Definition abstrakter Methoden

Abstrakte Methoden funktionieren nach dem selben Prinzip wie abstrakte Klassen: Ableitende Klassen sind gezwungen, die entsprechende Methode zu implementieren. Die Syntax zur Deklaration einer abstrakten Methode sieht dabei etwas anders aus, als Sie dies von normalen Methoden gewöhnt sind:

```
<Modifier> abstract <Rückgabetyp> <Name>(<Parameter>);
```

Achten Sie speziell darauf, dass sowohl der Zugriffsmodifier `abstract` als auch das Semikolon am Ende der Deklaration obligatorisch sind.

Achtung

Beachten Sie, das abstrakte Methoden nur innerhalb von Klassen erlaubt sind, die ebenfalls als `abstract` deklariert sind. Definieren Sie eine Methode als `abstract`, müssen Sie folglich auch die Klasse als `abstract` deklarieren.

Wenn Sie abstrakte Methoden deklarieren, können Sie für die entsprechende Methode keinen Rumpf – Sie sind also nicht in der Lage, eine Basisfunktionalität zu definieren.

Um dies zu verdeutlichen, hier ein kurzes Beispiel:

Listing 3.26
Innerhalb der Klasse wird eine abstrakte Methode deklariert

```
public abstract class AbstractMethods {

   public final int increment(int value) {
      return ++value;
   }

   public abstract int decrement(int value);
}
```

Innerhalb der abstrakten Klasse `AbstractMethods` deklarieren wir neben der bereits fertig umgesetzten Methode `increment()` (die wir hier übrigens als `final` kennnzeichnen, damit sie von ableitenden Klassen nicht überschrieben werden kann) eine von ableitenden Klassen zu implementierende Methode `decrement()`.

Eine ableitende Klasse sollte dann diesen Aufbau haben:

Listing 3.27
Ableitung von der Abstract-Methods-Klasse mit implementierter decrement-Methode

```
/**
 * Implementierung der AbstractMethods-Klasse
 */
public class AbstractMethodsImpl extends AbstractMethods {

   public int decrement(int value) {
      return --value;
   }
}
```

Die Ableitung von der Super-Klasse findet unter Verwendung des `extends`-Schlüsselworts statt. Die eigentliche Implementierung der `decrement()`-Methode ist nicht wirklich umfangreich ausgefallen, verringert sie doch lediglich den übergebenen Wert, bevor sie ihn zurückgibt, und verzichtet dabei auch noch auf sämtliche Prüfungen.

Zuletzt benötigen wir noch eine Klasse mit einer statischen `main()`-Methode, die von der Kommandozeile aus erreichbar ist:

Listing 3.28
Die AbstractMethods-Invoker-Klasse arbeitet mit der Implementierung der AbstractMethods-Klasse

```
public class AbstractMethodsInvoker {

   public static void main(String args[]) {
      AbstractMethods impl = new AbstractMethodsImpl();
      int value = (int)(Math.random() * 1000);

      System.out.println("Incrementing value of: " + value);
      System.out.println("Result is: " + impl.increment(value));
      System.out.println("Result of decrement is: " +
         impl.decrement(value));
   }
}
```

Nach der Erzeugung einer Zufallszahl im Bereich von 0 bis 1000 wird diese durch die Methoden `increment()` und `decrement()` erhöht und höchstwahrscheinlich auch verringert. Genau läßt sich dies leider nicht im Voraus sagen, da die tatsächliche Implementierung einer abstrakten Methode der ableitenden Klasse überlassen bleibt und von der Super-Klasse nicht zu beeinflussen ist.

Wenn Sie das Beispiel kompilieren und es aufrufen, werden Sie eine Ausgabe erhalten, die ähnlich dieser sein dürfte:

Incrementing value of: 828

Result is: 829

Result of decrement is: 827

3.14 Interfaces

Schnittstellen oder *Interfaces* (beide Begriffe sind synonym zu verwenden) definieren ebenso wie abstrakte Elemente Verhaltensweisen und Methoden, implementieren sie aber im Gegensatz zu diesen niemals selbst. Eine Klasse kann zwar nur von genau einer anderen Klasse erben, dafür aber beliebig viele Interfaces implementieren.

Interfaces sind, da sie reine Deklarationen darstellen, noch abstrakter als abstrakte Klassen – sie stellen quasi Baupläne dar, und lassen dem Baumeister freie Hand, wie er sie umsetzt. Die Verwendung von Interfaces erfordert also – aufgrund der rein deklarativen Beschaffenheit – mehr Arbeit vom Entwickler, sorgt aber gleichzeitig für deutlich mehr Flexibilität.

Verwenden Sie Interfaces immer dann, wenn Sie angeben wollen, welche Methoden oder statische Variablen von implementierenden Klassen bereitgestellt werden sollen und Sie davon ausgehen, das eine derartige Definition zusammen mit anderen Definitionen eingesetzt werden wird oder wenn Sie eine bestimmte Mindest-Struktur vorgeben wollen, ohne daß Sie sich für die konkrete Implementierung interessieren.

Setzen Sie dagegen auf abstrakte Klassen, wenn Sie eine Implementierung vorgeben wollen, die von abgeleiteten Klassen überschrieben werden kann, aber nicht muss. Sie können so Basis-Funktionalitäten definieren, auf die ableitende Klasse zurückgreifen kann.

3.14.1 Ein Interface definieren

Die Definition eines Interfaces hat syntaktisch große Ähnlichkeit mit einer Klassendefinition:

```
interface <Name> { ... }
```

Niemand schreibt vor, daß ein Interface tatsächlich auch Variablen und Methoden beinhalten muss. Das einfachste mögliche Interface könnte also so aussehen:

```
interface Nameable {}
```

Tatsächlich gibt es Interfaces, die einfach nur dafür gedacht sind, um bestimmte Eigenschaften zu dokumentieren – mit Hilfe des `instanceof`-Vergleichsoperators könnte das Vorhandensein der entsprechenden Eigenschaft ermittelt werden – und deshalb einen leeren Rumpf haben.

In der Praxis werden Sie aber in der überwiegenden Anzahl der Fälle innerhalb eines Interfaces Methoden und / oder Variablen deklarieren. Dabei sollten Sie folgende zwei Einschränkungen berücksichtigen:

- Methoden innerhalb einer Schnittstelle sind immer als `public abstract` deklariert
- Variablen sind immer als öffentliche Konstanten ausgeführt und besitzen den Zugriffmodifier `public static final`.

Andere Zugriffsmodifier – etwa `protected` oder `private` – können innerhalb von Schnittstellen nicht verwendet werden.

Umgesetzt in einem Interface könnte sich folgender Code ergeben:

Listing 3.29 Interface Nameable

```
interface Nameable {

   public abstract String getName();

   void setName(String name);
}
```

Unser Interface `Nameable` definiert, das ableitende Klassen zwei Methoden implementieren müssen: `getName()` und `setName()`. Dieses Getter-/Setter-Paar soll uns den Zugriff auf den Namen des jeweiligen Objekts erlauben – ohne das uns interessieren muss, ob und wo diese Information tatsächlich gehalten wird.

Beide Methodensignaturen sind hinsichtlich ihrer Zugriffsmodifier identisch – bei `getName()` wird explizit angegeben, das die Methode abstrakt und öffentlich ist, wogegen `setName()` diese Information implizit mitführt. Um dieses Verhalten müssen Sie sich nicht kümmern – das erledigt Java für Sie.

3.14.2 Implementieren eines Interfaces

Anders als bei der Ableitung von Klassen können Sie mehrere Interfaces in einem Objekt implementieren und somit eine einfache Form der Mehrfach-Vererbung simulieren. Zu diesem Zweck ergänzen Sie dessen Deklaration um das Schlüsselwort `implements`:

```
[<Zugriffsmodifier>] class <Name> [extends <Klasse>] implements
<Interfaces> { ... }
```

Achtung

Das Erben von einer Super-Klasse und das Implementieren von Interfaces geschieht oftmals parallel. Beachten Sie deshalb, das die Angabe, welche Interfaces implementiert werden, stets nach dem `extends`-Schlüsselwort für das Erweitern einer Klasse kommt.

Innerhalb der so definierten Klasse müssen nun die vom Interface erwarteten Methoden implementiert werden. Übertragen auf das Beispiel-Interface `Nameable` könnte dies dann so aussehen:

Listing 3.30 Implementierung des Nameable-Interfaces

```
public class NameableImpl implements Nameable {

   private String name;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
}
```

Im Kopf der Klasse erfolgt mit Hilfe des `implements`-Schlüsselwortes die Angabe, welche Interfaces implementiert werden sollen. In unserem Fall ist dies nur `Nameable` – mehrere Implementierungen trennen Sie einfach durch ein Komma zwischen den Namen der Interfaces.

Die Methoden `getName()` und `setName()` fungieren in unserer Implementierung als Getter und Setter für eine lokale Variable `name`.

Um nun so generisch wie möglich auf Methoden und Variablen einer Interface-Implementierung zugreifen zu können, empfiehlt es sich, so weit als möglich mit dem Schnittstellen-Typ zu arbeiten:

```
Nameable impl = new NameableImpl();
impl.setName("Paula");
```

Ein Wort noch zur Vererbung: Wenn die Super-Klasse, von der Sie erben lassen, ein Interface implementiert, vererbt sich dies auch auf die abgeleitete Klasse. Sie müssen die entsprechende Angabe mit Hilfe des `implements`-Schlüsselwortes also nicht wiederholen.

Der Vorteil dieser Vorgehensweise besteht darin, dass die eigentliche Implementierungsklasse von einem beliebigen Typ sein kann, was unsere Applikation nicht interessieren muss. Wir arbeiten mit dem Typ, den wir kennen und den wir auch erwarten: *Nameable*.

Die Prüfung darauf, ob ein Objekt unser Interface implementiert, erfolgt mit Hilfe des `instanceof`-Vergleichsoperators:

```
if(object instanceof Nameable) {
   Nameable impl = (Nameable)object;
   impl.setName("Paula");
}
```

3.14.3 Mehrere Interfaces implementieren

Wie bereits mehrfach erwähnt, können Sie in einer Klasse mehrere Interfaces implementieren. Damit haben Sie die Möglichkeit, die verschiedenen Anforderungen der unterschiedlichen Interfaces innerhalb einer Klasse zu implementieren, was im Sinne eines verringerten Schreib- und Entwicklungsaufwandes positiv einzuschätzen ist.

Lassen Sie uns als Beispiel ein weiteres Interface neben `Nameable` implementieren: `Comparable`. Das `Comparable`-Interface ist keine neue Erfindung des Autors, sondern Bestandteil der Java-Sprachspezifikation. Es erlaubt eine von den Elementen definierte Sortierung.

Erweitern wir also die Klasse `NameableImpl`, so das wir mehrere Instanzen sortieren können:

Listing 3.31
Erweiterung der NameableImpl-Klasse um das Interface Comparable

```
public class NameableImpl implements Nameable, Comparable {

   private String name;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
```

Listing 3.31 *(Forts.)* Erweiterung der NameableImpl-Klasse um das Interface Comparable

```
    public int compareTo(Object o) {
        if(o instanceof Nameable) {
            Nameable item = (Nameable)o;
            return this.getName().compareTo(item.getName());
        }

        return 0;
    }
}
```

Im Kopf der Klasse geben wir mit Hilfe des `implements`-Schlüsselwortes an, welche Interfaces wir implementieren wollen. Mehrere Interfaces trennen wir dabei durch Kommata. Im Klassen-Rumpf erfolgt dann wie gewohnt die Implementierung der einzelnen Methoden.

Das Interface `Comparable` erwartet, dass wir eine Methode `compareTo()` bereitstellen, die die beiden Objekt-Instanzen miteinander vergleicht. Wir machen uns an dieser Stelle die Umsetzung recht einfach und verwenden die vom `String`-Objekt bereitgestellte Implementierung. Dies gilt allerdings nur für den Fall, dass das übergebene Objekt ebenfalls das Interface `Nameable` implementiert, was wir zuvor mit Hilfe des `instanceof`-Vergleichsoperators ermitteln.

Sollte das übergebene Objekt unser Interface nicht implementieren, geben wir Null zurück. Dadurch enthalten wir uns jeder Stimme und behaupten steif und fest, die beiden Objekte wären gleich.

Die drei möglichen Rückgaben der `compareTo()`-Methoden lauten übrigens:

- 0: Die Werte der zu vergleichenden Objekte sind gleich
- <0: Der Wert des Objekts ist kleiner als der Wert des als Parameter übergebenen Objekts
- >0: Der Wert des Objekts ist größer als der Wert des als Parameter übergebenen Objekts

Ein Vergleich könnte nun beispielsweise so stattfinden:

Listing 3.32 Vergleich mit Hilfe des Comparable-Interfaces

```
import java.util.ArrayList;

public class NameableInvoker {

    private static void compare(Object a, Object b) {
        if(a instanceof Comparable) {
            Comparable c = (Comparable)a;
            System.out.println(": " + c.compareTo(b));
        } else {
            System.out.println(": This is not possible. :-(");
        }
    }

    public static void main(String args[]) {
        Nameable object = new NameableImpl();
        object.setName("Paula");

        Nameable differentObject = new NameableImpl();
        differentObject.setName("Paul");
```

Listing 3.32 *(Forts.)*
Vergleich mit Hilfe des Comparable-Interfaces

```
      System.out.print("Let's compare " + object.getName()
         + " to " + differentObject.getName());
      compare(object, differentObject);

      System.out.print("And now: Let's compare an ArrayList to "
         + object.getName());
      compare(new ArrayList(), object);
   }
}
```

Die statische Methode `main()` ist der Einsprungpunkt, wenn die Klasse von der Kommandozeile aufgerufen wird. Hier erfolgt zunächst die Instanzierung von zwei `Nameable`-Instanzen, denen jeweils ein Name zugeordnet wird. Anschließend werde beide Instanzen miteinander verglichen und danach erfolgt ein Vergleich einer `ArrayList` mit einer der beiden `Nameable`-Instanzen.

Die Vergleiche werden von der privaten Hilfsmethode `compare()` durchgeführt. Diese Methode überprüft zunächst mit Hilfe des Vergleichsoperators `instanceof`, ob das zuerst übergebene Objekt das Interface `Comparable` implementiert. Sollte dies der Fall sein, wird es in eine `Comparable`-Instanz gecastet, die dann den Vergleich vornimmt. Anderenfalls wird ein entsprechender Hinweistext ausgegeben.

Achtung

Die vom Interface `Comparable` definierte Methode `compareTo()` kann eine `ClassCastException` werfen. Dies geschieht dann, wenn die beiden Instanzen, die miteinander verglichen werden sollen, nicht vom selben Typ sind. Sie können diese Ausnahme abfangen, indem Sie folgenden Code verwenden:

```
if(a instanceof Comparable) {
   Comparable c = (Comparable)a;
   try {
      System.out.println(": " + c.compareTo(b));
   } catch (ClassCastException cce) {
      System.out.println(": Comparison not possible. :-(");
   }
}
```

Sichern Sie also den Vergleich mit Hilfe eines `try-catch`-Blocks ab.

Wenn Sie die Klassen kompilieren, sollten Sie folgende Ausgabe erhalten:

Let's compare Paula to Paul: 1

And now: Let's compare an ArrayList to Paula: This is not possible. :-(

Diese Aussagen sind nicht schwer zu interpretieren:

- Der Wert von „Paula" ist größer als der Wert von „Paul"
- Die ArrayList-Klasse implementiert das Interface Comparable nicht und kann deshalb nicht mit einer Nameable-Instanz verglichen werden

3.14.4 Interface-Ableitung

Interfaces können voneinander abgeleitet werden. Und noch viel schöner: Ein Interface kann von mehreren Interfaces gleichzeitig erben – das ist klassische *Mehrfach-Vererbung*! Der Vorteil für den Entwickler: Interfaces können kaskadierend gebaut und kleinteilig gehalten werden. So verringert sich der Schreibaufwand und das Rad muss nicht jedes Mal neu erfunden werden.

Syntaktisch gesehen funktioniert eine Interface-Vererbung ähnlich wie eine Klassen-Vererbung über das Schlüsselwort `extends`:

```
interface <Name> extends <Interfaces> { ... }
```

Wenn wir also das Interface `Nameable` erweitern und eventuell auch noch als `Comparable` markieren wollten, könnte dies so aussehen:

Listing 3.33 Das Interface AdvancedNameable erbt von Nameable und Comparable

```
public interface AdvancedNameable extends Nameable, Comparable {

   void setLastname(String lastname);
   String getLastname();
}
```

Klassen, die dieses Interface implementieren, müssen über alle fünf in den drei Schnittstellen geforderten Methoden verfügen: `getName()` und `setName()` aus `Nameable`, `compareTo()` aus `Comparable`, sowie `getLastname()` und `setLastname()` aus `AdvancedNameable`. Sollte eine der Methoden nicht implementiert sein, ist auch die Schnittstelle nicht umgesetzt.

Die Klassen, die `AdvancedNameable` implementieren, können für andere Klassen durch das Konzept der Polymorphie in verschiedene Rollen schlüpfen: Sie können von ihrem eigentlichen Typ sein, sie können vom Typ `Comparable` sein, sie sind vom Typ `Nameable` und natürlich auch vom Typ `AdvancedNameable`. Je nach Bedarf kann auf den Typ mit Hilfe des `instanceof`-Operators geprüft und anschließend in den benötigten Typ gecastet werden.

Sehen wir uns an, wie dies in einer implementierenden Klasse aussehen kann:

Listing 3.34 Die Klasse AdvancedNameableImpl implementiert gleich drei Schnittstellen

```
public class AdvancedNameableImpl implements AdvancedNameable {

   private String lastname;
   private String firstname;

   public void setLastname(String lastname) {
      this.lastname = lastname;
   }

   public String getLastname() {
      return this.lastname;
   }

   public String getName() {
      return this.firstname;
   }
```

```
    public void setName(String name) {
        this.firstname = name;
    }

    public int compareTo(Object o) {
        if(o instanceof AdvancedNameable) {
            AdvancedNameable an = (AdvancedNameable)o;
            int result =
                this.getLastname().compareTo(
                    an.getLastname());

            if(0 != result) {
                return result;
            }
        }

        if(o instanceof Nameable) {
            Nameable n = (Nameable)o;
            return this.getName().compareTo(n.getName());
        }

        throw new ClassCastException("Unable to cast instance of "
            + o.getClass().getName() + " into instance of Nameable "
            + "or AdvancedNameable!");
    }
}
```

Listing 3.34 *(Forts.)*
Die Klasse AdvancedNameableImpl implementiert gleich drei Schnittstellen

Die Implementierung der Getter und Setter-Methoden, die von `Nameable` und `AdvancedNameable` definiert worden sind, ist trivial. Sie verweisen auf die Instanzvariablen `name` und `lastname`.

Interessanter ist sicherlich die Methode `compareTo()`, die vom Interface `Comparable` definiert worden ist: Hier erfolgt zunächst eine Prüfung darauf, ob das übergebene Objekt vom Typ `AdvancedNameable` ist. Sollte dies zutreffen, erfolgt der Vergleich der Objekte anhand der Nachnamen. Sollte dieser Vergleich ergeben, dass die Objekte ungleich sind, wird dies zurückgegeben und die Vergleichsoperation ist beendet.

Sollte sich allerdings herausstellen, dass die Objekte gleich sind (weil die Nachnamen gleich sind), wird exakt genauso fortgefahren, als ob das übergebene Objekt keine `AdvancedNameable`-Instanz wäre: Es erfolgt eine Prüfung darauf, ob das übergebene Objekt die Schnittstelle `Nameable` implementiert. Wenn dem so ist, dann werden die durch `getName()` abgerufenen Werte mit Hilfe der `compareTo()`-Methode der `String`-Klasse miteinander verglichen und das Ergebnis zurückgegeben.

Wenn das übergebene Objekt allerdings weder `AdvancedNameable`, noch `Nameable` implementieren sollte, wird davon ausgegangen, dass es nicht mit der Klassen-Instanz verglichen werden kann. Entsprechend wird eine `ClassCastException` geworfen, die von der aufrufenden Methode abgefangen werden sollte.

Nach soviel Theorie wieder eine kurze praktische Demonstration:

Listing 3.35
Vergleich mehrerer Objekte mit Hilfe der compareTo-Methode

```
public class AdvancedNameableInvoker {

   public static void main(String args[]) {
      AdvancedNameable object = new AdvancedNameableImpl();
      object.setName("Paula");
      object.setLastname("Meier");

      AdvancedNameable differentObject = new AdvancedNameableImpl();
      differentObject.setName("Paul");
      differentObject.setLastname("Mueller");

      Nameable thirdObject = new NameableImpl();
      thirdObject.setName("Paula");

      System.out.println(String.format(
         "Let's compare %s %s to %s %s: %d",
            object.getName(), object.getLastname(),
            differentObject.getName(),
            differentObject.getLastname(),
            object.compareTo(differentObject)));

      System.out.println(String.format(
         "And now: Let's compare %s %s to %s: %d",
            object.getName(), object.getLastname(),
            thirdObject.getName(),
            object.compareTo(thirdObject)));
   }
}
```

Innerhalb der statischen Methode `main()` werden drei Objektinstanzen erzeugt: `object` und `differentObject` sind vom Typ `AdvancedNameableImpl` und implementieren das Interface `AdvancedNameable`. Das Objekt `thirdObject` ist dagegen vom Typ `NameableImpl` und implementiert folglich die Schnittstelle `Nameable`.

Mit Hilfe der `compareTo()`-Methode der `AdvancedNameable`-Implementierung werden nun zunächst die beiden Instancen der `AdvancedNameableImpl`-Klasse verglichen. Anschließend wird eine `AdvancedNameable`-Instanz mit dem Objekt `thirdObject` verglichen, das die Schnittstelle `Nameable` implementiert.

Lassen Sie uns an dieser Stelle zwei Voraussagen treffen:

- „Paula Meier“ wird ein Ergebnis kleiner als Null einfahren, wenn man sie mit „Paula Mueller“ vergleicht. Der Grund dafür: Wenn zwei `AdvancedNameable`-Instanzen miteinander verglichen werden, wird zunächst der per `getLastname()` erreichbare Nachname verglichen – und da im Alphabet der Buchstabe „e“ vor „u“ kommt, wird der Wert von „Paula Meier“ offensichtlich kleiner als der Wert von „Paula Mueller“ sein.
- Das Ergebnis des Vergleichs von „Paula Meier“ zu „Paula“ wird Null (also „gleich“) sein, denn „Paula“ ist eine `Nameable`-Instanz. Dadurch kann nicht anhand der Nachnamen verglichen werden und „Paula Meier“ reduziert sich für die `compareTo()`-Methode auf „Paula“. Und „Paula“ ist gleich „Paula“...

Das Ergebnis wird die getroffenen Aussagen bestätigen:

Let's compare Paula Meier to Paula Mueller: -16

And now: Let's compare Paula Meier to Paula: 0

3.15 Lokale und anonyme Klassen

Lokale Klassen sind Klassen, die innerhalb einer anderen Klasse definiert worden sind. Diese Klassen besitzen in der Regel nur eine Bedeutung für die deklarierende Klasse, können aber bei Bedarf auch von anderen Klassen angesprochen werden.

Sie könnten also in einer Klasse `WrapperClass` problemlos eine in ihr enthaltene Klasse `ContainedClass` erstellen:

```
public class WrapperClass {
   public class ContainedClass {}
}
```

Diese Vorgehensweise bietet sich insbesondere für Hilfsklassen an, die von vielen Methoden der äußeren Klasse genutzt werden sollen, aber von anderen Klassen nicht unbedingt erreichbar sein müssen.

Interessant ist die Sichtbarkeit von Variablen gelöst: Aus der lokalen Klasse heraus sind Membervariablen der umschließenden Klasse sichtbar! Dies bedeutet, das auf eine in der äußeren Klasse definierte Variable ebenso zugegriffen werden kann, wie auf eine in der lokalen Klasse definierte Variable:

Listing 3.36
Verwenden lokaler Klassen

```
public class WrapperClass {

   String name;
   int numericValue;

   WrapperClass() {
      name = "Outer class";
      numericValue = 200;
   }

   public void process() {
      ContainedClass inner = new ContainedClass();
      System.out.println(inner.getOutput());
   }

   class ContainedClass {

      private String name;

      ContainedClass() {
         name = "Inner class";
      }
```

Listing 3.36 *(Forts.)*
Verwenden lokaler Klassen

```
        private String getOutput() {
            return "My name: " + name + ", Outer class name: " +
                WrapperClass.this.name + ", Number: " +
                    numericValue;
        }

    }

    public static void main(String[] args) {
        WrapperClass outer = new WrapperClass();
        outer.process();
    }
}
```

Innerhalb der Klasse `WrapperClass` wurde eine lokale Klasse `ContainedClass` definiert. `WrapperClass` deklariert zwei Membervariablen – `name` und `numericValue`. Die lokale Klasse `ContainedClass` deklariert eine eigene Membervariable `name`, die aus Sicht der lokalen Klasse die Variable `name` der äußeren Klasse verdeckt.

Im Konstruktor der beiden Klassen werden jeweils die Standard-Werte der Variablen gesetzt. Aus der statischen Methode `main()` heraus wird eine neue `WrapperClass`-Instanz erzeugt und deren Methode `process()` aufgerufen. Innerhalb dieser Methode wird eine neue `ContainedClass`-Instanz erzeugt und die Rückgabe von deren Methode `getOutput()` ausgegeben.

In `getOutput()` wird die Rückgabe erzeugt, wobei auf drei unterschiedliche Arten auf Variablen zugegriffen wird: Zunächst erfolgt der Zugriff auf die lokale Membervariable `name` der `ContainedClass`-Instanz. Der Zugriff auf die Membervariable `name` der äußeren Klasse erfolgt mit Hilfe eines seltsam anmutenden Statements:

```
", Outer class name: " + WrapperClass.this.name
```

Dieser Ausdruck, der immer der Form

```
<Klassenname>.this
```

entspricht, erlaubt den Zugriff auf die für die lokale Klasse eigentlich nicht sichtbare Variable `name` der äußeren Klasse – diese Variable ist nicht sichtbar, weil die lokale Klasse eine eigene Membervariable namens `name` besitzt. Durch `<Klassenname>.this` können wir nun dennoch auf die Membervariable der äußeren Klasse zugreifen.

Die dritte Art des Zugriffs auf eine Variable ist der Zugriff auf eine nicht verdeckte Variable der äußeren Klasse, wie beim Abrufen des Wertes von `numericValue` demonstriert:

```
", Number: " + numericValue
```

Dieser Zugriff erfolgt ohne Qualifizierung und ohne, das per `<Klassenname>.this` explizit auf die äußere Klasse zugegriffen werden muss – der Java-Interpreter erledigt die Auflösung des Namens für uns, indem er zunächst prüft, ob in der aktuellen Methode eine derartige Variable existiert. Sollte dies nicht

der Fall sein, erfolgt eine Prüfung auf Klassen-Ebene. Wird er dort auch nicht fündig, setzt er seine Überprüfung in der äußeren Klasse fort – und zwar solange, bis er eine entsprechende Variable findet oder eine Exception geworden werden muss, weil eine derartige Variable tatsächlich nicht existiert.

Wenn Sie dieses Beispiel ausführen, werden Sie diese Ausgabe erhalten:

My name: Inner class, Outer class name: Outer class, Number: 200

3.15.1 Lokale Klassen in Methoden

Lokale Klassen müssen nicht zwingend außerhalb einer Methode oder eines Blocks definiert werden. Es ist ebenso möglich (aber nicht gebräuchlich), eine Klasse in einem Block oder einer Methode zu definieren. Dadurch haben wir die Möglichkeit, nicht nur auf Instanzvariablen der äußeren Klasse zuzugreifen, sondern ebenfalls auf die innerhalb der Methode oder des Blocks deklarierten Variablen.

Wandeln wir das gerade gezeigte Beispiel ein wenig ab:

Listing 3.37
Definition einer lokalen Klasse innerhalb einer Methode

```
public class MethodClass {

   String name;

   MethodClass() {
      name = "Outer class";
   }

   public void process() {
      final int numericValue = 400;

      class ContainedClass {
         private String name;

         ContainedClass() {
            name = "Inner class";
         }

         private String getOutput() {
            return "My name: " + name +
               ", Outer class name: " +
               MethodClass.this.name +
               ", Number: " + numericValue;
         }

      }

      ContainedClass local = new ContainedClass();
      System.out.println(local.getOutput());
   }

   public static void main(String[] args) {
      MethodClass outer = new MethodClass();
      outer.process();
   }
}
```

In diesem Beispiel wird innerhalb der Methode `process()` der äußeren Klasse `MethodClass` eine lokale Variable `numericValue` deklariert. Diese muss zwingend als `final` gekennzeichnet sein, wenn von einer inneren Klasse auf sie zugegriffen werden soll!

Anschließend erfolgt die Deklaration der inneren Klasse, die innerhalb ihrer Methode `getOutput()` auf die beiden `name`-Membervariablen von äußerer- und innerer Klassen-Instanz zugreifen. Ebenfalls erfolgt der Zugriff auf die Variable `numericValue`, die nur innerhalb der Methode existiert, in der die Klasse deklariert worden ist.

Bei Aufruf dieses Beispiels werden Sie folgende Ausgabe erhalten:

My name: Inner class, Outer class name: Outer class, Number: 400

Wie bereits erwähnt ist der Einsatz dieser Art von Klasse recht unüblich. Der Grund dafür liegt in zwei Nachteilen:

- Die Klasse ist außerhalb der aktuellen Methode oder des aktuellen Blocks nicht sichtbar und kann somit nicht weiter verwendet werden
- Der Schreibaufwand der Deklaration einer derartigen Klasse läßt sich noch verringern

Statt Klassen in Methoden zu deklarieren, setzt man oftmals das Konstrukt der anonymen Klasse ein, dem wir uns nun widmen werden.

3.15.2 Anonyme Klassen

Anonyme Klassen entsprechen weitestgehend dem Konzept lokaler Klassen innerhalb von Methoden, verringern aber den Schreibaufwand bei deren Deklaration. Dies geschieht, indem Definition und Instanzierung der Klasse in einer Anweisung erfolgen. Aus Gründen der Übersichtlichkeit werden anonyme Klassen in der Regel in Form von Ableitungen anderer Klassen oder der Implementierung von Interfaces verwendet. Ihr wichtigster Einsatzzweck ist die Definition von Event-Listenern beim Einsatz von *Swing* und *JFC*.

Lassen Sie uns dies an einem Beispiel verdeutlichen, in dem wir eine anonyme Klasse verwenden, um eine andere Klasse zu erweitern:

Listing 3.38 Deklaration und Verwendung einer anonymen Klasse

```
public abstract class AnonymousClass {

   String name;
   int numericValue;

   AnonymousClass() {
      name = "Anonymous class";
      numericValue = 200;
   }

   public abstract void process();
```

```
   public static void main(String[] args) {
      new AnonymousClass() {
         public void process() {
            System.out.println(
               "Name: " + name + ", value: " + numericValue);
         }
      }.process();
   }
}
```

Listing 3.38 *(Forts.)*
Deklaration und Verwendung einer anonymen Klasse

Die hier deklarierte Klasse `AnonymousClass` ist als `abstract` gekennzeichnet und erfodert die Implementierung der Methode `process()`, die eine wie auch immer geartete Verarbeitung vornehmen soll.

Innerhalb ihrer statischen Methode `main()` (auf die wir trotz der Kennzeichnung der Klasse als `abstract` zugreifen können), erzeugen wir eine neue Instanz der `AnonymousClass`-Klasse und implementieren dabei – wie gefordert – deren Methode `process()`. Direkt nach der Deklaration der Klasse (nach der schließenden geschweiften Klammer) rufen wir die soeben implementierte Methode `process()` auf. Dies ist möglich, da wir die Klasse ja bereits per `new` instanziert haben – wir haben hier also zwei Dinge auf einmal: Deklaration und Instanzierung.

Grundsätzlich sieht die Syntax für die Deklaration einer anonymen Klasse also so aus:

```
new <Klassenname> {
   // Implementierung
}[.<Methodenaufruf>]
```

Eine derart erzeugte anonyme Klasse kann wie eine gewöhnliche Objektinstanz an Methoden als Parameter übergeben werden. Innerhalb der Methode ist dann kein Unterschied bei der Arbeit mit der anonymen Klasse mehr feststellbar.

Darüber, ob der Einsatz anonymer Klassen im Hinblick auf Lesbarkeit oder Übersichtlichkeit Vorteile bringt, kann sicherlich gestritten werden. Ein unbestreitbarer Vorteil anonymer Klassen kann allerdings nicht wegdiskutiert werden: Sie sind absolut flexibel einsetzbar, denn sie werden dort deklariert, wo sie auch verwendet werden. Außerdem gelten für sie die gleichen Vorteile wie für lokale Klassen auch: Sie können auf Instanzvariablen der äußeren Klasse oder des umschließenden Blocks zugreifen – und dies auch, wenn sie an andere Methoden als Parameter weitergegeben worden sind.

Aus Gründen der Lesbarkeit und der Wartbarkeit sollten Sie den Einsatz anonymer Klassen jedoch auf kleine Konstrukte mit wenigen Zeilen Code beschränken.

3.16 Aufzählungen

Aufzählungen erlauben es, bestimmte Konstanten zu gruppieren und unter ihrem Namen innerhalb von Applikationen zu verwenden. In älteren Java-Versionen geschah dies, indem öffentliche Konstanten als Integer-Werte deklariert worden sind:

Listing 3.39 Herkömmliche Deklaration von Aufzählungen

```
public class OldEnums {
   public static final int DAY_SUNDAY = 0;
   public static final int DAY_MONDAY = 1;
   public static final int DAY_THUESDAY = 2;
   public static final int DAY_WEDNESDAY = 3;
   public static final int DAY_THURSDAY = 4;
   public static final int DAY_FRIDAY = 5;
   public static final int DAY_SATURDAY = 6;

   public static final int FILE_OPEN = 0;
   public static final int FILE_CLOSED = 1;
   public static final int FILE_DELETED = 2;
}
```

Zwar ist dieses Verfahren einfach und robust (wir haben es selbst in diesem Kapitel schon angewendet), es weist aber neben einigen anderen Nachteilen ein wesentliches Manko auf: Klassische Aufzählung unter Verwendung von konstanten Variablen sind nicht eindeutig und typsicher.

Der durch eine konstante Variable repräsentierte Wert kann genauso durch eine andere Variable repräsentiert werden. So können zwei konstante Variablen `FILE_OPEN` und `DAY_SUNDAY` beide den Wert `0` repräsentieren:

```
public static final int FILE_OPEN = 0;
public static final int DAY_SUNDAY = 0;
```

Innerhalb von Applikationen kann dies durchaus ein Problem darstellen, denn eine Typsicherheit und damit eine eindeutige Eingrenzung der Anwendung kann nicht garantiert werden. Nehmen wir an, wir hätten eine Klasse `Day` geschrieben, deren Konstruktor ein Element der `OldEnums.DAY`-Konstanten entgegennehmen sollte:

Listing 3.40 Die Klasse Day verfügt über einen Konstruktor, der ein Element einer Aufzählung entgegennimmt

```
public class Day {

   private int day = OldEnums.DAY_SUNDAY;

   Day(int day) {
      this.day = day;
   }
}
```

Alle nun folgenden Zuweisungen wären gültig:

```
Day day = new Day(OldEnums.DAY_SUNDAY);
Day secondDay = new Day(OldEnums.FILE_OPEN);
Day thirdDay = new Day(0);
```

Das Problem dabei ist: Ohne wirkliche Eindeutigkeit der Parameter-Typen sinkt die Sicherheit der Applikation.

3.16.1 Eindeutig und typsicher

Das Schlüsselwort *enum*, das mit Java 5 eingeführt worden ist, erlaubt es nunmehr, Aufzählungen analog zu Klassen- und Interface-Deklarationen zu definieren:

```
[Zugriffsmodifier] enum <Name> {
   Element[, ...]
}
```

Achtung

Beachten Sie, dass sich die Analogie zu Klassen und Interfaces nicht nur auf die Syntax beschränkt. Auch für alleinstehende Aufzählungen gilt: Eine Aufzählung – eine Datei gleichen Namens.

Mit Hilfe des `enum`-Schlüsselwortes und vor allem der dahinter stehenden Technologie, erschlagen wir mehrere Fliegen mit einer Klappe:

- Aufzählungen werden eindeutig und typsicher
- Die einzelnen Elemente der Aufzählung sind selbsterklärend
- Jede Aufzählung fungiert als eindeutiger Namensraum für deren Elemente – mehrere Aufzählungen können also mehrmals gleichartige Elemente definieren, und diese können dennoch unterschieden werden

Setzen wir nun zur Verdeutlichung unsere `DAY`- und `FILE`-Konstanten in Form von Aufzählungen um:

Listing 3.41
Aufzählungen unter Java 1.5

```
public class Globals {

   public enum Days {
      Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
   }

   public enum FileStatus {
      OPEN, CLOSED, DELETED
   }
}
```

Die einzelnen Elemente der Aufzählungen werden durch Kommata voneinander getrennt. Java weist ihnen selbstständig einen numerischen Wert zu, der ihrer Position in der Liste entspricht. Der erste Wert der Aufzählung hat dabei immer den Wert 0, der folgende 1 und so weiter.

Ändern wir die Klasse `Day` so ab, das sie die `Days`-Aufzählung aus der `Globals`-Klasse verwendet:

Listing 3.42
Die Day-Klasse verwendet nun Elemente der Days-Auflistung aus der Globals-Klasse

```
public class Day {

   private Globals.Days currentDay = Globals.Days.Sunday;

   Day(Globals.Days day) {
      this.currentDay = day;
   }
}
```

Werfen Sie zunächst einen Blick auf die Deklaration der internen Variable `currentDay`: Diese ist nunmehr kein `Integer`-Wert mehr, sondern ein Element vom Typ der `Globals.Days`-Auflistung. Beachten Sie auch, wie ein Standard-

Wert zugewiesen worden ist: Der Zugriff auf ein Element der Auflistung erfolgt genau so, als ob wir auf eine statische Variable zugreifen würden.

Wenn Sie nun versuchen würden, eine Element einer anderen Aufzählung oder eine Konstante zuzuweisen (auch wenn diese den gleichen Wert hätte), wird dies nicht funktionieren, da Java zunächst auf den Parameter-Typ prüft – und der muss vom Typ `Globals.Days` sein. Damit ist sichergestellt, das wirklich nur Elemente vom gewünschten Typ zugewiesen werden können, wodurch eines der größten Mankos der bisherigen Vorgehensweise beseitigt wurde.

3.16.2 Enumerations-Member durchlaufen

Mit Hilfe der neuen `enum`-Auflistungen ist es nunmehr auch möglich, die einzelnen Elemente einer Aufzählung zu durchlaufen:

Listing 3.43
Durchlaufen einer Aufzählung und Ausgabe aller Member

```
public class FileStatusItems {

   public static void main(String args[]) {
      System.out.println("Members of the FileStatus enumeration:");
      System.out.println("=====================================");
      for(Globals.FileStatus status : Globals.FileStatus.values()) {
         System.out.println(
            String.format("%d = %s",
            status.ordinal(), status));
      }
   }
}
```

Eine einfache `for`-Schleife genügt, um alle Elemente einer Auflistung auszugeben.

Der Grund dafür ist, dass uns jede Auflistung eine Methode `values()` bereitstellt. Diese liefert ein Array aus den einzelnen Aufzählungselementen, das wir anschließend durchlaufen können. Auf Ebene eines einzelnen Elements liefert uns dessen Methode `ordinal()` den repräsentierten Zahlenwert – und den Namen für das Element stellt das Element selbst dar.

Die Ausgabe der `FileStatusItems`-Klasse sollte uns über die einzelnen Elemente der `FileStatus`-Auflistung und deren Werte informieren:

Members of the FileStatus enumeration:
=====================================
0 = OPEN
1 = CLOSED
2 = DELETED

3.17 Pakete

Pakete erlauben es, Klassen, Interfaces oder Auflistungen hierarchisch zu organisieren. Somit können wir allen Elementen einen eindeutigen Namensraum voranstellen. Der praktische Nutzen erschließt sich schnell, wenn Sie Klassen entweder einfach organisieren wollen oder dazu gezwungen sind, weil Sie plötzlich gleich- oder sehr ähnlich benannte Elemente haben.

Pakete haben beim Einsatz folgende Vorteile:

- Sie organisieren Ihre Elemente in einzelne Einheiten – analog zu Ordnern auf Ihrer Festplatte oder Unterverzeichnissen in Ihrem Web-Auftritt
- Sie lösen das Benennungs-Problem – Elemente müssen nur innerhalb ihres Namensraums eindeutig benannt sein. Sie können somit je ein Element namens MyClass innerhalb eines Paketes packageA und innerhalb eines Paketes packageB definiert haben
- Pakete geben nur auf Elemente Zugriff, die als public deklariert worden sind
- Pakete erlauben es Ihnen, sprechende Organisationsnamen zu verwenden und somit mehr Klarheit über den Zweck von Elementen zu schaffen

3.17.1 Pakete definieren

Jedes Paket bildet einen eigenen Namensraum. Die Definition des Pakets, zu dem ein Element gehört, erfolgt mit Hilfe des `package`-Schlüsselworts, das vor der eigentlichen Element-Deklaration stehen muss:

```
package <Paketbezeichner>;
```

Der eigentliche Bezeichner kann dabei eine beliebige Zeichenkette sein. Die Konvention sieht dabei ein Modell vor, das am Anfang aus einem umgekehrten Domainnamen und danach einem eindeutigen Paket-Bezeichner besteht. Die einzelnen Bestandteile sind dabei analog zu Domainnamen durch Punkte getrennt. Generell sollte die Reihenfolge der Bestandteile einer Bedeutungshierarchie folgen – von Allgemein zu Speziell, von einer Länderdomain über den Namen des Entwicklers oder der Firma bis hin zum Namen des Produktes.

Ein Beispiel: Angenommen, sie besäßen die Domain *ksamaschke.de* (ohne *www* am Anfang) und wollten eine Klasse für das „MasterClass"-Buch schreiben, dann könnten Sie folgende Paket-Deklaration verwenden:

```
package de.ksamaschke.masterclass;
public class SampleClass { ... }
```

Innerhalb Ihrer Verzeichnis-Struktur müssen Sie nun den Namensraum widerspiegeln. Dabei entspricht jedes Element des Namensraums einem Verzeichnis, das unterhalb des vorhergehenden Elements liegt:

Abbildung 3.2
Verzeichnis-Struktur bei Verwendung des Namensraums de.ksamaschke.masterclass

Eine Klasse `SampleClass`, die zum Paket `de.ksamaschke.masterclass` gehört, liegt also im Verzeichnis `/de/ksamaschke/masterclass`.

3.17.2 Pakete verwenden

Um Elemente, die in Paketen organisiert sind, verwenden zu können, müssen Sie sich auf den kompletten Paketnamen beziehen und diesen der Element-Deklaration voranstellen. Wollten Sie also aus einer Applikation auf die Klasse

SampleClass aus dem Paket de.ksamaschke.masterclass zugreifen und diese verwenden, könnte das so vor sich gehen:

Listing 3.44
Verwenden einer Klasse aus einem anderen Paket

```
public class SampleClassSample {

   public static void main(String[] args) {
      de.ksamaschke.masterclass.SampleClass sample =
         new de.ksamaschke.masterclass.SampleClass();
      System.out.println(sample.sayHello());
   }
}
```

Die Syntax lautet also beim Instanzieren:

```
<Paket>.<Typ> = new <Paket>.<Typ>();
```

Wichtig ist, daß das Paket immer komplett angegeben wird. Sie können keine Platzhalter oder Kürzel definieren – wenn ein Paket zwanzig Ebenen hat, müssen Sie alle zwanzig Ebenen angeben.

3.17.3 Das import-Statement

Weil die Angabe des kompletten Palets durchaus lästig werden kann, wenn Sie dies mehrfach vornehmen müssen, verfügt Java über das import-Schlüsselwort. Dieses erlaubt es, innerhalb einer Klasse alle zu verwendenden Hierarchien verfügbar zu machen, ohne immer wieder den kompletten Namensraum der Elemente anzugeben.

Das import-Statement steht immer am Anfang einer Datei, meist sogar noch vor der Angabe des Pakets:

```
import <Paket>.<Typ>;
```

Wollten Sie beispielsweise auf die Klasse SampleClass aus dem Paket de.ksamaschke.masterclass zugreifen, könnten Sie dies mit Hilfe des import-Statement so umsetzen:

Listing 3.45
Importieren eines Pakets durch das import-Statement

```
import de.ksamaschke.masterclass.SampleClass;

public class SampleClassSample2 {

   public static void main(String[] args) {
      SampleClass sample = new SampleClass();
      System.out.println(sample.sayHello());
   }
}
```

Nach dem Importieren des Namensraums des SampleClass-Typs können wir nunmehr also statt

```
de.ksamaschke.masterclass.SampleClass sample =
   new de.ksamaschke.masterclass.SampleClass();
```

viel kürzer und einfacher

```
SampleClass sample = new SampleClass();
```

schreiben.

3.17.4 Das import-Statement mit Platzhaltern

Beim Import von Elementen spart man sich bereits in der bekannten Form eine Menge Schreibarbeit. Ebenfalls möglich ist aber die Verwendung eines Platzhalters. Dies erlaubt es, statt eines konkreten Typs eine Namensraum-Ebene anzugeben, die mitsamt der direkt untergeordneten Elemente importiert werden soll. Die Syntax ändert sich nur marginal:

```
import <Paket>.*;
```

Achtung

Erwarten Sie bei Verwendung des `import`-Statements mit dem Platzhalter-Zeichen nicht, daß auch untergeordnete Namensräume importiert werden. Diese müssen Sie weiterhin per `import`-Anweisung einbinden. Ein Beispiel soll dies verdeutlichen: Angenommen, Sie wollten Elemente der Pakete `de.ksamaschke` und `de.ksamaschke.masterclass` einbinden, dann müssen Sie dies auch entsprechend angeben:

```
import de.ksamaschke.*;
import de.ksamaschke.masterclass.*;
```

Ebenso ist es nicht möglich, den Platzhalter an einer anderen Stelle als der Angabe des Typs zu verwenden. Dieses Statement würde also nicht funktionieren:

```
import de.*.masterclass;
```

Der Nutzen des `import`-Statements kommt dann zum Tragen, wenn Sie sehr viele Typen eines Pakets einbinden und verwenden wollen. Statt beispielsweise zehn `import`-Statements für zehn Typen zu verwenden, können Sie dies – falls die Elemente im gleichen Paket liegen – per `import`-Anweisung mit Platzhalter in nur einem Statement erledigen.

Achtung

Es ist aus Performance- und Lesbarkeitsgründen immer vorzuziehen, die zu verwendenden Typen direkt zu importieren. Der Platzhalter sollte immer nur dann zum Einsatz kommen, wo eine große Anzahl (cirka ab sieben oder acht Elementen) von `import`-Statements nötig wäre, um Elemente zu importieren.

3.17.5 Namensraum-Konflikte

Mit Hilfe von Paketen können gleich benannte Elemente sehr gut voneinander getrennt werden, denn die Namen der Pakete stellen einen eindeutigen Namensraum dar. Was aber geschieht, wenn Sie zwei gleichnamige Elemente per `import`-Statement einbinden und verwenden wollen? Angenommen, Sie hätten folgenden Code:

Listing 3.46 Namensraum-Konflikt

```
import de.masterclass.java.objects.SampleClass;
import de.masterclass.java.others.*;

public class NamingConflict {

   public static void main(String[] args) {
      SampleClass sample = new SampleClass();
      SampleClass otherSample = new SampleClass();
   }
}
```

Wie verhält sich der Java-Interpreter an dieser Stelle? Ganz einfach: Er tut überhaupt nichts! Sie werden die Klasse schon nicht kompiliert bekommen, da der jeweils zu verwendende Typ nicht eindeutig identifizierbar ist.

Wenn Sie also gleichnamige Klassen aus unterschiedlichen Namensräumen importieren und verwenden wollen, müssen Sie dem Compiler schon deutlich sagen, welche Klasse Sie instanzieren wollen:

Listing 3.47 Auflösung des Namensraums-Konflikts

```
import de.masterclass.java.objects.SampleClass;
import de.masterclass.java.others.*;

public class NamingConflict {

   public static void main(String[] args) {
      de.masterclass.java.objects.SampleClass sample =
         new de.masterclass.java.objects.SampleClass();
      de.masterclass.java.others.SampleClass otherSample =
         new de.masterclass.java.others.SampleClass();
   }
}
```

Nunmehr kann das Beispiel kompiliert und ausgeführt werden.

3.18 Zusammenfassung

Objektorientierung bedeutet für uns, dass wir gewaltig umdenken müssen. Gleichzeitig ist sie aber auch ein Garant dafür, dass wir Code schreiben können, der wiederverwendbar und sehr gut wartbar ist. Setzen wir Klassen richtig ein, können wir mit Hilfe von Vererbung Funktionalitäten so kapseln, dass wir sie nur einmal definieren müssen.

Auch Pakete und Aufzählungen sind sinnvolle Sprach-Elemente von Java, erlauben sie es doch, Inhalte zu strukturien und einfacher kontrollierbar zu halten. Und mit Hilfe von Interfaces dürfen wir Architekten spielen – wir geben nur vor, wie etwas auszusehen hat. Um die eigentliche Umsetzung dürfen sich dann Andere kümmern.

Aus eigener Erfahrung kann Ihnen der Autor bestätigen, dass objektorientiertes Denken nicht immer einfach ist – insbesondere der Einstieg fällt dem ein oder anderen besonders schwer. Sind Sie aber erst einmal in der Thematik drin, werden Sie schnell feststellen, dass Sie überhaupt nicht mehr anders arbeiten wollen – zu einfach ist es, objektorientiert zu arbeiten. Und zu gut sind die Ergebnisse, die mit diesem Ansatz erzielt werden können.

4

Listen

Listen erlauben es uns, Objekte und Informationen in strukturierter Form abzulegen. Im Laufe dieses Kapitels wollen wir uns kurz anschauen, welche Struktur für welchen Zweck am besten einsetzbar ist und worin Unterschiede oder Gemeinsamkeiten der verschiedenen Listen bestehen. Mit Hilfe von `Generics` können Listen erzeugt werden, die nur bestimmte Datentypen aufnehmen (homogene Liste). Somit kann sichergestellt werden, welche Art von Daten in einer Liste enthalten ist.

Wir beschränken uns hier auf die in der Praxis gebräuchlichsten Listen- und Aufzählungstypen `Array`, `ArrayList`, `HashMap` und `Properties`. Andere Listen werden am Ende des Kapitels kurz beschrieben.

4.1 Array

Arrays sind die einfachste Listenform, die Java anbietet. Ein Array ist eine Gruppe von Elementen desselben Typs. Arrays erlauben es, die Anzahl ihrer enthaltenen Elemente abzufragen und die einzelnen Elemente zu durchlaufen. Arrays erlauben es nicht, zur Laufzeit Elemente hinzuzufügen – die Anzahl der Elemente wird bereits bei der Deklaration festgelegt.

4.1.1 Deklaration

Die Deklaration eines Arrays erfolgt, als ob Sie eine normale Instanz-Variable des entsprechenden Typs deklarieren wollten. Der einzige Unterschied besteht darin, dass entweder Variablen-Typ oder Variablen-Name von eckigen Klammern gefolgt werden:

```
<Typ>[] <Name>;
<Typ> <Name>[];
```

Beide Arten der Deklaration sind äquivalent. Welche Art der Deklaration verwendet wird, hängt primär von den speziellen Vorlieben des Entwicklers oder

von firmeninternen Vorgaben ab. Wir werden hier im Folgenden mit der ersten Notationsart arbeiten und Arrays durch eckige Klammern nach dem Typ kennzeichnen.

Beispiele für die Deklaration von Arrays können sein:

```
String[] data;
Integer[] ints;
Object[] unknown;
```

Arrays sind Objekte. Als solche unterliegen sie den gleichen Restriktionen, wie normale Objekte auch – so können ihnen beispielsweise nur Objekte vom selben oder von abgeleiteten Typen zugewiesen werden.

4.1.2 Instanzierung

Ebenso wie einzelne Objekte müssen Arrays erst instanziert werden, bevor mit ihnen gearbeitet werden kann. Die Instanzierung erfolgt dabei unter Verwendung des Schlüsselworts *new* und unter Angabe der maximalen Anzahl an Elementen des Arrays:

```
<Typ>[] <Name> = new <Typ>[<Maximale Anzahl>];
```

Um also ein *String*-Array mit maximal fünf Elementen zu erstellen, würden Sie

```
String[] data = new String[5];
```

schreiben. Nun können die einzelnen Elemente des Arrays zugewiesen werden – dies geschieht nämlich nicht automatisch! Stattdessen sind alle Speicherplätze für die einzelnen Elemente mit Standard-Werten vorbelegt:

- 0 für numerische Arrays
- false für boolesche Arrays
- \0 (Null-terminierter String) für String-Arrays
- null für Objekt-Arrays

Bei einem *String*-Array wurden demnach allen Elementen das Token „\0“ zugewiesen. Für uns heißt das: Wir müssen Werte zuweisen, um mit dem Array arbeiten zu können.

4.1.3 Zuweisung und Abruf von Werten

Über den so genannten *Indexer* können Sie einem Array Werte zuweisen und diese auch wieder abrufen. Beachten Sie, dass *Indexer* immer nullbasiert zählen – das erste Element hat den Index *0*, das zweite Element den Index *1* und das letzte Element den Index *Länge-1*.

Der *Indexer* wird – ebenso wie die Größe des Arrays und die Angabe, dass es sich überhaupt um ein Array handelt – mit Hilfe von eckigen Klammern ausgedrückt:

```
<Variable>[<Index>] = <Wert>;
<Wert> = <Variable>[<Index>];
```

Um nun dem zweiten Element eines *String*-Arrays einen Wert zuzuweisen, müsste folgender Code verwendet werden:

```
String[] data = new String[5];
data[1] = "Two";
String myElement = data[1];
```

Beachten Sie, dass Zuweisung und Abruf exakt genauso erfolgen, als würden Sie mit einer einzelnen Instanz der `String`-Klasse arbeiten.

Der Zugriff auf Elemente eines Arrays kann auf zweierlei Arten durchgeführt werden:

Zählschleife

Die Zählschleife bedient sich der Eigenschaft `length` eines Arrays und zählt einen Zähler von Null bis zur *Array-Länge –1* hoch:

```
String[] data =
   new String[] { "One", "Two", "Three", "Four", "Five" };

for(int i=0; i<data.length; i++) {
   String current = data[i];
   System.out.println(
      String.format("Item #%d at Index #%d = \"%s\"",
         i+1, i, current));
}
```

Die Umsetzung der Zählschleife erfolgt als eine einfache *for*-Schleife. In deren Kopf erfolgt zunächst die Deklaration und Initialisierung der Variablen `i` als Zähler, gefolgt von Abbruch-Bedingung (`i` ist nicht mehr kleiner als die Anzahl der in `data` enthaltenen Elemente) und der Anweisung, den Zähler bei jedem Schleifen-Durchlauf zu erhöhen.

Im Rumpf der Schleife wird der Variablen `current` das durch `i` bezeichnete Element des Arrays zugewiesen und anschließend ausgegeben. Bei obigem Code ergibt sich folgende Ausgabe:

Item #1 at Index #0 = "One"

Item #2 at Index #1 = "Two"

Item #3 at Index #2 = "Three"

Item #4 at Index #3 = "Four"

Item #5 at Index #4 = "Five"

Wenn Sie alle Werte des Arrays zuweisen wollen, empfiehlt es sich, dies bei der Instanzierung mit zu erledigen. Zu diesem Zweck können wir ein Array deklarieren und weisen anschließend in geschweiften Klammern die enthaltenen Werte zu.

Die einzelnen Elemente werden dabei durch Kommata getrennt. Übertragen auf ein *String*-Array könnte dies so aussehen:

```
String[] data =
   new String[] { "One",
    "Two", "Three", "Four",
   "Five"};
```

Beachten Sie, dass Sie bei einer derartigen Notation die Anzahl der enthaltenen Elemente nicht direkt angeben können – diese ergibt sich anhand der zugewiesenen Werte.

Listing 4.1
Ausgabe aller Array-Elemente mit Hilfe einer for-Schleife

Falls Sie sich die Frage stellen, wieso die Angabe des aktuellen Elements durch *i+1* ausgedrückt worden ist: Erinnern Sie sich daran, dass Arrays nullbasierend sind. Das erste Element befindet sich an der Index-Position 0 – um also auf die Nummer des aktuellen Elements zu kommen, müssen wir den Zähler um 1 erhöhen.

for-each-Schleife

Mit Hilfe der *for-each*-Schleife, eines neu bei Java 1.5 eingeführten Konstrukts, können Sie ein Array ebenfalls sehr einfach durchlaufen:

Listing 4.2
Ausgabe aller Elemente eines Arrays mit Hilfe einer for-each-Schleife

```
String[] data =
   new String[] { "One", "Two", "Three", "Four", "Five" };

for(String item : data) {
   System.out.println(String.format("Item: \"%s\"", item));
}
```

Nach der Deklaration des Arrays erfolgt analog zum letzten Beispiel die Ausgabe aller Elemente mit Hilfe einer Schleife. Diesmal wird von uns die *for-each*-Schleife eingesetzt, die uns zwar keinen Zugriff auf den aktuellen Index gewährt, dafür aber wesentlich einfacher zu programmieren ist – die Deklaration einer lokalen Variable (`item`) und das Setzen eines Doppelpunktes vor dem zu durchlaufenden Array (`data`) genügt hier völlig. Java erledigt den Rest für uns und weist bei jedem Schleifendurchlauf das aktuelle Element des Arrays der lokalen Variablen zu.

Wie zu erwarten war, werden alle enthaltenen Elemente des Arrays untereinander ausgegeben:

Item: "One"

Item: "Two"

Item: "Three"

Item: "Four"

Item: "Five"

4.1.4 Mehrdimensionale Arrays

Mehrdimensionale Arrays gibt es bei Java nicht direkt. Sie sind aber recht einfach emulierbar. Angenommen, Sie wollten ein zweidimensionales Array für einen Termin-Kalender einsetzen, dann könnten Sie dies erreichen, indem Sie die einzelnen Elemente bereits als Array deklarieren:

```
Object[][] calendar = new Object[12][31];
```

Das Statement sollten Sie so lesen, dass Sie ein neues Object-Array mit 12 Elementen aus je einem Object-Array mit 31 Instanzen erstellen möchten. Das Array mit zwölf Elementen repräsentiert dabei die Monate, während die jedem Monat zugeordneten 31 Elemente die Tage darstellen.

Der 21. März wäre in diesem Modell dann so erreichbar:

```
Object day = calendar[3][21];
```

Die Anzahl der Elemente jedes Arrays erhalten Sie mit Hilfe der Methode `length()`:

```
System.out.println(String.format(
   "Elements in 1st dimension: %d", calendar.length));
System.out.println(String.format(
   "Elements in 2nd dimension: %d", calendar[0].length));
```

4.1.5 Vor- und Nachteile

Arrays sind eine sehr leistungsfähige Struktur für die Aufnahme gleichartiger Daten. Sie bieten dabei folgende Vorteile:

- Sehr einfach zu verwenden
- Einfache und typsichere Deklaration
- Universell einsetzbar

Daneben sollen einige Nachteile von Arrays gegenüber anderen Speicherstrukturen nicht verschwiegen werden:

- Kein direktes Vergrößern oder Verkleinern der Array-Grenzen möglich
- Keine Möglichkeit des Zugriffs über einen Schlüssel
- Keine Möglichkeit der Prüfung, ob eine Element bereits im Array enthalten ist
- Keine echte Mehrdimensionalität

4.2 ArrayList

Eine `ArrayList` stellt die einfachste Form der Liste nach dem Array dar. Sie verfügt über alle Vorteile eines Arrays, vermeidet aber die meisten von dessen Nachteilen. Die `ArrayList`-Klasse befindet sich im Paket *java.util* und ist seit Java 1.5 in der Lage, typsicher zu arbeiten.

Diese Typsicherheit ist mit Hilfe von so genannten *Generics* implementiert worden – eine Technologie, die es erlaubt, die Zuweis- und Abrufbarkeit bestimmter Typen bei generischen Listen zu beschränken, ohne die Liste neu implementieren zu müssen. Klingt Ihnen zu abstrakt? Keine Sorgen – wir werden *Generics* ganz einfach nutzen, und Sie werden schnell feststellen, worin deren Sinn liegt.

4.2.1 Deklaration und Instanzierung

Die Deklaration einer *ArrayList* erfolgt ebenso einfach, wie Sie dies von anderen Objekten gewohnt sind:

```
ArrayList list = new ArrayList();
```

Ohne weitere Angabe von Parametern haben Sie so eine *ArrayList* mit einer initialen Kapazität von zehn Elementen angelegt. Manchmal möchten Sie aus Performance-Gründen allerdings gleich eine größere Kapazität vorgeben:

```
ArrayList biggerList = new ArrayList(100);
```

Achtung

Die Kapazitätsvorgabe einer *ArrayList* hat nichts damit zu tun, wie viele Elemente später tatsächlich in der Liste gespeichert werden können. Sie sorgt lediglich dafür, dass die Liste vorab genügend Platz reserviert. Eine geringere Kapazität der Liste könnte im Extremfall dazu führen, dass bei jedem Einfügen eines neuen Elements der entsprechende Platz reserviert und die Liste entsprechend erweitert werden müsste, was insbesondere im Hinblick auf Performance echte Nachteile aufwerfen könnte.

Andererseits kann die Definition einer zu großen Listenkapazität ebenfalls nachteilige Folgen haben – insbesondere auf Systemen mit geringem Arbeitsspeicher. Wenn Sie also Performance-Engpässe haben, sollten Sie einen Blick auf Listendefinitionen und deren Kapazitäten werfen, da dort jede Menge Optimierungspotenzial vermutet werden kann.

Ebenso können Sie der *ArrayList* eine Liste von Elementen schon bei der Instanzierung zuweisen:

```
import java.util.ArrayList;
import java.util.Arrays;

String[] data =
   new String[] { "One", "Two", "Three", "Four", "Five" };
ArrayList listFromList = new ArrayList(Arrays.asList(data));
```

Betrachten Sie doch das obige Statement noch einmal genauer: Statt das *String*-Array direkt an den Konstruktor der *ArrayList* zu übergeben, verwenden wir folgende Anweisung:

```
Arrays.asList(data)
```

Mit Hilfe von *Arrays.asList* konvertieren wir das Array in eine *List*-Instanz, die einer *ArrayList* zugewiesen wird. Eine direkte Zuweisung des Arrays zu einer *ArrayList* ist leider nicht zulässig. Die einzige Alternative zur Verwendung von *Arrays.asList* wäre gewesen, alle Elemente der *ArrayList* in einer *for*- oder *for-each*-Schleife auszulesen und manuell zuzuweisen ... Dann doch lieber *Arrays.asList* einsetzen!

Wenn Sie sichergehen wollen, dass nur Elemente eines bestimmten Typs oder davon abgeleiteter Typen zugewiesen werden, können Sie dies ebenfalls bei der Deklaration der Liste angeben, indem Sie direkt hinter dem Typnamen der Liste den erlaubten Datentyp definieren:

```
ArrayList<String> typeSafeList = new ArrayList<String>();
```

Jede Zuweisung von Elementen, die nicht dem angegebenen Typ entsprechen, würde in *Exceptions* oder Kompiler-Fehlermeldungen enden. Diese Technologie nennt man *Generics*.

4.2.2 Zuweisen von Werten

Um einer *ArrayList* Werte hinzuzufügen, können Sie deren Methoden `add()` und `addAll()` verwenden. Erstere erlaubt die Zuweisung eines einzelnen Elements, letztere einer ganzen Liste.

add

Die Methode `add()` ist überladen. Die erste Variante nimmt eine *Object*-Instanz (oder den Typ, der bei der Listendefinition angegeben worden ist) als Parameter entgegen und fügt sie am Ende der Liste an:

```
import java.util.ArrayList;

ArrayList<String> list = new ArrayList<String>();
list.add("One");
```

Um vor einem bereits existierenden Element oder ganz am Anfang der Liste ein Element einzufügen, können Sie die Position des neu einzufügenden Elements als Parameter mit angeben:

```
import java.util.ArrayList;

ArrayList<String> list = new ArrayList<String>();
list.add("One");
list.add(0, "Two");
```

Lassen Sie uns hier mit Hilfe einer *for-each*-Schleife alle Elemente der Liste ausgeben:

```
for(String item : list) {
   System.out.println(item);
}
```

Die Ausgabe zeigt die Reihenfolge der Elemente:

Two
One

Mit Hilfe der Methode `add()` sind Sie also nicht in der Lage, Elemente zu überschreiben, sondern können „lediglich“ neue Elemente ein- oder anfügen.

addAll

Mit Hilfe von `addAll()` können Sie der *ArrayList* eine Liste von Elementen hinzufügen:

```
import java.util.ArrayList;
import java.util.Arrays;

ArrayList<String> list = new ArrayList<String>();
list.add("One");
```

Was geschieht eigentlich, wenn Sie eine *ArrayList* typsicher deklarieren und dann eine Variable eines anderen Typs zuweisen wollen?

Angenommen, wir hätten folgende *ArrayList*-Deklaration:

```
import
  java.util.ArrayList;

ArrayList<String> list =
  new ArrayList<String>();
list.add("One");
list.add(0, "Two");
list.add(1, 1);
```

Würde uns dies der Compiler durchgehen lassen? Nein, jedenfalls nicht, wenn er korrekt arbeitet. Sollte er es aus irgendeinem Grund dennoch tun, können wir uns darauf verlassen, dass Java zur Laufzeit mit einer *Exception* aufwarten wird.

Durch die explizite Angabe des erwarteten Typs kann also sichergestellt werden, welche Art von Daten zugewiesen werden darf. Das langwierige Ableiten eigener typsicherer Versionen von *ArrayList* (oder anderen Listen-Typen), das man aus früheren Java-Versionen kannte, dürfte also in der Regel der Vergangenheit angehören.

```
String[] data = new String[] { "Two", "Three", "Four", "Five" };
list.addAll(Arrays.asList(data));


for(String item : list) {
   System.out.println(item);
}
```

Wenn wir den Code ausführen, stellen wir fest, dass alle Elemente von `data` am Ende der *ArrayList* eingefügt worden sind:

One
Two
Three
Four
Five

Wenn wir dies nicht wollen, müssen wir auch bei `addAll()` die Einfüge-Position angeben. Alle folgenden Elemente werden dann entsprechend in der Liste nach hinten verschoben:

```
import java.util.ArrayList;
import java.util.Arrays;

ArrayList<String> list = new ArrayList<String>();
list.add("One");

String[] data = new String[] { "Two", "Three", "Four", "Five" };
list.addAll(0, Arrays.asList(data));


for(String item : list) {
   System.out.println(item);
}
```

Die Ausgabe bestätigt diese Aussage:

Two
Three
Four
Five
One

4.2.3 Abrufen von Werten

Das Abrufen von Werten aus einer *ArrayList* gestaltet sich ähnlich simpel, wie es bei Arrays der Fall war: Auf ein bestimmtes Element greifen Sie über dessen Index zu. Und die Anzahl der Elemente einer *ArrayList* erhalten Sie über deren Methode `size()`. Wenn wir dann noch berücksichtigen, dass eine *ArrayList* ihre Elemente intern nullbasiert numeriert, können wir aus der Größe der Liste auch ableiten, dass deren letztes Element den Index-Wert *Größe –1* besitzen muss.

Mit Hilfe der Methode `get()`, die als Parameter den Index-Wert des abzurufenden Elements entgegen nimmt, kann der eigentliche Zugriff auf ein Element der Liste geschehen:

```
import java.util.ArrayList;
import java.util.Arrays;

ArrayList<String> list = new ArrayList<String>();
list.add("One");

String[] data = new String[] { "Two", "Three", "Four", "Five" };
list.addAll(Arrays.asList(data));


System.out.println(
   String.format("Last element: %s", list.get(list.size() - 1)));
```

Wenn Sie das Code-Fragment in einer Klasse kapseln und ausführen, sollten Sie folgende Ausgabe erhalten:

Last element: Five

Achtung

Wenn Sie auf ein Element zugreifen wollen, dessen Index die Größe der Liste überschreitet, werden Sie eine entsprechende *IndexOutOfBoundsException* erhalten. Prüfen Sie deshalb vor einem Zugriff immer ab, ob das entsprechende Element überhaupt in der Liste existiert – dessen Index muss zwischen 0 und der *Listengröße -1* liegen.

Oft werden Sie aber nicht nur einen Wert, sondern eine Menge von Werten abrufen wollen. In diesem Fall verwendet man in der Regel *for*- und *for-each*-Schleifen.

for

Eine *for*-Schleife zum Abruf von Werten einer *ArrayList* könnte so aussehen:

Listing 4.3
Ausgabe aller Elemente einer ArrayList mittels for-Schleife

```
import java.util.ArrayList;
import java.util.Arrays;

/**
 * Created by Karsten Samaschke
 */
public class ArrayList_For {

   public static void main(String args[]) {

      ArrayList<String> list = new ArrayList<String>();
      list.add("One");
```

Listing 4.3 *(Forts.)*
Ausgabe aller Elemente einer ArrayList mittels for-Schleife

```
        String[] data = new String[] {
           "Two", "Three", "Four", "Five" };

        list.addAll(Arrays.asList(data));

        for(int i=0; i<list.size(); i++) {
           System.out.println(
              String.format("Element at Index #%d: \"%s\"", i,
              list.get(i)));
        }
     }
}
```

Der Zähler `i` entspricht bei dieser Listen-Variante einer Zahl zwischen *0* und der *Listengröße – 1*. Mit Hilfe dieser Information kann der Wert unter Verwendung der Methode `get()` abgerufen und ausgegeben werden. Die Ausgabe bei obigem Beispiel sollte so aussehen:

Element at Index #0: "One"

Element at Index #1: "Two"

Element at Index #2: "Three"

Element at Index #3: "Four"

Element at Index #4: "Five"

for-each

Noch einfacher gestaltet sich die Ausgabe mit Hilfe einer *for-each*-Schleife:

Listing 4.4
Ausgabe aller Elemente einer ArrayList per for-each-Schleife

```
import java.util.ArrayList;
import java.util.Arrays;

/**
 * Created by Karsten Samaschke
 */
public class ArrayList_ForEach {

   public static void main(String args[]) {


      ArrayList<String> list = new ArrayList<String>();
      list.add("One");

      String[] data =
         new String[] { "Two", "Three", "Four", "Five" };
      list.addAll(Arrays.asList(data));

      for(String item : list) {
         System.out.println(String.format("%s", item));
      }
   }
}
```

Hier verzichten wir zwar auf die Information darüber, an welcher Stelle der Liste wir uns befinden, sparen aber auch jede Menge Schreibarbeit ein. Und die Ausgabe erfüllt ihren Zweck:

One
Two
Three
Four
Five

Iterator

Eine dritte Alternative ist der Einsatz eines so genannten *Iterators*. Dieser erlaubt es, analog zur *for-each*-Schleife durch die einzelnen Elemente der *ArrayList* zu laufen, ohne diese im Speziellen kennen zu müssen:

Listing 4.5
Abruf der Daten einer ArrayList mit Hilfe eines Iterators

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;

/**
 * Created by Karsten Samaschke
 */
public class ArrayList_Iterator {

   public static void main(String args[]) {

      ArrayList<String> list = new ArrayList<String>();
      list.add("One");

      String[] data =
         new String[] { "Two", "Three", "Four", "Five" };
      list.addAll(Arrays.asList(data));

      Iterator<String> it = list.iterator();
      while(it.hasNext()) {
         String current = it.next();
         System.out.println(current);
      }
   }
}
```

Die Vorteile eines *Iterators* gegenüber der neuen *for-each*-Schleife beschränken sich darauf, dass der *Iterator* einerseits auch schon in älteren Java-Versionen verfügbar war und andererseits etwas mehr Kontrolle darüber lässt, wann er den Wert tatsächlich abruft, was durch *<Iterator>.next()* vorgenommen wird.

Ansonsten sind *Iterator* und *for-each*-Schleife in diesem Szenario ziemlich äquivalent, was auch durch die Ausgabe unterstrichen wird:

One
Two
Three
Four
Five

4.2.4 Ersetzen von Werten

Neben dem Zuweisen und Ersetzen werden oftmals auch Aktualisierungen von Daten vorgenommen. Dies ist bei einer *ArrayList* mit Hilfe der Methode `set()` möglich, die als Parameter den Index-Wert des zu überschreibenden Wertes und den neuen Wert entgegennimmt:

Listing 4.6
Ersetzen eines Listen-Elements mit Hilfe der Methode set() der ArrayList

```
import java.util.ArrayList;
import java.util.Arrays;

/**
 * Created by Karsten Samaschke
 */
public class ArrayList_Set {

   public static void main(String args[]) {

      String[] data =
         new String[] { "One", "Two", "Three", "Four", "Five" };
      ArrayList<String> list =
         new ArrayList<String>(Arrays.asList(data));

      list.set(0, "Zero");

      for(String item : list) {
         System.out.println(String.format("%s", item));
      }
   }
}
```

Achtung

Die Methode `set()` wirft eine *IndexOutOfBoundsException*, falls für den Index-Wert eine negative Zahl oder eine Zahl, die größer/gleich der Listengröße ist, angegeben wird. Sie sollten also zunächst mit Hilfe von `size()` überprüfen, ob der Indexwert gültig ist.

Wenn Sie die Klasse ausführen, werden Sie sehen, dass das erste Element der *ArrayList* überschrieben worden ist:

Zero
Two
Three
Four
Five

4.2.5 Löschen von Werten

Sie können Werte aus einer *ArrayList* einzeln oder komplett löschen. Die einfachste Variante des Löschens stellt die Methode `clear()` dar, die einfach alle Elemente entfernt:

```
list.clear();
```

Wenn Sie nur einen bestimmten Wert löschen wollen, können Sie dies über dessen Index mit Hilfe der Methode `remove()` erledigen:

```
list.remove(0);
```

Sollten Sie den Wert kennen, können Sie diesen als Parameter angeben:

```
list.remove("Two");
```

Wenn Sie Werte zwischen zwei Indizes entfernen wollen, können Sie die Methode `removeRange()` verwenden:

```
list.removeRange(0, 3);
```

Die Methode `removeRange()` entfernt dabei alle Elemente, deren Indizes zwischen dem ersten Parameter (einschließlich) und vor dem zweiten Parameter liegen. Obiges Beispiel sollte also so gelesen werden, dass die Elemente an den Positionen *0*, *1* und *2* entfernt werden. Das Element mit dem Index *3* bleibt dagegen unberührt.

4.2.6 Überprüfen, ob ein Element in der Liste existiert

Die Methode `contains()` gibt uns Aufschluss darüber, ob ein bestimmtes Element innerhalb einer *ArrayList* existiert:

```
if(list.contains("Two")) {
   System.out.println(
      list.indexOf("Two"));
}
```

Die Rückgabe der Methode `contains` ist entweder *true* oder *false*.

Wenn wir darüber hinaus noch feststellen wollen, an welcher Position in der Liste sich das gewünschte Element befindet, können wir die Methode `indexOf()` verwenden, die auf Gleichheit mit Hilfe der generischen `equals()`-Methode testet.

Wenn das gesuchte Element nicht gefunden werden konnte, wird *-1* zurückgegeben, anderenfalls erhalten wir den Index-Wert des gesuchten Elements. Während `indexOf()` die *ArrayList* von vorne durchsucht, verfügt die Methode `lastIndexOf()` über die gleiche Funktionalität, sucht aber vom Ende der Liste her.

4.2.7 Vor- und Nachteile der ArrayList

Der Einsatz der *ArrayList* bringt im Vergleich zum Array in fast jeder Beziehung nur Vorteile mit sich:

- Die Größe der Liste kann geändert werden
- Es können problemlos Elemente eingefügt und wieder entfernt werden
- Eine ArrayList kann aus einem Array erzeugt werden
- Einer ArrayList können andere Listen hinzugefügt werden
- In einer ArrayList können Elemente gesucht werden
- ArrayLists arbeiten oftmals performanter als Arrays
- ArrayLists sind flexibler als Arrays

Nachteile von *ArrayLists* gegenüber Arrays mag es zwar geben, jedoch sind diese in der Praxis meist nicht relevant. Deshalb gilt: Wann immer Sie ein Array einsetzen, sollten Sie intensiv über den Einsatz einer *ArrayList* nachdenken.

4.3 HashMap

Die *HashMap* bietet eine einfache und performante Lösung, um Daten in Form von Schlüssel-/ Wert-Paaren zu erfassen. Bei der *HashMap* können Sie beliebige Objekte als Schlüssel verwenden und anhand dieser Schlüssel die zugewiesenen Werte wieder abrufen.

Ein typischer Einsatzbereich von *HashMaps* könnte beispielsweise eine Personen-Verwaltung sein: Der Name der Person ist der Schlüssel, der Wert repräsentiert deren Detail-Informationen.

4.3.1 Deklaration und Instanzierung

Die Deklaration einer *HashMap* erfolgt im einfachsten Fall genauso, wie wir es von anderen Objekten gewohnt sind:

```
java.util.HashMap map = new java.util.HashMap();
```

Dabei wird eine *HashMap* mit einer Kapazität von elf Elementen und einem *Load-Faktor* (zu Deutsch: Lastfaktor) von 0,75 erzeugt. Das bedeutet, dass die *HashMap* Platz für elf Elemente bietet und bei einer Füllung von 75% automatisch Platz für neue Elemente schafft. Stellen Sie sich dies ähnlich wie bei einer *ArrayList* vor, die auch zwischen Kapazität und tatsächlich genutztem Platz unterscheidet.

In der Regel müssen Sie keine Änderungen an initialer Kapazität und *Load-Faktor* vornehmen, da die *HashMap* diese Informationen sehr gut selbst verwaltet. Wenn Sie wissen, dass Ressourcen kein Problem darstellen, können Sie selbstverständlich die initiale Kapazität besonders hoch und den *Load-Faktor* besonders niedrig festlegen – entsprechend viele System-Ressourcen werden durch die *HashMap* gebunden, und entsprechend langsam wird Ihre Anwendung auf nicht so leistungsfähigen Systemen laufen. Deshalb: Gebrauchen Sie diese Möglichkeit mit Vorsicht!

Sie können der *HashMap* natürlich auch explizite Vorgaben hinsichtlich initialer Kapazität machen:

```
java.util.HashMap map = new java.util.HashMap(15);
```

legt eine *HashMap* mit Platz für 15 Elemente an, deren *Load-Faktor* 0,75 ist.

Selbstverständlich können Sie mit Hilfe des *Load-Faktors* auch angeben, ab welchem Füllgrad wieder Platz für neue Elemente geschaffen werden soll. Dies könnte beispielsweise so aussehen:

```
java.util.HashMap map = new java.util.HashMap(15, 0.90);
```

Mit obigem Code hätten Sie eine neue *HashMap* mit einer Kapazität von 15 Elementen und einem *Load-Faktor* von 0,90 erzeugt.

Wenn Sie eine Collection haben, die – ebenso wie die *HashMap* – das *Map*-Interface von Java unterstützt, können Sie diese ebenfalls schon beim Instanzieren einer neuen *HashMap* zuweisen:

```
java.util.HashMap table = new java.util.HashMap(myMap);
```

4.3.2 HashMap und Generics

Die `HashMap`-Klasse unterstützt das neu bei Java 1.5 eingeführte *Generics*-Konzept von Java, mit dem Sie festlegen können, welche Typen erwartet werden. Da die `HashMap` Schlüssel und Werte unterscheidet, können Sie dies getrennt für beide Elemente festlegen.

Angenommen, Sie wollten eine `HashMap` definieren, die nur `Strings` als Schlüssel entgegennimmt und deren Werte vom Typ `Person` sein müssen, dann sollte dies so aussehen:

```
java.util.HashMap<String, Person> map =
   new java.util.HashMap<String, Person>();
```

Die anderen Konstruktor-Überladungen bleiben weiterhin gültig.

4.3.3 Werte zuweisen

Die Zuweisung von Werten erfolgt bei einer *HashMap* immer unter Angabe eines Schlüssels und des zuzuweisenden Wertes. Der Schlüssel darf dabei ein beliebiges Objekt sein, so lange dieses die Methoden `hashCode()` und `equals()` unterstützt.

Die eigentliche Zuweisung einer Schlüssel-/Wert-Kombination erfolgt mit Hilfe der Methode `put()`:

Listing 4.7
Zuweisen von Elementen zu einer HashMap

```
import java.util.HashMap;

/**
 * Works with a HashMap and adds some values
 */
public class HashMap_Put {

   public static void main(String[] args) {
      HashMap<String, Person> map =
         new HashMap<String, Person>();
      map.put("Samaschke",
         new Person("Samaschke", "Karsten"));
      map.put("McNealy", new Person("McNealy", "Scott"));
      map.put("Torvalds", new Person("Torvalds", "Linus"));
      map.put("Gates", new Person("Gates", "Bill"));
      map.put("Jobs", new Person("Jobs", "Steve"));
   }
}
```

Unsere HashMap-Instanz wird in diesem Beispiel mit Hilfe von Generics so deklariert, dass sie ausschließlich Schlüssel vom Typ String und Werte vom Typ Person entgegennimmt:

```
HashMap<String, Person> map =  new HashMap<String, Person>();
```

Dies müssen wir berücksichtigen, wenn wir einen Wert zuweisen wollen. Deshalb übergeben wir mit Hilfe von put() einen Schlüssel vom Typ String und einen Wert vom Typ Person:

```
map.put("Samaschke", new Person("Samaschke", "Karsten"));
```

Analog zum Hinzufügen einzelner Werte können Sie ebenfalls eine bereits existierende *Map*-Implementierung anfügen. In diesem Fall verwenden Sie die Methode `putAll()`, der als Parameter die *Map*-Implementierung übergeben wird:

Listing 4.8 Hinzufügen einer Map-Implementierung zu einer HashMap

```
import java.util.HashMap;

/**
 * Works with a HashMap and adds a list
 */
public class HashMap_PutAll {

   public static void main(String[] args) {
      HashMap<String, Person> map = new HashMap<String, Person>();
      map.put("Samaschke", new Person("Samaschke", "Karsten"));
      map.put("McNealy", new Person("McNealy", "Scott"));

      HashMap<String, Person> innerMap =
         new HashMap<String, Person>();
      innerMap.put("Torvalds", new Person("Torvalds", "Linus"));
      innerMap.put("Gates", new Person("Gates", "Bill"));
      innerMap.put("Jobs", new Person("Jobs", "Steve"));

      map.putAll(innerMap);
   }
}
```

Was geschieht übrigens, wenn Sie einen Schlüssel mehrfach verwenden? Nun, die *HashMap* ersetzt den dem Schlüssel zugeordneten Wert durch den neuen Wert. Wenn Sie also

```
map.put("Samaschke", new Person("Samaschke", "Karsten"));
map.put("Samaschke", new Person("Samaschke", "Johannes"));
```

schrieben, würde unter dem Schlüssel „Samaschke“ nur der zuletzt zugewiesene Eintrag abgelegt werden.

4.3.4 Abrufen von Werten

Das Abrufen eines Schlüssels geschieht, indem Sie der Methode `get()` das Objekt übergeben, das den Schlüssel repräsentiert. Angenommen, Sie wollten den Wert auslesen, der sich hinter dem Schlüssel „Samaschke“ befände, dann könnten Sie dies so erreichen:

Listing 4.9 Abrufen eines Wertes aus einer HashMap

```
import java.util.HashMap;

/**
 * Retrieves values from a HashMap
 */
public class HashMap_Get {

   public static void main(String[] args) {
      HashMap<String, Person> map =
         new HashMap<String, Person>();
      map.put("Samaschke", new Person("Samaschke", "Karsten"));
      map.put("McNealy", new Person("McNealy", "Scott"));
```

```
        map.put("Torvalds", new Person("Torvalds", "Linus"));
        map.put("Gates", new Person("Gates", "Bill"));
        map.put("Jobs", new Person("Jobs", "Steve"));

        Person p = map.get("Samaschke");
        System.out.println(
           String.format("Person represented by \"Samaschke\":
              %s, %s",
              p.getName(), p.getFirstName()));
    }
}
```

Listing 4.9 *(Forts.)* Abrufen eines Wertes aus einer HashMap

Die verwendete *HashMap*-Instanz wird mit Hilfe von *Generics* als Schlüssel-/Wert-Liste aus *String*- und *Person*-Elementen deklariert:

```
HashMap<String, Person> map = new HashMap<String, Person>();
```

Sinngemäß passiert hier Folgendes: Wir geben einer *Collection*-Implementierung (in diesem Fall *HashMap*) an, welche Typen von Elementen zugelassen sein sollen. Unsere *HashMap*-Instanz wird nun nur noch die Typen als Parameter akzeptieren, die bei ihrer Deklaration angegeben worden sind.

Sollte es sich dabei um *String*- und *Person*-Elemente handeln, würden nur diese als Parameter akzeptiert werden. Die *HashMap* würde sogar darauf achten, dass die *String*-Elemente nur als Schlüssel und die *Person*-Instanzen nur als Werte eingesetzt werden können. Für uns ist diese Technologie also völlig transparent und ohne großen Aufwand umsetzbar. Der unschätzbare Vorteil: Wir können mit Java-Bordmitteln sicherstellen, dass nur uns genehme Typen verwendet werden. Dies war früher nur durch das Schreiben eigener *Collection*-Implementierungen (und dort auch nur teilweise) möglich.

Nach dem Zuweisen einiger Werte mit Hilfe der Methode `put()`, können wir nun einen bestimmten Wert wieder abrufen. Wir erledigen dies, indem wir der Methode eine Variable vom Typ des Schlüssels (in unserem Fall also einen *String*) als Parameter übergeben:

```
Person p = map.get("Samaschke");
```

Den Inhalt der ermittelten *Person*-Instanz können wir anschließend einfach ausgeben.

Achtung

Die Methode `get()` der *HashMap* liefert uns, falls wir nicht von Generics Gebrauch machen oder eine Java-Version vor 1.5 einsetzen, immer eine Instanz vom Typ *Object* zurück. Alternativ ist die Rückgabe auch *null*, falls der entsprechende Schlüssel nicht existierte.

Haben wir also keine Typen angegeben oder setzen eine ältere Java-Version ein, müssen wir das erhaltene Objekt noch in den Zieltyp *casten*. Dies könnte dann so aussehen:

```
Person p = (Person)map.get("Samaschke");
```

Wenn Sie die Klasse ausführen, werden Sie sicherlich das korrekte Ergebnis erhalten:

Person represented by "Samaschke": Samaschke, Karsten

4.3.5 Abrufen aller Elemente einer HashMap

Mit Hilfe ihrer Methode `keySet()` gibt uns die *HashMap* eine *Set*-Implementierung einer Liste aller ihrer Schlüssel zurück.

Diese Liste können wir mit Hilfe eines *Iterators* oder nach Umwandlung in ein Array durchlaufen. Damit verfügen wir dann über die Möglichkeit, alle zugeordneten Werte abzurufen:

Listing 4.10 Ausgabe aller Elemente einer HashMap

```
import java.util.HashMap;
import java.util.Iterator;

/**
 * Iterates through a HashMap
 */
public class HashMap_KeySet {

   public static void main(String[] args) {
      HashMap<String, Person> map =
         new HashMap<String, Person>();

      map.put("Samaschke", new Person("Samaschke", "Karsten"));
      map.put("McNealy", new Person("McNealy", "Scott"));
      map.put("Torvalds", new Person("Torvalds", "Linus"));
      map.put("Gates", new Person("Gates", "Bill"));
      map.put("Jobs", new Person("Jobs", "Steve"));

      Iterator<String> it = map.keySet().iterator();

      while(it.hasNext()) {
         String current = it.next();
         Person p = map.get(current);

         System.out.println(
            String.format("Person represented by \"%s\": %s, %s",
               current, p.getName(), p.getFirstName()));
      }
   }
}
```

Nach der Deklaration der HashMap als solche mit den Typen *String* und *Person* weisen wir ihr einige Daten zu.

Ein *Iterator* erlaubt es uns, alle seine enthaltenen Elemente ähnlich einer *for-each*-Schleife abzurufen. Ein Iterator kann ebenfalls Gebrauch von Generics machen – wie in unserem Beispiel, wo er als Iterator vom Typ String definiert worden ist:

```
Iterator<String> it = map.keySet().iterator();
```

Der eigentliche Abruf erfolgt in zwei Schritten innerhalb einer while-Schleife. Zunächst wird mit Hilfe von `<Iterator>.hasNext()` geprüft, ob weitere Elemente vorhanden sind. Sollte dem so sein, wird der Body der while-Schleife durchlaufen, in dem dann per `<Iterator>.current()` der aktuelle Schlüssel abgerufen und der Variablen current zugewiesen werden kann. Nun kann der Inhalt von current als Schlüssel für die Methode `get()` der *HashMap* verwendet werden.

Das Ergebnis der Mühen sieht doch zumindest interessant aus:

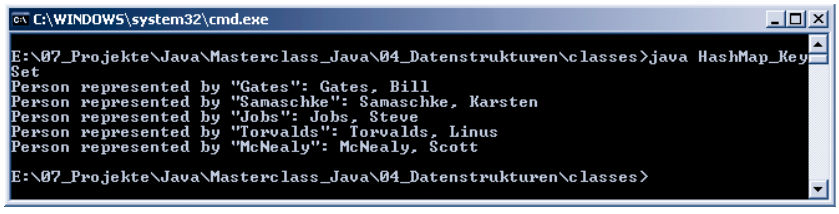


Abbildung 4.1
Ausgabe der Elemente einer HashMap

Sie wundern sich über die Reihenfolge? Die resultiert daraus, dass die `HashMap` die Elemente intern anhand ihrer *Hash*-Werte sortiert – und ein derartiger *Hash*-Wert hat wenig bis nichts mit der alphabetischen Repräsentation eines *Strings* zu tun.

Eine weitere Möglichkeit, alle Werte einer *HashMap* abzurufen, bietet die Methode `values()` an, die eine *Collection* aus den enthaltenen Werten zurückgibt:

```
for(Person p : map.values()) {
   System.out.println(
      String.format("Current person: %s, %s",
         p.getName(), p.getFirstName()));
}
```

Hier haben wir die einzelnen Elemente der *HashMap* mit Hilfe einer *for-each*-Schleife durchlaufen – weniger Schreibaufwand, dafür aber haben wir zumindest alle Werte erhalten:

Current person: Gates, Bill

Current person: Samaschke, Karsten

Current person: Jobs, Steve

Current person: Torvalds, Linus

Current person: McNealy, Scott

4.3.6 Löschen von Elementen

Das Löschen einzelner oder aller Elemente der *HashMap* bewerkstelligen Sie mit Hilfe der Methoden `remove()` und `clear()`. Erstere Methode entfernt einzelne Elemente anhand ihrer Schlüssel aus der *HashMap*, letztere räumt gründlich auf und leert die komplette Liste.

Um ein bestimmtes Element aus der Liste zu entfernen, können Sie folgenden Code verwenden:

```
String key = "Gates";
Person removed = map.remove(key);
if(null != removed) {
   System.out.println(String.format(
      "%s, %s has been successfully removed!",
         removed.getName(), removed.getFirstName());
} else {
   System.out.println(String.format(
      "The element %s has not been part of the list!", key));
}
```

Beachten Sie, dass die *HashMap* uns das entfernte Objekt als Rückgabe der Methode `remove()` zur weiteren Verwendung zur Verfügung stellt. Davon haben wir in obigem Code auch Gebrauch gemacht, indem wir die Rückgabe daraufhin untersucht haben, ob sie *null* wäre. Eine Rückgabe des Wertes *null* lässt darauf schließen, dass das entsprechende Element nicht in der Liste existiert hat.

Um alle Elemente zu entfernen, verwenden Sie die Methode `clear()`:

```
map.clear();
System.out.println(map.size());
```

Die Ausgabe in diesem Fall wird mit ziemlicher Sicherheit „0“ lauten.

Übrigens: Die Anzahl aller Schlüssel in einer *HashMap* bestimmen Sie wie oben demonstriert mit Hilfe von deren Methode `size()`. Und wenn Sie wissen wollen, ob die *HashMap* Elemente enthält, verwenden Sie die Methode `isEmpty()`, die einen booleschen Wert zurückgibt – *false* bedeutet, dass Elemente in der HashMap existieren, und *true* sagt uns, dass die Map leer war.

4.3.7 Überprüfen, ob bestimmte Schlüssel oder Werte in der Liste existieren

Die *HashMap* stellt uns zwei Methoden zur Verfügung, mit deren Hilfe wir feststellen können, ob bestimmte Schlüssel oder Werte in der Liste enthalten sind: `containsKey()` und `containsValue()`.

Um festzustellen, ob ein bestimmter Schlüssel in der Liste enthalten ist, verwenden wir die Methode `containsKey()`:

Listing 4.11
Feststellen, ob eine Liste einen bestimmten Schlüssel enthält: containsKey()

```
import java.util.HashMap;

/**
 * Checks, whether a key exists in the HashMap-instance
 */
public class HashMap_ContainsKey {

   public static void main(String[] args) {
      HashMap<String, Person> map =
         new HashMap<String, Person>();
      map.put("Samaschke", new Person("Samaschke", "Karsten"));
      map.put("McNealy", new Person("McNealy", "Scott"));
```

```
      map.put("Torvalds", new Person("Torvalds", "Linus"));
      map.put("Gates", new Person("Gates", "Bill"));
      map.put("Jobs", new Person("Jobs", "Steve"));

      String key = "Samaschke";
      System.out.println(
         String.format("Element %s is part of the list: %s",
            key,
            map.containsKey(key)));
   }
}
```

Listing 4.11 *(Forts.)*
Feststellen, ob eine Liste einen bestimmten Schlüssel enthält: containsKey()

Die Methode `containsKey()` erwartet als Argument einen Schlüssel. Dieser Schlüssel muss sowohl vom Wert als auch vom Datentyp her mit einem Schlüssel aus der Liste übereinstimmen. Da wir unsere `HashMap` als `HashMap` mit den Typen `String` und `Person` deklariert haben, muss der Schlüssel vom Typ `String` sein.

Übergeben wir entsprechend einen String als Parameter, erhalten wir den booleschen Wert `true`, wenn der Schlüssel in der Liste existierte. Anderenfalls wird uns die HashMap mit der Rückgabe *false* beglücken.

Bei obigem Beispiel lautet die Ausgabe bei der Prüfung darauf, ob ein Schlüssel *Samaschke* in der Liste existierte:

Element Samaschke is part of the list: true

Das Gegenstück zu `containsKey()` ist `containsValue()`. Diese Methode liefert dann *true*, wenn das gesuchte Element in der *HashMap* enthalten ist, wobei sowohl Typ als auch Wert und Hash-Code übereinstimmen müssen. Übertragen auf unser Beispiel bedeutet dies, dass wir eine `Person`-Instanz als Parameter übergeben müssen:

```
import java.util.HashMap;

/**
 * Checks, whether a specific value is contained in the list or not
 */
public class HashMap_ContainsValue {

   public static void main(String[] args) {
      HashMap<String, Person> map =
         new HashMap<String, Person>();

      Person item = new Person("Samaschke", "Karsten");
      map.put("Samaschke", item);
      map.put("McNealy", new Person("McNealy", "Scott"));
      map.put("Torvalds", new Person("Torvalds", "Linus"));
      map.put("Gates", new Person("Gates", "Bill"));
      map.put("Jobs", new Person("Jobs", "Steve"));

      System.out.println(
         String.format("The list contains the Person-instance: %s",
            map.containsValue(item)));
   }
}
```

Listing 4.12
Feststellen, ob ein Element mit einem bestimmten Wert in der HashMap existiert

Unsere HashMap wurde als `HashMap` mit `String`-Schlüsseln und `Person`-Elementen deklariert:

```
HashMap<String, Person> map = new HashMap<String, Person>();
```

Entsprechend weisen wir einige Werte zu, die diesem Schema entsprechen:

```
Person item = new Person("Samaschke", "Karsten");
map.put("Samaschke", item);
map.put("McNealy", new Person("McNealy", "Scott"));
...
```

Beachten Sie, dass wir die `Person`-Instanz, auf deren Existenz wir später prüfen wollen, der lokalen Variablen item zuweisen.

Anschließend prüfen wir mit Hilfe von `containsValue()` darauf, ob das Element in der Liste existiert. Als Parameter wird dabei das Objekt übergeben, auf das geprüft werden soll – in unserem Fall ist dies der Wert der Variablen item, die eine `Person`-Instanz überprüft.

Die Rückgabe bestätigt, dass das gesuchte Element in der Liste existiert:

The list contains the Person-instance: true

Ändern wir das Beispiel doch einmal minimal ab und weisen die `Person`-Instanz, auf die wir später prüfen wollen, nicht direkt bei Erzeugung der Variablen `item` zu.

```
map.put("Samaschke", new Person("Samaschke", "Karsten"));
```

Stattdessen erzeugen wir die Person-Instanz erst später:

```
Person item = new Person("Samaschke", "Karsten");
System.out.println(
   String.format("The list contains the Person-instance: %s",
      map.containsValue(item)));
}
```

Wenn Sie dieses Beispiel ausführen, werden Sie feststellen, dass keine Übereinstimmung festgestellt werden kann:

The list contains the Person-instance: false

Der Grund dafür: Die beiden Instanzen der `Person`-Klasse (einmal die direkt beim Hinzufügen zur `HashMap` erzeugte Instanz und die der Variablen `item` zugewiesene Instanz) sind nicht identisch, obwohl sie die gleichen Werte enthalten, denn ihr `HashCode` unterscheidet sich. Unterschiedliche Instanzen einer Klasse weisen also auch immer unterschiedlichen `HashCode` auf, auch wenn die enthaltenen Werte identisch sind.

Überschreiben der Methode equals()

Der Lösungsansatz für dieses Problem besteht darin, die Methode `equals()` der `Person`-Klasse zu überschreiben:

Listing 4.13
Überschreiben der Methode equals der Person-Klasse

```
/**
   * Overwrites the generic equals-method provided by
   * java.lang.Object
   */
public boolean equals(Object o) {
   if (this == o) {
      return true;
   }

   if (!(o instanceof Person)) {
      return false;
   }

   Person person = (Person) o;
   boolean result = (person != null &&
      person.firstName != null &&
      person.name != null);

   if(result) {
      result = (this.firstName != null &&
         this.firstName == person.firstName);
      if(result) {
         result = (this.name != null &&
            this.name == person.name);
      }
   }

   return result;
}
```

Die Methode `equals` führt einige Prüfungen durch, bevor sie ihr abschließendes Urteil zurückgibt. Die erste Prüfung ist darauf, ob die aktuelle Klassen-Instanz mit der übergebenen Klassen-Instanz identisch ist. Falls dem so sein sollte, wird `true` zurückgegeben.

Sollten die beiden Objekte nicht identisch sein, wird geprüft, ob das übergebene Objekt überhaupt vom Typ der aktuellen Instanz ist. Sollte dies nicht der Fall sein, dann können wir die Prüfung an dieser Stelle abbrechen – wenn die Typen nicht übereinstimmen, dann nutzt die gründlichste Untersuchung nichts, denn dann können die Objekte oder deren Werte einfach nicht gleich sein.

Nun kann das übergebene Objekt in eine Instanz der `Person`-Klasse gecastet werden. Damit sind wir in der Lage, zu überprüfen, ob sowohl `firstName` als auch `name` ungleich *null* sind. Nur wenn die der Fall ist, werden wir weitere Prüfungen vornehmen. Ablesen lässt sich das Ergebnis dieser Checks an dem Inhalt von `result`:

Wenn `result` `true` ist, dann werden wir die Felder `firstName` miteinander vergleichen. Und nur, wenn diese beiden Felder den gleichen Inhalt haben, behält `result` den Wert `true`. Falls dem wirklich so sein sollte, können wir auch noch

die Inhalte der Variablen `name` der beiden Klassen-Instanzen miteinander vergleichen.

Die Variable `result`, die das Ergebnis der Überprüfungen repräsentiert, wird also nur `true` zurückgeben, wenn

- sowohl `name` als auch `firstName` ungleich null sind,
- die Inhalte von `name` und `firstName` übereinstimmen.

In jedem anderen Fall ist die Rückgabe der Methode `false`, was signalisiert, dass die beiden Objekte nicht identisch sind.

Erneutes Prüfen, ob das Element in der HashMap enthalten ist

Wenn wir diese Änderungen in die Klasse Person einbauen, dann sind wir nunmehr in der Lage, eine erneute Überprüfung auf das Vorhandensein einer Person-Instanz mit dem Vornamen Karsten und dem Nachnamen Samaschke vorzunehmen. Wir verwenden erneut den weiter oben gezeigten Code:

```
Person item = new Person("Samaschke", "Karsten");
System.out.println(
   String.format("The list contains the Person-instance: %s",
      map.containsValue(item)));
}
```

Diesmal ist die Ausgabe eindeutig:

The list contains the Person-instance: true

4.4 Properties

Die `Properties`-Klasse ist eine Ableitung der `Hashtable`-Klasse. Eine *Hashtable* wiederum entspricht weitestgehend der *HashMap*, ist aber im Grunde eine ältere Version mit minimalen Unterschieden hinsichtlich der internen Funktionalität. Die `Hashtable`-Klasse basiert auf der als *obsolete* (also als überflüssig) gekennzeichneten Klasse *Dictionary*. Im Gegensatz zur *HashMap* oder anderen *Collection*-Implementierungen ist sie nicht *Generics*-fähig.

Die `Properties`-Klasse verwaltet Name-/Wert-Paare und bietet ganz besondere Methoden zum Laden und Speichern der in ihr enthaltenen Werte, wodurch sie sich besonders für die Sicherung von Einstellungen einer Applikation eignet.

4.4.1 Deklaration und Instanzierung

Die Deklaration und Instanzierung einer `Properties`-Instanz erfolgt mit Hilfe des Default-Konstruktors ohne Argumente:

```
java.util.Properties props = new java.util.Properties();
```

Eine überladene Version der `Properties`-Klasse erlaubt es uns, eine bereits existierende `Properties`-Instanz für die Definition von Default-Werten zu verwenden:

```
java.util.Properties props = new java.util.Properties(defaults);
```

4.4.2 Zuweisen und Abrufen von Werten

Die Zuweisung von Werten erfolgt mit Hilfe der Methode `setProperty()`, die als Parameter zwei *String*-Werte entgegennimmt. Der erste Parameter dient dabei als Schlüssel, der zweite Parameter stellt den Wert dar:

Listing 4.14
Zuweisen von Werten zu einer Properties-Instanz

```
import java.util.Properties;

/**
 * Defining a simple Properties-instance
 */
public class Properties_SetProperty {

   public static void main(String args[]) {
      Properties props = new Properties();
      props.setProperty("Name", "Samaschke");
      props.setProperty("FirstName", "Karsten");
      props.setProperty("City", "Berlin");
   }
}
```

Das Abrufen der enthaltenen Werte kann nun mit Hilfe der Methode `getProperty()` stattfinden, die als Parameter den Schlüssel entgegennimmt, unter dem der gesuchte Wert abgelegt worden ist:

Listing 4.15
Abrufen von Werten einer Property-Instanz

```
import java.util.Properties;

/**
 * Retrieves a value from a Properties-instance
 */
public class Properties_GetProperty {

   public static void main(String[] args) {
      Properties props = new Properties();
      props.setProperty("Name", "Samaschke");
      props.setProperty("FirstName", "Karsten");
      props.setProperty("City", "Berlin");

      System.out.println(
         String.format("Name: %s",
            props.getProperty("Name")));
      System.out.println(
         String.format("First name: %s",
            props.getProperty("FirstName")));
      System.out.println(
         String.format("City: %s",
            props.getProperty("City")));
      System.out.println(
         String.format("Country: %s",
            props.getProperty("Country", "Germany")));
   }
}
```

Wenn Sie das Beispiel ausführen, werden Sie folgende Ausgabe erhalten:

Name: Samaschke

First name: Karsten

City: Berlin

Country: Germany

Die letzte Ausgabe sollte Sie etwas stutzig machen: Wenn Sie keinen Schlüssel *Country* definiert haben, woher kam dann die Länder-Angabe?

Was genau geschah also? Des Rätsels Lösung liegt darin, dass die Methode `getProperty()` der `Properties`-Klasse überladen ist – die Standard-Variante liefert uns den Wert eines Schlüssels und *null*, wenn der Schlüssel nicht gefunden werden konnte.

Die überladene Variante jedoch akzeptiert als zweiten Parameter einen Default-Rückgabe-Wert, der zum Einsatz kommt, falls der angegebene Schlüssel nicht gefunden werden konnte. Im obigen Beispiel haben wir genau diese überladene Variante der Methode `getProperty()` eingesetzt und *Germany* als Standard-Wert definiert.

4.4.3 Speichern der Werte in Textdateien

Das Speichern einer *Properties*-Liste in eine Textdatei kann mit Hilfe von deren Methode `store()` erledigt werden. Diese Methode erwartet als Argument die Angabe eines *OutputStream*-Objekts.

Folgender Code speichert eine *Properties*-Instanz im Unterverzeichnis *data* in die Datei *samaschke.props*:

Listing 4.16 Speichern einer Properties-Instanz in eine Text-Datei

```
import java.util.Properties;
import java.io.IOException;
import java.io.FileOutputStream;

/**
 * Saves the content of a Properties-instance to the disk
 */
public class Properties_SaveToFile {

   public static void main(String[] args) {
      Properties props = new Properties();
      props.setProperty("Name", "Samaschke");
      props.setProperty("FirstName", "Karsten");
      props.setProperty("City", "Berlin");

      try {
         FileOutputStream fos =
            new FileOutputStream("data/samaschke.props");
         props.store(fos, "Data for Samaschke");
         fos.close();
      } catch (IOException ignored) {}
   }
}
```

Der Ablauf des Programms in groben Zügen: Zunächst erstellen wir eine *Properties*-Instanz, der wir einige Werte zuweisen. Anschließend erzeugen wir einen so genannten *FileOutputStream*, der es uns erlaubt, Daten in Dateien zu speichern.

Da es dabei zu einer *IOException* kommen kann, kapseln wir dieses Statement zusammen mit dem eigentlichen Schreibvorgang und dem Schließen des Streams in einem *try-catch*-Block – schließlich kann es immer geschehen, dass das Verzeichnis nicht existiert oder andere Fehler auftreten.

Das eigentliche Speichern erfolgt mit Hilfe der Methode `store()`. Deren ersten Parameter stellt besagter *FileOutputStream* dar, mit Hilfe des zweiten Parameters sind wir in der Lage, einen Kommentar anzugeben. Nach dem Schließen des Streams können wir die Datei in einem Text-Editor öffnen und werden folgenden Inhalt vorfinden:

#Data for Samaschke

#Mon Jul 19 00:02:11 CEST 2004

Name=Samaschke

FirstName=Karsten

City=Berlin

Die Daten sind demnach erfolgreich abgelegt worden. Und der Kommentar, zu dem uns die Methode `store()` genötigt hat, befindet sich ganz am Anfang der Datei.

4.4.4 Speichern der Werte in XML-Dateien

Ebenso einfach wie das Speichern der Daten in eine Textdatei stellt sich die *Serialisierung* (das ist der Fachbegriff für das Speichern von Daten einer Instanz) in eine XML-Datei dar:

Listing 4.17
Serialisieren einer Properties-Instanz in eine XML-Datei

```
import java.util.Properties;
import java.io.IOException;
import java.io.FileOutputStream;

/**
 * Saves the contents of a Properties-instance to a xml file
 */
public class Properties_SaveToXml {

   public static void main(String[] args) {
      Properties props = new Properties();
      props.setProperty("Name", "Samaschke");
      props.setProperty("FirstName", "Karsten");
      props.setProperty("City", "Berlin");

      try {
         FileOutputStream fos =
            new FileOutputStream("data/samaschke.xml");
```

Listing 4.17 *(Forts.)* Serialisieren einer Properties-Instanz in eine XML-Datei

```
            props.storeToXML(fos, "Data for Samaschke");
            fos.close();
        } catch (IOException ignored) {}
    }
}
```

Nach dem Erstellen von `Properties`-Instanz und dem Zuweisen von Werten erzeugen wir – ebenso wie im letzten Beispiel – eine neue `FileOutputStream`-Instanz, die diesmal auf die Datei *samaschke.xml* im Unterverzeichnis *data* der aktuellen Applikation zeigt.

Diese Instanz der `FileOutputStream`-Klasse wird als erster Parameter der Methode `storeToXML()` der `Properties`-Klasse übergeben. Der zweite Parameter bezeichnet einen beliebig wählbaren Kommentar, der in der Datei abgelegt wird.

Wenn wir die Klasse ausgeführt haben, werden wir im Unterverzeichnis *data* die Datei *samaschke.xml* vorfinden, die folgenden Inhalt besitzt:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">

<properties>

<comment>Data for Samaschke</comment>

<entry key="Name">Samaschke</entry>

<entry key="FirstName">Karsten</entry>

<entry key="City">Berlin</entry>

</properties>

Standardmäßig wird als Encoding im Kopf der Datei *UTF-8* angegeben. Dies sollte für die meisten Einsatzzwecke genau richtig sein – manchmal allerdings ist man gezwungen, ein anderes Encoding zu verwenden. Aus diesem Grund bietet die Methode `storeToXML()` eine Überladung an, die einen dritten Parameter akzeptiert, der das auszugebende Encoding repräsentiert:

```
props.storeToXML(fos, "Data for Samaschke", "iso-8859-1");
```

4.4.5 Laden der Werte aus Textdateien

Wenn Sie eine *Properties*-Instanz in eine Textdatei serialisiert haben, werden Sie in den meisten Fällen versucht sein, diese Informationen auch wieder einzulesen. Aus diesem Grund stellt die `Properties`-Klasse die Methode `load()` zur Verfügung, die die Informationen aus einem *Stream* einliest und verarbeitet:

Listing 4.18 Laden von Properties-Daten aus einer Textdatei

```
import java.util.Properties;
import java.io.FileInputStream;
import java.io.IOException;

/**
 * Loads the content of a Properties-instance from a file
 */
```

Listing 4.18 *(Forts.)*
Laden von Properties-Daten aus einer Textdatei

```
public class Properties_Load {

   public static void main(String[] args) {
      Properties props = new Properties();
      try {
         FileInputStream in =
            new FileInputStream("data/samaschke.props");
         props.load(in);
         in.close();
      } catch (IOException ignored) {}

      System.out.println(
         String.format("Name: %s",
            props.getProperty("Name")));
      System.out.println(
         String.format("First name: %s",
            props.getProperty("FirstName")));
      System.out.println(
         String.format("City: %s",
            props.getProperty("City")));
   }
}
```

Das Vorgehen beim Laden der serialisierten Daten ist einfach: Zunächst erzeugen wir eine neue *Properties*-Instanz, die die Daten aufnehmen wird. Anschließend instanzieren wir einen `FileInputStream`, dem wir als Parameter den Namen der zu ladenden Datei übergeben. Diese `FileInputStream`-Instanz übergeben wir der Methode `load()` der `Properties`-Klasse als Parameter. Nach dem Laden der Daten und dem Schließen des *Streams* können wir den Inhalt der `Properties`-Instanz ausgeben – das Ergebnis sollte uns nicht überraschen:

Name: Samaschke

First name: Karsten

City: Berlin

4.4.6 Laden der Werte aus XML-Dateien

Exakt genauso einfach wie das Laden einer Text-Datei gestaltet sich das Laden der Daten aus einer XML-Datei:

Listing 4.19
Laden von Daten aus einer XML-Datei

```
import java.util.Properties;
import java.io.FileInputStream;
import java.io.IOException;

/**
 * Loads the contents of a Properties-instance from a xml file
 */
public class Properties_LoadXML {

   public static void main(String[] args) {
      Properties props = new Properties();
```

Listing 4.19 *(Forts.)* Laden von Daten aus einer XML-Datei

```
        try {
           FileInputStream in =
              new FileInputStream("data/samaschke.xml");
           props.loadFromXML(in);
           in.close();
        } catch (IOException e) {
           System.out.println(e.getMessage());
        }

        System.out.println(
           String.format("Name: %s",
              props.getProperty("Name")));
        System.out.println(
           String.format("First name: %s",
              props.getProperty("FirstName")));
        System.out.println(
           String.format("City: %s",
              props.getProperty("City")));
     }
}
```

Die Vorgehensweise beim Laden der Properties-Daten aus einer XML-Datei ist fast identisch zum Laden von Daten aus einer Text-Datei: Zunächst erzeugen wir eine `Properties`-Instanz. Anschließend wird ein `FileInputStream` mit dem Namen der zu ladenden Datei als Parameter erzeugt. Das eigentliche Laden der Daten erfolgt mit Hilfe der Methode `loadFromXML()` der `Properties`-Klasse, die als Parameter den erzeugten *Stream* erwartet. Nach dem Schließen des Eingabestroms können die geladenen Informationen ausgegeben werden.

Die `Properties`-Klasse sollten Sie immer dann einsetzen, wenn Sie Namen-/Werte-Paare aus `Strings` verwalten und möglichst einfach speichern wollen. Wenn Ihnen das Laden und Speichern nicht so wichtig ist und Sie auf die Definition von Default-Werten verzichten können, sollten Sie statt zur `Properties`-Klasse lieber zu einer typisierten `HashMap` greifen – die Deklaration einer `HashMap` in der Form

```
HashMap<String, String> =
    new HashMap<String,
    String>();
```

stellt Ihnen eine ähnliche Funktionalität wie die `Properties`-Klasse bereit, vermeidet aber die geschilderten Nachteile aufgrund der älteren `Hashtable` als *Superklasse*.

4.4.7 Vor- und Nachteile der Properties-Klasse

Die `Properties`-Klasse stellt eine einfache Methode dar, um beispielsweise Programmeinstellungen oder einfache Namen-/Werte-Paare zu verwalten. Sie erbt von der `Hashtable`, was einerseits ein Vorteil ist, da sie deren Funktionalität nutzen kann, andererseits aber ein echter Nachteil werden kann, da ein Zugriff auf die Methoden `get()` und `put()` der `Hashtable` weiterhin möglich ist – und diese Methoden nehmen beliebige Objekte statt `Strings` als Parameter entgegen. Dies kann in der Folge zu undefinierten Zuständen der Liste führen.

Der große Vorteil der `Properties`-Klasse ist ihre Fähigkeit, sich schnell und einfach zu serialisieren – egal, ob in Form einer Text- oder einer XML-Datei. Und ebenso einfach können die serialisierten Informationen wieder geladen werden.

4.5 Weitere Listen

Neben `ArrayList`, `Properties` und `HashMap` stellt uns Java noch weitere, teils besonders spezialisierte Listen bereit. Hier finden Sie eine kurze Übersicht der wichtigsten Elemente.

4.5.1 HashSet

Einfache Implementierung des *Set*-Interfaces. Im Hintergrund arbeitet eine *HashMap*. Das *HashSet* eignet sich für die Erfassung von Daten, auf die mit Hilfe eines *Iterators* zugegriffen werden soll.

4.5.2 TreeSet

Implementierung des *Set*-Inferfaces. Sortiert die Daten intern in „natürlicher" Reihenfolge – Strings beispielsweise alphabetisch, Zahlen nach ihrem Wert. Eignet sich für sortierte Daten, auf die mit Hilfe eines *Iterators* zugegriffen werden soll.

4.5.3 LinkedList

Implementierung der *List*- und *Queue*-Interfaces. Bietet spezielle Methoden, um das jeweils erste oder letzte Element der Liste zu erhalten. Kann eine bessere Performance als die *ArrayList* bieten, wenn Elemente häufig hinzugefügt oder gelöscht werden.

4.5.4 TreeMap

Implementierung des *Map*-Interfaces analog zur *HashMap*. Garantiert, dass die Elemente in „natürlicher" Reihenfolge gehalten und abgerufen werden können.

4.5.5 LinkedHashMap

Implementierung des *Map*-Interfaces analog zu *HashMap*. Sortiert die Elemente in der Reihenfolge des Einfügens. Kann deshalb eine bessere Performance als die *HashMap* bieten, wenn Elemente häufig hinzugefügt oder entfernt werden müssen.

4.5.6 Vector

Ältere *ArrayList*-Implementierung. Bietet keine Vorteile gegenüber der *ArrayList*. Wird für Abwärtskompatibilität mit älteren Java-Versionen (1.0, 1.1) bereitgestellt. Heutzutage sollte stattdessen die *ArrayList* verwendet werden.

4.6 Wichtige Interfaces

Da Listen in der Regel eine eigene Logik beinhalten, kommt die generelle Ableitung von Basisklassen nicht in Frage. Stattdessen implementieren die meisten Listen einige Interfaces, von denen die wichtigsten hier besprochen werden sollen.

4.6.1 Enumeration

Klassen, die das Interface *Enumeration* implementieren, bieten uns die Möglichkeit, eine Serie von Elementen sequentiell zu durchlaufen. Dabei wird mit Hilfe der Methode `hasMoreElements()` überprüft, ob weitere Elemente durchlaufen werden können. Das jeweils nächste Element wird mit Hilfe von `nextElement()` abgerufen.

Beim Durchlaufen einer *Enumeration* können keine Werte verändert oder gelöscht werden. Eine *Enumeration* bietet uns einen vorwärtsgerichteten und schreibgeschützten Ansatz für das Abrufen von Daten.

Eine Klasse, die das Interface *Enumeration* implementiert, ist die `Properties`-Klasse. Wenn wir aus dieser alle Schlüssel abrufen wollten, könnten wir dies mit folgendem Code erledigen:

```
import java.util.Properties;
import java.util.Enumeration;

Properties props = new Properties();
props.setProperty("Name", "Samaschke");
props.setProperty("FirstName", "Karsten");
props.setProperty("City", "Berlin");

Enumeration keys = props.keys();
while(keys.hasMoreElements()) {
   String key = (String)keys.nextElement();
   System.out.println(
      String.format("%s = %s", key, props.getProperty(key)));
}
```

In diesem Beispiel deklarieren wir zunächst eine neue *Properties*-Instanz, der wir einige Werte zuweisen. Mit Hilfe der Methode `keys()` der *Properties*-Instanz ermitteln wir alle in ihr enthaltenen Schlüssel und weisen diese der lokalen *Enumeration* `keys` zu. Der Inhalt von `keys` wird nun so lange abgerufen, bis keine weiteren Elemente mehr zurückgeliefert werden – wir verwenden dafür eine *while*-Schleife, die so lange läuft, bis `keys.hasMoreElements()` *false* zurückgibt.

Innerhalb der *while*-Schleife rufen wir das nächste Element mit Hilfe der Methode `nextElement()` der *Enumeration* ab. Da dieses Element jedoch als *Object*-Instanz zurückgegeben wird, müssen wir es noch in einen *String* casten, bevor wir es der lokalen Variablen `key` zuweisen können.

Zuletzt geben wir den Schlüssel und seinen Wert aus. Den Wert ermitteln wir übrigens mit Hilfe der Methode `getProperty()` der `Properties`-Instanz, der wir den Schlüssel als Parameter übergeben.

Wenn wir dieses Beispiel ausführen würden, sollten wir folgende Ausgabe erhalten:

Name = Samaschke

FirstName = Karsten

City = Berlin

4.6.2 Iterator

Das Interface *Iterator* wird gerne auch als Nachfolger der *Enumeration* bezeichnet. Genau wie das Interface *Enumeration* bietet es uns die Möglichkeit, eine Serie von Elementen abzurufen und zu durchlaufen. Es gibt zwei prägnante Unterschiede zwischen einer *Iterator*- und einer *Enumeration*-Instanz:

- Die Methodennamen des Iterators sind kürzer
- Der Iterator verfügt über eine Methode remove(), die das aktuelle Element aus der Aufzählung entfernt

Die Empfehlung von Sun lautet, bei eigenen Implementierungen nach Möglichkeit immer einen *Iterator* statt einer *Enumeration* zu verwenden.

Ein typischer Einsatzzweck eines *Iterators* ist das Durchlaufen aller Elemente einer *ArrayList*. Sehen wir uns an, wie dies vonstatten gehen könnte:

```
import java.util.ArrayList;
import java.util.Iterator;

ArrayList<Person> list = new ArrayList<Person>();
list.add(new Person("Samaschke", "Karsten"));
list.add(new Person("McNealy", "Scott"));
list.add(new Person("Torvalds", "Linus"));
list.add(new Person("Gates", "Bill"));
list.add(new Person("Jobs", "Steve"));

Iterator<Person> it = list.iterator();
while(it.hasNext()) {
   Person current = it.next();

   System.out.println('
      String.format("Current Person: %s, %s",
         current.getName(), current.getFirstName())
   );
}
```

Mit Hilfe von *Generics* definieren wir am Anfang des Listings eine *ArrayList* vom Typ *Person*:

```
ArrayList<Person> list = new ArrayList<Person>();
```

Der *ArrayList*-Instanz weisen wir anschließend einige Werte zu.

Nun kommt der *Iterator* ins Spiel: Wir deklarieren die Variable `it` als *Iterator* vom Typ *Person* und weisen ihr die *Iterator*-Instanz der *ArrayList*-Instanz zu, die wir per `list.iterator()` erreichen können:

```
Iterator<Person> it = list.iterator();
```

Jetzt können wir den Inhalt der *Iterator*-Instanz so lange laufen lassen, bis sie keine weiteren Elemente enthält. Wir können dies mit Hilfe der Methode `hasNext()` feststellen und nutzen es in einer *while*-Schleife:

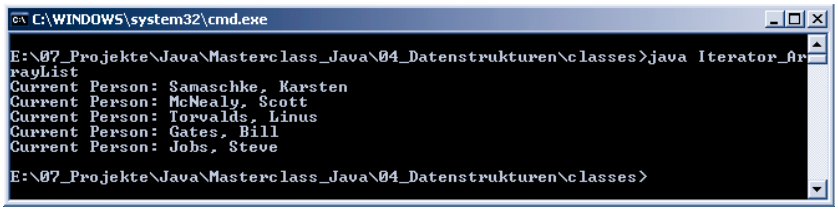
```
while(it.hasNext()) { ... }
```

Im Körper der Schleife weisen wir der lokalen Variablen `current` das aktuelle Element des *Iterators* zu. Dieses Element ermitteln wir mit Hilfe der Methode `next()` der Implementierung:

```
Person current = it.next();
```

Zur Kontrolle lassen wir die ermittelten Informationen ausgeben:

Abbildung 4.2
Ausgabe aller Elemente einer ArrayList mit Hilfe eines Iterators



Falls Ihnen die Arbeit mit dem *Iterator*-Interface nicht so zusagen sollte: Mit Version 1.5 verfügt Java auch über die Möglichkeit, Werte per *for-each*-Schleife auszugeben. Damit verringert sich der Schreibaufwand beim Abrufen von Werten aus einer *ArrayList* noch einmal ein wenig, allerdings ist die *for-each*-Schleife auch nicht so flexibel wie ein *Iterator*.

4.6.3 Collection

Das Interface *Collection* definiert die Basis-Funktionalität, die von allen Java-Collections implementiert werden muss. Eine *Collection*-Instanz repräsentiert eine Gruppe von Elementen, die sie enthält.

Im Java SDK existiert keine explizite Implementierung des *Collection*-Interfaces – stattdessen wird *Collection* als Basis-Interface für ableitende Interfaces, etwa *Set* oder *List*, verwendet.

Wenn Sie irgendwo auf Methoden stoßen, die *Collection* als Parameter erwarten oder zurückgeben, dient dies in der Regel nur dem vereinfachten Austausch von Daten.

Das *Collection*-Interface definiert folgende Methoden:

- `boolean add(E o)`: Fügt ein Element der Liste hinzu.
- `boolean addAll(Collection<? extends E> c)`: Fügt die Elemente einer Collection der Liste hinzu.
- `void clear()`: Entfernt alle Elemente der Liste.
- `boolean contains(Object o)`: Ermittelt, ob das übergebene Element in der Liste enthalten ist.
- `boolean containsAll(Collection<?> c)`: Ermittelt, ob alle Elemente der übergebenen Liste in der Collection enthalten sind.
- `boolean equals(Object o)`: Vergleicht das übergebene Objekt mit der Liste. Prüft auf Gleichheit.
- `int hashCode()`: Gibt den Hash-Code für diese Collection zurück.
- `boolean isEmpty()`: Gibt true zurück, wenn die Collection leer ist.

- `Iterator<E> iterator()`: Gibt eine Iterator-Implementierung vom Typ der enthaltenen Elemente zurück.
- `boolean remove(Object o)`: Entfernt das übergebene Element aus der Collection.
- `boolean removeAll(Collection<?> c)`: Entfernt alle Elemente der übergebenen Liste aus der Collection.
- `boolean retainAll(Collection<?> c)`: Entfernt alle Elemente, die nicht in der übergebenen Liste enthalten sind, aus der Collection.
- `int size()`: Gibt die Anzahl der enthaltenen Elemente zurück.
- `Object[] toArray()`: Gibt die Elemente der Collection als Array zurück.
- `<T> T[] toArray(T[] a)`: Gibt ein Array zurück, das alle Elemente der Collection beinhaltet. Der Runtime-Typ des Arrays entspricht dabei dem Runtime-Typ des übergebenen Arrays.

Wie bereits erwähnt, werden Sie bei den von Java mitgelieferten Klassen keine einzige direkte Implementierung von *Collection* finden – stattdessen werden Ihnen wesentlich häufiger `Set`- und `List`-Implementierungen unter die Augen kommen.

4.6.4 Map

Das Interface *Map* beschreibt eine Struktur, die aus Schlüssel-/Wert-Paaren besteht. Klassen, die *Map* implementieren, können keine doppelten Schlüssel aufnehmen und jeder Schlüssel muss auf mindestens einen Wert verweisen. Das Interface *Map* ersetzt die abstrakte Klasse *Dictionary*, da es einen generischeren Ansatz verfolgt.

Das *Map*-Interface erlaubt drei Sichten auf die enthaltenen Daten: als Satz von Schlüsseln, als Satz von Werten und als Schlüssel-/Werte-Paare. Die Reihenfolge der Schlüssel und Werte ist immer abhängig vom jeweiligen *Iterator*, der die Werte zurückgibt.

Das Interface *Map* deklariert folgende Methoden, die von implementierenden Klassen umgesetzt werden müssen:

- `void clear()`: Entfernt alle Mappings.
- `boolean containsKey(Object key)`: Prüft, ob der angegebene Schlüssel existiert.
- `boolean containsValue(Object value)`: Prüft, ob auf den angegebenen Wert gemappt wird.
- `Set<Map.Entry<K, V>> entrySet()`: Gibt eine `Set`-Implementierung zurück, die aus `Map.Entry`-Elementen besteht. Die Typen von Schlüssel und Wert des `Map.Entry`-Objekts entsprechen dabei denen der Map-Instanz – ist diese als `Map<String, String>` (Map, deren Schlüssel und Werte vom Typ `String` sind) deklariert, wird das `Map.Entry`-Element auch ausschließlich `Strings` für Schlüssel und Wert zurückgeben.
- `boolean equals(Object o)`: Stellt fest, ob die aktuelle Map-Instanz dem übergebenen Objekt entspricht.

Seit Java 1.5 und dessen Unterstützung für die so genannten *Generics* sind in der Notation von Elementen neue Typen hinzugekommen, die auf den ersten Blick recht verwirrend wirken, uns aber darüber informieren, dass hier typsicher gerarbeitet werden kann.

Einige dieser Notations-Elemente sind:

- **`E`: Steht für ein Element eines bestimmten Typs. Der Typ des Elements wird in der Regel bei der Deklaration des übergeordneten Elements bestimmt. Wenn wir eine ArrayList als `ArrayList<String>` (sprich: Array-List vom Typ `String` – nimmt nur String-Werte entgegen) definieren, entspricht `E` demTyp `String`. Definieren wir die ArrayList als `ArrayList<Long>` (ArrayList vom Typ `Long` – akzeptiert nur Long-Parameter) wäre `E` vom Typ `Long`.**
- **`K`: Steht bei einer Schlüssel-/Wert-Auflistung für den Typ des Schlüssels.**
- **`V`: Steht bei einer Schlüssel-/Wert-Auflistung für den Typ des Wertes.**
- **`<? extends E>`: Steht für ein Element eines unbekannten oder nicht definierten Typs, das vom angegebenen Typ erbt oder dem angegebenen Typ entspricht.**
- **`<?>`: Steht für einen unbekannten Typ.**
- **`<T>`: Steht für einen bestimmten Typ – etwa `String` oder `Long`.**

- `V get(Object key)`: Gibt das durch den übergebenen Schlüssel bezeichnete Element zurück.
- `int hashCode()`: Liefert den HashCode der aktuellen Instanz.
- `Set<K> keySet()`: Liefert eine `Set`-Implementierung zurück, die aus allen Schlüsseln der HashMap besteht.
- `V put(K key, V value)`: Fügt das angegebene Schlüssel-/Werte-Paar der Map hinzu. Schlüssel und Wert müssen dabei vom Typ sein, der bei der Deklaration der Map angegeben worden ist. Ist diese beispielsweise als `Map<String, String>` (Map, deren Schlüssel und Werte vom Typ `String` sind) deklariert, dürfen nur Schlüssel und Werte vom Typ String hinzugefügt werden.
- `void putAll(Map<? extends K, ? extends V>)`: Fügt die Schlüssel und Werte einer Map, deren Schlüssel und Werte den Typen der aktuellen Map-Instanz entsprechen oder von diesen abgeleitet worden sind, der aktuellen Map-Instanz hinzu.
- `V remove(Object key)`: Entfernt das durch den als Parameter übergebenen Schlüssel gekennzeichnete Element aus der Map und gibt es zurück.
- `int size()`: Gibt die Anzahl der Schlüssel-/Werte-Paare zurück.
- `Collection<V> values()`: Gibt eine Collection der Werte zurück.

Klassen, die *Map* implementieren, haben wir bereits in diesem Kapitel kennen gelernt – etwa die *HashMap*.

4.6.5 List

Das Interface *List* erweitert das Interface *Collection* um Methoden, mit deren Hilfe Daten auf Basis eines Index-Wertes zugewiesen oder abgerufen werden können.

Dieser Index ist 0-basierend – das erste Element hat den Index 0, das zweite Element 1 und so weiter.

Klassen, die *List* implementieren, erlauben in der Regel die Zuweisung von doppelten Werten und können auch *null* als Wert enthalten.

Das Interface *List* definiert neben den von *Collection* ererbten Methoden folgende eigene Member:

- `void add(int index, <E> o)`: Fügt ein Element an der angegebenen `Index`-Position ein.
- `void addAll(int index, Collection<? extends E> c)`: Fügt die Elemente der übergebenen Collection ab der durch `index` bezeichneten Position ein.
- `Object get(int index)`: Gibt das Element an der angegebenen Position zurück.
- `int indexOf(Object o)`: Gibt das erste Vorkommen des übergebenen Objekts in der Liste zurück. Gibt -1 zurück, wenn die Liste das Objekt nicht enthält.
- `int lastIndexOf(Object o)`: Gibt das letzte Vorkommen des übergebenen Objekts in der Liste zurück. Gibt -1 zurück, wenn die Liste das Objekt nicht enthält.

- `ListIterator<E> listIterator()`: Gibt einen `ListIterator` für die Elemente der Liste zurück.
- `ListIterator<E> listIterator(int index)`: Gibt einen `ListIterator` für alle Elemente der Liste zurück, die am oder hinter dem angegebenen `Index` liegen.
- `E remove(int index)`: Entfernt das Element an der angegebenen Position und gibt es zurück.
- `E set(int index, E element)`: Ersetzt das Element an der durch `index` angegebenen Position mit dem übergebenen Element und gibt das ersetzte Element zurück.
- `List<E> subList(int fromIndex, int toIndex)`: Gibt eine Liste aller Elemente zurück, die sich an den Positionen zwischen `fromIndex` und `toIndex` befinden. Das Element an der Position fromIndex wird dabei eingeschlossen, das Element an der Position toIndex nicht.

Das *List*-Interface wird bei allen indexbasierenden *Collections* implementiert. Dazu gehören beispielsweise *ArrayList*, *Vector* und *LinkedList*.

4.6.6 Set

Das Interface *Set* stellt eine Ableitung von *Collection* dar. Es definiert keine neuen Methoden, sondern wird für alle Listen verwendet, die keine doppelten Elemente und keine *null*-Werte zulassen.

Mit einem *Set* können Sie wie mit einer *Collection* arbeiten, sollten aber großen Wert auf eine Prüfung der zu übergebenden Werte legen – diese dürfen nicht *null* und nicht bereits in der Liste enthalten sein!

Die einfachste Möglichkeit, über ein *Set* zu stolpern, bieten uns das Interface *Map* und dessen Ableitungen. Die verfügen über eine Methode `entrySet()`, die uns alle enthaltenen Schlüssel-/Werte-Mappings als *Set*-Implementierung zurückgibt:

Listing 4.20
Durchlaufen einer HashMap mit Hilfe einer Set-Implementation

```
import java.util.HashMap;
import java.util.Set;
import java.util.Map;
import java.util.Iterator;

/**
 * Uses a Set to iterate through the contents of a Map
 * implementation
 */
public class EntrySet_HashMap {

   public static void main(String args[]) {
      HashMap<String, Person> map =
         new HashMap<String, Person>();

      map.put("Samaschke", new Person("Samaschke", "Karsten"));
      map.put("McNealy", new Person("McNealy", "Scott"));
      map.put("Torvalds", new Person("Torvalds", "Linus"));
```

Listing 4.20 *(Forts.)*
Durchlaufen einer HashMap mit Hilfe einer Set-Implementation

```
        map.put("Gates", new Person("Gates", "Bill"));
        map.put("Jobs", new Person("Jobs", "Steve"));

        Set<Map.Entry<String, Person>> entries = map.entrySet();
        Iterator<Map.Entry<String, Person>> it =
           entries.iterator();

        while(it.hasNext()) {
           Map.Entry<String, Person> entry = it.next();
           Person person = entry.getValue();

           System.out.println(
              String.format("Current person is %s, %s",
                 person.getName(), person.getFirstName())
           );
        }
     }
}
```

In diesem Beispiel erzeugen wir zunächst eine *HashMap*, die als Schlüssel *String*- und als Werte *Person*-Instanzen entgegennimmt:

```
HashMap<String, Person> map = new HashMap<String, Person>();
```

Nun fügen wir einige Daten hinzu und rufen dann eine *Set*-Implementierung ab. Die einzelnen Elemente des *Sets* sind *Map.Entry*-Instanzen, die wiederum der Deklaration der *Map* entsprechen, aus der sie entstammen: Sie lassen ebenfalls nur *Strings* als Schlüssel und `Person`-Instanzen als Werte zu.

Etwas schwierig lesbar wird es, weil wir gezwungen sind, die Deklarationen der erlaubten Typen zu schachteln:

```
Set<Map.Entry<String, Person>> entries = map.entrySet();
```

Auch im danach deklarierten *Iterator* spiegelt sich die geschachtelte Typ-Deklaration wider – er gibt *Map.Entry*-Instanzen zurück, die *Strings* und *Person*-Instanzen für Schlüssel und Wert erlauben:

```
Iterator<Map.Entry<String, Person>> it = entries.iterator();
```

Nun wird es wieder etwas leichter fassbar: Wir werden den *Iterator* in einer *while*-Schleife so lange durchlaufen, bis er keine weiteren Elemente beinhaltet und seine Methode `hasNext()` den Wert *false* zurückgibt:

```
while(it.hasNext()) { ... }
```

Das jeweils aktuelle Element des *Iterators* wird nun der lokalen Variable `entry` vom Typ *Map.Entry* zugewiesen. Dazu bedienen wir uns der Methode `next()` der *Iterator*-Instanz:

```
Map.Entry<String, Person> entry = it.next();
```

Eine derartige *Map.Entry*-Instanz erlaubt es uns, sowohl den Schlüssel als auch den zugehörigen Wert abzurufen – den Schlüssel liefert uns die Methode `getKey()` und den Wert die Methode `getValue()`.

Wir interessieren uns hier nur für den Wert des Elements. Dieser Wert muss vom Typ *Person* sein, weshalb wir ihn problem- und bedenkenlos einer *Person*-Instanz zuweisen können:

```
Person person = entry.getValue();
```

Nun können Vorname und Nachname der repräsentierten Person ausgegeben werden:

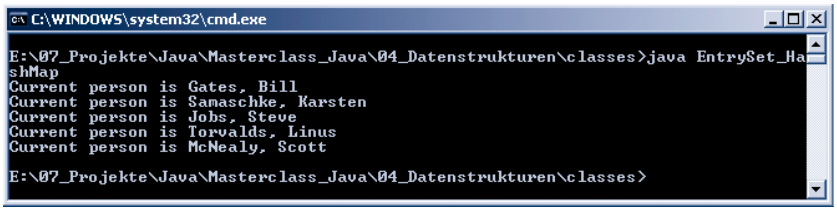


Abbildung 4.3
Ausgabe von Daten mit Hilfe eines Sets

4.7 Generics

Wir haben im Verlauf dieses Kapitels bereits mehrmals das Konzept der *Generics* (generische Definition) angesprochen und auch in der Praxis demonstriert. Dennoch wollen wir an dieser Stelle noch einmal kurz zusammenfassen, was *Generics* sind und wie sie uns helfen.

4.7.1 Was sind Generics?

Generics sind eine allgemeine Erweiterung von Java, die grundsätzlich für beliebige Klassen verwendet werden können. Der Begriff bezeichnet eine Technologie, die es uns erlaubt, Klassen mit Informationen darüber zu versehen, welche Datentypen sie zulassen. Werden andere Datentypen als die definierten oder davon abgeleitete Typen übergeben, wird eine Exception geworfen und die Zuweisung ist nicht möglich.

Der Vorteil des Konzepts: Größere Typsicherheit bei der Entwicklung und die Verringerung von ClassCastExceptions zur Laufzeit.

4.7.2 Was bringen Generics?

Generics sind ein einfaches Konzept, das gerade dadurch so mächtig wird. Ihr Sinn liegt darin, nicht mehr mit dem generischen `Object`-Datentyp im Zusammenhang mit `Collection`- und `Map`-Instanzen arbeiten zu müssen, sondern explizit angeben zu können, welche Datentypen erwartet werden.

Wir sparen uns so jede Menge Programmier- und Denk-Arbeit, denn schliesslich fällt das oftmals langwierige und staubtrockene Erstellen eigener `Collection`- oder `Map`-Ableitungen in den meisten Fällen ebenso weg, wie ein Casting der Elemente von und nach `Object`.

Das Beste beim Einsatz von *Generics*: Java übernimmt die ganze Arbeit für uns! Wir müssen die Datentypen nur noch angeben – den Rest erledigen Compiler und Interpreter.

4.7.3 Wie werden die Datentypen angegeben?

Die erlaubten Datentypen werden bei `Collection`- und `Map`-Instanzen in spitzen Klammern direkt nach dem Listentyp angegeben, Wenn Sie eine `ArrayList` aus `Person`-Objekten deklarieren wollten, sollte dies dann beispielsweise so aussehen:

```
ArrayList<Person> list = new ArrayList<Person>();
```

Jeder Versuch, der ArrayList `list` Elemente, die nicht vom Typ *Person* oder von diesem zumindest abgeleitet sind, zu übergeben, ist zum Scheitern verurteilt. Schon der Compiler wird Sie deutlich auf Ihren Fehler hinweisen.

Das gleiche Vorgehen wenden Sie an, wenn Sie beispielsweise `Map`-Implementierungen deklarieren wollten. Der einzige Unterschied zu `Collection`-Ableitungen besteht darin, dass `Map`-Ableitungen zwei Datentypen akzeptieren: Eine Datentyp-Angabe für die Schlüssel und eine Angabe für die Werte:

```
HashMap<String, Person> map = new HashMap<String, Person>();
```

Die Angaben für die erlaubten Datentypen trennen Sie mit einem Komma.

Achtung

Eine Angabe von alternativen Datentypen oder mehreren Datentypen für ein Element ist nicht möglich! Sie können also nicht angeben, dass eine `Collection`-Ableitung entweder `Person`- oder `String`-Elemente zulässt. Sie sollten in einem derartigen Fall auf Generics verzichten oder ein gemeinsames Interface für Ihre Elemente definieren und dieses angeben.

Anders sieht der Fall aus, wenn ein Basistyp für Parameter oder Elemente vereinbart werden soll. Dann kann der Basistyp angegeben werden, und sowohl dieser Typ, als auch von ihm abgeleitete Typen werden als Parameter akzeptiert. Um beispielsweise eine `ArrayList` zu verwenden, die nur Zahlen aufnimmt, dabei aber keine Festlegung auf den konkreten Typ der Zahl (etwa `Long`, `Double` oder `Integer`) trifft, könnte man diese Deklaration verwenden:

```
ArrayList<Number> numberList = new ArrayList<Number>();
```

Dies funktioniert übrigens nicht nur mit Klassen oder Datentypen, sondern auch mit Interfaces.

4.7.4 Müssen Generics zwingend deklariert werden?

Nein, Generics müssen nicht zwingend deklariert werden. Der Compiler verwendet implizit die Klasse `Object`, wenn kein konkreter Datentyp angegeben worden ist. Die Deklaration einer `ArrayList` als

```
ArrayList list = new ArrayList();
```

ist also mit der Deklaration einer `ArrayList` vom Typ `Object` gleichzusetzen:

```
ArrayList<Object> list = new ArrayList<Object>();
```

4.7.5 Ab welcher Java-Version sind Generics verfügbar?

Generics sind ab Java 5 verfügbar. In vorherigen Java-Versionen musste man eigene Ableitungen definieren, die entsprechende Prüfungen auf den Datentyp selbstständig vorgenommen haben. Die Wiederverwendbarkeit eigener Klassen war dadurch eingeschränkter.

4.7.6 Generics in eigenen Klassen und Methoden verwenden

Die Deklaration von Klassen und Methoden, die das Konzept der Generics nutzen, ist nicht allzu schwierig. Statt der Angabe eines konkreten Datentyps nutzen wir die Generics-Notation:

Listing 4.21
Deklaration einer Klasse, die Generics verwendet

```
public class GenericClass<T1, T2> {

   private T1 first;
   private T2 second;

   GenericClass(T1 first, T2 second) {
      this.first = first;
      this.second = second;
   }

   public T1 getFirst() {
      return first;
   }

   public T2 getSecond() {
      return second;
   }
}
```

Das Geheimnis der Deklaration von Klassen, die Generics verwenden, besteht darin, im Kopf der Klasse die Typen zu deklarieren:

```
public class GenericClass<T1, T2> {
```

Die so deklarierten Typen, die faktisch nichts anderes als Platzhalter sind und zunächst wie ein Object behandelt werden, können nun innerhalb der Klasse wie gewöhnliche Membervariablen verwendet werden – die durch sie repräsentierten Werte können etwa durch Getter zurückgegeben werden.

Die Instanzierung und Verwendung der Klasse `GenericClass` geschieht nun analog zur Verwendung von anderen Klassen, die das Konzept der Generics nutzen:

```
GenericClass<Integer, String> instance =
   new GenericClass<Integer, String>(10, "Hello!");
int first = instance.getFirst();
String second = instance.getSecond();
```

Statt auf Klassen-Ebene können Generics auch auf Methoden-Ebene verwendet werden:

```
public class GenericMethods() {

   public <T3> void process(T3 item) {
      // ...
   }

}
```

Wenn Generics auf Methoden-Ebene eingeführt werden sollen, muss deren Deklaration vor der Angabe des Rückgabe-Typs der Methode stehen. Dies gilt auch, wenn die Methode `void` als Rückgabe-Typ hat.

Generische Typen einschränken

Beim Einsatz von Generics ist es möglich, einen Basis-Typ zu definieren, etwa um ausschließlich Zahlen-Werte zuzulassen:

Listing 4.22
Definition einer Klasse, die nur Zahlen akzeptiert

```
public class NumberPair<T1 extends Number, T2 extends Number> {

   private T1 first;
   private T2 second;

   NumberPair(T1 first, T2 second) {
      this.first = first;
      this.second = second;
   }

   public T1 getFirst() {
      return first;
   }

   public T2 getSecond() {
      return second;
   }
}
```

Nun können nur noch Instanzen der `NumberPair`-Klasse erzeugt werden, die Zahlen verwenden.

Diese Deklarationen wären dabei beispielsweise gültig:

```
NumberPair<Double, Integer> =
   new NumberPair<Double, Integer>(10.5, 10);
NumberPair<Long, Byte> = new NumberPair<Long, Byte>(10000000, 1);
```

Diese Art von Deklaration wäre nicht mehr gültig:

```
NumberPair<Double, String> =
   new NumberPair<Double, String>(10.5, "Ten");
```

Die Einschränkung der generischen Typen auf einen bestimmten Basistyp geschieht unter Verwendung des `extends`-Schlüsselworts:

```
<T extends <Basis-Typ>>
```

Diese Art von Einschränkungen ist auch möglich, um etwa Listen zu erzeugen, die nur Zahlen als Parameter akzeptieren:

Listing 4.23
Deklaration einer Liste, die nur Zahlen akzeptiert

```
public class NumberList<T extends Number>
   extends ArrayList<T> {

   public double computeSum() {
      // ...
   }

   public double computeAverage() {
      // ...
   }
}
```

Die Methoden `computeSum()` und `computeAverage()` können nun auf simple Art und Weise implementiert werden, da sichergestellt ist, das in einer `NumberList`-Instanz stets nur Zahlen enthalten sind. Die Zuweisung anderer Werte würde mit entsprechenden Exceptions bestraft werden.

4.7.7 Nachteile von Generics

Der Einsatz von Generics sorgt für einen Nachteil im Hinblick auf Performance, denn der Compiler und Interpreter müssen ständig die Datentypen überprüfen. Außerdem findet intern ein Casting der Werte in die erlaubten Datentypen statt, was für weiteren Overhead sorgt. `Collection`- und `Map`-Instanzen arbeiten aus diesem Grund langsamer, als dies bei ihren nicht-generischen Vergängern der Fall war.

Die Vorteile bei der Entwicklung und Wartung von Applikationen erkauft man sich also mit Nachteilen im Hinblick auf das Laufzeitverhalten.

4.8 Zusammenfassung

Java gibt uns die Freiheit, `Collections` und `Maps` in der Form einzusetzen, wie wir es für richtig halten. Nahezu jede `Collection`- oder `Map`-Ableitung ist dabei leichter zu handhaben, als dies beim Einsatz von Arrays zum Halten gleichartiger Werte der Fall wäre.

Diese Vorteile werden allerdings mit Nachteilen im Hinblick auf Performance erkauft – insbesondere im Hinblick darauf, dass die meisten Listen bei Java 5 *Generics* verwenden. Wenn es also stark auf Performance ankommt, führt kaum ein Weg am simplen Array vorbei.

Aber Generics haben große Vorteile im Hinblick auf Typsicherheit und damit auf die generelle Applikationssicherheit, denn nunmehr ist es möglich, auch mit Listen

typsicher zu arbeiten und somit anzugeben, welche Datentypen erlaubt sind. Dies ist eine deutliche Verbesserung im Vergleich zu älteren Java-Versionen.

Generell bleibt festzuhalten: Der Einsatz von Generics bei den verschiedenen Listen-Arten ist ein zweischneidiges Schwert, denn die Vorteile bei der Entwicklung und bei der Typsicherheit werden mit Nachteilen im Hinblick auf Performance erkauft. Dies sollte Sie nach einer Abwägung der verschiedenen Aspekte jedoch nicht davon abhalten, in Ihren Applikationen vom Konzept der Generics Gebrauch zu machen – jedoch nur dort, wo es wirklich sinnvoll erscheint.

5

Ausnahmen

Ausnahmen stellen Zustände dar, die den Ablauf des Programms stören – das können Fehler, falsche Eingaben des Benutzers, nicht vorhandene Dateien oder andere Dinge sein. Derartige Ausnahmen können bedacht und abgefangen werden – und genau mit diesem Thema beschäftigt sich dieses Kapitel.

5.1 Ausnahmen werfen

In jedem Buch, das sich mit Programmierung und Ausnahmen befasst, wird man Ihnen schnell beibringen wollen, wie Sie Ausnahmen abfangen können. Hier nicht! Wir werden nun einfach einmal unsere destruktive Seite ausleben und eine Ausnahme in einem Programm werfen:

Listing 5.1
Werfen einer Exception

```
/**
 * Created by Karsten Samaschke
 */
public class ExceptionThrow {

   public static void main(String args[])
      throws Exception {
      System.out.println("Going to throw an exception...");
      if(1 == 1) {
         throw new Exception("Sample exception");
      }
      System.out.println("...exception thrown");
   }
}
```

Führen Sie die Klasse ohne weitere Erklärung einmal aus, nachdem Sie sie kompiliert haben:

Going to throw an exception...

java.lang.Exception: Sample exception

at ExceptionThrow.main(ExceptionThrow.java:9)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)

at java.lang.reflect.Method.invoke(Method.java:494)

at com.intellij.rt.execution.application.AppMain.main(AppMain.java:78)

Exception in thread "main"

So etwas nennt man ein Ende mit einem Knall – aber irgendwie sieht die Ausgabe dann doch nicht wirklich gesund aus.

Je nach Einstellung kann die Fehlermeldung auch weniger umfangreich ausfallen. Dennoch sollte Ihnen auffallen, dass die zweite Ausgabe *...exception thrown* nicht mehr ausgegeben worden ist. Offensichtlich wurde die Verarbeitung an der Stelle, an der wir die Ausnahme geworfen haben, unterbrochen.

Werfen wir nun einen etwas genaueren Blick auf den Code. Zunächst sollte uns ins Auge fallen, dass der Kopf der Methode `main()` etwas anders als gewohnt aussieht:

```
public static void main(String args[]) throws Exception
```

Neu ist, dass wir mit Hilfe des Schlüsselworts *throws* angeben, welche Ausnahmen die Methode werfen kann. Diese Angabe bezieht sich auf Ausnahmen, die vom Code selbst generiert werden oder die von verwendeten Klassen geworfen werden könnten und innerhalb der Methode nicht abgefangen werden. Mehrere Ausnahmen trennen wir einfach durch Kommata:

```
public static void main(String args[])
   throws ArithmeticException, NumberFormatException
```

Die eigentliche Ausgabe werfen wir mit Hilfe des Schlüsselworts *throw*:

```
throw new Exception("Sample exception");
```

Ausnahmen stellen offensichtlich Objekte dar. Diese können ebenso wie gewöhnliche Objekte deklariert werden und Werte aufnehmen. Die `Exception`-Klasse bietet uns dabei einen mehrfach überladenen Konstruktor, der – wie in unserem Beispiel gezeigt – unter anderem auch eine Zeichenkette aufnehmen kann, die dann als Fehlermeldungstext verwendet wird.

Fassen wir also kurz zusammen:

- Methoden, die Exceptions werfen sollen, müssen speziell gekennzeichnet werden
- Eine neue Exception wird wie ein Objekt erzeugt
- Code, der nach dem Werfen einer Exception folgt, wird nicht mehr ausgeführt

5.2 Ausnahmen abfangen

Nachdem wir nun also eine Ausnahme definiert haben, wollen wir sehen, wie wir sie abfangen und somit verhindern können, dass Fehler bis zum Benutzer durchschlagen.

Ausnahmen eskalieren nach oben – und zwar so lange, bis sie behandelt werden. Das bedeutet, wenn die Ausnahme in einer Methode nicht behandelt worden ist, dann wird sie an die aufrufende Methode weitergegeben und kann dort behandelt werden. Geschieht dies nicht, eskaliert die Ausnahme weiter nach oben – so lange, bis sie aufgefangen wird.

Sehen wir uns an, wie wir dies erreichen können:

```
/**
 * Created by Karsten Samaschke
 */
public class ExceptionCatch {

   public static void main(String args[]) throws Exception {
      System.out.println("Going to throw an exception...");
      try {
         throw new Exception("Sample exception");
      } catch (Exception e) {
         System.out.println(
            String.format("CATCHED EXCEPTION: %s",
               e.getMessage()));
      }
      System.out.println("...exception thrown and catched");
   }
}
```

Listing 5.2
Abfangen einer Exception

Der Hauptunterschied zum vorherigen Beispiel liegt darin, dass wir die Stelle, an der potentiell (und in unserem Fall mit absoluter Sicherheit) eine Ausnahme geworfen wird, in einem *try-catch*-Block behandeln. Die Syntax eines *try-catch*-Konstrukts sieht übrigens so aus:

```
try {
   // ...
} catch (<Exception> <Name>) {
   // ...
}
```

Innerhalb des *try*-Blocks können beliebig viele Anweisungen stehen, die unter dem Schutz des *catch*-Statements ausgeführt werden. Sollte in diesem Bereich eine *Exception* auftreten, wird diese vom *catch*-Block behandelt, so dieser eine zutreffende Signatur hat.

Klingt kompliziert? Ist es nicht. Sehen wir uns doch einfach die Ausgabe an, die von dieser Klasse generiert wird:

Going to throw an exception...
CATCHED EXCEPTION: Sample exception
...exception thrown and catched

5.2.1 Mehrere Ausnahmen behandeln

Ausnamen sind leider die Regel. Insbesondere bei der Programmierung. Wenn Sie mit mehreren Klassen in Ihren Lösungen arbeiten, kann es durchaus vorkommen, dass Sie mehrere Ausnahmen für ein Code-Fragment behandeln müssen.

Achtung

Erliegen Sie nicht der Versuchung, alle Ausnahmen mit einem generischen *try-catch*-Statement erschlagen zu wollen, das als Ausnahmetyp *Exception* hat:

```
try {
   // ...
} catch (Exception e) {
   // ...
}
```

Dieser Ansatz funktioniert zwar, ist jedoch wirklich übelste Art der Entwicklung – Sie fangen so nicht nur Ausnahmen ab, die von Ihrem Code geworfen werden, sondern auch Ausnahmen, die von der Runtime generiert worden sind und somit Ausnahmen auf System-Ebene entsprechen. Damit verbauen Sie Java die Möglichkeit, angemessen auf derartige Ereignisse zu reagieren, und führen möglicherweise eine generelle Instabilität der Java-Runtime herbei.

Behandeln Sie also nur die Ausnahmen, die Ihr Code oder von Ihrem Code verwendete Objekte generieren. Und wenn Sie diese Ausnahmen nicht behandeln wollen, deklarieren Sie sie explizit per *throws* im Methoden-Kopf.

Dies könnte dann beispielsweise so aussehen:

Listing 5.3
Mehrere Ausnahmen abfangen

```
/**
 * Created by Karsten Samaschke
 */
public class ExceptionMultiple {

   public static void main(String args[]) throws Exception {
      System.out.println("Going to throw an exception...");
      int first = (int) (10 * Math.random());
      int second = (int) (2 * Math.random());
      int result = 0;

      int secondResult = 0;
      String data = "abc";
      if((Math.random() * 2) > 0) {
         data = "1";
      }

      try {
         result = first / second;
         secondResult =
            Integer.parseInt(data);
```

Listing 5.3 *(Forts.)*
Mehrere Ausnahmen abfangen

```
        } catch (NumberFormatException e) {
            System.out.println(
                String.format("NumberFormatException: %s",
                    e.getMessage()));
        } catch (ArithmeticException e) {
            System.out.println(
                String.format("ArithmeticException: %s",
                    e.getMessage()));
        }

        System.out.println(
            String.format("%d / %d = %d", first, second, result));

        System.out.println(
            String.format("Parsed Integer: %d", secondResult));
    }
}
```

In diesem Beispiel erzeugen wir zwei Zufallszahlen – die Variable `first` erhält eine Zahl zwischen *0* und *10*, `second` dagegen eine Zufallszahl zwischen *0* und *2*. Mit Hilfe einer weiteren Zufallszahl bestimmen wir, ob die Variable `data` die Zeichenkette *„abc“* oder *„1“* als Inhalt bekommt.

Innerhalb des `try`-Blocks führen wir nun zwei Operationen durch: Zunächst dividieren wir die erste Zahl durch die zweite Zahl, was nur funktionieren wird, wenn die zweite Zahl ungleich *0* ist. Anschließend lassen wir die in `data` enthaltene Zeichenkette (*„abc“* oder *„1“*) in eine Zahl umwandeln. Logisch, dass dies nur gut gehen kann, wenn `data` keine Buchstaben enthält.

Potentiell können also innerhalb des `try`-Blocks zwei verschiedene Ausnahmen auftreten: Eine `ArithmeticException`, falls eine Division durch *0* stattfinden sollte, und eine `NumberFormatException` für den Fall, dass `data` keine gültige Zahl enthält.

Beide Ausnahmen fangen wir innerhalb getrennter `catch`-Blöcke ab. Dies bedeutet für uns, dass sich offensichtlich ein `catch`-Block an einen anderen `catch`-Block anschließen kann. Die Syntax dafür ist sehr einfach:

```
try {
    // ...
} catch (<Exception> <Name>) {
    // ...
} catch (<AndereException> <Name>) {
    // ...
}
```

Das getrennte Abfangen der Ausnahmen erlaubt es uns, gezielter auf die einzelnen Zustände oder Fehler zu reagieren. In unserem Beispiel ist diese Reaktion auf eine entsprechende Ausgabe beschränkt – in natura werden Sie sicherlich korrekte Fehlerbehandlungen implementieren.

Wenn Sie das Beispiel ausführen, erhalten Sie eine der folgenden Ausgaben:

ArithmeticException: / by zero

8 / 0 = 0

Parsed Integer: 0

Hier ist die zweite Zahl offensichtlich *0* gewesen. Daraufhin wurde eine `ArithmeticException` geworfen, die wir abgefangen und behandelt haben.

NumberFormatException: For input string: ”abc”

7 / 1 = 7

Parsed Integer: 0

In diesem Fall konnte die Division erfolgreich durchgeführt werden, allerdings scheiterte die Umwandlung von *„abc“* in eine Zeichenkette. Das Resultat: eine `NumberFormatException`.

Sollte alles einwandfrei funktionieren – also sowohl Division als auch Umwandlung erfolgreich durchgeführt werden –, sollte die Ausgabe ähnlich dieser aussehen:

9 / 1 = 9

Parsed Integer: 1

5.2.2 Das finally-Schlüsselwort

Wie auch immer Sie Ausnahmen abfangen – manchmal haben Sie Code, der in jedem Fall ausgeführt werden muss, und zwar unabhängig davon, ob ein Fehler auftritt oder nicht – etwa Aufräumaktionen oder ein Logging. Mit Hilfe des `finally`-Schlüsselwortes können Sie dies erzwingen:

Listing 5.4
Einatz des finally-Schlüsselworts

```
/**
 * Created by Karsten Samaschke
 */
public class ExceptionFinally {

   public static void main(String args[]) throws Exception {
      int first = (int) (10 * Math.random());
      int second = (int) (2 * Math.random());
      int result = 0;


      try {
         result = first / second;

         throw new Exception("Sample Exception!");
      } catch (ArithmeticException e) {
         System.out.println(
            String.format("ArithmeticException: %s",
               e.getMessage()));
      } finally {
         System.out.println("Finished!");
      }

      System.out.println(
         String.format("%d / %d = %d", first, second, result));
   }
}
```

In diesem Beispiel wird in jedem Fall eine Ausnahme geworfen – entweder die bekannte `ArithmeticException` aufgrund einer Division durch Null oder eine Standard-Ausnahme, die wir nach der Division werfen, aber nicht abfangen.

Welche Ausnahme auch immer geworfen, aber nicht abgefangen wird – die Ausgabe beinhaltet in jedem Fall den im `finally`-Block ausgegebenen Text:

Finished!

java.lang.Exception: Sample Exception!

at ExceptionFinally.main(ExceptionFinally.java:15)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessor Impl.java:39)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethod AccessorImpl.java:25)

at java.lang.reflect.Method.invoke(Method.java:494)

at com.intellij.rt.execution.application.AppMain.main(AppMain.java:78)

Exception in thread "main"

Interessant ist an der Stelle, dass die Ausnahme zwar vor dem *finally*-Block geworfen wurde, der im `finally`-Block enthaltene Code aber offensichtlich ausgeführt worden ist, bevor die Ausnahme an die übergeordnete Methode weitergegeben worden ist.

Machen Sie also Gebrauch vom `finally`-Block – besser können Sie die Ausführung bestimmter Code-Fragmente trotz möglicher Ausnahmen nicht erzwingen.

5.3 Eigene Ausnahmen definieren

Wie bereits weiter oben beschrieben, sind Ausnahmen auch nur Klassen – und zwar in der Regel Ableitungen der `Exception`-Klasse. Dies bedeutet für uns, dass wir durchaus auch eigene `Exception`-Ableitungen deklarieren und werfen können:

Listing 5.5
Eigene Exception-Klasse

```
/**
 * Created by Karsten Samaschke
 */
public class DivisionByZeroException
   extends ArithmeticException {

   private int first;
   private int second;

   public DivisionByZeroException(int first, int second) {
      this.first = first;
      this.second = second;
   }
```

Listing 5.5 *(Forts.)*
Eigene Exception-Klasse

```
    public int getFirst() {
       return first;
    }

    public int getSecond() {
       return second;
    }

    public String getMessage() {
       return "Division by zero is not allowed!";
    }
}
```

Wie jede andere Ausnahme auch, erbt die `DivisionByZeroException` von der Basis-Klasse `Exception` oder einer ihrer Ableitungen (in diesem Fall von der `ArithmeticException`). Wir deklarieren einen Konstruktor, der zwei *Integer*-Werte als Parameter erwartet – eben die beiden Werte, bei deren Division ein Fehler aufgetreten ist. Zudem überschreiben wir die Methode `getMessage()` der Basis-Klasse, so dass immer die Fehlermeldung zurückgegeben wird, dass eine Division durch Null nicht erlaubt ist.

Nun können wir unsere Exception werfen, falls eine Division durch Null versucht wird:

Listing 5.6
Diese Klasse wirft eventuell eine DivisionByZeroException und behandelt sie auch selbst

```
/**
 * Created by Karsten Samaschke
 */
public class ExceptionDivisonByZero {

   /**
    * Divides two randomly created Integers
    * @throws DivisionByZeroException
    */
   public void divide() throws DivisionByZeroException  {

      int first = (int) (10 * Math.random());
      int second = (int) (2 * Math.random());
      int result = 0;

      if(second == 0) {
         throw new DivisionByZeroException(first, second);
      }

      result = first / second;

      System.out.println(
         String.format("%d / %d = %d",
            first, second, result));
   }

   public static void main(String args[]) {
      try {
         ExceptionDivisonByZero edbz =
            new ExceptionDivisonByZero();
         edbz.divide();
```

```
        } catch (DivisionByZeroException e) {
            System.out.println("EXCEPTION");
            System.out.println("=========");
            System.out.println(String.format("Message: %s",
               e.getMessage()));
            System.out.println(String.format("First: %d",
               e.getFirst()));
            System.out.println(String.format("Second: %d",
               e.getSecond()));
        }
    }
}
```

Listing 5.6 *(Forts.)*
Diese Klasse wirft eventuell eine DivisionByZeroException und behandelt sie auch selbst

Innerhalb der statischen `main()`-Methode der Klasse wird in einem `try-catch`-Block eine Instanz der Klasse erzeugt, der lokalen Variablen `edbz` zugewiesen und die Methode `divide()` der Klasseninstanz aufgerufen.

Im Kopf der Methode `divide()` wird mit Hilfe des Schlüsselwortes `throws` angegeben, dass möglicherweise eine `DivisionByZeroException` geworfen werden könnte. Anschließend werden die beiden Zahlen, die durcheinander dividiert werden sollen, als Zufallszahlen ermittelt. Nun folgt eine Prüfung darauf, ob die zweite der beiden Zahlen, `second`, Null ist. Sollte dem so sein, werfen wir die `DivisionByZeroException` mit den beiden ermittelten Werten als Parameter.

Da die `DivisionByZeroException` nicht aufgefangen wird, eskaliert sie zur aufrufenden Methode – der statischen Methode `main()`, die sich innerhalb des `try-catch`-Statements befindet. Der `catch`-Block fängt die geworfene Ausnahme ab und gibt die vorhandenen Informationen aus.

Wenn Sie die Klasse ausführen, werden Sie zwei verschiedene Ausgaben bewundern können. Uns interessieren an dieser Stelle aber nur die von unserer `DivisionByZeroException`-Klasse generierte Ausnahme und deren Informationen:

EXCEPTION

=========

Message: Division by zero is not allowed!

First: 9

Second: 0

Damit sollte auch der grundsätzliche Einsatzbereich eigener Ausnahmen etwas klarer geworden sein: Ausnahme- oder Fehlersituationen, in denen zusätzliche oder andere Informationen transportiert werden müssen. In anderen Fällen können Sie in der Regel auf eine der existierenden Ausnahmen zurückgreifen oder eine einfache Ableitung der `Exception`-Klasse verwenden, die keine zusätzlichen Informationen mitführt.

Achtung

Was Sie nie machen sollten: Als Ausnahmetyp im Kopf einer Methode oder innerhalb eines `try-catch`-Statements die generische `Exception`-Klasse angeben. Die möglichen Folgen sind weiter oben beschrieben – im schlimmsten Fall bringen Sie die komplette Java-Runtime an den Rand des Abstürzens.

Verwenden Sie stattdessen in solchen Fällen eine Ableitung der Standard-`Exception`-Klasse, bei der Sie lediglich den Standard-Konstruktor überschreiben:

Listing 5.7
Standard-Exception-Ableitung

```
public class OwnException extends Exception {
   OwnException(String message) {
      super(message);
   }
}
```

Nunmehr können Sie statt Standard-*Exceptions* die Ableitung *OwnException* werfen. Diese können Sie auch explizit abfangen und würden somit keinerlei Beeinträchtigungen der Umgebung verursachen – auch potentiell nicht.

5.4 Strategien beim Einsatz von Ausnahmen

Bevor Sie auch nur die erste Ausnahme selbst werfen, sollten Sie sich über einige Dinge im Klaren sein:

- Ausnahmen kennzeichnen keine Regelzustände
- Ausnahmen können Ressourcen kosten, wenn sie nicht dicht abgefangen werden
- Ausnahmen dienen nicht dem Datenaustausch
- Ausnahmen sollten mit Bedacht und Überlegung eingesetzt werden

Diese Aussagen sollte man sich unterstreichen und verinnerlichen. Denn sie sind wesentlich, wenn Sie mit Ausnahmen arbeiten und diese sinnvoll verwenden wollen.

5.4.1 Behandeln Sie Ausnahmen so lokal wie möglich

Vermeiden Sie das Eskalieren von Ausnahmen und fangen Sie diese so dicht als möglich ab, denn: Je näher am Ursprung eine Ausnahme abgefangen und behandelt wird, desto geringer werden die Auswirkungen auf die Applikation sein.

Deshalb sollte die Regel sein: Behandeln Sie Ausnahmen so nah wie möglich. Umgeben Sie nur die Statements mit *try-catch*-Blöcken, die potentiell Ausnahmen werfen können. Vermeiden Sie Konstrukte wie dieses:

```
void method() {
   try {
      // ...
```

```
      <potentiell fehlerträchtiges Statement>;
      // ...
   } catch (<Exception> e) {
      // Fehlerbehandlung
   }
}
```

Sinnvoller im Sinne von Performance und Leistungsfähigkeit ist es, wenn Sie nur das fehlerträchtige Statement kapseln:

```
void method() {
   // ...
   try {
      <potentiell fehlerträchtiges Statement>;
   } catch (<Exception> e) {
      // Fehlerbehandlung
   }
   // ...
}
```

5.4.2 Behandeln Sie nur Ausnahmen, für die Sie zuständig sind

Wie bereits mehrfach im Verlauf dieses Kapitels beschrieben, sollten Sie nur die Ausnahmen behandeln, für die Sie tatsächlich auch zuständig sind. Fangen Sie also niemals Ausnahmen mit Hilfe der generischen `Exception`-Klasse ab, sondern fangen Sie exakt die Ausnahme-Typen ab, die auch geworfen werden könnten.

Dies sorgt dafür, dass sich die Teile Ihrer Applikation um die Fehler kümmern, von denen sie am meisten verstehen. So sparen Sie sich unnötigen Arbeitsaufwand und Ihre Applikation wird besser wartbar.

5.4.3 Verwenden Sie spezifische Ausnahme-Typen

Wenn Sie selbst Ausnahmen werfen, sollten Sie – soweit möglich – spezialisierte Ausnahme-Typen verwenden. Diese Typen sollten die Informationen aufnehmen, die nötig sind, damit abfangende Codes den Fehler eindeutig identifizieren und behandeln können.

Statt einer normalen `ArithmeticException` im Falle einer Division durch Null sollten Sie bei Bedarf lieber auf eine selbst definierte Exception zurückgreifen – möglicherweise erbt diese von der `ArithmeticException`, so dass andere Methoden, die eine `ArithmeticException` behandeln, nicht umgeschrieben werden müssen.

Catch-Statements, die Ihren Code verwenden, könnten nun statt der generischeren `ArithmeticException` Ihre spezifischere Ausnahme behandeln. Dadurch wären Sie in der Lage, auf den speziellen Fehler einer Division durch Null anders zu reagieren als auf eine allgemeinere Ausnahme, wie sie die `ArithmeticException` darstellt. Und Sie könnten beispielsweise einem Nutzer genauer mitteilen, worin dessen Fehler bestand.

Ein Beispiel für eine derartige Implementierung ist die `DivisionByZeroException`, die Sie weiter vorne in diesem Kapitel finden können.

So erreichen Sie gleich mehrere Dinge auf einmal:

- Auf Ihre Ausnahme kann zielgerichteter reagiert werden
- Die Benutzerfreundlichkeit Ihrer Applikation steigt, denn Fehler können besser präsentiert werden
- Ihre Ausnahme ist abwärtskompatibel – bestehender Code muss nicht umgeschrieben werden

5.4.4 Fazit

Ausnahmen erlauben es uns, fehlerhafte Zustände unserer Applikation zu erkennen und zu behandeln. Sie sollten möglichst nah am Ursprungsort abgefangen werden und nie unmotiviert geworfen werden. Ausnahmen sollten allerdings auch nicht einfach unterdrückt werden, denn sie erlauben eine zielgerichtete Reaktion spezialisierter Routinen auf ein Ereignis.

Sie sollten in Ihren Klassen eigene Exceptions deklarieren, um damit fehlerhafte Zustände von Komponenten oder Applikationsbestandteilen anzuzeigen. Mit Hilfe des Schlüsselwortes throws können Sie aufrufende Methoden darüber informieren, welche Ausnahmen von Ihren Methoden geworfen werden können – die aufrufende Methode kann somit entweder reagieren oder die Ausnahme selbst nach oben durchreichen.

Ausnahmen sind ein zwar manchmal sperriges, nichts desto trotz sinnvolles und nützliches Konstrukt bei der Entwicklung von Applikationen.

6

Threads

Kennen Sie das? In Ihrer Applikation haben Sie eine Aufgabe zu erledigen, die das Programm zeitlich in Anspruch nimmt und somit dessen Ressourcen frisst – aber eigentlich nicht zu den Kernaufgaben des Programms gehört? Oder Ihr Programm muss komplexe Berechnungen ausführen, die es für Minuten blockieren würden?

Dann suchen Sie nach Wegen, derartige Bestandteile des Programms auszulagern, und zwar so, dass sie das Programm nicht mehr stören und dennoch Bestandteil des Programms bleiben. Die Lösung für dieses Problem heißt *Threading*.

Threading bedeutet, dass das Programm einen Teil von sich eigenständig laufen lässt und damit Zeit bekommt, sich um andere Dinge zu kümmern. Kommt Ihnen bekannt vor? Kein Wunder: Auf Betriebssystem-Ebene heißt das auch Multitasking.

Threads (also die Komponenten, die eigenständig ausgeführt werden sollen) eignen sich ideal für alles, was potentiell viel Rechenzeit in Anspruch nehmen und dabei kontinuierlich ausgeführt werden soll.

6.1 Einen Thread erstellen

Threads sind grundsätzlich normale Klassen, die die Schnittstelle *Runnable* implementieren oder von der Basis-Klasse `Thread` erben.

Die Schnittstelle *Runnable* erfordert, dass eine Methode `run()` in der Klasse vorhanden ist.

6.1.1 Eine threadingfähige Klasse

Ein einfaches Beispiel für eine threadingfähige Klasse, die die Schnittstelle `Runnable` implementiert, könnte so aussehen:

Listing 6.1
Die Klasse SimpleThread implementiert die Schnittstelle Runnable und kann als eigenständiger Thread laufen

```
import java.util.Calendar;
import java.text.DateFormat;

/**
 * A simple thread implementing the Runnable interface
 */
public class SimpleThread implements Runnable {

   /**
    * The run method as requested by Runnable
    */
   public void run() {
      while(true) {
         displayMessage();
         try {
            Thread.sleep(5000);
         } catch (InterruptedException e) { }
      }
   }

   /**
    * Displays a message indicating that the tread is still running
    */
   private void displayMessage() {
      String date = DateFormat.getTimeInstance().format(
         Calendar.getInstance().getTime());
      System.out.println(
         String.format("%s: Still running (%s)!",
         "SimpleThread", date));
   }
}
```

Im Kopf der Klassendefinition geben wir mit Hilfe des Schlüsselworts `implements` an, dass das Interface `Runnable` implementiert werden soll. Dies erfordert dann in der Konsequenz, dass wir eine Methode `run()` definieren, die über keine Parameter verfügt.

In der `run()`-Methode machen wir nun der Einfachheit halber nichts weiter, als alle fünf Sekunden die Meldung *Still running* samt aktueller Uhrzeit auszugeben.

Zu diesem Zweck beinhaltet sie eine Endlosschleife, die in Form einer *while*-Schleife ausgeführt worden ist. Wie wir wissen, erwartet eine *while*-Schleife immer eine Bedingung, die als Ergebnis `true` haben muss, damit die Schleife durchlaufen werden kann. Wenn die Bedingung selbst `true` ist, haben wir eine Schleife, die nicht abbrechen wird – also den typischen Fall einer Endlosschleife.

Innerhalb dieser Endlosschleife rufen wir zuerst die Methode `displayMessage()` auf, die mit Hilfe eines `Calendar`-Objekts und einer `DateFormat`-Instanz die aktuelle Uhrzeit anhand der Benutzereinstellungen formatiert und zusammen

mit dem Namen der Klasse ausgibt. Diese `DateFormat`-Instanz erhalten wir, indem wir die statische Methode `getTimeInstance()` der `DateFormat`-Klasse aufrufen, die uns eine Instanz mit den im System hinterlegten Formatierungsinformationen zurückgibt.

Zurück in der Methode `run()`: Hier lassen wir nun den aktuellen Thread für fünf Sekunden pausieren, indem wir die entsprechende Anzahl an Millisekunden an die Methode `Thread.sleep()` übergeben.

Achtung

Wenn Sie einen Thread per `Thread.sleep()` pausieren lassen, müssen Sie dies innerhalb eines `try-catch`-Blocks machen. Dieser `try-catch`-Block muss eine `InterruptedException` abfangen können, da diese von `Thread.sleep()` geworfen wird, wenn der aktuelle Thread abgebrochen worden ist. Die Ausnahme müssen Sie dabei in der Regel nicht weiter behandeln – der `catch`-Block kann also leer bleiben, es sei denn, Sie müssten bestimmte Aufräumarbeiten vornehmen.

6.1.2 Einen Thread starten

Um einen Thread zu starten, müssen wir zunächst eine Referenz auf ein *Thread*-Objekt erzeugen. Falls wir eine Klasse verwenden, die nicht direkt von *Thread* erbt, sondern stattdessen die Schnittstelle `Runnable` implementiert, müssen wir diese bei der Instanzierung der `Thread`-Klasse als Parameter übergeben. Anschließend kann der Thread mit Hilfe seiner Methode `start()` aktiviert werden.

Umgesetzt in einer Klasse könnte dies so aussehen:

Listing 6.2
Starten eines Threads

```
import java.text.DateFormat;
import java.util.Calendar;

/**
 * Starts a new thread
 */
public class CreateThread {

   public static void main(String args[]) {'
      Thread thread;
      SimpleThread instance = new SimpleThread();

      thread = new Thread(instance);
      thread.start();

      while(true) {
         displayMessage();
         try {
            thread.sleep(3500);
         } catch (InterruptedException e) { }
      }
   }
```

Listing 6.2 *(Forts.)*
Starten eines Threads

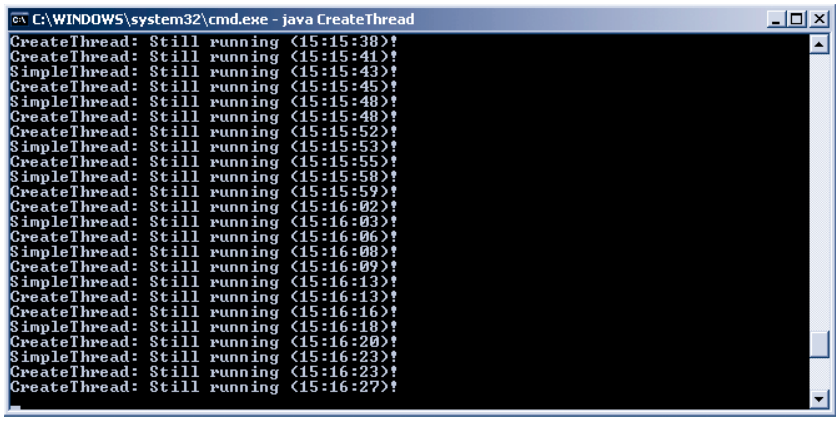
```
    /**
     * Displays a message indicating that the tread is still running
     */
    private static void displayMessage() {
       String date = DateFormat.getTimeInstance().format(
          Calendar.getInstance().getTime());
       System.out.println(String.format(
          "%s: Still running (%s)!", "CreateThread", date));
    }
}
```

Innerhalb der statischen Methode `main()` deklarieren wir hier zunächst eine Instanz der `Thread`-Klasse, weisen ihr aber noch keine Instanz zu. Dies geschieht erst, nachdem wir die Klasse, die im Thread laufen soll, mit Hilfe des Schlüsselwortes `new` instanziert haben. Das eigentliche Starten des Threads geschieht dann wie bereits erwähnt mit Hilfe von dessen Methode `start()`.

Nachdem wir den neuen Thread gestartet haben, lassen wir auch in der Methode `main()` regelmäßig eine Status-Information ausgeben – im Unterschied zur Klasse `SimpleThread` geschieht dies hier im Abstand von dreieinhalb Sekunden (3500 Millisekunden), was wir mit Hilfe von `Thread.sleep()` erreichen.

Wenn Sie die Klasse ausführen, sollten Sie eine Ausgabe ähnlich dieser erhalten:

Abbildung 6.1
Beide Threads laufen parallel



Sie stellen sicherlich erfreut fest, dass die beiden Threads parallel laufen – beide warten völlig unabhängig voneinander einen gewissen Zeitraum, bevor sie erneut ihre Ausgaben tätigen.

6.2 Timer statt Thread

Wenn Sie Aufgaben haben, die wiederholt in regelmäßigen Zeitabständen ausgeführt werden sollen, müssen Sie nicht zwingend auf die *Thread*-Ebene herabsteigen. Der *Timer*, der seit Java 1.3 zur Verfügung steht, erledigt genau diese Aufgabe.

Damit eine Klasse vom Timer regelmäßig ausgeführt werden kann, muss sie von der Klasse `TimerTask` erben und deren `run()`-Methode überschreiben. Dies könnte für eine Klasse, die eine Funktionalität analog zu obigem Beispiel implementieren soll, etwa so aussehen:

```
import java.util.TimerTask;
import java.util.Calendar;
import java.text.DateFormat;

/**
 * Created by Karsten Samaschke
 */
public class SimpleTimerTask extends TimerTask {

   /**
    * The action to be performed by this timer task.
    */
   public void run() {
      String date = DateFormat.getTimeInstance().format(
         Calendar.getInstance().getTime());
      System.out.println(String.format(
         "%s: Still running (%s)!",
         "SimpleTimerTask", date));
   }
}
```

Die Klasse `SimpleTimerTask` erbt von der Basisklasse `TimerTask`, was wir mit Hilfe des `extends`-Schlüsselwortes angeben. Sie überschreibt die Methode `run()` der `TimerTask`-Klasse und legt dabei den Code fest, der während der `Timer`-Laufzeit ausgeführt werden soll.

Dieser Code ist dabei bewusst simpel gehalten: Bei jedem Aufruf von `run()` durch den `Timer` ermitteln wir die aktuelle Uhrzeit anhand der Benutzer-Einstellungen im System und geben sie zusammen mit einer kleinen Hinweis-Meldung aus.

Um den `TimerTask` auszuführen, muss er eingebunden werden. Dies geschieht in diesem Fall in einer externen Klasse, die eine neue `Timer`-Instanz erzeugt und mit Hilfe von deren `schedule()`-Methode festlegt, wann und in welchen Abständen unsere `SimpleTimerTask`-Klasse ausgeführt wird:

Listing 6.3
In der Klasse CreateTimer wird der Timer gestartet

```
import java.text.DateFormat;
import java.util.Calendar;
import java.util.Timer;
import java.util.TimerTask;

/**
 * Created by Karsten Samaschke
 */
public class CreateTimer {

   public static void main(String args[]) {
      Timer timer = new Timer();
      SimpleTimerTask task = new SimpleTimerTask();
```

Listing 6.3 *(Forts.)*
In der Klasse CreateTimer wird der Timer gestartet

```
        long delay = 0;
        long period = 5000;
        timer.schedule(task, delay, period);

        while(true) {
           displayMessage();
           try {
              Thread.sleep(3500);
           } catch (InterruptedException e) { }
        }
     }

     /**
      * Displays a message indicating that the tread is still running
      */
     private static void displayMessage() {
        String date = DateFormat.getTimeInstance().format(
           Calendar.getInstance().getTime());
        System.out.println(String.format(
           "%s: Still running (%s)!", "CreateTimer", date));
     }
}
```

Der *Timer* empfiehlt sich also immer dann für einen Einsatz, wenn Sie wiederholt und vor allem regelmäßig Code-Fragmente ausführen wollen. Für einen Einsatz in Szenarien, in denen Sie Code einmalig auslagern und unabhängig extern laufen lassen wollen, empfiehlt sich stattdessen nach wie vor der Einsatz von Threads.

Innerhalb der statischen Methode `main()` erzeugen wir mit Hilfe des Schlüsselworts `new` eine `Timer`- und eine `TimerTask`-Instanz. Die `TimerTask`-Instanz ist dabei eine Instanz unserer Klasse `SimpleTimerTask` – wir arbeiten hier intern aber mit der Basis-Klasse, da wir keinerlei weiterführende Funktionalität definiert haben.

Der Methode `schedule()` der *Timer*-Instanz werden nun als Parameter unsere `SimpleTimerTask`-Instanz, die Verzögerung bis zur ersten Ausführung des *Timers* in Millisekunden und die Wiederholverzögerung vor einer erneuten Ausführung in Millisekunden als Parameter übergeben. In unserem Beispiel ist dies eine Ausführungsverzögerung von 0 Millisekunden und einer Wiederhol-Verzögerung von 5000 Millisekunden – also fünf Sekunden.

Der Rest der Methode `main()` ist identisch mit dem vorherigen *Thread*-Beispiel – wir geben hier zur Kontrolle alle dreieinhalb Sekunden einen Text aus, so dass wir sehen, dass die Methode weiterhin ausgeführt wird. Diese Verzögerung erreichen wir aber nicht, indem wir einen weiteren `Timer` einsetzen, sondern den aktuellen Thread mit Hilfe von `Thread.sleep()` pausieren lassen.

Wenn Sie die Klasse `CreateTimer` kompilieren und von der Kommandozeile per

```
java CreateTimer
```

ausführen lassen, werden Sie folgende Ausgabe erhalten:

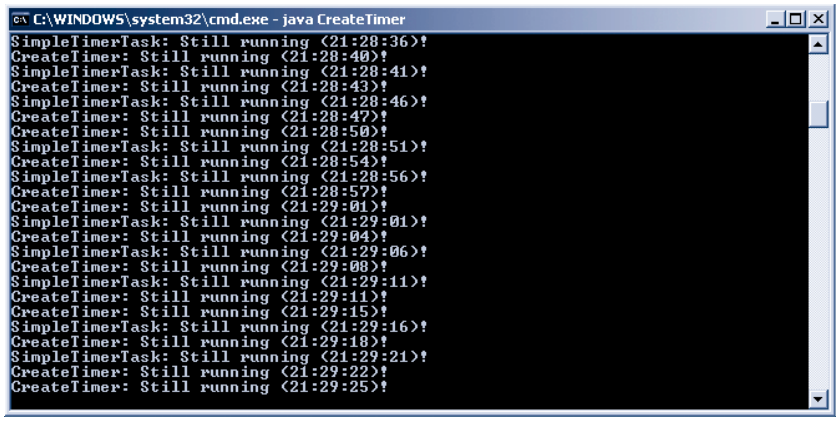


Abbildung 6.2
Auch per Timer können Klassen in anderen Threads laufen

6.3 Priorität von Threads setzen

Threads können nicht nur parallel laufen. Sie können zusätzlich auch mit einer Priorität versehen werden. Die Priorität von Threads kommt dann zum Tragen, wenn verschiedene Threads zeitgleich ausgeführt werden sollen – Java entscheidet dann anhand der Priorität des Threads, welchen es tatsächlich ausführen soll und in welcher Reihenfolge weitere Threads ablaufen sollen.

Wenn mehrere Threads mit der gleichen Priorität auf Ausführung warten, entscheidet Java, welcher Thread tatsächlich als Erstes zur Ausführung kommt. Die weiteren Threads werden anschließend einer nach dem anderen abgearbeitet – man spricht in diesem Fall von einem Round-Robin-Verfahren, bei dem die Thread-Last gleichmäßig verteilt wird.

Wenn Sie einen neuen Thread ohne explizite Priorität starten, erbt dieser die Priorität von dem erzeugenden Thread. Dies bedeutet, dass ein Thread, der von einem Thread mit `MAX_PRIORITY` erzeugt worden ist, selber auch diese Priorität besitzt.

Die Priorität von Threads setzen Sie mit Hilfe der Methode `setPriority()` einer *Thread*-Instanz. Die Methode nimmt einen *Integer*-Wert entgegen, der zwischen `MIN_PRIORITY` und `MAX_PRIORITY` liegen muss. `MIN_PRIORITY` kennzeichnet die geringste Priorität, `MAX_PRIORITY` steht für die höchste mögliche Priorität eines Threads. Beide Konstanten sind in der Klasse `Thread` definiert.

Sehen wir uns ein Beispiel für die Priorität von Threads an:

Listing 6.4
Priorisierung von Threads

```
/**
 * Created by Karsten Samaschke
 */
public class PriorityThread implements Runnable {

   private String name;
   private int priority;

   /**
    * Gets the name of the current instance
    * @return The name of the instance
    */
   public String getName() {
      return name;
   }
```

Listing 6.4 *(Forts.)*
Priorisierung von Threads

```
    /**
     * Sets the name of the current instance
     * @param name Name of the instance
     */
    public void setName(String name) {
       this.name = name;
    }

    /**
     * Gets the priority of the thread
     * @return The priority of the current thread
     */
    public int getPriority() {
       return priority;
    }

    /**
     * Sets the priority of the current thread
     * @param priority Priority of the thread
     */
    public void setPriority(int priority) {
       this.priority = priority;
    }

    /**
     * Implementation of Runnable's run-method
     */
    public void run() {
       System.out.println(
          String.format(
             "Thread %s with priority %d has executed!",
             this.getName(), this.getPriority()));
    }

    public static void main(String args[]) {
       PriorityThread minThread = new PriorityThread();
       minThread.setName("minPrio");
       minThread.setPriority(Thread.MIN_PRIORITY);

       PriorityThread maxThread = new PriorityThread();
       maxThread.setName("maxPrio");
       maxThread.setPriority(Thread.MAX_PRIORITY);

       Thread minPrio = new Thread(minThread);
       Thread maxPrio = new Thread(maxThread);

       minPrio.setName(minThread.getName());
       minPrio.setPriority(minThread.getPriority());
       maxPrio.setName(maxThread.getName());
       maxPrio.setPriority(maxThread.getPriority());

       minPrio.start();
       maxPrio.start();
    }
}
```

Die Klasse `PriorityThread` implementiert das Interface *Runnable* mit dessen Methode `run()` – wobei diese Methode sehr simpel gehalten ist, denn sie gibt einfach nur aus, dass der Thread beendet worden ist.

Innerhalb der statischen Methode `main()` werden mit Hilfe des Schlüsselwortes *new* zwei neue `PriorityThread`-Instanzen erzeugt: `minThread` und `maxThread`. Beiden wird mit Hilfe ihrer Methode `setName()` ein Name zugewiesen – die Instanz, die die geringere Priorität haben soll, erhält den Namen `minPrio`, die andere heißt `maxPrio`.

Entsprechend der Namen werden auch die Werte für die Prioritäten verteilt – `minThread` erhält die Priorität `MIN_PRIORITY`, `maxThread` wird der Wert von `MAX_PRIORITY` zugewiesen. Die eigentliche Zuweisung erfolgt mit Hilfe der Methode `setPriority()` der jeweiligen `Thread`-Instanz. Beachten Sie, dass wir in diesem Beispiel noch keine wirklichen Thread-Prioritäten verteilt haben – wir haben die Werte zunächst nur unseren `PriorityThread`-Instanzen zugewiesen, damit wir sie später ausgeben können.

Nun erzeugen wir die beiden Thread-Instanzen `minPrio` und `maxPrio`, denen wir die Prioritäts-Werte aus den beiden `PriorityThread`-Instanzen zuweisen. Dies bedeutet, dass `minPrio` mit einer geringeren Priorität als `maxPrio` laufen wird.

Um dies zu beweisen, werden die beiden Threads am Ende mit Hilfe ihrer `start()`-Methoden gestartet.

Beachten Sie, dass wir zunächst `minPrio` und erst anschließend `maxPrio` starten – denn die Ausgabe ist interessant:

Thread maxPrio with priority 10 has executed!

Thread minPrio with priority 1 has executed!

Offensichtlich führt die höhere Priorisierung dazu, dass `maxPrio` zuerst ausgeführt worden ist, obwohl `minPrio` vorher gestartet worden ist. Sie können also die Ausführungsreihenfolge von Threads mit Hilfe ihrer Priorität ändern.

6.4 Threads beenden

Können laufende Threads beendet werden? Laut Dokumentation ist dies nicht (mehr) möglich, denn die Methode `stop()` einer `Thread`-Instanz ist als *deprecated* gekennzeichnet – von der Verwendung wird also im Sinne einer Zukunftskompatibilität dringend abgeraten.

6.4.1 Verwendung von interrupt() zum Abbrechen eines Threads

In verschiedenen Foren wird gerne empfohlen, die Methode `interrupt()` der jeweiligen *Thread*-Instanz zu verwenden, denn die sei im Gegensatz zur Methode `stop()` nicht *deprecated*.

Um es kurz zu sagen: Wenn es Ihnen nichts ausmacht, dass dabei jedes Mal eine `InterruptedException` (was im Sinne von sauberer Programmierung und Stabilität suboptimal ist) geworfen wird und dem Thread keine Möglichkeit gelassen wird, sich sauber zu beenden, dann nutzen Sie diese Methode.

Wir werden davon hier Abstand nehmen, da diese Lösung genau genommen keine Lösung ist, sondern mehr eine letzte Möglichkeit darstellt, um die Beendigung eines Threads zu erzwingen.

Achtung

Die Verwendung von `stop()` oder `interrupt()` führt dazu, dass ein Thread unter Umständen keine Möglichkeit mehr hat, seine Ressourcen aufzuräumen. In der Konsequenz kann dies bedeuten, dass Dateien geöffnet bleiben oder Änderungen nicht übernommen werden.

6.4.2 Verwendung einer eigenen Thread-Ableitung

Viel eleganter als die Verwendung der Methode `interrupt()` ist der Einsatz einer eigenen `Thread`-Ableitung. Diese `Thread`-Ableitung gibt uns Auskunft darüber, ob eine Beendigung des Threads gewollt ist oder nicht:

Listing 6.5
Die Klasse StoppableThread teilt interessierten Klassen mit, dass der Thread stoppen soll

```
/**
 * Base class, which allows inheriting Threads
 * to determine their status
 */
public class StoppableThread extends Thread  {

   private boolean stop = false;

   /**
    * Default constructor
    */
   StoppableThread() {
      super();
   }

   /**
    * Constructor which takes a String as argument
    * @param name Name of the current thread
    */
   StoppableThread(String name) {
      super(name);
   }

   /**
    * Constructor which takes a String and a ThreadGroup-instance
    * as arguments
    * @param g ThreadGroup the thread belongs to
    * @param name Name of the thread
    */
   StoppableThread(ThreadGroup g, String name) {
      super(g, name);
   }
```

Listing 6.5 *(Forts.)*
Die Klasse StoppableThread teilt interessierten Klassen mit, dass der Thread stoppen soll

```
    /**
     * Returns true if the thread needs to stop
     */
    public boolean getStop() {
       return this.stop;
    }

    /**
     * Sets a status-flag indicating that the thread has to stop
     * (true) or not (false)
     * @param value Status as boolean value
     */
    public void setStop(boolean value) {
       this.stop = value;
    }
}
```

Die Klasse `StoppableThread` erbt von `Thread`, was wir mit Hilfe des Schlüsselworts *extends* definieren. Sie implementiert darüber hinaus Getter- und Setter-Methoden für die private Variable `stop`. Diese boolesche Variable gibt Auskunft darüber, ob der aktuelle Thread angehalten werden soll (dann ist der Wert `true`) oder nicht.

Um die Arbeit mit unserer neuen Klasse zu erleichtern, verfügt diese über drei Konstruktoren. Der Standard-Konstruktor nimmt keine Argumente entgegen und ruft mit Hilfe des Schlüsselworts `super` den Standard-Konstruktor der *Super-Klasse* auf. Die beiden anderen Konstruktoren nehmen den Namen des Threads und eine Thread-Gruppe, zu der der Thread gehören soll, entgegen. Alle Werte werden via `super` an die übergeordnete `Thread`-Klasse übergeben – diese kümmert sich dann auch um deren Verarbeitung.

6.4.3 StoppableThread statt Thread als Super-Klasse

Leider ist dies nur die Hälfte der Arbeit – Klassen, die als Thread laufen sollen, müssen nun regelmäßig prüfen, ob der Thread noch weiterlaufen soll oder nicht. Diese Prüfung ist aber zum Glück nicht allzu komplex, wie das folgende Beispiel beweist:

Listing 6.6
Dieses Thread-Element kann sich kontrolliert beenden

```
/**
 * Stoppable thread implementation
 */
public class SimpleStoppableThread extends StoppableThread {

   /**
    * The Runnable's run-method
    */
   public void run() {
      System.out.println("Thread started!");

      while(!this.getStop()) {
         try {
            Thread.sleep(100);
            System.out.print(".");
         } catch (InterruptedException e) { }
      }
```

Listing 6.6 *(Forts.)*
Dieses Thread-Element kann sich kontrolliert beenden

```
        System.out.println("\nThread stopped!");
    }

    public static void main(String args[]) {
        SimpleStoppableThread stoppable = new SimpleStoppableThread();
        stoppable.start();

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) { }

        stoppable.setStop(true);
    }
}
```

Das Prinzip, das uns jede Menge Arbeit erspart, ist: Alle neuen Threads erben nicht mehr von der Basis-Klasse `Thread`, sondern von der eben definierten Klasse `StoppableThread`. Diese stellt mit Hilfe ihrer Methoden `getStop()` und `setStop()` alles bereit, was wir benötigen, um eine Stop-Anforderung zu erkennen oder zu setzen.

Nebenbei ersparen wir uns mit dem Erben von `Thread` oder `StoppableThread`, dass wir das Interface `Runnable` explizit implementieren müssen – Thread und damit auch alle abgeleiteten Klassen implementiert dies bereits und wir müssen die Methode `run()` nur noch mit unserer Logik überschreiben. Außerdem können sich unsere Threads nunmehr selbst starten – wir müssen also nicht mehr auf eine externe `Thread`-Instanz zurückgreifen.

Innerhalb der überschriebenen Methode `run()` definiert unsere `Thread`-Implementierung eine `while`-Schleife, deren Abbruchbedingung darin besteht, dass der Getter `getStop()` `true` zurückliefert.

Sollte nun also dem aktuellen Thread der Befehl zum Stoppen geben werden, kann dies recht schnell und einfach ermittelt werden – `getStop()` würde in diesem Fall *true* zurückgeben. Spätestens nach 100 Millisekunden (so lange warten wir innerhalb der Schleife mit Hilfe der statischen Methode `sleep` der `Thread`-Klasse) würde sich unser Thread beenden.

Diese Lösung ist viel sauberer und eleganter als ein Abbruch per `stop()`- oder `interrupt()`-Methode, denn sie erlaubt es uns, in Ruhe alle benötigen Ressourcen freizugeben oder andere Aufräumarbeiten vorzunehmen.

Die Ausführung von der Kommandozeile findet mit Hilfe der statischen Methode `main()` statt, die wir der Einfachheit halber ebenfalls innerhalb der Klasse definieren. Die Methode erstellt und startet eine neue Instanz der `SimpleStoppableThread`-Klasse. Anschließend pausiert sie zwei Sekunden mit Hilfe von `Thread.sleep()`.

Am Ende wird der aktuelle Thread beendet, indem `setStop()` mit dem Wert `true` aufgerufen wird. Da der Thread regelmäßig den Wert der privaten Variablen `stop` mit Hilfe des Getters `getStop()` überprüft, kann er sich nun kontrolliert beenden.

Wenn Sie die Lösung per

```
java SimpleStoppableThread
```

starten, sollten Sie folgende Ausgabe erhalten:

Thread started!

.....

Thread stopped!

Würden wir stattdessen die als *deprecated* (also als veraltet und potentiell nicht mehr unterstützt) gekennzeichnete Methode `stop()` verwenden, erhielten wir folgende Ausgabe, die verdeutlicht, warum wir Threads lieber kontrolliert abbrechen sollten:

Thread started!

.....

Hier hatte der Thread offensichtlich keine Möglichkeit mehr, sich kontrolliert zu beenden.

6.5 Zugriff auf den aktuellen Thread

Aus jeder Klasse heraus kann auf den Thread zugegriffen werden, in dem sie läuft. Dies geschieht mit Hilfe der statischen Methode `currentThread()` der `Thread`-Klasse:

Listing 6.7
Ausgabe von Informationen zum aktuellen Thread

```
/**
 * Demonstrates, how to access the current thread
 */
public class CurrentThread {

   /**
    * The application's run method
    * @param args Commandline arguments
    */
   public static void main(String args[]) {
      Thread current = Thread.currentThread();
      System.out.println(
         String.format(
            "ID: %d\nName: %s\nPriority: %d\nGroup-Name: %s",
            current.getId(),
            current.getName(),
            current.getPriority(),
            current.getThreadGroup().getName()));
   }
}
```

Die Funktion der statischen Methode `main()` ist schnell erläutert: Am Anfang holen wir uns eine Referenz auf den aktuellen Thread mittels `Thread.currentThread()` und weisen diese der lokalen Variablen `current` zu. Dann geben wir einige Informationen zum Thread aus – im Einzelnen sind dies:

- ID des Threads
- Name des Threads
- Priorität des Threads
- und der Name der Thread-Gruppe, zu der der Thread gehört

Wenn Sie die Klasse kompilieren und ausführen, sollten Sie eine Ausgabe ähnlich dieser erhalten:

ID: 1

Name: main

Priority: 5

Group-Name: main

6.6 Ermitteln aller laufenden Threads

Mit Hilfe der `Thread`-Klasse können wir alle laufenden Threads ermitteln. Dies geschieht, indem wir ein `Thread`-Array erzeugen, das Platz für die Gesamtzahl der laufenden Threads bietet, und dieses Array der statischen Klassen-Methode `enumerate()` der `Thread`-Klasse übergeben:

Listing 6.8
Auflisten aller laufenden Threads einer Applikation

```
/**
 * Displays a list of all running threads
 */
public class AllThreads extends StoppableThread {

   /**
    * The thread's run method
    */
   public void run() {
      while(!getStop()) {
         try {
            Thread.sleep(100);
         } catch (InterruptedException e) { }
      }
   }

   /**
    * Starts some threads and displays all threads
    * @param args Commandline arguments
    */
   public static void main(String[] args) {
      for(int i=0; i<18; i++) {
         Thread t = new AllThreads();
         t.setName(String.format("Thread %d", i));
         t.start();
      }
```

```
        Thread[] threads = new Thread[Thread.activeCount()];
        Thread.enumerate(threads);

        System.out.println("ALL THREADS");
        System.out.println("===========");
        for (Thread ct : threads) {
           System.out.println(ct.getName());
           if(ct instanceof StoppableThread) {
              ((StoppableThread)ct).setStop(true);
           }
        }
     }
}
```

Listing 6.8 *(Forts.)*
Auflisten aller laufenden Threads einer Applikation

Die Klasse `AllThreads`, die das Auflisten aller laufenden Threads erledigen wird, erbt selbst von der Basis-Klasse `StoppableThread`, was wir mit Hilfe des Schlüsselworts `extends` angeben. Die Klasse `StoppableThread` erlaubt es uns, einen Thread kontrolliert zu beenden. Wir haben sie bereits weiter oben in diesem Kapitel ausführlich behandelt.

Die Verwendung von `StoppableThread` als Basis-Klasse versetzt uns in die Lage, `AllThreads` selbst als Thread zu verwenden. Wir nehmen diese Möglichkeit innerhalb der statischen Methode `main()` wahr und erzeugen gleich einmal achtzehn neue Threads vom Typ `AllThreads`.

Jedem dieser achtzehn Threads wird mit Hilfe der Setter-Methode `setName()` ein eigener Name zugewiesen. Dieser Name beinhaltet die Thread-Nummer. Anschließend wird der jeweilige Thread mit Hilfe seiner Methode `start()` gestartet. Er wird so lange laufen, bis er per `setStop()` das Stop-Signal erhalten hat. Alternativ könnte der Thread natürlich auch per `stop()` oder `interrupt()` unterbrochen werden, was aber nicht empfehlenswert ist.

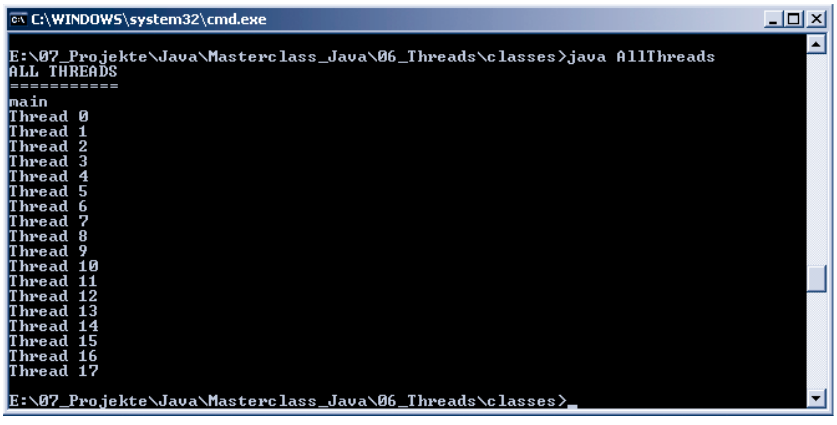
Nachdem wir alle Threads gestartet haben, erzeugen wir ein neues `Thread`-Array mit der Anzahl der laufenden Threads als Obergrenze. Diese Anzahl kann mit Hilfe der statischen Methode `activeCount()` der `Thread`-Klasse ermittelt werden. Dem *Thread*-Array werden nun mit Hilfe der statischen Methode `enumerate()` der *Thread*-Klasse alle laufenden Thread-Instanzen zugewiesen.

Jetzt verfügen wir über eine Liste aller Threads, die wir nun per `for-each`-Schleife durchlaufen können. Dabei wird bei jedem Durchlauf der lokalen Variablen `ct` eine Referenz auf den jeweiligen Thread zugewiesen und dessen Name kann mit Hilfe von `getName()` ermittelt und ausgeben werden.

Zuletzt überprüfen wir jeden Thread mit Hilfe des Schlüsselworts `instanceof` darauf, ob er eine `StoppableThread`-Instanz ist. Sollte dies der Fall sein, wird er in den Typ `StoppableThread` gecastet und ihm kann mit Hilfe seiner Methode `setStop()` signalisiert werden, dass er sich beenden soll – schließlich benötigen wir ihn nicht mehr in unserer Applikation und möchten die entsprechenden Ressourcen wieder freigeben.

Wenn Sie die Klasse kompilieren und ausführen, sollten Sie eine Ausgabe analog zu dieser erhalten:

Abbildung 6.3
Ausgabe aller laufenden Threads



Der oberste Thread *main* entspricht übrigens dem Thread, den wir gestartet haben, indem wir die Klasse von der Kommandozeile aus aufgerufen haben. Er wird als letzter Thread beendet werden.

6.7 Organisation von Threads in Thread-Gruppen

Thread-Gruppen stellen eine Organisationsform von Threads dar, mit deren Hilfe diese leichter verwaltet und gemanaged werden können. Jeder Thread kann genau einer Thread-Gruppe angehören und von dieser beeinflusst werden. Sehen wir uns dies in einem Beispiel an:

Listing 6.9
ThreadGroups erlauben die Organisation der enthaltenen Threads

```
/**
 * Demonstrates the usage of ThreadGroups
 */
public class ThreadGroupExample extends StoppableThread {

   /**
    * The thread's constructor
    * @param g Group the thread belongs to
    * @param name Name of the thread
    */
   ThreadGroupExample(ThreadGroup g, String name) {
      super(g, name);
   }

   /**
    * The thread's main method
    */
```

Listing 6.9 *(Forts.)* ThreadGroups erlauben die Organisation der enthaltenen Threads

```
    public void run() {
       while(!this.getStop()) {
          try {
             Thread.sleep(100);
          } catch (InterruptedException e) { }
       }
    }

    /**
     * Displays the name of a ThreadGroup and iterates through
     * its members
     * @param tg ThreadGroup to display and iterate through
     */
    private static void enumerateGroup(ThreadGroup tg) {
       System.out.println("ENUMERATING GROUP");
       System.out.println("================");
       System.out.println(String.format("Group: %s", tg.getName()));

       Thread[] tgItems = new Thread[tg.activeCount()];
       tg.enumerate(tgItems);

       for(Thread t : tgItems) {
          System.out.println(
             String.format("     Thread %s (%d)",
             t.getName(), t.getPriority()));
          if(t instanceof StoppableThread) {
             ((StoppableThread)t).setStop(true);
          }
       }

       System.out.println();
    }

    /**
     * Creates two new ThreadGroups and sets their thread's priority
     * @param args Commandline arguments
     */
    public static void main(String args[]) {
       ThreadGroup group = new ThreadGroup("sampleGroup");
       group.setMaxPriority(3);
       for(int i=0; i<6; i++) {
          Thread t = new ThreadGroupExample(
             group, String.format("%d", i));
          t.start();
       }

       ThreadGroup secondGroup =
          new ThreadGroup("secondGroup");
       for(int i=5; i<12; i++) {
          Thread t = new ThreadGroupExample(
             secondGroup, String.format("%d", i));
          t.start();
       }

       enumerateGroup(group);
       enumerateGroup(secondGroup);
    }
}
```

Die Klasse `ThreadGroupExample` ist selbst eine Ableitung der bereits weiter vorne in diesem Kapitel beschriebenen Klasse `StoppableThread`, was wir mit Hilfe des Schlüsselwortes `extends` gefolgt von der zugrunde liegenden Basis-Klasse `StoppableThread` angeben. Dadurch verfügt sie über die Getter und Setter `getStop()` und `setStop()` der `StoppableThread`-Klasse, die es uns erlauben, einen Thread kontrolliert zu beenden. Voraussetzung dafür ist natürlich, dass die `Thread`-Instanz dies in ihrer `run()`-Methode überprüft – was hier der Fall ist.

Die Klasse `ThreadGroupExample` verfügt neben der Methode `run()` über einen Konstruktor, der Thread-Gruppe und den Namen des Threads entgegennimmt.

Die Methode main()

Die eigentlich interessanten Dinge finden jedoch in den Methoden `main()` und `enumerateGroup()` statt. Sehen wir uns zunächst die `main()`-Methode etwas näher an: Diese stellt die Start-Methode der Klasse dar, die eingebunden wird, wenn ein Aufruf über Kommandozeile erfolgt.

Zuerst erzeugen wir hier eine neue `ThreadGroup` mit Namen `sampleGroup`, für die wir dann festlegen, dass alle enthaltenen Threads eine maximale *Thread-Priorität* von 3 erhalten sollen. Eine derartige Festlegung muss vor dem Zuweisen von Threads erfolgen, denn sie zeigt keinerlei Auswirkungen auf bereits vorhandene Elemente.

Danach werden sechs neue Threads mit Hilfe des Schlüsselworts `new` erzeugt und durch `<ThreadInstanz>.start()` aktiviert. Bei ihrer Erzeugung werden der neuen Instanz die Thread-Gruppe, zu der sie gehört, und ihr jeweiliger Name als Parameter übergeben (wobei der Name hier der Einfachheit halber nur aus einer laufenden Nummer besteht).

Jetzt erzeugen wir eine weitere Thread-Gruppe namens `secondGroup`. Dieser Thread-Gruppe werden ebenfalls sechs Threads zugeordnet, indem diesen bei der Erzeugung die `secondGroup`-Instanz als Parameter übergeben wird.

enumerateGroup()

Nachdem wir nunmehr über zwei Thread-Gruppen mit jeweils sechs Elementen verfügen, lassen wir uns deren Detail-Informationen mit Hilfe der ebenfalls statischen Methode `enumerateGroup()` ausgeben. Zuerst erfolgt die Ausgabe des Gruppen-Namens, den wir mit Hilfe von `getName()` ermitteln können.

Anschließend erzeugen wir ein leeres `Thread`-Array mit der von `<ThreadGroup>.activeCount()` zurückgegebenen Anzahl an Elementen. Dieses lassen wir dann durch die Methode `enumerate()` der Thread-Gruppe befüllen. Wir erhalten so alle der Thread-Gruppe zugeordneten Threads in einem Array, das wir anschließend per `for-each`-Schleife durchlaufen.

Für jeden enthaltenen Thread geben wir dabei seinen Namen und seine Priorität aus, was mit Hilfe der beiden Getter `getName()` und `getPriority()` vonstatten geht. Der guten Ordnung halber überprüfen wir zuletzt mit Hilfe des Schlüsselworts `instanceof`, ob der Thread eine Instanz von `StoppableThread` ist, und

beenden ihn, falls dies zutrifft, indem wir ihn zunächst nach `StoppableThread` casten und dann seiner `setStop()`-Methode den Wert `true` übergeben.

Nach dieser Menge an Informationen sollten wir unsere Klasse einfach von der Kommandozeile ausführen. Kompilieren Sie die Klasse zu diesem Zweck und rufen Sie sie auf:

```
java ThreadGroupExample
```

Die Ausgabe sollte dann so aussehen:

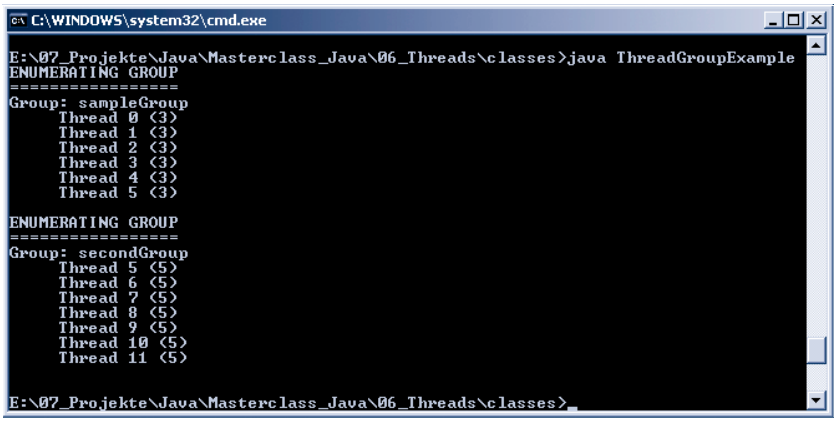


Abbildung 6.4
Gruppierung von Threads

Thread-Gruppen eignen sich also sehr gut, um Threads zu organisieren. Gerade dann, wenn Sie viele Threads in einer Applikation verwenden (müssen), bietet sich ihr Einsatz an.

6.8 Synchronisierung von Threads

Wenn Threads miteinander kommunizieren oder Dateisystemzugriffe durchführen wollen, ist eine Synchronisierung dieser Prozesse unerlässlich – beispielsweise könnte der Zugriff auf eine Datei, in die von einem anderen Thread gerade geschrieben wird, fehlschlagen oder Variablen könnten undefinierte Werte halten.

Nehmen wir als Beispiel eine gemeinsame Zählvariable, die von zwei Threads parallel genutzt und hochgezählt werden soll:

```
public class SynchronizedThread extends Thread {

   static int counter = 0;

   public void run() {
      while(true) {
         int current = counter++;
         try {
            Thread.sleep(10);
         } catch (InterruptedException ex) {}
```

```
            System.out.println(current);
        }
    }

    public static void main(String[] args) {
        Thread t1 = new SynchronizedThread();
        t1.start();

        Thread t2 = new SynchronizedThread();
        t2.start();
    }
}
```

Wenn Sie dieses Beispiel eine Weile laufen lassen, werden Sie früher oder später feststellen, dass der Zugriff auf die Zählvariable offensichtlich nicht synchronisiert ist:

```
650
653
652
655
654
657
656
659
658
661
660
663
662
665
664
667
666
669
668
671
670
```

Wie anders ist es zu erklären, das die Zahlen nicht in korrekter Reihenfolge ausgegeben werden?

Der Hintergrund dieses Verhaltens liegt darin, wie Multitasking und die Verwaltung von Prozessen implementiert ist: Der Scheduler, der für das Zuweisen von Rechenzeit zu den einzelnen Threads zuständig ist, hat den einen Thread nach dem Erhöhen der Zählvariablen aber vor dem Ausgeben des neuen Wertes unterbrochen und den anderen Thread aktiviert. Dieser hat nun seinerseits die Zählvariable hochgezählt und deren Wert ausgegeben.

Um dieses Verhalten zu verhindern, benötigen wir einen Prozess, mit dem wir die Aktionen der einzelnen Threads synchronisieren können.

6.8.1 Monitor

Die Synchronisation von Threads erfolgt mit Hilfe von *Monitoren*: Kritische Code-Fragmente werden automatisch gesperrt und erst nach dem Durchlauf des Fragments wieder freigegeben. Ist der Code-Bereich bereits gesperrt, muss ein Thread warten, bis dieser Bereich wieder freigegeben worden ist.

In Java erfolgt die Synchronisierung mit Hilfe des Schlüsselworts `synchronized`. Dieses Schlüsselwort ist in der Lage, entweder einen Bereich oder eine ganze Methode zu schützen. Der Einsatz von `synchronized` erfordert, das entweder eine Objektvariable oder der Pointer `this` als Parameter angegeben werden. Wichtig dabei ist, das die Objektvariablen unterschiedlicher Threads, die miteinander synchronisiert werden sollen, stets auf die gleiche Objekt-Instanz zeigen. Gibt es keine Membervariable in unterschiedlichen Threads, die auf die gleiche Objektinstanz zeigt, kann man sich damit behelfen, das man mit Hilfe der Methode `getClass()` eine derartige Referenz selbst erzeugt.

Um die Threads des obigen Beispiels zu synchronisieren, kapseln wir den kritischen Bereich mit Hilfe eines `synchronized`-Blocks:

Listing 6.10
Synchronisation zweier Threads per Monitor

```
public class SynchronizedThread extends Thread {

   static int counter = 0;


   public void run() {
      while(true) {
         synchronized(this.getClass()) {
            int current = counter++;
            try {
               Thread.sleep(10);
            } catch (InterruptedException ex) {}
            System.out.println(current);
         }
      }
   }

   public static void main(String[] args) {
      Thread t1 = new SynchronizedThread();
      t1.start();

      Thread t2 = new SynchronizedThread();
      t2.start();
   }
}
```

Nunmehr wird die Ausgabe der beiden Threads in der korrekten Reihenfolge geschehen:

```
0
1
2
3
4
5
```

Um eine Methode eines Objekts, das etwa von mehreren Instanzen parallel verwendet wird, zu synchronisieren, reicht es aus, das Schlüsselwort `synchronized` in die Methoden-Signatur aufzunehmen.

Nehmen wir wieder ein Beispiel: Hier haben wir das Verwalten und Hochzählen eines Zählers ausgelagert – die Klasse `Counter` erledigt genau dies. Mit Hilfe ihrer Methode `increment()` wird der Zähler erhöht und das Ergebnis dieser Operation zurück gegeben. Um dabei eine zeitaufwändigere Operationen zu simulieren, lassen wir den jeweils aktuellen Thread bei der Ausführung ein wenig pausieren:

Listing 6.11
Die Instanz der Klasse Counter ist nicht synchronisiert

```
class Counter {

   private int id = 0;

   public int increment() {
      int result = id++;
      try {
         Thread.sleep(100);
      } catch (InterruptedException ie) {}

      return result;
   }
}

public class SyncMethod extends Thread {

   private Counter counter;
   private String name;

   SyncMethod(Counter counter, String name) {
      this.counter = counter;
      this.name = name;
   }

   public void run() {
      while(true) {
         System.out.println("Name: " + name +
            ", Counter: " + counter.increment());
      }
   }
```

```
   public static void main(String[] args) {
      Counter counter = new Counter();
      Thread[] t = new Thread[4];
      for(int i=0; i<4; i++) {
         t[i] = new SyncMethod(counter, "Thread_" + i);
         t[i].start();
      }
   }
}
```

Listing 6.11 *(Forts.)*
Die Instanz der Klasse Counter ist nicht synchronisiert

Beim Start der Applikation werden fünf Instanzen der `SyncMethod`-Klasse erzeugt. Diese erben jeweils von `Thread` und geben in ihrer `run()`-Methode ihren Namen und den Zählerstand aus. Im Konstruktor wird ihnen eben dieser Name und eine Instanz der `Counter`-Klasse als Parameter übergeben.

Noch einmal zur Verdeutlichung: Alle fünf Thread-Instanzen greifen auf die selbe Counter-Instanz zurück. Deren Methode `increment()` ist (noch) nicht synchronisiert. Und das kommt dann dabei heraus:

```
Name: Thread_0, Counter: 0
Name: Thread_1, Counter: 1
Name: Thread_2, Counter: 2
Name: Thread_3, Counter: 3
Name: Thread_3, Counter: 7
Name: Thread_2, Counter: 6
Name: Thread_1, Counter: 5
Name: Thread_0, Counter: 4
Name: Thread_0, Counter: 11
Name: Thread_1, Counter: 10
Name: Thread_2, Counter: 9
Name: Thread_3, Counter: 8
```

Um dieses Verhalten zu vermeiden, kennzeichnen wir die Methode `increment()` der `Counter`-Klasse als `synchronized`:

```
class Counter {

   private int id = 0;

   public synchronized int increment() {
      int result = id++;
      try {
         Thread.sleep(100);
      } catch (InterruptedException ie) {}

      return result;
   }
}
```

Listing 6.12
Nun ist die Methode increment() als synchronized gekennzeichnet

Wird die Methode `increment()` als `synchronized` gekennzeichnet, sieht die Ausgabe komplett anders aus:

```
Name: Thread_0, Counter: 0
Name: Thread_1, Counter: 1
Name: Thread_2, Counter: 2
Name: Thread_3, Counter: 3
Name: Thread_0, Counter: 4
```

```
Name: Thread_1, Counter: 5
Name: Thread_2, Counter: 6
Name: Thread_3, Counter: 7
Name: Thread_0, Counter: 8
Name: Thread_1, Counter: 9
Name: Thread_2, Counter: 10
Name: Thread_3, Counter: 11
Name: Thread_0, Counter: 12
Name: Thread_1, Counter: 13
Name: Thread_2, Counter: 14
Name: Thread_3, Counter: 15
```

Beachten Sie auch die Reihenfolge der Threads – diese ist nunmehr wesentlich geordneter, denn das Schlüsselwort `synchronized` bewirkt, das die komplette Methode solange gesperrt bleibt, bis sie komplett durchlaufen worden ist. Salopp gesagt bedeutet das für die Threads: Hinten anstellen und warten, bis man an der Reihe ist.

6.8.2 wait und notify

Ein synchronisiertes Objekt oder ein synchronisierter Block verfügt über eine sogenannte Warteliste. In dieser Liste werden vom Scheduler alle Threads eingetragen, die auf die Beendigung der Sperre des Objekts oder Blocks warten.

Der Aufruf der Methode `wait()` innerhalb eines synchronisierten Blocks oder einer synchronisierten Methode führt dazu, das diese Methode oder der Block als wartend markiert und in die Warteliste des Objekts, anhand dessen synchronisiert wird, aufgenommen werden kann. Ein Objekt kann sich somit aktiv selbst aus dem Verkehr ziehen und anderen Objekten die Abarbeitung des Blocks oder der Methode ermöglichen.

Der Aufruf von `notify()` sorgt dafür, das die als wartend markierten Threads die entsprechenden Methoden oder Blöcke ausgeführen können. Der Aufrufer wird dagegen aus der Warteliste entfernt und stellt sich somit quasi an deren Ende wieder an. Die Warteliste wird danach wie gewohnt abgearbeitet.

Wichtig ist, das stets die `wait()`- und `notify()`-Methoden der als Parameter des `synchronized`-Statements verwendeten Objekt-Instanz aufgerufen werden.

Beide Methoden eignen sich somit für Szenarien, in denen die zeitliche Abfolge der Prozesse gesteuert werden sollen.

Nehmen wir ein Beispiel: Dieses demonstriert anhand eines *Producer/Consumer*-Ansatzes die Synchronisation und zeitliche Steuerung von Prozessen. Der Producer (Produzent) stellt etwas her – in unserem Fall einige zufällig erzeugte Zahlen, die von einem der Consumer (Verbraucher) dann ausgegeben werden sollen. Dabei soll stets nur ein Consumer die Ausgabe übernehmen und in der Zeit dürfen auch keine anderen Daten in die zum Datenaustausch verwendete ArrayList eingestellt werden. Um den ganzen Prozess noch etwas komplexer zu gestalten, pausieren sowohl Producer als auch Consumer jeweils für einen zufälligen Zeitraum.

Sehen wir uns an, wie dies umgesetzt werden könnte:

Listing 6.13
wait() und notify() im Einsatz

```
import java.util.ArrayList;

class Producer extends Thread {

   private ArrayList<Integer> data;

   Producer(ArrayList<Integer> data) {
      this.data = data;
   }

   public void run() {
      int current;

      while(true) {
         synchronized(data) {
            for(int i=0;i<3;i++) {
               current = (int)(Math.random() * 100);
               data.add(current);
               System.out.println("Producer produced: " + current);
            }

            data.notify();
         }

         try {
            Thread.sleep((int)Math.random() * 100);
         } catch (InterruptedException e) {}
      }
   }
}

class Consumer extends Thread {

   private String name;
   private ArrayList<Integer> data;

   Consumer(String name, ArrayList<Integer> data) {
      this.name = name;
      this.data = data;
   }

   public void run() {
      while(true) {
         synchronized(data) {
            if(data.size() < 1) {
               try {
                  data.wait();
               } catch (InterruptedException e) {}
            } else {
               for(int current : data) {
                  System.out.println(
                     "Consumer " + name +
                        " found value: " + current);
               }
```

Listing 6.13 *(Forts.)*
wait() und notify() im Einsatz

```
                data.clear();
            }

            try {
               Thread.sleep((int) Math.random() * 100);
            } catch (InterruptedException e) {}
         }
      }
   }
}

public class ProducerConsumer {

   public static void main(String[] args) {
      ArrayList<Integer> data = new ArrayList<Integer>();

      Producer producer = new Producer(data);
      producer.start();

      Consumer c1 = new Consumer("#1", data);
      c1.start();

      Consumer c2 = new Consumer("#2", data);
      c2.start();
   }
}
```

Sowohl Consumer- als Producer-Instanzen verfügen über eine Instanz einer `ArrayList`. Es handelt sich dabei immer um die gleiche Objekt-Instanz. Anhand dieser Instanz können Synchronisierung, Warten und Aktivieren der verschiedenen Threads erfolgen. Die Consumer-Threads prüfen in ihrer `run()`-Methode, ob in der ArrayList Elemente enthalten sind. Wenn das nicht der Fall sein sollte, wird der aktuelle Block mit Hilfe von `wait()` in die Warteschleife eingereiht und wartet somit, bis er wieder aktiviert wird. Sobald das geschieht, gibt der Block die in der `ArrayList` enthaltenen Daten aus und leert diese danach.

Der Produzent erzeugt innerhalb seiner `run()`-Methode stets drei neue Elemente für die `ArrayList`. Da dies auch in einem `synchronized`-Block geschieht, findet die Erzeugung nur statt, wenn dieser Block nicht gesperrt ist. Nach der Erzeugung der Daten benachrichtigt der Produzent alle wartenden Blöcke mit Hilfe der Methode `notify()`.

Dieser Vorgang wiederholt sich nun kontinuierlich - und die Ausgabe zeigt, das `wait()` und `notify()` gute Dienste leisten:

```
Producer produced: 82
Producer produced: 2
Producer produced: 79
Consumer #1 found value: 82
Consumer #1 found value: 2
Consumer #1 found value: 79
Producer produced: 38
Producer produced: 46
Producer produced: 53
Consumer #2 found value: 38
Consumer #2 found value: 46
```

```
Consumer #2 found value: 53
Producer produced: 9
Producer produced: 98
Producer produced: 64
Consumer #2 found value: 9
Consumer #2 found value: 98
Consumer #2 found value: 64
Producer produced: 37
Producer produced: 48
Producer produced: 47
Consumer #1 found value: 37
Consumer #1 found value: 48
Consumer #1 found value: 47
Producer produced: 7
Producer produced: 60
Producer produced: 49
Consumer #1 found value: 7
Consumer #1 found value: 60
Consumer #1 found value: 49
```

6.9 Fazit

Die Arbeit mit Threads ist komplex und spannend zugleich. Threads bieten uns viele Möglichkeiten, sich wiederholende und ressourcenintensive Vorgänge auszulagern und somit die Ausführung der Haupt-Applikation nicht zu behindern.

Die Zuordnung verschiedener Threads zu Thread-Gruppen bietet interessante Möglichkeiten der Verwaltung. Das die Arbeit mit Threads auch komplex werden kann, belegen die gezeigten Synchronisations-Szenarien. Doch mit Hilfe des Schlüsselworts `synchronized` und der beiden Methoden `wait()` und `notify()` sind wir in der Lage, auch komplexere Synchronisations-Aufgaben erfolgreich zu bewältigen.

7

Prozesse und System-Umgebung

Prozesse erlauben es, andere Programme aus Java heraus zu starten und laufen zu lassen. Prozesse können auch weiterlaufen, wenn das Java-Programm, das sie gestartet hat, schon beendet worden ist. Es ist auch möglich externen Prozessen Parameter zu übergeben oder ihre Ausgaben (soweit sie über die Standard-Ausgabe erfolgen) auszulesen.

Entwickler erhalten somit eine mächtige Möglichkeit des Zugriffs auf Betriebssystem-Komponenten und können teils sehr aufwändige Operationen mit Hilfe des Betriebssystems durchführen.

Achtung

Die Verwendung von Prozessen innerhalb von Java-Programmen birgt ein enormes Risiko in sich: Da Sie angeben müssen, welches Programm Sie ausführen wollen, wird Ihre Java-Applikation selbst nicht mehr portabel und kann auf anderen Systemen unter Umständen gar nicht mehr ausgeführt werden. Überlegen Sie sich also genau, ob Sie die gewünschte Funktionalität in Form einer externen Applikation einbinden wollen oder ob Sie nicht eine Möglichkeit sehen, das gewünschte Ziel mit Bordmitteln zu erreichen.

7.1 Starten einer externen Applikation

Mit Hilfe der Methode `exec()` der `Runtime`-Klasse können Sie eine externe Applikation starten. Lassen Sie uns diesen Vorgang am Beispiel des Windows-Notepads einmal durchspielen:

Listing 7.1
Starten eines externen Prozesses

```
import java.io.IOException;

/**
 * This class starts a new process
 */
public class SimpleProcess {

   /**
    * Starts the new process
    */
   public static void main(String[] args) {
      try {
         Runtime r = Runtime.getRuntime();
         Process p = r.exec("notepad.exe c:\\boot.ini");
         p.exitValue();
      }
      catch (IOException e) { }
      catch (IllegalThreadStateException e) { }
   }
}
```

Die Klasse *SimpleProcess* definiert eine statische Methode `main()`, die als Einsprungspunkt für eine Kommandozeilen-Applikation dient. Über die statische Methode `getRuntime()` der `Runtime`-Klasse ermitteln wir eine `Runtime`-Instanz. Damit haben wir die Möglichkeit, auf das umgebende Betriebssystem zuzugreifen.

Diese Möglichkeit des Zugriffs nehmen wir auch wahr: Mit Hilfe der Methode `exec()` starten wir einen externen Prozess und weisen die resultierende *Process*-Instanz der lokalen Variablen `p` zu. Die von uns gestartete Applikation ist das Windows Notepad, mit dessen Hilfe wir die Datei *boot.ini* öffnen wollen. Letztere Datei wird *notepad.exe* als Parameter übergeben.

Da eine `IOException` auftreten kann, wenn sich die Datei *notepad.exe* nicht innerhalb des durchsuchten Pfades befindet, werden die Anweisungen in einem *try-catch*-Block gekapselt.

Durch `exitValue()` rufen wir den Rückgabe-Code des Prozesses ab. Da dieser aber noch läuft, wird Java eine `IllegalThreadStateException` werfen – die wir mit Hilfe eines `catch`-Statements abfangen. Uns stört dies nicht, denn unsere Applikation wird unter Windows eine neue Notepad-Instanz gestartet haben, was das Ziel dieser Bemühungen war. Unter anderen Betriebssystemen werden wir leider keine neue Notepad-Instanz gestartet bekommen, denn das ist der Haken, wenn via Process ein externes Programm gestartet wird: Das zu startende Programm existiert möglicherweise überhaupt nicht auf dem Zielsystem.

Kompilieren Sie die Klasse und führen Sie sie mit Hilfe von

```
java SimpleProcess
```

aus. Soweit Sie auf einem Windows-System arbeiten und sich die Datei *notepad.exe* in einem Verzeichnis befindet, auf das die *PATH*-Umgebungsvariable von Windows zeigt, werden Sie unversehens eine neue Notepad-Instanz zu sehen bekommen:

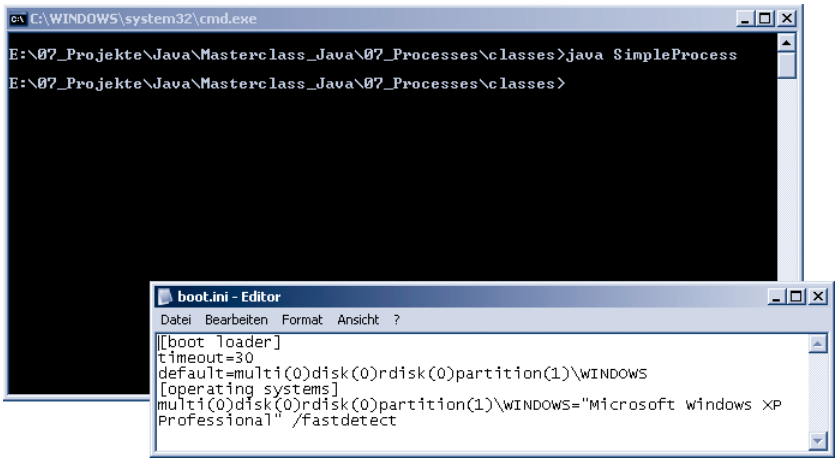


Abbildung 7.1
Die Datei *boot.ini* wird innerhalb des Windows-Notepads angezeigt – der Prozess wurde von Java gestartet

Sollten Sie auf einem anderen System arbeiten oder sich *notepad.exe* nicht finden lassen, werden Sie keinerlei Ausgabe zu sehen bekommen – die resultierende `IOException` wurde zuvor in der Klasse abgefangen.

7.2 Einlesen der Rückgabe eines Prozesses

Natürlich können Sie nicht nur neue Prozesse starten, sondern auch deren Beendigung abwarten und anschließend die Ausgabe der Prozesse, die diese an die Standard-Ausgabe-Pipe richten, abfangen und verarbeiten.

Versuchen wir diesmal, einen Prozess zu starten, der einen Internet-Webserver anpingt und somit dessen Erreichbarkeit auf einer sehr niedrigen Protokoll-Ebene prüft:

Listing 7.2
Anpingen eines Web-Servers per externem Prozess

```
import java.io.BufferedInputStream;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.io.InputStreamReader;
import java.io.IOException;

/**
 * Pings an internet web-server and checks its availability
 */
public class Ping {

   /**
    * The application's main method which pings the remote server
    */
   public static void main(String[] args) {
      // Get Runtime
      Runtime r = Runtime.getRuntime();
      BufferedInputStream in = null;
      BufferedReader rdr = null;
```

Listing 7.2 *(Forts.)*
Anpingen eines Web-Servers per externem Prozess

```
        PrintWriter out = null;

       // Create new PrintWriter for use with German Umlauts
        try {
           out = new PrintWriter(
              new OutputStreamWriter(System.out, "Cp850") );
        } catch (UnsupportedEncodingException e) {
           e.printStackTrace();
        }

        try {
           // Create Process
           // For Unix-systems: use "ping -c 4 -i 1 java.sun.com"
           Process p = r.exec("ping java.sun.com");

          // Wait until the process has finished
           p.waitFor();

           // Create BufferedReader for the data
           in = new BufferedInputStream(p.getInputStream());
           rdr = new BufferedReader(
              new InputStreamReader(in, "cp850"));

           // Read all data
           String line = null;
           while(null != (line = rdr.readLine())) {
              if(line.length() > 0) {
                 out.println(line);
                 out.flush();
              }
           }
        } catch (IOException e) {
           e.printStackTrace();
        } catch (InterruptedException e) {
           e.printStackTrace();
        } finally {
           try {
              rdr.close();
              in.close();
           } catch (IOException e) {}
        }
     }
}
```

Dieses Beispiel ist zugegebenermaßen etwas umfangreicher – dennoch wollen wir uns die Mühe machen und es detailliert betrachten.

Innerhalb der statischen Methode `main()` erzeugen wir zunächst eine Referenz auf die die aktuelle System-Umgebung repräsentierende Klasse `Runtime`, indem wir deren statische Methode `getRuntime()` aufrufen.

Nun fügen wir etwas ein, was nur Sinn macht, wenn man Umlaute über die Java-Konsole ausgeben will – wir erzeugen eine neue `PrintWriter`-Instanz, der wir im Konstruktor eine neue `OutputStreamWriter`-Instanz übergeben. Dieser wiederum weisen wir in deren Konstruktor den Standard-Ausgabestrom von Java, der über `System.out()` erreichbar ist, zu und teilen mit Hilfe des zweiten

Parameters mit, welches Encoding dabei verwendet werden soll – in unserem Fall die *Codepage* 850, die zumindest bei Windows-NT-Systemen und deren Nachfolgern eine Ausgabe von Umlauten ermöglicht.

Nachdem wir den Ausgabestrom erfolgreich auf unsere lokale `PrintWriter`-Instanz umgeleitet haben, können wir den eigentlichen Prozess starten. Wir bedienen uns dabei der *ping*-Applikation, die zumindest unter Linux und Windows NT / 2000 / XP und Windows Server 2003 zur Verfügung steht. Dabei handelt es sich um ein kleines Kommandozeilen-Utility, das als Parameter die IP-Adresse oder den Namen des anzupingenden Servers entgegennimmt.

Das Umbiegen des Standard-Ausgabe-Stroms auf eine eigene `PrintWriter`-Instanz mit einem speziellen Encoding ist leider auch bei den neuesten Java-Versionen nötig, denn `System.out()` unterstützt kein Encoding, das Umlaute erlaubt. Dies wurde bis dato als Bug bezeichnet und wird wohl auch zukünftig nicht korrigiert werden – schade für alle nicht-englischen Entwickler, die unter Umständen Umlaute ausgeben wollen.

Achtung

Sollten Sie dieses Beispiel unter Linux oder einem anderen Unix-System ausführen wollen, müssen Sie den Aufruf von `exec()` ändern – statt *ping java.sun.com* muss es hier *ping -c 4 -i 1 java.sun.com* heißen. Wieder ein Beispiel dafür, dass man `exec()` nur mit äußerster Vorsicht einsetzen sollte.

Den Prozess starten wir mit Hilfe der Methode `exec()` unserer *Runtime*-Instanz. Diese liefert uns eine Instanz der `Process`-Klasse als Rückgabe. Deren Methode `waitFor()` wartet dann so lange, bis der Prozess beendet worden ist – in diesem Fall also, bis der Server angepingt worden ist.

Nun wollen wir die Rückgabe des Prozesses einlesen. Diese Rückgabe erreichen wir in ihrer Reinform mit Hilfe der Methode `getInputStream()` der *Process*-Instanz. Diese Reinform ist allerdings nicht besonders gut für unsere Zwecke geeignet – wir müssten selbst prüfen, ob und wann Daten vorliegen und wann das Auslesen der Daten beendet werden kann. Aus diesem Grund weisen wir den von `getInputStream()` zurückgegebenen `InputStream` der `Process`-Instanz einer neuen `BufferedInputStream`-Instanz zu, die darauf spezialisiert ist, mit Daten umzugehen, die nicht am Stück, sondern nach und nach eintreffen.

Aber auch aus der `BufferedInputStream`-Instanz kann man die Rückgabe leider nicht so problemlos auslesen, wie man sich das wünschen würde – hier müsste man sich mit Bytes herumschlagen und ebenfalls diverse Überprüfungen vornehmen, bis das Einlesen beendet werden könnte.

Also machen wir es uns erneut ein wenig leichter und verwenden eine `InputStreamReader`-Instanz, deren Konstruktor wir die `Stream`-Instanz zuweisen, aus der sie lesen soll und der wir zusätzlich als Parameter mitgeben, dass die Daten mit dem Zeichensatz der Codepage 850 (*Latin-1*) codiert sind. Damit könnten wir die Daten recht einfach auslesen – aber leider immer noch nur als Byte-Array.

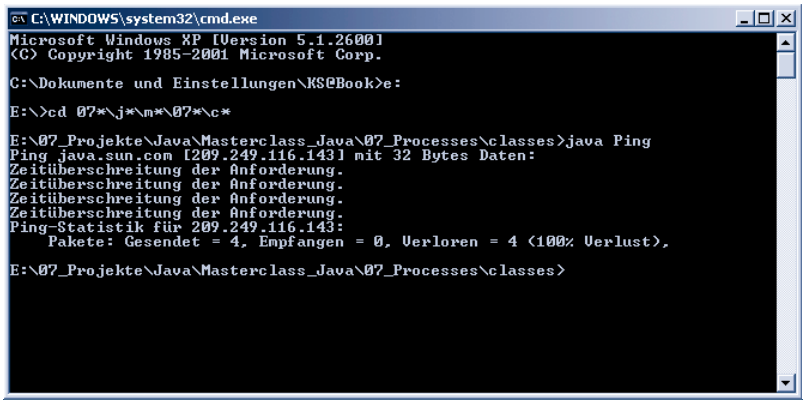
Damit wir uns auch damit nicht herumschlagen müssen, erzeugen wir einen `BufferedReader`, dem wir die eben erzeugte `InputStreamReader`-Instanz im Konstruktor übergeben. Jetzt können wir tatsächlich Daten auslesen!

Auf den ersten Blick mag das Schachteln von *Streams* und *StreamReadern* verwirrend und komplex erscheinen. Auf den zweiten Blick offenbart sich jedoch, dass die Java-Entwickler damit größtmögliche Flexibilität bei der Verwendung unterschiedlichster Datenquellen erlauben.

Das eigentliche Auslesen der Daten geschieht, indem wir die Informationen mit Hilfe der Methode `readLine()` unserer *BufferedReader*-Instanz abrufen. Dies geschieht innerhalb des Kopfes einer `while`-Schleife, die so lange durchlaufen wird, wie wir Daten erhalten und somit die lokale Variable `line`, die eine Zeile aufnimmt, nicht `null` wird. Sollte `line` mindestens ein Zeichen enthalten, geben wir ihren Inhalt aus – samt aller Umlaute und Sonderzeichen, die auf der Konsole tatsächlich dargestellt werden!

Lange Rede, kurzer Sinn: Mit Hilfe von `getInputStream()` können wir die Rückgabe eines Prozesses auslesen. Und mit Hilfe einer eigenen `PrintWriter`-Instanz können wir auf der Konsole via Java auch Umlaute ausgeben.

Abbildung 7.2
Ping per Java mit korrekter Ausgabe von Umlauten



7.3 Freien Speicher ermitteln

Die `Runtime`-Klasse erlaubt es uns, auf einfache Art und Weise zu ermitteln, wie viel Speicher der aktuellen Java-Instanz zur Verfügung steht:

Listing 7.3
Ermitteln des freien Speichers einer Java-Instanz

```
/**
 * Displays the amount of memory of this Java-instance
 */
public class Memory {

   /**
    * The application's main method
    */
   public static void main(String[] args) {
      Runtime r = Runtime.getRuntime();

      long mem = r.freeMemory();
      double kBytes = ((mem / 1024) * 100) / 100;
      double mBytes = ((kBytes / 1024) * 100) /100;

      System.out.println(
         String.format(
            "This Java instance has %d Bytes "
               + "(=%g KBytes and %g MBytes) free!",
            mem, kBytes, mBytes));
   }
}
```

Am Anfang der statischen Klasse `main()` der `Memory`-Klasse erzeugen wir mit Hilfe von `Runtime.getRuntime()` eine Referenz auf die die aktuelle System-Umgebung repräsentierende `Runtime`-Instanz. Nun haben wir Zugriff auf die Methode `freeMemory()`, die uns als Rückgabe die Anzahl der freien Bytes als `long`-Wert liefert.

Dieser wird anschließend durch 1.024 dividiert, damit wir die Anzahl der KBytes bekommen. Nach einer weiteren Division des gerundeten KByte-Wertes haben wir die Größe des zugewiesenen Arbeitsspeichers in MByte ermittelt. Diese Werte werden am Ende ausgegeben.

Wenn Sie die Klasse kompilieren und danach per

```
java Memory
```

ausführen, werden Sie eine Ausgabe analog zu dieser erhalten:

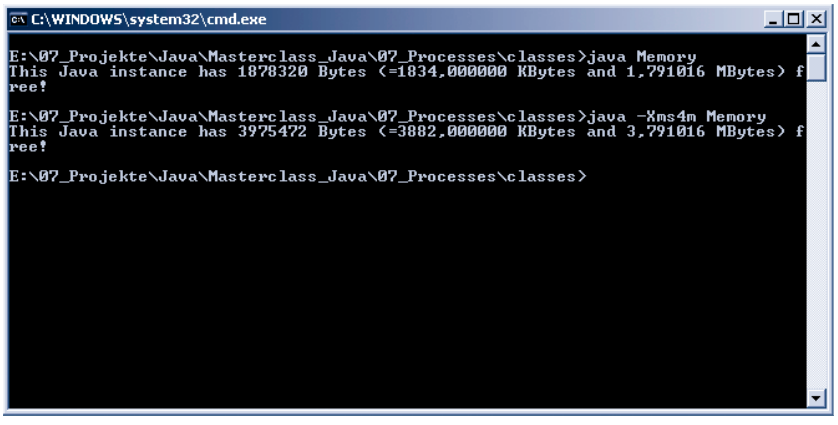


Abbildung 7.3
Ausgabe des verfügbaren Arbeitsspeichers einer Java-Applikation

Tipp

Den Arbeitsspeicher, den eine Java-Applikation initial zur Verfügung hat, können Sie übrigens beeinflussen: Verwenden Sie den Parameter *–Xms<Größe>* beim Aufruf einer Java-Klasse. Der Wert von *<Größe>* kann dabei in Bytes, KBytes oder MBytes angegeben werden. Dies wird angegeben, indem *<Größe>* kein Buchstabe, der Buchstabe *k* oder der Buchstabe *m* nachgestellt wird.

Beispiele:

- Definition eines initialen Arbeitsspeichers von 10.000 Bytes:

  ```
  java -Xms10000 <Class>
  ```

- Initialer Arbeitsspeicher von 100 KBytes:

  ```
  java -Xms100k <Class>
  ```

- Initialer Arbeitsspeicher von 10 MByte:

  ```
  java -Xms10m <Class>
  ```

Neben dem initialen Arbeitsspeicher können Sie auch den maximalen Arbeitsspeicher beeinflussen: Verwenden Sie zu diesem Zweck den Parameter *-Xmx<Größe>*.

7.4 Garbage Collector erzwingen

Mit Hilfe der `Runtime`-Klasse können wir einen *Garbage Collector* für die aktuelle Java-Instanz erzwingen. Dies wird in der Regel einigen Speicher freigeben, kostet aber auch jede Menge System-Ressourcen. Überlegen Sie sich den Einsatz des *Garbage Collectors* deshalb genau – zumal er von Java selbst auch ohne unser Zutun ausgeführt wird.

Um den *Garbage Collector* dennoch zu erzwingen, können Sie sich der Methode `gc()` der `Runtime`-Klasse bedienen:

Listing 7.4 Ausführen des Garbage Collectors

```
/**
 * Executes a garbage collector
 */
public class Garbage {

   /**
    * The application's main method
    */
   public static void main(String[] args) {
      Runtime r = Runtime.getRuntime();

      System.out.println(
         String.format("This Java instance has %d Bytes free!",
            r.freeMemory()));

      System.out.println();
      System.out.println("Performing garbage collector!");
      System.out.println();

      r.gc();

      System.out.println(
         String.format("This Java instance has %d Bytes free!",
            r.freeMemory()));

   }
}
```

In diesem Beispiel erzeugen wir zunächst eine Referenz auf die `Runtime`-Instanz, die die aktuelle System-Umgebung repräsentiert. Dies geschieht wie üblich mit Hilfe der statischen Methode `getRuntime()` der `Runtime`-Klasse.

Nun können wir per `freeMemory()` die Anzahl der freien Bytes ausgeben. Der *Garbage Collector* wird mit Hilfe der Methode `gc()` ausgeführt. Ein nochmaliges `freeMemory()` wird uns bestätigen, dass zumindest einige Bytes freigegeben worden sind.

Kompilieren Sie die Klasse und führen Sie sie per

```
java Garbage
```

aus. Sie werden eine Ausgabe analog zu dieser sehen:

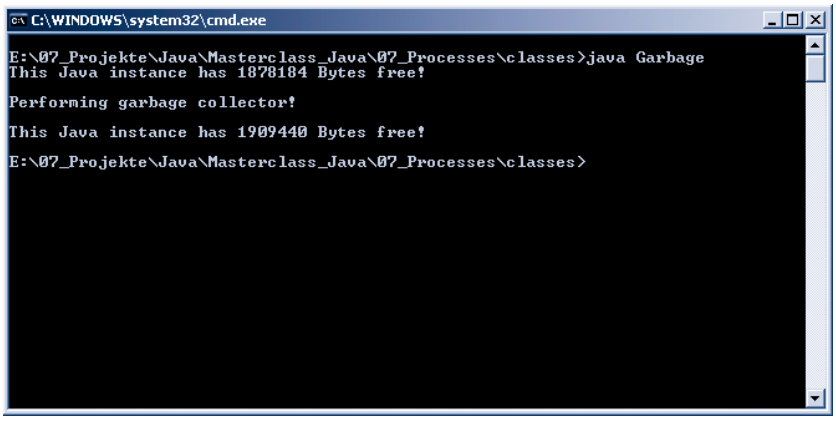


Abbildung 7.4
Der Garbage Collector hat Speicher freigegeben

Achtung

Im Normalfall besteht kein Grund dazu, den *Garbage Collector* selbst aufzurufen. Java erledigt dies von selbst. Für einen derartigen Aufruf sollten aufgrund des Performance-Hungers des *Garbage Collectors* gute Gründe gegeben sein – etwa ein Speicher-Engpass oder das sofortige und gründliche Aufräumen nicht mehr benötiger Klassen.

7.5 Umgebungs-Informationen abrufen

Die `System`-Klasse stellt uns über die Methode `getProperties()` eine elegante Möglichkeit des Abrufens von Umgebungs-Informationen zur Verfügung. Diese Informationen können recht einfach durchlaufen werden, da sie in Form einer `Properties`-Instanz vorliegen:

Listing 7.5
Ausgabe von Javas Umgebungs-Informationen

```
import java.util.Properties;
import java.util.Enumeration;

/**
 * Displays Java's runtime variables
 */
public class Env {


   /**
    * The main method of this class
    */
   public static void main(String[] args) {
      Properties env = System.getProperties();
      Enumeration keys = env.keys();
```

Listing 7.5 *(Forts.)*
Ausgabe von Javas Umgebungs-Informationen

```
        while(keys.hasMoreElements()) {
            String key = (String)keys.nextElement();
            String value = env.getProperty(key);

            System.out.println(String.format("%s = %s", key, value));
        }
    }
}
```

Der Abruf der Umgebungs-Informationen geschieht, indem wir mit Hilfe von `System.getProperties()` eine `Properties`-Instanz erhalten und der lokalen Variablen `env` zuweisen.

Wenn Sie den Wert eines bestimmten Schlüssels bestimmen wollen, müssen Sie nicht über `System.getProperties().getProperty()` gehen. Stattdessen können Sie auch die bereitgestellte Abkürzung via `System.getProperty()` verwenden und dieser Methode den gewünschten Schlüssel als Parameter übergeben – das tippt sich etwas besser und ist auch leichter lesbar.

Nun können wir alle Schlüssel aus der lokalen `Properties`-Instanz `env` auslesen. Diese Schlüssel stehen uns in Form einer *Enumeration* zur Verfügung – wir können sie deshalb mit Hilfe einer `while`-Schleife durchlaufen. Deren Abbruchbedingung ist, dass keine weiteren Schlüssel in der *Enumeration* vorhanden sind, was wir mit Hilfe der Methode `hasMoreElements()` feststellen können. Den eigentlichen Schlüssel können wir nun durch `nextElement()` ermitteln, allerdings wird er in Form einer `Object`-Instanz zurückgegeben, die wir deshalb noch in einen *String* casten müssen, bevor wir sie der lokalen Variablen `key` zuweisen können.

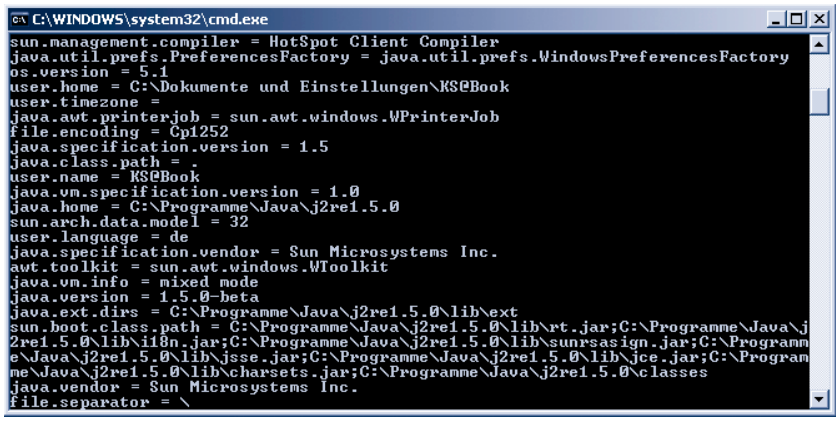
Jetzt sind wir in der Lage, den Wert aus der *Properties*-Instanz auszulesen – wir nutzen dazu deren Methode `getProperty()`, der wir als Parameter den zuvor ermittelten und nun in `key` enthaltenen Schlüssel übergeben.

Diese Informationen geben wir anschließend aus. Wenn Sie die Klasse kompilieren, können Sie sie mit Hilfe von

```
java Env
```

ausführen und werden eine Ausgabe ähnlich dieser erhalten:

Abbildung 7.5
Von Java bereitgestellte Umgebungs-Informationen



7.6 Betriebssystem und Java-Version bestimmen

Nun, da Sie wissen, wie Sie auf die von Java bereitgestellten Informationen zugreifen können, sollte es kein Problem mehr darstellen, Betriebssystem und -Version zu bestimmen.

Noch leichter wird es, wenn man weiß, dass diese Informationen mit Hilfe der Schlüssel *os.name* und *os.version* abgerufen werden können. Und mit Hilfe des Schlüssels *java.version* erhalten wir die Versions-Informationen von Java:

Listing 7.6
Ermitteln von Betriebssystem- und Java-Versions-Informationen

```
/**
 * Displays OS and Version
 */
public class OS {

   public static void main(String[] args) {
      System.out.println(
         String.format("OS: %s",
            System.getProperty("os.name")));
      System.out.println(
         String.format("Version: %s",
            System.getProperty("os.version")));
      System.out.println(
         String.format("Java-Version: %s",
            System.getProperty("java.version")));
   }
}
```

Innerhalb der statischen Methode `main()` greifen wir auf die Java-Umgebungs-Informationen zu, indem wir die entsprechenden Schlüssel `os.name`, `os.version` und `java.version` der Methode `getProperty()` der `System`-Klasse als Parameter übergeben.

Wenn Sie die Klasse kompilieren und via

```
java OS
```

ausführen, werden Sie unter Windows XP folgende Ausgabe erhalten:

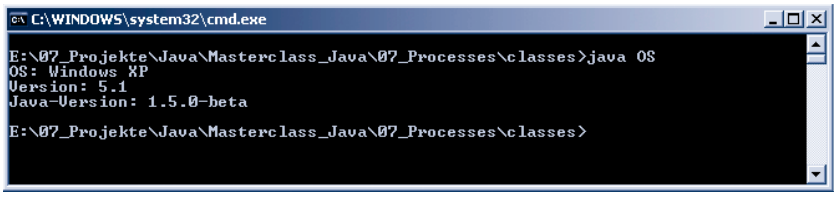


Abbildung 7.6
Ausgabe von Betriebssystem, -Version und Java-Version

7.7 Informationen zum aktuellen Nutzer ermitteln

Die Java-Runtime gibt uns eine Menge Informationen über den aktuellen Nutzer an die Hand, die Sie mit Hilfe von `System.getProperty()` abrufen können.

Diese Informationen sind im Einzelnen:

- `user.country (*)`: Kürzel des Landes, das der Nutzer in den System-Einstellungen angegeben hat – beispielsweise *DE* für Deutschland oder *AT* für Österreich
- `user.dir`: Aktuelles Arbeitsverzeichnis
- `user.variant (*)`: Verwendete Variante der Länder- und Sprach-Einstellungen
- `user.home`: Home-Verzeichnis des Nutzers (bei Windows beispielsweise der Ordner „Eigene Dateien“)
- `user.timezone (*)`: Verwendete Zeitzone
- `user.name`: Anmeldename des Nutzers
- `user.language (*)`: Kürzel der Sprache, die der Nutzer aktiviert hat – beispielsweise *de* für Deutsch oder *en* für Englisch

Mit einem Stern *(*)* gekennzeichnete Elemente sind nicht auf jedem System vorhanden – die drei Schlüssel `user.dir`, `user.home` und `user.name` werden aber in jedem Fall einen Wert zurückgeben, da sie zu den zugesicherten System-Informationen gehören.

Der Abruf der Informationen erfolgt mit Hilfe von `System.getProperty()`:

```
Properties env = System.getProperties();
Enumeration keys = env.keys();

while(keys.hasMoreElements()) {
   String key = (String)keys.nextElement();

   if(key.startsWith("user.")) {
      String value = System.getProperty(key);
      System.out.println(String.format("%s = %s", key, value));
   }
}
```

In diesem Fragment weisen wir der lokalen Variablen `env` eine Referenz auf die von `System` bereitgestellte `Properties`-Instanz zu. Die Schlüssel der `Properties`-Instanz lesen wir mit Hilfe von deren *Enumeration* `keys` aus.

Solange Schlüssel ausgelesen werden können (`keys.hasMoreElements()` also `true` zurückgibt), rufen wir nun jeweils einen Schlüssel ab, konvertieren ihn explizit in einen `String` und weisen ihn der Variablen `key` zu. Wenn deren Inhalt mit der Zeichenfolge „user.“ beginnt (was mit Hilfe der Methode `startsWith()` überprüft werden kann), ermitteln wir den durch den Schlüssel repräsentierten Wert mit Hilfe von `System.getProperty()` und geben beide Informationen aus.

Wenn Sie dies auf einem Windows-System ausführen, werden Sie eine Ausgabe analog zu dieser erhalten:

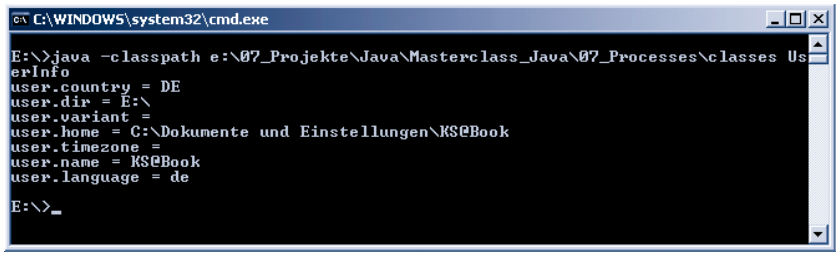


Abbildung 7.7
Ausgabe von Benutzer-Informationen

7.8 Zugesicherte Umgebungsvariablen

Java stellt potentiell eine große Anzahl Umgebungsvariablen bereit, die über `System.getProperty()` abgerufen werden können. Nicht alle dieser Variablen sind auf jedem System verfügbar, aber Java sichert die Existenz zumindest einiger Umgebungsvariablen zu:

- `java.version`: Java-Version
- `java.vendor`: Anbieter
- `java.vendor.url`: Anbieter-Homepage
- `java.home`: Installations-Verzeichnis
- `java.vm.specification.version`: JVM Spezifikations-Version
- `java.vm.specification.vendor`: JVM Spezifikations-Anbieter
- `java.vm.specification.name`: JVM Spezifikations-Name
- `java.vm.version`: JVM-Version
- `version java.vm.vendor`: JVM-Anbieter
- `java.vm.name`: JVM-Name
- `java.specification.version`: JRE-Version
- `java.specification.vendor`: JRE-Anbieter
- `java.specification.name`: JRE-Spezifikation
- `java.class.version`: Java Class Format-Version
- `java.class.path`: Klassenpfad
- `java.library.path`: Pfade, die durchsucht werden, wenn Java-Libraries geladen werden sollen
- `java.io.tmpdir`: Temporäres Verzeichnis
- `java.compiler`: Kompiler-Name
- `java.ext.dirs`: Pfade, die durchsucht werden, wenn Java-Extensions geladen werden sollen
- `os.name`: Betriebssystem-Name
- `os.arch`: Prozessor-Architektur

- `os.version`: Betriebssystem-Version
- `file.separator`: Trenner zwischen Pfaden und Verzeichnissen
- `path.separator`: Trenner zwischen mehreren Pfaden
- `line.separator`: Zeilen-Umbruch-Zeichenfolge (`"\n"` bei Unix, `"\r\n"` bei Windows)
- `user.name`: Anmeldename des aktuellen Benutzers
- `user.home`: Home-Verzeichnis des aktuellen Benutzers
- `user.dir`: Aktuelles Arbeitsverzeichnis

Interessant für einen produktiven Einsatz werden sicherlich die Informationen zum Betriebssystem, zum Benutzer, zu den Pfaden und möglicherweise auch die Java-Version sein. Andere Informationen – etwa zur *JRE*- oder *JVM*-Version – werden in der Praxis nicht allzu häufig angewendet.

Achtung

Wenn Sie andere Java-Umgebungs-Variablen als die zugesicherten verwenden wollen, geschieht dies ausdrücklich auf eigenes Risiko. Sie sollten deshalb die Existenz eines Wertes, den Sie mit Hilfe von `getProperty()` ermitteln, immer hinterfragen und sowohl auf `null` als auch auf eine Mindestlänge von einem Zeichen prüfen:

```
String value = System.getProperty(key);
if(null != value && value.length > 0) {
   System.out.println(String.format("Wert von %s: %s", key, value));
}
```

7.9 System-Umgebungs-Informationen abrufen

Neu bei Java 1.5 ist die Möglichkeit, auf die Umgebungsvariablen des Betriebssystems zuzugreifen. Sie können so beispielsweise das *Home*-Verzeichnis des aktuell angemeldeten Nutzers auslesen oder die *PATH*-Angabe interpretieren:

Listing 7.7
Ausgabe aller Umgebungs-Variablen des Betriebssystems

```
import java.util.Map;
import java.util.Iterator;

/**
 * Displays system environmental variables
 */
public class SystemEnv {

   /**
    * The main method of this class
    */
   public static void main(String[] args) {
      Map<String, String>env = System.getenv();
```

```
        Iterator<String>keys = env.keySet().iterator();

        while(keys.hasNext()) {
            String key = keys.next();
            String value = env.get(key);

            System.out.println(String.format("%s = %s", key, value));
        }
    }
}
```

Listing 7.7 *(Forts.)*
Ausgabe aller Umgebungs-Variablen des Betriebssystems

Am Anfang der Methode `main()` deklarieren wir in der Variablen `env` eine neue Instanz der `Map`-Klasse, die sowohl für Schlüssel als auch für Werte nur Strings entgegennehmen darf. Zugleich weisen wir `env` den Inhalt von `System.getenv()` zu und haben somit Zugriff auf alle Umgebungs-Informationen, die das Betriebssystem zur Verfügung stellt.

Mit Hilfe des *String-Iterators* `keys` werden wir alle Schlüssel durchlaufen, damit wir mit deren Hilfe die entsprechenden Werte abrufen können. Wir deklarieren also den *Iterator* und weisen ihm die via `env.keySet().iterator()` erreichbare *Iterator*-Instanz zu.

Das eigentliche Durchlaufen der Informationen findet mit Hilfe einer `while`-Schleife statt. Die Abbruchbedingung der Schleife ist dann erreicht, wenn die Methode `hasNext()` der `Iterator`-Instanz `false` zurückgibt und somit keine weiteren Schlüssel mehr vorhanden sind.

Das Abrufen des Wertes des aktuellen Schlüssels ist dann fast trivial – dies geschieht mit Hilfe der Methode `get()` der `Map`-Instanz, der wir den Schlüssel als Parameter übergeben. Anschließend werden Schlüssel und Wert ausgegeben.

Wenn Sie die Klasse kompilieren und per

```
java SystemEnv
```

ausführen, werden Sie beispielsweise bei einem Windows XP-System eine Ausgabe ähnlich dieser erhalten:

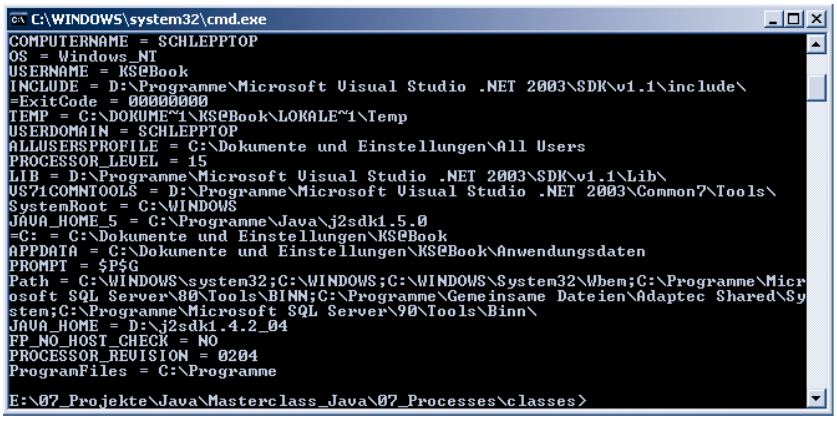


Abbildung 7.8
Ausgabe von Umgebungsvariablen

7.10 Zusammenfassung

Die `System`-, `Runtime`- und `Process`-Klassen erlauben es uns, mit dem umgebenden Betriebssystem zu interagieren. Das Problem bei der Sache ist, dass diese Interaktion meist zu Lasten von Portierbarkeit oder Performance geht. Der Einsatz von externen Prozessen und das sich Verlassen auf das Vorhandensein bestimmter Umgebungs-Informationen setzt also stets ein gründliches Abwägen der Vor- und Nachteile voraus. Ebenso sollten Sie nicht vergessen, stets damit zu rechnen, dass manche Dinge nicht so funktionieren werden, wie Sie es beim Testen unter einem bestimmten System erwartet haben. Kapseln Sie deshalb kritische Elemente in `try-catch`-Blöcken und überprüfen Sie die Werte von Variablen, bevor Sie mit ihnen arbeiten.

8

Arbeiten mit der Kommandozeile

Die Kommandozeile stellt die einfachste Möglichkeit dar, Daten einzulesen und auszugeben. Meist wird sie von einfacheren Utilities genutzt, die vielen Benutzern völlig verborgen bleiben – dabei bietet sie (wenn man von einer grafischen Oberfläche einmal absieht) eigentlich alles, was wir benötigen, um effektiv mit dem Benutzer zu interagieren.

8.1 Aufbau einer Kommandozeilen-Applikation

Eine Klasse, die von der Kommandozeile (*cmd.exe* unter Windows 2000 / XP / 2003, *command.com* unter Windows 95-SE, *shell* unter Unix) laufen soll, muss genau eine Anforderung erfüllen: Sie muss über eine als `public` gekennzeichnete statische Methode `main()` verfügen, die Parameter in Form eines *String*-Arrays entgegennimmt.

```
class <Klassenname> {
   public static void main(String[] args) { ... }
}
```

Umgesetzt in einer tatsächlich ausführbaren Lösung ergäbe sich beispielsweise dieser Code:

Listing 8.1
Einfache Kommandozeilen-Applikation

```
/**
 * Simple commandline application
 */
public class SimpleCommandLine {

   /**
    * Entry-point when called from the commandline
    */
   public static void main(String[] args) {
      System.out.println("Simple Commandline application");
      if(null != args && args.length > 0) {
         System.out.println("==============================");
```

Listing 8.1 *(Forts.)*
Einfache Kommandozeilen-Applikation

```
            System.out.println(
               "You provided me with the following arguments:");
            for(String arg : args) {
               System.out.println(arg);
            }
            System.out.println("==============================");
         }
         System.out.println("Finished...");
      }
}
```

Unsere Klasse `SimpleCommandLine` ist von der Kommandozeile aus ausführbar, denn sie deklariert eine statische Methode `main()`, die als `public` gekennzeichnet ist, und nimmt ein `String`-Array als Parameter entgegen. Innerhalb dieser Methode wird zunächst ein kleiner Hinweistext ausgegeben.

Anschließend wird festgestellt, ob das `String`-Array `args` ungleich `null` ist und mindestens ein Element beinhaltet, was mit Hilfe von `args.length` überprüft werden kann:

```
if(null != args && args.length > 0) { ... }
```

Falls beide Bedingungen zutreffen, kann davon ausgegangen werden, dass Parameter übergeben worden sind. Mit Hilfe eine `for-each`-Schleife werden die Parameter nacheinander durchlaufen, der Variablen `arg` zugewiesen und ausgegeben:

```
for(String arg : args) {
   System.out.println(arg);
}
```

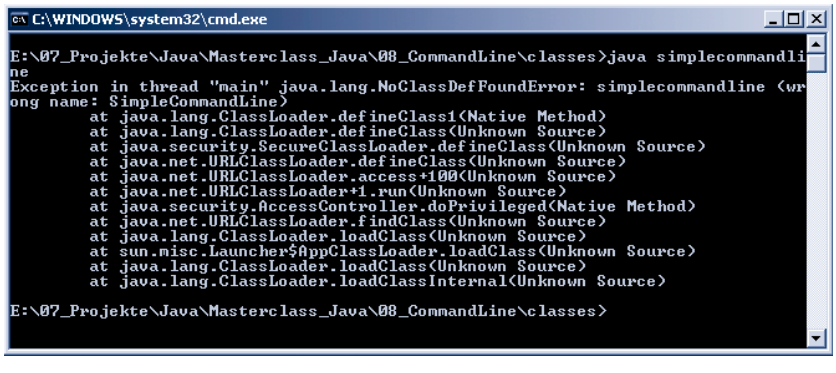
Wenn Sie die Klasse kompilieren, können Sie sie von der Kommandozeile aus ausführen.

Öffnen Sie dazu eine Kommandozeile auf Ihrem System, wechseln Sie in das Verzeichnis, in dem sich die kompilierte Klasse befindet, und rufen Sie sie auf:

```
java simplecommandline
```

Wenn Sie nun eine sinnvolle Ausgabe bekommen, haben Sie definitiv getrickst! Sie werden wahrscheinlich eine Fehlermeldung erhalten, die dieser an Schönheit in nichts nachsteht:

Abbildung 8.1
Fehlermeldung, falls eine Klasse nicht gefunden werden konnte



Inhaltlich besagt diese Fehlermeldung, dass die Klasse `simplecommandline` nicht gefunden werden konnte. Kein Wunder, denn die JVM unterscheidet sehr genau zwischen Groß- und Kleinschreibung. Und unsere Kommandozeilen-Applikation heißt nicht `simplecommandline`, sondern `SimpleCommandLine` – worauf wir in der Fehlermeldung übrigens auch aufmerksam gemacht worden sind.

Rufen wir unsere Klasse nun also richtig auf:

```
java SimpleCommandLine
```

Diesmal erhalten wir keine Fehlermeldung:

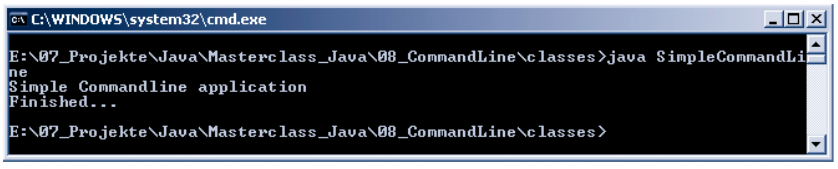


Abbildung 8.2
Unsere Kommandozeilen-Applikation konnte geladen und ausgeführt werden

Wenn wir nun auch noch Parameter übergeben wollten, könnten wir das recht leicht machen: Wir hängen sie einfach durch Leerzeichen getrennt an und rufen die Klasse beispielsweise so auf:

```
java SimpleCommandLine 1 2 3 4 5 6 7 8 9 0 "11 + 12"
```

Bevor Sie *Enter* drücken: Erinnern Sie sich noch kurz, wie die Klasse funktioniert hatte: Wenn wir Parameter übergeben, dann werden diese freundlicherweise wieder ausgegeben:

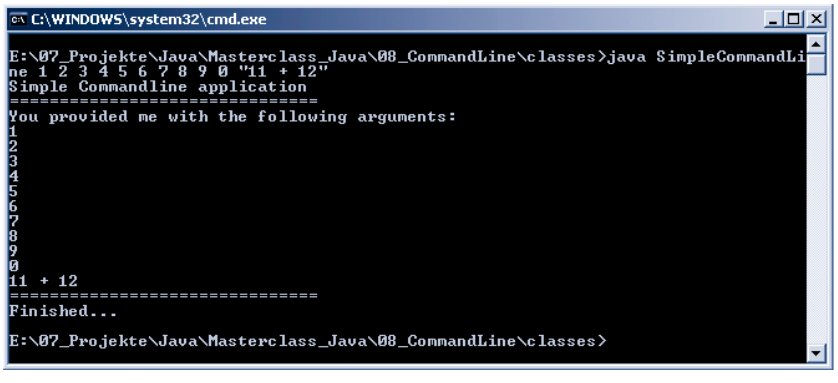


Abbildung 8.3
Über- und Ausgabe von Kommandozeilen-Parametern

Zeichenketten, die Leerzeichen enthalten und als ein Parameter aufgefasst werden sollen, müssen in Anführungsstriche gesetzt werden. Java interpretiert dann den Bereich zwischen den Anführungsstrichen als einen zusammengehörigen Parameter. Wollen Sie innerhalb der Anführungsstriche weitere Anführungsstriche übergeben, so müssen Sie diese escapen – aus " wird dann \"

```
java SimpleCommandLine "Hi, I am in \"good\" mood"
```

Das sieht doch gut aus, oder? Beachten Sie aber die letzte Parameter-Ausgabe – wieso wird *11 + 12* auf einer Zeile ausgegeben? Handelt es sich nicht eigentlich um drei Parameter – schließlich wurden sie durch Leerzeichen ordnungemäß voneinander getrennt?

Nein, die Ausgabe ist korrekt: *11 + 12* wird tatsächlich und völlig zu Recht als ein Parameter behandelt – wenn Sie sich noch einmal kurz ansehen wollen, wie wir die Klasse mit den Parametern aufgerufen haben, dann werden Sie feststellen, dass wir *11 + 12* in Anführungsstriche gesetzt hatten (*"11 + 12"*).

Java interpretiert dies als zusammenhängende Zeichenkette – und demnach als einen Parameter.

8.2 Parameter einlesen

Viele Applikationen lassen sich über Parameter steuern. Tatsächlich kann man empfehlen, in fast jede Applikation eine derartige Steuerung einzubauen – die Flexibilität steigt und damit auch der potentielle Nutzen für einen Anwender.

Um Parameter zu verwenden und auszuwerten, bieten sich zwei Ansätze an:

- Voraussetzen einer festen Parameter-Reihenfolge
- Zulassen einer dynamischen Parameter-Reihenfolge mit benannten Parametern

Wir werden hier im Folgenden auf beide Ansätze kurz eingehen und erläutern, welcher Ansatz sich für welchen Zweck am besten eignet.

8.2.1 Feste Parameter-Reihenfolge

Eine feste Parameter-Reihenfolge ist am einfachsten umzusetzen, denn sie ist am wenigsten flexibel. Bei einer festen Parameter-Reihenfolge gehen wir davon aus, dass – falls Parameter übergeben worden sind – diese eine bestimmte Bedeutung haben. Sollten keine Parameter angegeben worden sein, können wir entweder Standard-Werte verwenden oder einen entsprechenden Hinweis ausgeben. Wir werden hier beide Varianten kurz besprechen.

Feste Parameter-Reihenfolge mit Standard-Werten

Wenn wir keinerlei Aufwand betreiben und einfach nur die Möglichkeit zur Verfügung stellen wollen, Parameter optional zu übergeben, dann bietet sich die Verwendung von Standard-Werten an.

Dieser Ansatz impliziert folgende Schritte, die wir im Code ausführen müssen:

- Definieren der Standard-Werte
- Prüfen, ob Werte übergeben worden sind
- Prüfen der übergebenen Werte auf Datentyp und Inhalt

In einer Klasse *FixedParametersDefaults* umgesetzt, ergäbe sich folgender Code:

Listing 8.2
Diese Klasse akzeptiert optional zwei Parameter

```
import java.util.regex.Pattern;

/**
 * This class accepts two parameters - if they are missing,
 * it will use its default parameters
 */
public class FixedParametersDefaults {

   /**
    * Calculates the sum of two numbers
    */
   public static void main(String[] args) {
      String regexPattern = "^\\d+?$";

      long numberOne = 10;
      long numberTwo = 20;

      if(null != args && args.length == 2) {
         boolean useParams =
            Pattern.matches(regexPattern, args[0]) &&
            Pattern.matches(regexPattern, args[1]);

         if(useParams) {
            numberOne = Long.parseLong(args[0]);
            numberTwo = Long.parseLong(args[1]);

            System.out.println(
               "Successfully retrieved two numbers!");
         }
      }

      System.out.println(
         String.format("Calculating the sum of %d and %d: %d",
            numberOne, numberTwo, numberOne + numberTwo));
   }
}
```

Im Kopf der Klasse deklarieren wir, dass sie die Klasse `java.util.regex.Pattern` importieren soll – reguläre Ausdrücke sind vielen Entwicklern ein Graus und dabei dennoch der Preis, den wir zahlen müssen, wenn wir flexibel mit Kommandozeilen-Parametern arbeiten und dabei nach Möglichkeit keine Ausnahmen riskieren wollen.

Innerhalb unserer statischen Methode `main()` definieren wir als Erstes das so genannte *Muster* für die Prüfung mit Hilfe der `Pattern`-Klasse. Das Muster

^\\d+?$

besagt, dass die zu überprüfende Zeichenkette nur aus Ziffern bestehen darf, wobei mindestens eine Ziffer vorhanden sein muss.

Reguläre Ausdrücke erlauben es, Werte anhand von Mustern, die die Struktur der enthaltenen Daten beschreiben, zu überprüfen.

Ein griffiges Beispiel wäre die Validierung einer E-Mail-Adresse. Wenn Sie dies per normalem Code erledigen wollten, müssten Sie eine Unmenge an Bedingungen definieren und überprüfen:

- Existiert ein @-Symbol in der zu überprüfenden Zeichenkette?
- Existieren Zeichen vor dem @-Symbol und handelt es sich dabei um erlaubte Zeichen?
- Existieren Zeichen nach dem @-Symbol und handelt es sich dabei um Buchstaben und / oder Zahlen?
- Existiert ein Punkt, gefolgt von zwei bis vier Zeichen (Top-Level-Domain, etwa .de oder .com)?
- Existieren mindestens zwei Zeichen vor der Top-Level-Domain?

Die Anzahl der möglichen und sinnvollen Prüfungen bei E-Mail-Adressen geht schier ins Unendliche – und jede dieser Prüfungen müsste in Java-Code umgesetzt werden. Viel Spaß beim Programmieren!

Alternativ können Sie aber auch einen regulären Ausdruck verwenden:

```
^([0-9a-zA-Z]([-\\.\\w]*[0-9a-zA-Z])*@([0-9a-zA-Z][-\\w]*[0-9a-zA-Z]\\.)+
  [a-zA-Z]{2,9})$
```

Sie werden jetzt möglicherweise völlig zu Recht anmerken, dass dieser Ausdruck mehr als unverständlich ist. An dieser Stelle fehlt auch die Möglichkeit, detailliert auf das Muster einzugehen.

So viel sei aber dann doch erläutert: Das Muster definiert, welche Buchstaben und Zahlen an welcher Stelle der übergebenen Zeichenkette erlaubt sind:

- Am Anfang sind Zahlen und Buchstaben zugelassen
- Bei Bindestrich oder einem Punkt muss diesem mindestens ein Buchstabe folgen
- Vor dem @-Symbol muss ein Buchstabe stehen
- Nach dem @-Symbol dürfen Buchstaben und Zahlen, Bindestriche oder Punkte folgen
- Am Ende muss eine Top-Level-Domain definiert sein

Alle diese Angaben sind in obigem Muster enthalten. Logisch, dass es da ein wenig unübersichtlich wirkt.

Um Ihnen einen Einstieg in reguläre Ausdrücke zu geben, sei Ihnen das Tutorial unter *http://www.regular-expressions.info/tutorial.html* ans Herz gelegt.

Viele oft gebrauchte reguläre Ausdrücke können Sie in der *RegexLib* unter *http://www.regexlib.com* nachschlagen. Für den Einsatz dieser Muster müssen Sie übrigens nicht zwingend reguläre Ausdrücke beherrschen – ein einfaches Copy & Paste reicht in der Regel völlig aus.

Die beiden Long-Variablen numberOne und numberTwo definieren unsere Standardwerte – numberOne bekommt den Wert 10 zugewiesen und numberTwo erhält den Wert 20:

```
long numberOne = 10;
long numberTwo = 20;
```

Nun kann überprüft werden, ob das String-Array args ungleich null ist und genau zwei Elemente enthält:

```
if(null != args && args.length == 2) { ... }
```

Nur in diesem Fall werden wir mit Hilfe der Methode matches() der java.util.regex.Pattern-Klasse prüfen, ob beide Werte dem in unserem regulären Ausdruck definierten Muster entsprechen. Das Ergebnis dieser Prüfung wird der booleschen Variablen useParams zugewiesen. Diese kann nur den Wert true haben, wenn sowohl der erste als auch der zweite übergebene Parameter ausschließlich aus Ziffern bestehen und beide mindestens ein Zeichen lang sind:

```
boolean useParams =
   Pattern.matches(regexPattern, args[0]) &&
   Pattern.matches(regexPattern, args[1]);
```

Wenn beide Parameter unseren Anforderungen genügen und somit die Variable useParams den Wert *true* hat, erfolgt mit Hilfe von Long.parseLong() die Umwandlung der beiden Zeichenketten in Zahlen:

```
numberOne = Long.parseLong(args[0]);
numberTwo = Long.parseLong(args[1]);
```

Zuletzt errechnen wir noch die Summe aus numberOne und numberTwo und geben diesen Wert aus. Dabei spielt es für uns keine Rolle, ob Parameter übergeben worden sind oder nicht. Ebenso interessiert es uns hier nicht mehr, ob die Parameter gültig waren – schließlich prüfen wir deren Inhalt ab und haben Standard-Werte definiert:

```
System.out.println(
   String.format("Calculating the sum of %d and %d: %d",
      numberOne, numberTwo, numberOne + numberTwo));
```

Wenn Sie die Klasse kompilieren, dann können Sie zunächst die Standard-Werte zur Berechnung verwenden. Öffnen Sie dazu eine Kommandozeile und geben Sie

```
java FixedParametersDefaults
```

ein. Anschließend sollten Sie eine Ausgabe wie diese zu Gesicht bekommen:

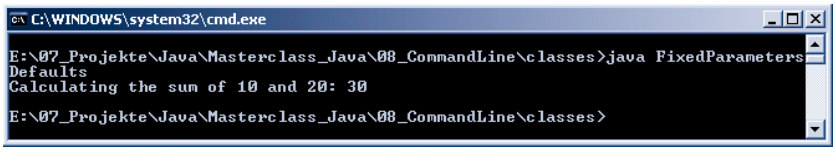


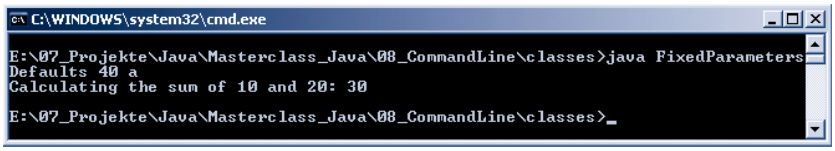
Abbildung 8.4
Die Klasse FixedParameters-Defaults rechnet mit Standardwerten

Übergeben Sie nun mindestens eine Nicht-Ziffer als Parameter:

```
java FixedParametersDefaults 40 a
```

Bei Ausführung werden Sie feststellen, dass auch hier die Standard-Werte zur Anwendung kamen:

Abbildung 8.5
Bei ungültigen Parametern werden die Standardwerte verwendet

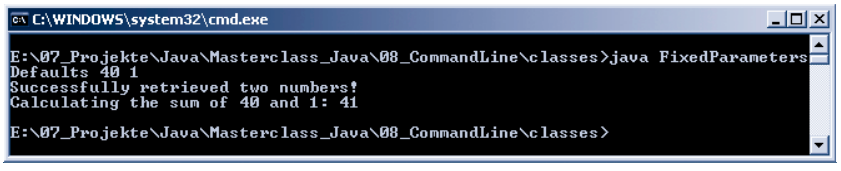


Zuletzt übergeben wir zwei Ziffern als Parameter:

```
java FixedParametersDefaults 40 1
```

Wie wir sehen, verwendet die Klasse nunmehr die übergebenen Parameter – und bestätigt uns dies sogar:

Abbildung 8.6
Wenn zwei Zahlen übergeben werden, rechnet die Klasse auch mit diesen



Ausgabe eines Hinweises bei nicht vorhandenen oder falschen Parametern

Wie in unserer Klasse demonstriert, bietet sich der Einsatz von regulären Ausdrücken zur Validierung von Daten an vielen Stellen an und ist – mit Ausnahme der Muster – nicht allzu komplex. Daneben gibt es noch weitere Vorteile, die Ihnen an dieser Stelle nicht verschwiegen werden sollen: Reguläre Ausdrücke sind meist performanter als vergleichbarer Java-Überprüfungs-Code. Und reguläre Ausdrücke sind in jedem Fall performanter als das blinde Casting von Werten bei gleichzeitigem Abfangen von möglichen *NumberFormatExceptions*.

Die zweite Variante beim Einsatz von statischen Parameter-Listen stellt die Ausgabe von Hinweistexten bei nicht vorhandenen oder syntaktisch falschen Parametern dar. Dies ist dann sinnvoll, wenn die Parameter nicht optional sind, sondern von der Applikation vorausgesetzt werden.

Wenn wir unsere Klasse `FixedParametersDefaults` ein wenig abändern, können wir dieses Verhalten implementieren:

Listing 8.3
Die Klasse FixedParamsRequired setzt zwei Parameter voraus

```
import java.util.regex.Pattern;

/**
 * This class requires two parameters -
 * if they are missing or have the wrong syntax,
 * it will display an error message
 */
public class FixedParametersRequired {

   /**
    * Calculates the sum of two numbers
    */
   public static void main(String[] args) {
      String regexPattern = "^\\d+?$";
      boolean useParams = false;

      long numberOne = 0;
      long numberTwo = 0;
```

```
      if(null != args && args.length == 2) {
         useParams = Pattern.matches(regexPattern, args[0]) &&
            Pattern.matches(regexPattern, args[1]);
         if(useParams) {
            numberOne = Long.parseLong(args[0]);
            numberTwo = Long.parseLong(args[1]);
         }
      }

      if(useParams) {
         System.out.println(
            String.format("Calculating the sum of %d and %d: %d",
               numberOne, numberTwo, numberOne + numberTwo));
      } else {
         System.out.println("Wrong number or syntax of arguments!");
         System.out.println(
            "Usage: FixedParametersRequired <Long> <Long>");
      }
   }
}
```

Listing 8.3 *(Forts.)*
Die Klasse FixedParamsRequired setzt zwei Parameter voraus

Die Unterschiede der Klasse `FixedParamsRequired` zur weiter oben besprochenen Klasse `FixedParamsDefaults` sind vergleichsweise gering.

Besagte *NumberFormatExceptions* treten dann auf, wenn Sie versuchen, Zeichenketten in Zahlen zu casten und die Zeichenketten entweder keine Zahlen oder keine Zahlen des erforderlichen Formats darstellen – etwa bei einer Zeichenkette, die eine Zahl mit Nachkomma-Stellen repräsentiert, wo *Integer*-Werte erwartet werden.

Die Empfehlung des Autors lautet also noch einmal und besonders nachdrücklich: Setzen Sie reguläre Ausdrücke ein, wo es geht.

Ebenso wie im letzten Beispiel wird im Kopf der Klasse per `import`-Statement die Klasse `java.util.regex.Pattern` eingebunden, die eine Prüfung eines Wertes per regulärem Ausdruck erlaubt:

```
import java.util.regex.Pattern;
```

Den angesprochenen regulären Ausdruck definieren wir als Erstes innerhalb der statischen Methode main():

```
String regexPattern = "^\\d+?$";
```

Anschließend erfolgt die Deklaration der booleschen Variablen `useParams`, die uns Auskunft darüber geben wird, ob die übergebenen Parameter verwendet werden dürfen. Die beiden Variablen `numberOne` und `numberTwo` werden ebenfalls deklariert und mit dem Standardwert *0* vorbelegt – eine andere Zahl ist hier nicht nötig, da der eigentliche Wert, mit dem später gerechnet werden soll, als Parameter übergeben wird.

Wurden genau zwei Werte als Parameter übergeben, dann wird die folgende Untersuchung des String-Arrays `args` `true` ergeben – und die beiden Parameter können mit Hilfe der `Pattern`-Klasse und deren statischer Methode `matches()` strukturell überprüft werden:

```
if(null != args && args.length == 2) {
   useParams = Pattern.matches(regexPattern, args[0]) &&
      Pattern.matches(regexPattern, args[1]);
   ...
}
```

Da das Ergebnis dieser Prüfung ein boolescher Wert ist, kann es der Variablen `useParams` übergeben werden. Sollte `useParams` den Wert *true* haben, dann wer-

den die übergebenen Parameter in Long-Werte konvertiert und den beiden Variablen `numberOne` und `numberTwo` zugewiesen:

```
numberOne = Long.parseLong(args[0]);
numberTwo = Long.parseLong(args[1]);
```

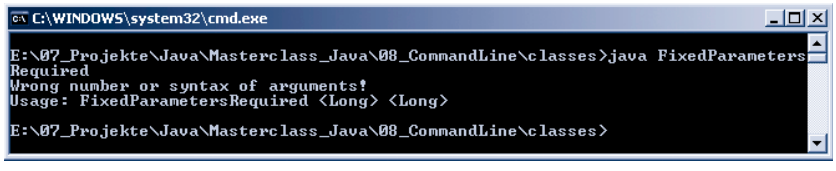
Jetzt können wir anhand von `useParams` entscheiden, ob zwei Parameter übergeben worden sind und diese auch noch syntaktisch korrekte Werte enthielten. Falls ja, dann würde `useParams` den Wert *true* haben und wir könnten sowohl die Werte von `numberOne` und `numberTwo` als auch deren Summe ausgeben. Falls nicht, dann würden wir den Nutzer darüber informieren, dass er entweder keine Parameter übergeben hat oder diese einfach nicht gültig waren.

Um uns von der einwandfreien Funktion der Klasse zu überzeugen, sollten wir sie kompilieren und ausführen. Lassen Sie uns zunächst testen, wie die Reaktion aussieht, wenn wir keine Parameter übergeben:

```
java FixedParamsRequired
```

Die Ausgabe erfüllt unsere Erwartungen: Die Klasse beschwert sich darüber, dass keine oder falsche Parameter übergeben worden sind:

Abbildung 8.7
Die Klasse FixedParamsRequired erwartet die Angabe von zwei Parametern

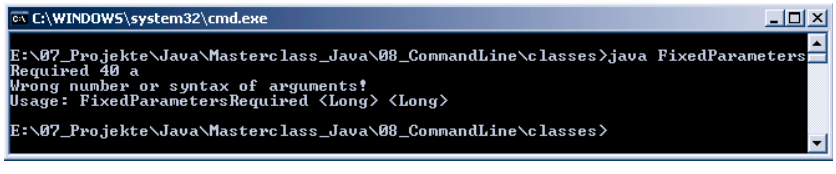


Geben wir nun eine Zahl und einen Buchstaben als Parameter an:

```
java FixedParamsRequired 40 a
```

Wir werden erneut darüber aufgeklärt, dass unsere Parameter nicht korrekt seien:

Abbildung 8.8
Buchstaben werden nicht als gültige Parameter akzeptiert

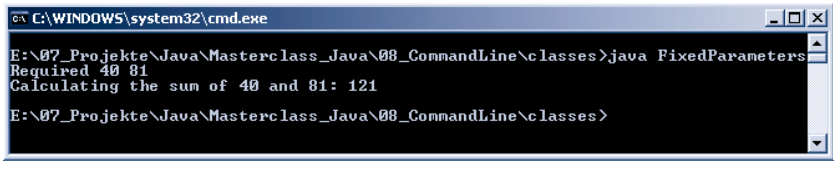


Nun übergeben wir zwei Ziffern als Parameter:

```
java FixedParamsRequired 40 81
```

Diesmal rechnet die Klasse brav das Ergebnis der Addition beider Zahlen aus:

Abbildung 8.9
Wenn zwei Zahlen übergeben werden, dann erfolgt eine Addition



8.2.2 Benannte Parameter

In der Praxis werden wir häufiger auf benannte Parameter treffen, denn diese erlauben es uns, die Reihenfolge, in der die Parameter angegeben werden müssen, dynamisch zu halten. Benannte Parameter folgen in der Regel einer bestimmten Namenskonvention, die der Entwickler oder die Firmenrichtlinien festlegen.

Prinzip

Das Verarbeitungsprinzip bei der Verwendung benannter Parameter besteht darin, dass sämtliche Parameter innerhalb der Klasse durchlaufen und analysiert werden. Die Trennung von Parameter-Namen und -Wert erfolgt in der Regel durch einen Doppelpunkt (:). Normalerweise würde der Name eines Parameters aus einem bis zwei Zeichen bestehen – eventuell einem Bindestrich und einem Buchstaben. Ein Parameter hätte nach diesem Modell also folgenden Aufbau:

```
-<Name>:<Wert>[ ]
```

Das Leerzeichen am Ende wäre nur nötig, wenn wir mehrere Parameter übergeben wollten, da diese durch Leerzeichen getrennt werden.

Enthält der Wert des Parameters selber Leerzeichen, so muss er in Anführungsstriche gesetzt werden:

```
-<Name>:"<Wert>"[ ]
```

Anführungszeichen, die im Wert enthalten sind, werden *escaped* – Ihnen wird ein Backslash (\) vorangestellt. Wollten Sie also die Zeichenkette

Hello, "Mike" is just a codeword!

als Parameter übergeben, sollte dies so aussehen:

```
-n:"Hello, \"Mike\" is just a codeword!"
```

Um das Entfernen der Anführungsstriche und der zusätzlichen Backslashes müssen Sie sich nicht kümmern. Dies erledigt Java von allein. In Ihrem Programm kommt als Wert nur die ursprüngliche Zeichenkette an.

Praxis

Wir wollen im Folgenden eine Lösung umsetzen, die es uns erlaubt, per Kommandozeile drei Parameter anzugeben. Erwartet werden ein Name, ein Vorname und eine Stadt-Kennung – intern werden diese Informationen dann Instanzvariablen zugewiesen und anschließend wieder ausgegeben.

Sehen wir uns an, wie diese Lösung aussehen könnte:

Listing 8.4
Die Klasse NamedParameters verwendet benannte Parameter, um Werte zu erfassen

```
/**
 * Takes a name, a first name and a city and displays the data
 */
public class NamedParameters {

   private String firstName = null;
   private String lastName = null;
   private String city = null;
   private boolean showHelp = false;

   /**
    * Constructor of this class
    * @param args Parameters from the commandline
    */
   NamedParameters(String[] args) {
      for(String arg : args) {
         this.setValue(arg);
      }
   }

   /**
    * Maps a value to a field
    * @param arg Value to be mapped
    */
   private void setValue(String arg) {
      if (null != arg && arg.length() > 3) {
         if(arg.startsWith("-f:")) {
            this.firstName = arg.substring(3);
         } else if(arg.startsWith("-n:")) {
            this.lastName = arg.substring(3);
         } else if(arg.startsWith("-c:")) {
            this.city = arg.substring(3);
         }
      } else if(null != arg && arg.startsWith("-h")) {
         this.showHelp = true;
      }
   }

   /**
    * Indicates, whether the help should be displayed or not
    */
   private boolean displayHelp() {
      boolean result = this.showHelp;
      if(!result) {
         result = !(null == this.lastName
            || null == this.firstName
            || null == this.city);
      }

      return result;
   }
```

Listing 8.4 *(Forts.)*
Die Klasse NamedParameters verwendet benannte Parameter, um Werte zu erfassen

```
    /**
     * Generates the String representation of this class -
     * which is either a text containing
     * the given values or an error message
     */
    public String display() {
        StringBuilder sb = new StringBuilder();

        if(!displayHelp()) {
            sb.append(
                String.format("Hello %s %s from %s!",
                    firstName, lastName, city)
            );
        } else {
            sb.append("Usage: java NamedParameters "
                + "-c:<Value> -n:<Value> -f:<Value> | -h\n\n");
            sb.append("Parameters:\n");
            sb.append("-h             Displays this help\n");
            sb.append("-c:<Value>     Sets the city\n");
            sb.append("-n:<Value>     Sets the last name\n");
            sb.append("-f:<Value>     Sets the first name\n\n");
            sb.append("Replace <Value> with actual data!");
        }

        return sb.toString();
    }

    /**
     * Static entrypoint when called from console
     * @param args Parameters to be passed to the actual instance
     */
    public static void main(String[] args) {
        NamedParameters instance = new NamedParameters(args);
        System.out.println(instance.display());
    }
}
```

Lassen Sie sich nicht von der schieren Größe des Listings irritieren – die Funktionsweise der Klasse ist recht einfach zu verstehen und zu erklären.

main()

Fangen wir einfach mit der statischen Methode `main()` an, die aufgerufen wird, wenn die Klasse von der Kommandozeile aus gestartet wurde. Hier erzeugen wir eine neue Instanz der Klasse und übergeben deren Konstruktor das Parameter-Array:

```
NamedParameters instance = new NamedParameters(args);
```

Danach geben wir die Rückgabe der Methode `display()` der `NamedParameters`-Instanz aus.

```
System.out.println(instance.display());
```

Sehen wir uns nun an, welche Verarbeitungsschritte innerhalb der Klassen-Instanz stattfinden.

Konstruktor

Werfen wir zunächst einen Blick auf den Konstruktor. Dieser nimmt die Parameter, die auf der Kommandozeile übergeben worden sind, entgegen und durchläuft dieses `String`-Array mit Hilfe einer `for-each`-Schleife:

```
for(String arg : args) {
   this.setValue(arg);
}
```

Bei jedem Durchlauf wird versucht, die übergebenen Parameter auf die internen Felder der Klasse zu mappen. Dieser Vorgang findet innerhalb der Methode `setValue()` statt, die den aktuellen Parameter als Argument beim Aufruf übergeben bekommt.

setValue()

Speziell in der Methode `setValue()` gilt der Grundsatz: *Security by obscurity*! Dies ist nötig, weil sie die einzige Möglichkeit der Klasse darstellt, Daten von außen zu erhalten – werden hier inkonsistente oder ungültige Daten übergeben, dann kann für das einwandfreie Funktionieren der Anwendung nicht mehr garantiert werden.

Deshalb wird jeder Parameter darauf hin überprüft, ob er einen Wert ungleich `null` und eine Länge von mehr als drei Zeichen (drei Zeichen kennzeichnen in dieser Applikation einen Parameter-Namen) besitzt. Sollten diese beiden Bedingungen zutreffen, dann wird versucht, die ersten drei Zeichen des Parameters als Parameter-Namen zu interpretieren.

Folgende Parameter werden von der Methode `setValue()` unterschieden:

- `-h`
  Hilfetext anzeigen
- `-n:`
  Namen angeben
- `-f:`
  Vornamen angeben
- `-c:`
  Stadt angeben

Sollte einer der drei letzten Parameter-Namen identifiziert werden, was mit Hilfe der Methode `startsWith()` der `String`-Instanz überprüft wird, dann wird der Inhalt ab der vierten Stelle als Wert interpretiert und der entsprechenden Klassen-Variablen zugewiesen, was wir mit Hilfe von `arg.substring(3)` erledigen:

```
if(arg.startsWith("-f:")) {
   this.firstName = arg.substring(3);
} else if(arg.startsWith("-n:")) {
   this.lastName = arg.substring(3);
} else if(arg.startsWith("-c:")) {
   this.city = arg.substring(3);
}
```

Wenn der Name des Parameters nicht identifiziert werden konnte, dann wird sein Wert ignoriert.

Sollte übrigens ein String mit einer Länge von weniger als drei Zeichen übergeben worden sein, erfolgt eine Prüfung darauf, ob dieser String mit der Zeichenfolge `-h` beginnt – dies würde darauf hinweisen, dass der Anwender den Hilfe-Text sehen möchte.

display()

Die Methode `display()` hat den Zweck, die Ausgabe zu generieren. Dabei können genau zwei Fälle eintreten:

- Alle erforderlichen Parameter konnten übergeben werden und deshalb wird die Zusammenfassung der Daten angezeigt.
- Die Hilfeseite wird angezeigt, weil dies entweder vom Nutzer mit Hilfe des Parameters `-h` angefordert worden ist oder nicht alle Werte zugewiesen werden konnten.

Unterschieden werden diese Fälle, indem die Rückgabe der Methode `displayHelp()` ausgewertet wird. Diese Rückgabe ist ein boolescher Wert – sie wird `true`, wenn die Hilfeseite angezeigt werden soll, und `false`, wenn dies nicht der Fall ist.

Anhand dieser Informationen kann nun die Rückgabe generiert und einer `StringBuilder`-Instanz zugewiesen werden.

Das eigentliche Zuweisen der Daten erfolgt dabei unter Verwendung der Methode `append()` des `StringBuilders`, wobei die Zeilenumbrüche mit Hilfe des Tokens `\n` angegeben werden:

```
sb.append("Parameters:\n");
```

Die Umwandlung des Inhalts des `StringBuilders` in einen `String` erfolgt mit Hilfe von dessen Methode `toString()`. Das Ergebnis wird anschließend an die aufrufende Methode zurückgegeben:

```
return sb.toString();
```

Der `StringBuilder` ist seit Java 1.5 verfügbar und stellt so etwas wie den kleinen Bruder der `StringBuffer`-Klasse dar. Beide verfügen über identische Methoden-Signaturen – der Unterschied zwischen ihnen liegt darin, dass der `StringBuilder` nicht thread-sicher ist. Dies muss uns nur insofern interessieren, als dadurch ein wenig Overhead für den Java-Interpreter wegfällt und der `StringBuilder` deshalb in der Regel etwas schneller als der `StringBuffer` ist.

displayHelp()

Die bereits angesprochene Methode `displayHelp()` stellt fest, ob die Hilfe-Ausgabe zurückgegeben oder die der Klasse übergebenen Daten angezeigt werden sollten. Sie zeigt dies mit Hilfe eines booleschen Wertes deutlich an. Dieser boolesche Wert wird dann `true`, wenn

- entweder der Parameter `-h` übergeben wurde und deshalb die private Variable `showHelp` den Wert `true` hat oder
- eine der drei Klassenvariablen `firstName`, `lastName` und `city` `null` ist und ihr also kein Wert zugewiesen wurde.

Sollte keine der beiden Bedingungen zutreffen, kann davon ausgegangen werden, dass die übergebenen Werte korrekt waren und den entsprechenden Variablen zugewiesen werden konnten.

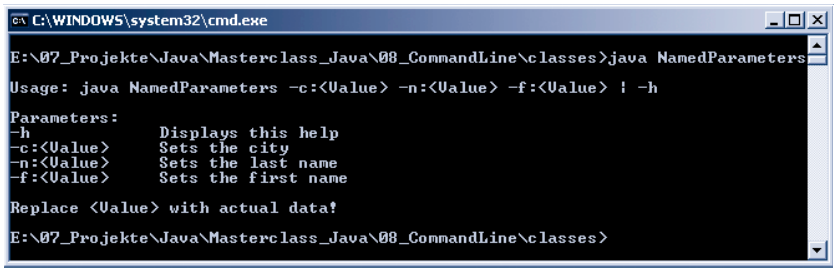
Keine Parameter übergeben, Parameter fehlt oder Hilfe explizit anfordern

Kompilieren Sie nun die Klasse und rufen Sie sie ohne Angabe irgendwelcher Parameter auf:

```
java NamedParameters
```

Die Ausgabe wird Sie über die möglichen Parameter und deren Bedeutung aufklären:

Abbildung 8.10
Ausgabe der Hilfe bei nicht angegebenen Parametern



Die gleiche Ausgabe erhalten Sie, wenn Sie an irgendeiner Stelle den Parameter *-h* angeben.

Wenn Sie die Klasse aufrufen und dabei einen der Parameter vergessen, dann werden Sie ebenfalls diesen Hilfe-Bildschirm zu Gesicht bekommen.

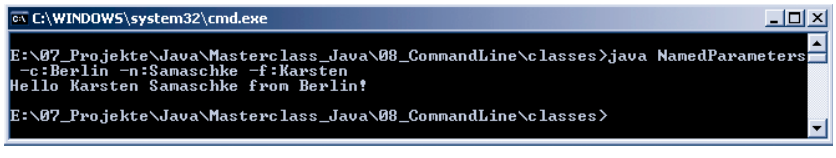
Angabe aller Parameter

Geben wir nun alle erforderlichen Parameter der Klasse an. Lassen Sie uns zu diesem Zweck die Klasse so aufrufen:

```
java NamedParameters -c:Berlin -n:Samaschke -f:Karsten
```

Die Ausgabe wird die übergebenen Parameter in Form einer kleinen Begrüßung präsentieren:

Abbildung 8.11
Ausgabe der übergebenen Parameter



Ändern Sie nun die Reihenfolge der Parameter ab:

```
java NamedParameters -n:Samaschke -c:Berlin -f:Karsten
```

Die Ausgabe ist identisch zum vorherigen Aufruf:

Hello Karsten Samaschke from Berlin!

8.2.3 Welcher Parameter-Typ für welchen Zweck?

Wir haben nun zwei Typen von Parametern kennen gelernt: Parameter mit fester Reihenfolge und benannte Parameter.

Parameter mit fester Reihenfolge ihrer Elemente erlauben es, sehr schnell und ohne größeren Aufwand Parameter zu verwenden. Sie gestatten es, eine festgelegte Abfolge von Parametern einzulesen. Der größte Nachteil dieser Art von Parametern ist ihre mangelhafte Flexibilität: Die Reihenfolge, in der die Elemente erwartet werden, ist festgeschrieben – jede Abweichung kann zu unvorhergesehenen Ergebnissen führen.

Benannte Parameter erlauben es, viel flexibler mit Argumenten zu arbeiten. Die einzelnen Elemente können eindeutig benannt und entsprechend zugeordnet werden. Allerdings erhöht sich der Aufwand für das Einlesen und Verarbeiten der Parameter, da nunmehr zunächst der Name des Arguments ermittelt und anschließend dessen Wert werden muss. Dies sind Vorgänge, die ein deutliches Mehr an Programmier-Aufwand voraussetzen – insbesondere dann, wenn noch die Datentypen überprüft werden sollen.

Dennoch lautet die klare Aussage: Verwenden Sie ab mehr als zwei Argumenten, die Ihre Klasse entgegennehmen und verarbeiten soll, benannte Parameter.

8.2.4 Das CLI-Paket der Apache Foundation

Die *Apache Foundation* bietet im Rahmen ihrer *commons*-Utilities eine Lösung namens *CLI* an, die den Umgang mit Kommandozeilen-Parametern stark vereinfacht. Sie können sich dieses Paket unter der Adresse *http://www.apache.de/dist/jakarta/commons/cli/binaries/cli-1.0.zip* kostenlos herunterladen.

Wenn Sie Parameter unterstützen, sollten Sie die Nutzer Ihrer Applikation nicht über deren Zweck im Unklaren lassen. Bei benannten Parametern bietet es sich an, einen Parameter *-h* oder */?* anzubieten, der eine genaue Hilfe ausgibt. Ebenfalls können Sie die Hilfe ausgeben, falls Parameter erforderlich, aber nicht zugewiesen worden sind oder wenn die Datentypen der übergebenen Parameter nicht korrekt sind.

Nach dem Download des Paketes sollten Sie das *zip*-Archiv öffnen und zumindest die Datei *commons-cli-1.0.jar* in ein Verzeichnis Ihrer Wahl entpacken. Zusätzlich können Sie selbstverständlich auch die API-Dokumentation und die Anmerkungen der Entwickler extrahieren und speichern.

Prinzip

Mit Hilfe des *commons-cli*-Pakets definieren Sie die möglichen Kommandozeilen-Parameter samt Hinweistexten direkt in Ihrem Code. Sie organisieren diese Parameter in einer eigenen Liste, die anhand dieser Informationen selbstständig eine Hilfe-Anzeige generieren und anzeigen kann. Sie können ebenfalls ohne großen Aufwand auf das Vorhandensein von Parametern prüfen und deren Werte bequem abrufen.

Parameter und deren Werte können beim Einsatz des *commons-cli*-Pakets durch Leerzeichen getrennt werden. Dies entspricht mehr der üblichen Praxis beim Einsatz von Kommandozeilen-Parametern. Die Syntax für den Einsatz von Parametern sieht so aus:

```
-<Name>[ <Wert>][ ]
```

Das abschließende Leerzeichen müssen Sie nur angeben, wenn weitere Parameter folgen sollen.

Das *commons-cli*-Paket unterstützt übrigens den Einsatz von Parameter-Schaltern – also Parameter, deren reines Vorkommen oder Nicht-Vorkommen schon Zustände signalisiert oder Aktionen auslöst.

Für uns als Entwickler ist der Einsatz des *commons-cli*-Pakets eine reine Wohltat – die Parameter (hier werden sie Optionen genannt) sind schnell deklariert und das Abrufen der Werte geschieht problemlos. Gleiches gilt für die Verwendung von Parameter-Schaltern und das Generieren der Hilfe-Übersicht.

Praxis

Der Umgang mit dem *commons-cli*-Paket ist nicht allzu schwierig. Lassen Sie uns zur Verdeutlichung eine Klasse entwickeln, die – ebenso wie das Beispiel zu benannten Parametern – drei Werte entgegennimmt und anschließend wieder ausgibt:

Listing 8.5
Verwenden des CLI-Pakets

```
import org.apache.commons.cli.*;

/**
 * Takes a name, a first name and a city
 * as parameters from the command line
 * and displays the data
 */
public class CLIParams {

   private String firstName = null;
   private String lastName = null;
   private String city = null;
   private boolean showHelp = false;

   /**
    * Constructor of this class
    * @param args Parameters from the commandline
    */
   CLIParams(String[] args) {
      Options options = new Options();

      options.addOption(
         new Option("f", true, "Sets the first name"));
      options.addOption(
         new Option("n", true, "Sets the last name"));
      options.addOption(
         new Option("c", true, "Sets the city"));

      options.addOption(
         new Option("h", false, "Displays the help page"));

      try {
         CommandLineParser cmdParser = new PosixParser();
         CommandLine cmd = cmdParser.parse(options, args);
```

Listing 8.5 *(Forts.)*
Verwenden des CLI-Pakets

```
            this.firstName = cmd.getOptionValue("f");
            this.lastName = cmd.getOptionValue("n");
            this.city = cmd.getOptionValue("c");

            this.showHelp = cmd.hasOption("h");

            if(displayHelp()) {
               System.out.println();
               HelpFormatter hf = new HelpFormatter();
               hf.printHelp("CLIParams", options);
            }
         } catch (ParseException e) {
            e.printStackTrace();
         }
      }


      /**
       * Indicates, whether the help should be displayed or not
       */
      public boolean displayHelp() {
         boolean result = this.showHelp;
         if(!result) {
            result = (
               null == this.lastName ||
               null == this.firstName ||
               null == this.city);
         }

         return result;
      }

      /**
       * Generates the String representation of this class
       */
      public String display() {
         return String.format("\nHello %s %s from %s!",
            firstName, lastName, city);
      }

      /**
       * Main method of this application
       * @param args Arguments to pass to the instance
       */
      public static void main(String[] args) {
         CLIParams instance = new CLIParams(args);
         if(!(instance.displayHelp())) {
            System.out.println(instance.display());
         }
      }
}
```

Bei der Ausführung der Klasse `CLIParams` wird zunächst die statische Methode `main()` mit den von der Kommandozeile angegebenen Parametern als Argument aufgerufen. Hier erzeugen wir eine neue Instanz der Klasse, die die weitere

Verarbeitung der Informationen übernehmen wird. Zu diesem Zweck reichen wir ihrem Konstruktor die uns übergebenen Parameter als Argument weiter – er wird sie dann schon auf geeignete Art und Weise verarbeiten:

```
CLIParams instance = new CLIParams(args);
```

Definition der Parameter und Einlesen ihrer Werte im Konstruktor

Innerhalb des Konstruktors erfolgen die Definition der erwarteten Kommandozeilen-Parameter und das Einlesen ihrer Werte. Dafür erzeugen wir eine Instanz der `Options`-Klasse aus dem `org.apache.commons.cli`-Paket, die alle erlaubten Kommandozeilen-Parameter transportieren wird:

```
Options options = new Options();
```

Der *Options*-Instanz weisen wir nun drei Kommandozeilen-Parameter zu. Jeder dieser Parameter wird durch eine Instanz der `Option`-Klasse aus dem *org.apache.commons.cli*-Paket repräsentiert – deren Konstruktor übergeben wir folgende Informationen:

- Name des Parameters (beispielsweise „n“) ohne vorangestellten Bindestrich
- boolesches Flag, das uns angibt, ob ein Wert erwartet wird (*true*) oder der Parameter selbst als Schalter fungiert (*false*)
- Erklärungstext zum Parameter, der ausgegeben wird, wenn eine Hilfe-Seite angezeigt werden soll

Für drei Parameter *f*, *n* und *c*, die die Informationen zu Vor- und Nachname sowie Stadt repräsentieren, sieht dies dann so aus:

```
options.addOption(
   new Option("f", true, "Sets the first name"));
options.addOption(
   new Option("n", true, "Sets the last name"));
options.addOption(
   new Option("c", true, "Sets the city"));
```

Nach diesen drei Optionen, für die Werte erwartet werden, definieren wir einen Schalter, der uns Auskunft darüber gibt, ob die Hilfe-Seite angezeigt werden soll. Dieser erhält den Namen *h* und erwartet keinen Wert – das reine Vorkommen des Parameters im Aufruf reicht uns bereits aus, um den Wunsch des Nutzers auf eine Übersicht der Parameter zur Nutzung der Klasse zu erkennen:

```
options.addOption(
   new Option("h", false, "Displays the help page"));
```

Nun können wir die Kommandozeilen-Informationen einlesen und verarbeiten lassen. Zu diesem Zweck erzeugen wir eine neue Instanz einer Klasse, die das Interface `CommandLineParser` aus dem `org.apache.commons.cli`-Paket implementiert. Für die Kommandozeile bietet sich hier die Klasse *PosixParser* an:

```
CommandLineParser cmdParser = new PosixParser();
```

Diese `CommandLineParser`-Instanz liest nun die Informationen von der Kommandozeile ein und stellt sie uns mit Hilfe einer `CommandLine`-Instanz zum Abruf bereit:

```
CommandLine cmd = cmdParser.parse(options, args);
```

Die Werte liegen nunmehr innerhalb der *CommandLine*-Instanz `cmd` vor, so dass wir sie abrufen können.

Für Parameter, die Werte enthalten sollen, rufen wir die Inhalte mit Hilfe der Methode `getOptionValue()` ab, der wir als Argument den Namen des Parameters übergeben. Für unsere drei Parameter *f*, *n* und *c* sieht das dann so aus:

```
this.firstName = cmd.getOptionValue("f");
this.lastName = cmd.getOptionValue("n");
this.city = cmd.getOptionValue("c");
```

Achtung

Sollte ein abzurufender Wert nicht per Kommando-Zeile übergeben worden sein, ist die Rückgabe von `getOptionValue()` für diesen Parameter *null*. Sie sollten unbedingt auf *null* prüfen, bevor Sie die Werte von Kommandozeilen-Parametern in Ihrer Applikation verwenden!

Da ein Parameter-Schalter keinen Wert besitzt, sondern allein durch seine An- oder Abwesenheit eine Information trägt, müssen wir eine Prüfung vornehmen, ob der Name des Schalters überhaupt angegeben worden ist. Dies erledigt für uns die Methode `hasOption()` der `CommandLine`-Instanz. Sie wird `true` zurückgeben, wenn der als Argument übergebene Parameter-Name tatsächlich von der Kommandozeile aus angegeben worden ist. Anderenfalls lautet die Rückgabe `false`.

Wenn wir also darauf prüfen wollen, ob der Schalter *h* angegeben worden ist, dann erledigen wir das mit diesem Code-Fragment:

```
this.showHelp = cmd.hasOption("h");
```

Nun kann mit Hilfe der privaten Methode `displayHelp()` darauf hin geprüft werden, ob die Hilfe-Informationen der Klasse angezeigt werden sollen. Dies wäre der Fall, wenn entweder einer der drei Werte-Parameter *c*, *f* oder *n* nicht angegeben worden wäre oder der Schalter *h* gesetzt worden ist.

Sollte die Hilfe angezeigt werden, dann erzeugen wir diese nicht etwa selbst, sondern verwenden eine Instanz der `HelpFormatter`-Klasse:

```
HelpFormatter hf = new HelpFormatter();
```

Deren Methode `printHelp()` gibt die Informationen zur Benutzung der Klasse und der möglichen Parameter in der Standard-Ausgabe aus:

```
hf.printHelp("CLIParams", options);
```

Die Ersetzung der Platzhalter erledigt die Methode `String.format()`, die seit Java 1.5 verfügbar ist. Ihr werden der String mit den Platzhaltern (*%s*) sowie die Variablen, mit deren Wert die Platzhalter ersetzt werden sollen, als Parameter übergeben.

Fortsetzung der Marginalie von S. 223

Die Ersetzung findet dann von links nach rechts statt – der erste Platzhalter wird durch den ersten folgenden Parameter ersetzt, der zweite Platzhalter dann durch die zweite Ersetzung. Bezogen auf die Ausgabe der drei Kommandozeilen-Parameter sieht dies dann so aus:

```
String.format("\nHello %s %s from %s!", firstName, lastName, city);
```

Der erste Platzhalter erhält den Wert von `firstName`, der zweite Platzhalter wird durch den Inhalt von `lastName` ersetzt und `city` stellt den Inhalt bereit, der an Stelle des dritten Platzhalters eingefügt wird.

Sollten die drei Variablen `firstName`, `lastName` und `city` die Werte *Karsten*, *Samaschke* und *Berlin* tragen, wird die Rückgabe von `String.format()` (unter Vernachlässigung des am Anfang der Zeichenkette befindlichen Zeilenumbruchs)

Hello Karsten Samaschke from Berlin!

lauten.

Der erste Parameter von `printHelp()` entspricht in aller Regel dem Klassen-Namen, der von der Kommandozeile aus aufgerufen werden soll. Das zweite Argument ist der weiter oben definierte *Options*-Container, der alle definierten Kommandozeilen-Parameter beinhaltet.

Hier ist die Abarbeitung des Konstruktors beendet. Die Kontrolle wird nun wieder der aufrufenden Methode übergeben – in unserem Fall handelt es sich um die statische Methode `main()`.

Dort wird nun geprüft, ob die Hilfeseite bereits angezeigt worden ist. Dies geschieht durch einen erneuten Aufruf der Methode `displayHelp()`, die uns *true* zurückgeben wird, falls der Schalter *h* für die Hilfeseite gesetzt worden ist oder einer der Werte-Parameter nicht vorhanden war und deshalb *null* ist:

```
result = (
   null == this.lastName ||
   null == this.firstName ||
   null == this.city);
```

Wenn alle Werte gesetzt worden sind und der Schalter *h* nicht vorhanden war, dann ist die Rückgabe der Methode `displayHelp()` *false*. In diesem Fall gibt die statische Methode `main()` die Ausgabe der Klasse, die durch deren Methode `display()` erzeugt und zurückgegeben worden ist, aus.

Die Methode `display()` erzeugt dabei einen String, bei dem drei Platzhalter durch die eingelesenen Kommandozeilen-Informationen ersetzt werden.

Ausführen der Klasse

Wenn wir die Klasse nun kompilieren und anschließend per

```
java CLIParams
```

aufrufen, dann werden wir eine wirklich unangenehme Überraschung erleben:

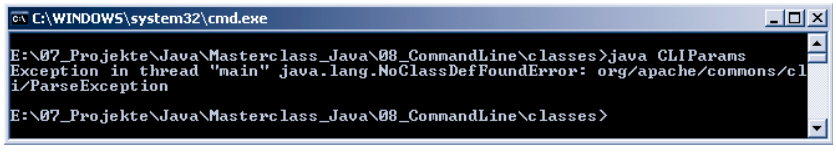


Abbildung 8.12
Die Klasse ParseException aus dem Paket org.apache.commons.cli konnte nicht gefunden werden

Was ist hier los? Warum funktioniert die Klasse nicht?

Denken Sie noch mal ein wenig zurück und sehen Sie sich den Code noch mal genau an. Direkt am Anfang der Klasse importieren wir das Paket `org.apache.commons.cli` mit allen seinen Klassen-Definitionen:

```
import org.apache.commons.cli.*;
```

Erinnern Sie sich daran, dass Sie dieses Paket zunächst von *http://www.apache.de/dist/jakarta/commons/cli/binaries/cli-1.0.zip* herunterladen mussten? Und anschließend haben Sie das Java-Archiv *commons-cli-1.0.jar* in ein Verzeichnis auf Ihrem Rechner entpackt?

Das erste Element der Pfad-Angabe sollte der Punkt (.) sein. Dieser steht für das aktuelle Verzeichnis, so dass die Klassen, die sich darin befinden, ebenfalls von Java gefunden werden können. Diese Art der Angabe ist dann sinnvoll und funktioniert hervorragend, wenn Sie die Klasse aus ihrem Speicher-Verzeichnis heraus aufrufen.

Das alles weiß der Java-Interpreter aber nicht. Die Information darüber, welches Paket wir zusätzlich verwenden wollen, müssen wir ihm explizit geben.

Wir verwenden zu diesem Zweck den Kommandozeilen-Parameter *classpath*, der die Pfade zu den zu verwendenden Paketen und Pfaden aufnehmen wird. Die einzelnen Pfade werden durch Semikola voneinander getrennt.

Der Autor hat das Archiv *commons-cli-1.0.jar* auf seinem Windows-System im Ordner

E:\12_Lib\02_Java\commons-cli-1.0

abgelegt. Die *classpath*-Angabe für sein System sieht dann so aus:

.;E:\12_Lib\02_Java\commons-cli.1.0\commons-cli-1.0.jar

Fügen wir den Klassenpfad zum Aufruf der Klasse hinzu, ergibt sich dieser Aufruf:

```
java -classpath .;E:\12_Lib\02_Java\commons-cli-1.0\
  commons-cli-1.0.jar CLIParams
```

Das sieht zwar deutlich unübersichtlicher aus, als es vorher der Fall war, dafür funktioniert aber nun unsere Klasse:

Sollten Sie sie aus einem anderen Ordner heraus aufrufen wollen, dann müssen Sie auch den kompletten Pfad zum Ordner angeben, in dem sich die Klasse befindet – die Angabe sollte dann für den Rechner des Autors so aussehen:

E:\07_Projekte\Java\Masterclass_Java\08_CommandLine\classes;E:\12_Lib\02_Java\commons-cli.1.0\commons-cli.1.0.jar

Ersetzen Sie die Pfad-Angaben durch die Angaben auf Ihrem System, damit der Klassenpfad gültig wird.

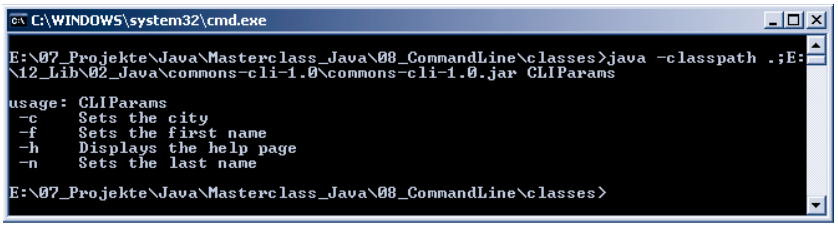


Abbildung 8.13
Bei Angabe des classpath-Parameters funktioniert die Klasse und zeigt ihre Hilfe an

Da wir die Klasse ohne Parameter aufgerufen haben, sind die Werte für `firstName`, `lastName` und `city` natürlich *null* – deshalb wird die Hilfe-Seite angezeigt.

Rufen wir die Klasse nun unter Angabe der drei Parameter *c* (für die Stadt), *f* (Vorname) und *n* (Nachname) auf:

```
java -classpath .;E:\12_Lib\02_Java\commons-cli-1.0\
  commons-cli-1.0.jar CLIParams -c Berlin -f Karsten -n Samaschke
```

Jetzt erhalten wir eine andere Ausgabe, die uns zeigt, dass die Parameter korrekt verarbeitet werden konnten:

Abbildung 8.14
Aufruf der Klasse unter Angabe von Parametern

Zuletzt können wir den Schalter *h* noch aktivieren:

```
java -classpath .;E:\12_Lib\02_Java\commons-cli-1.0\
  commons-cli-1.0.jar CLIParams -c Berlin -f Karsten -n Samaschke -h
```

Nun erfolgt wieder die Ausgabe der Hilfe-Seite – exakt so, wie wir es auch gewollt haben!

8.2.5 Commons-cli oder eigene benannte Parameter?

Die Entscheidung, ab welcher Parameter-Anzahl der Einsatz des *commons-cli*-Paketes sinnvoll wird, liegt allein bei Ihnen. Der Autor hat es sich angewöhnt, etwa ab fünf Parametern darauf zurückzugreifen, weil ab etwa dieser Parameter-Anzahl der Schreibaufwand für das Auslesen und Zuweisen benannter Parameter größer wird als die Definition von Parameter mit Hilfe der `Option`-Klasse.

Das *commons-cli*-Paket ist darüber hinaus noch weit mächtiger, als hier angedeutet – ein Blick auf die *Usage*-Seite sollte Ihnen einige interessante Ansätze aufzeigen können: *http://jakarta.apache.org/commons/cli/usage.html*.

8.3 Einlesen von Informationen

Neben der Verarbeitung von Parametern kann eine Kommandozeilen-Applikation auch auf anderem Wege an Informationen kommen: Sie kann sie direkt beim Nutzer erfragen. Dieser Prozess klingt mehr als trivial. Lassen Sie uns trotzdem kurz analysieren, wie wir Daten von der Kommandozeile einlesen können:

Grundsätzlich steht uns zu diesem Zweck die `InputStream`-Instanz unter `System.in` zur Verfügung. Diese kann aber leider nur Bytes oder einzelne Zeichen einlesen – für uns wäre das auf Dauer ziemlich anstrengend, Eingaben zeichenweise erfassen – und dabei auf einen Zeilenumbruch prüfen zu wollen.

Deshalb müssen wir wohl einen etwas anderen Weg gehen und die `InputStream`-Instanz mit Hilfe eines passenden *Readers* einlesen – da bietet sich ein `BufferedReader` an, denn dieser kann sowohl zeilenweise lesen als auch Strings zurückgeben. Leider kann dieser Reader nicht direkt auf eine `InputStream`-Instanz angesetzt werden – dies kann aber sehr wohl ein `InputStreamReader`, der wiederum auf die `InputStream`-Instanz zugreift. Also werden wir einen `BufferedReader` erzeugen müssen, der wiederum einen `InputStreamReader` als Parameter übergeben bekommt, der seinerseits den `InputStream` unter `System.in` kapselt.

Um also Daten von der Kommandozeile einlesen zu können, müssen wir wider Erwarten doch ein wenig Aufwand betreiben. Dass sich dieser Aufwand aber durchaus in Grenzen hält und eigentlich nur das Erzeugen der `BufferedReader`-Instanz sowie das Überprüfen der eingebenen Daten umfasst, zeigt das folgende Beispiel, das zwei Zahlen entgegennimmt und diese addiert:

Listing 8.6
Einlesen von Daten von der Kommandozeile

```
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.regex.Pattern;

/**
 * Retrieves two values and adds them to each other
 */
public class Input {

   /**
    * Retrieves a number from command line
    * @param message Message to display when asking for the number
    * @return Number entered
    */
   private static int getNumber(String message) {
      int result = 0;
      boolean finished = false;
      String pattern = "^\\d+$";

      InputStreamReader in = new InputStreamReader(System.in);
      BufferedReader cmd = new BufferedReader(in);

      while(!finished) {
         try {
            System.out.print(message);
            String input = cmd.readLine();

            if(Pattern.matches(pattern, input)) {
               result = Integer.parseInt(input);
               finished = true;
            } else {
               System.out.println(
                  "You did not enter an integer value!");
               System.out.println();
            }
         } catch (IOException e) {
            e.printStackTrace();
         }
      }

      return result;
   }

   /**
    * The main application of this class
    */
   public static void main(String[] args) {
      int firstNumber = 0;
      int secondNumber = 0;

      firstNumber =
         getNumber("Please enter the first number: ");
      secondNumber =
         getNumber("Please enter the second number: ");
```

Listing 8.6 *(Forts.)* Einlesen von Daten von der Kommandozeile

```
        System.out.println();
        System.out.println(
           String.format("%d + %d = %d",
              firstNumber,
              secondNumber,
              firstNumber + secondNumber));
    }
}
```

Innerhalb der statischen Methode `main()` der Klasse `Input` deklarieren wir zwei Variablen vom Typ `int`. Beiden lassen wir ihren Wert mit Hilfe der Methode `getNumber()` zuweisen, die als Parameter den Text der Eingabe-Aufforderung übergeben bekommt. Anschließend geben wir die eingegebenen Zahlen und das Ergebnis ihrer Addition aus.

Innerhalb von `getNumber()` wird die Eingabe des Nutzers entgegengenommen. Zu diesem Zweck deklarieren wir am Anfang der Methode drei lokale Variablen: `result` wird die eingegebene Zahl aufnehmen, `finished` informiert uns darüber, ob der Nutzer eine gültige Eingabe vorgenommen hat, und `pattern` beinhaltet einen regulären Ausdruck, mit dessen Hilfe wir die eingegebene Zeichenkette untersuchen wollen:

```
int result = 0;
boolean finished = false;
String pattern = "^\\d+$";
```

Der reguläre Ausdruck in `pattern` sieht etwas kryptisch aus, drückt aber Folgendes aus:

- In der zu untersuchenden Zeichenkette dürfen vom Anfang (das Dach-Zeichen ^) bis zum Ende (ausgedrückt durch das Dollar-Symbol $) ausschließlich Ziffern (\\d) verwendet werden.
- Es muss mindestens eine Ziffer angegeben werden (ausgedrückt durch das Plus-Zeichen +).
- Falls andere Zeichen auftreten, wird die Verarbeitung abgebrochen (wird durch das Frage-Zeichen ? sinngemäß ausgedrückt).

Sie fragen sich, wieso der Autor so viel Wert auf die Überprüfung von Eingaben legt? Sehen Sie es doch einfach so: Sie wissen nie, wer das Programm ausführt und was er im Schilde führt. Kontrollieren Sie deshalb immer alle Werte, die auf irgendeine Art und Weise aus externen Quellen kommen!

Nun erzeugen wir zunächst eine `InputStreamReader`-Instanz, der im Konstruktor die zu verwendende `InputStream`-Instanz übergeben wird. In unserem Fall ist dies `System.in` – also die Standard-Eingabe. Diese `InputStreamReader`-Instanz wird dem Konstruktor des `BufferdReaders` als Parameter übergeben – nun können wir mit dessen Hilfe bequem zeilenweise Daten einlesen!

```
InputStreamReader in = new InputStreamReader(System.in);
BufferedReader cmd = new BufferedReader(in);
```

Dieses Einlesen geschieht innerhalb einer `while`-Schleife, deren Abbruchbedingung darin besteht, dass die lokale Variable `finished` den Wert `true` hat. Beim ersten Durchlauf ist dies definitiv nicht der Fall, denn `finished` ist mit dem Wert `false` instanziert worden. Also wird zumindest ein Mal der Körper der Schleife ausgeführt.

Hier geben wir zuerst die Eingabe-Aufforderung aus und lesen die Eingabe des Nutzers mit Hilfe der Methode `readLine()` der `BufferedReader`-Instanz in die lokale Variable `input` ein. Dies geschieht allerdings erst, wenn der Benutzer die ↵-Taste gedrückt hat. Bis dahin passiert für unsere Applikation scheinbar nichts – tatsächlich aber prüft der `BufferedReader` beständig, ob Zeichen eingegeben worden sind:

```
String input = cmd.readLine();
```

Sobald die ↵-Taste gedrückt worden ist, erfolgt mit Hilfe der Methode `matches()` der `Pattern`-Klasse aus dem `java.util.regex`-Paket die Überprüfung der in `input` enthaltenen Eingabe anhand des definierten Musters. Die Bedingungen, unter denen diese Überprüfung erfolgreich abgeschlossen werden kann und somit *true* zurückgegeben wird, sind weiter oben geschildert.

```
if(Pattern.matches(pattern, input)) { ... }
```

Sollte `Pattern.matches()` `true` zurückgeben, weil der Inhalt von `input` dem in `pattern` definierten Muster entspricht, wird dieser Inhalt mit Hilfe von `Integer.parseInt()` in einen `int`-Wert umgewandelt und der Variablen `result` zugewiesen. Die Variable `finished` erhält den Wert `true`, da erfolgreich eine Zahl eingelesen werden konnte und deshalb die Schleife nicht weiter durchlaufen werden muss:

```
result = Integer.parseInt(input);
finished = true;
```

Wenn die Überprüfung von `input` nicht erfolgreich verlaufen ist, wird eine entsprechende Fehlermeldung ausgegeben und die Schleife ein weiteres Mal durchlaufen. Dies wiederholt sich solange, bis der Nutzer aufgibt und eine ganze Zahl eingibt oder die Applikation abgebrochen wird.

Zuletzt erfolgt die Rückgabe der ermittelten Zahl an die aufrufende Methode `main()`.

Wenn wir die Klasse kompiliert haben, können wir sie von der Kommandozeile ausführen:

```
java Input
```

Nun werden Sie diese Ausgabe erhalten:

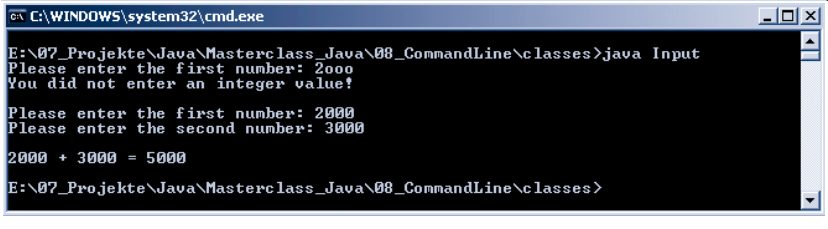


Abbildung 8.15
Abfragen und Überprüfen von Werten aus der Kommandozeile

Experimentieren Sie ruhig ein wenig mit den Werten herum, die Sie eingeben. Verwenden Sie Buchstaben oder andere Sonderzeichen, versuchen Sie, Kommata oder Punkte zu setzen. Sie werden feststellen, dass tatsächlich nur Ziffern erlaubt sind – alles andere wird durch den oben definierten und erklärten regulären Ausdruck abgewiesen.

Dieses Prinzip nennt sich *security-by-obscurity* (Sicherheit durch Vorsicht und Kontrolle) und ist der einfachste Weg, Applikationen abzusichern.

Dies setzt allerdings voraus, dass ein entsprechendes Problem-Bewusstsein auf Seiten des Entwicklers ebenso vorhanden ist, wie die Bereitschaft, Dinge oder Daten regelmäßig und immer wieder zu hinterfragen.

Anmerkung: Der Sinn der Überprüfung von Benutzereingaben liegt darin, Fehlverhalten der Software, falsche Ausgaben und Sicherheitslücken (Vulnerabilities) zu vermeiden (z.B. Parameter, die ohne Überprüfung an eine SQL-Engine geschickt werden). Exceptions haben nach wie vor ihre Berechtigung!

8.4 Ausgabe von Informationen

Die Ausgabe von Informationen auf der Kommando-Zeile haben Sie bereits häufig praktiziert. Sie haben sich zu diesem Zweck der *PrintWriter*-Instanz unter `System.out` bedient und somit den Standard-Ausgabestrom des Systems verwendet.

8.4.1 Einfache Ausgabe

Das einfache Ausgeben von Informationen findet mit Hilfe der mehrfach überladenen Methoden `print()` und `println()` statt. Beide nehmen Argumente der folgenden Datentypen als Parameter entgegen:

- `Object`
- `String`
- `char[]`
- `int`
- `long`
- `float`
- `double`
- *boolean*

Die Methode `println()` bietet darüber hinaus auch eine Überladung ohne Parameter an. Der Unterschied zwischen `print()` und `println()` besteht darin, dass `println()` am Ende der Ausgabe einen Zeilenumbruch anfügt und somit die nächste Ausgabe in einer neuen Zeile erfolgt.

Wenn Sie Objekte ausgeben wollen, ist dies aufgrund der Überladung der Methoden mit einem Argument vom Typ *Object* kein Problem, da jede Klasse in Java implizit von *Object* erbt. Beide Methoden werden zur Generierung die Methode `toString()` des jeweiligen Objekts aufrufen – auch die steht implizit bei jedem Objekt zur Verfügung, da sie bereits von der Basis-Klasse bereitgestellt wird.

Wenn Sie beispielsweise eine `Thread`-Instanz erzeugen und per `println()` ausgeben wollen, ist das sehr wohl möglich:

```
Thread t = new Thread();
System.out.println(t);
```

Die Ausgabe in diesem Fall wäre ähnlich dieser:

Thread[Thread-0,5,main]

8.4.2 Überschreiben der Methode toString()

Eine Ausgabe wie bei der `Thread`-Klasse mag für Debugging-Zwecke toll sein. Wenn Sie aber eine eigene Ausgabe für Ihre Objekte wünschen, dann überschreiben Sie einfach deren `toString()`-Methode mit einer eigenen Rückgabe:

```
public class CustomOutput {

   public String toString() {
      return "Ehem, yes. This is my output. "
         + "Now you are stunned, aren't you?";
   }
}
```

Wenn wir nun eine neue Instanz dieser `CustomOutput`-Klasse erzeugen würden, können wir sie ausgeben, indem wir sie der Methode `println()` als Parameter übergeben:

```
CustomOutput co = new CustomOutput();
System.out.println(co);
```

Die Ausgabe würde etwas umfangreicher ausfallen, als Sie dies bei der `Thread`-Klasse gesehen haben:

Ehem, yes. This is my output. Now you are stunned, aren't you?

8.4.3 Sichere Ausgabe von Zeilen-Umbrüchen

Die Methode `println()` bietet eine Überladung an, die ohne Parameter arbeitet. Mit Hilfe dieser Überladung können Sie eine Leerzeile ausgeben:

```
System.out.println();
```

Da `println()` ohnehin am Ende jeder Ausgabe einen Zeilenumbruch setzt, können Sie eine Zeichenkette entsprechend zerlegen und dann zeilenweise ausgeben lassen:

```
System.out.println("Hello!");
System.out.println("This is a text in a new line!");
System.out.println("This is a new line, too!");
```

Dies erzeugt folgende Ausgabe:

Hello!

This is a text in a new line!

This is a new line, too.

Aber auch mit Hilfe der Methode `print()` können Sie Zeilenumbrüche ausgeben. In den online zugänglichen Foren und Newsgroups wird gerne empfohlen, zu diesem Zweck das Token *\n* in die Zeichenkette an der Stelle einzufügen, an der der Zeilenumbruch stattfinden soll:

```
System.out.print("Hello!\nThis is a text in a new line!\n
   + "This is a new line, too.");
```

Auf den meisten Systemen wird die Verwendung des Tokens \n keine Probleme bereiten. Leider gilt dies nicht unter allen Umständen und ist darüber hinaus vom System selbst abhängig. Oder, um es kurz zu sagen: Die Ausgabe eines Zeilenumbruchs durch \n ist nicht in jedem Fall garantiert.

Glücklicherweise gibt es eine Lösung für das Problem. Diese bedingt zwar etwas mehr Schreibaufwand für den Entwickler, funktioniert dafür aber auf jedem System:

Listing 8.7
Ersetzen des Tokens \n durch eine sichere Variante

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * Replaces the default token \\n with a safe variant
 */
public class NewLine {

   public static String lf(String message) {
      String ls = System.getProperty("line.separator");
      Pattern p = Pattern.compile("\n");
      Matcher m = p.matcher(message);
      message = m.replaceAll(ls);

      return message;
   }
}
```

Die Klasse `NewLine` deklariert eine statische Methode `lf()`, die in übergebenen Texten das übliche Zeilenumbruch-Token \n so ersetzt, dass das vom System definierte Token verwendet wird. Dieses ermitteln wir mit Hilfe der Methode `getProperty()` der `System`-Klasse – unter dem Schlüssel *line.separator* können wir das systemspezifische Token für den Zeilenumbruch finden und abrufen:

```
String ls = System.getProperty("line.separator");
```

Anschließend ersetzen wir alle Vorkommen des Default-Tokens \n durch die systemspezifische Variante.

Wir bedienen uns dabei der `Matcher`-Klasse des `java.util.regex`-Pakets. Eine Instanz dieser Klasse erzeugen wir mit Hilfe einer `Pattern`-Instanz. Eine `Pattern`-Instanz wiederum erhalten wir, wenn wir der statischen Methode `compile()` der `Pattern`-Klasse einen regulären Ausdruck als Parameter übergeben:

```
Pattern p = Pattern.compile("\n");
Matcher m = p.matcher(message);
```

Jetzt können wir alle Vorkommen des Tokens \n in der als Parameter übergebenen Zeichenkette auf einen Schlag ersetzen. Dies erledigt die Methode `replaceAll()` der `Matcher`-Instanz:

```
message = m.replaceAll(ls);
```

Nach der Rückgabe des bearbeiteten Strings kann die aufrufende Methode diese Zeichenkette ausgeben.

Eine Methode, die `lf()` der `NewLine`-Klasse verwenden möchte, könnte dies dann so erledigen:

```
String output = "Hello!\nThis is a text in a new line!\n"
   + "This is a new line, too.";
System.out.println(NewLine.lf(output));
```

Die Ausgabe unterscheidet sich rein optisch nicht von einer nicht bearbeiteten Ausgabe – das Zeilenumbruch-Token ist ja kein druckbares Zeichen, sondern wird als Zeilenumbruch dargestellt. Würde man es im Text wieder sichtbar machen, sähe dieser unter Windows (Zeilenumbruch-Token: *\r\n*) nun so aus:

Hello!\r\nThis is a text in a new line!\r\nThis is a new line, too.

Unter Linux hätte sich dagegen nichts geändert, da hier *\n* das Token für den Zeilenumbruch ist:

Hello!\nThis is a text in a new line!\nThis is a new line, too.

In jedem Fall ist nunmehr sichergestellt, dass der Zeilenumbruch korrekt ausgegeben wird.

8.4.4 Umleiten der Ausgabe in eine Datei

Um eine Ausgabe dauerhaft vorhalten zu können, müssen wir sie auf irgendeine Art und Weise in einer Datei ablegen können. Die einfachste Möglichkeit, dies zu tun, ist den Standard-Ausgabestrom von Java in eine Datei umzuleiten. Dies kann aus einer Java-Applikation heraus erfolgen:

Listing 8.8
Umleiten der Konsolen-Ausgabe in eine Datei

```
import java.io.FileOutputStream;
import java.io.File;
import java.io.PrintStream;
import java.io.FileNotFoundException;

/**
 * Shows how to redirect the System.out to a file
 */
public class RedirectSystemOut {

   public static void main(String[] args) {
      try {
         FileOutputStream fos =
            new FileOutputStream(
               new File("./output.txt"), true);
         PrintStream out = new PrintStream(fos);

         System.setOut(out);

         System.out.println("Hello!");
      } catch (FileNotFoundException e) {
         e.printStackTrace();
      }
   }
}
```

Um die Standard-Ausgabe umleiten zu können, müssen wir der Methode `setOut()` der `System`-Klasse eine `PrintStream`-Instanz zuweisen. Diese `PrintStream`-Instanz soll ihrerseits die Daten mit Hilfe eines `FileOutputStreams` in eine Datei schreiben. Eine Datei wiederum wird durch eine `File`-Instanz repräsentiert.

Wenn wir das alles in eine logische Reihenfolge bringen, würden wir zunächst die `File`-Instanz erzeugen, die auf die Datei *output.txt* im aktuellen Verzeichnis verweist. Der Konstruktor der `FileOutputStream`-Instanz nimmt diese `File`-Instanz als Parameter entgegen. Der zweite Parameter ist vom Typ `boolean` und gibt an, dass eine eventuell vorhandene Datei *output.txt* nicht überschrieben werden soll, sondern der neue Inhalt am Datei-Ende angehängt werden soll:

```
FileOutputStream fos =
   new FileOutputStream(
      new File("./output.txt"), true);
```

Diese `FileOutputStream`-Instanz dient nun als Parameter für den Konstruktor der neuen `PrintStream`-Instanz `out`, die den Standard-Ausgabe-Strom überschreiben soll:

```
PrintStream out = new PrintStream(fos);
```

Das eigentliche Überschreiben des Standard-Ausgabe-Stroms findet mit Hilfe der Methode `setOut()` der `System`-Klasse statt:

```
System.setOut(out);
```

Wenn wir nun per `System.out.print()` oder `System.out.println()` eine Ausgabe erzeugen, wird diese nicht angezeigt, sondern der Datei *output.txt* angehängt.

Das gleiche Vorgehen können Sie übrigens für den Standard-Fehler-Ausgabe-Strom, der unter `System.err` erreicht werden kann, anwenden. Sie müssen in diesem Fall nur statt `System.setOut()` die Methode `System.setErr()` verwenden. Diese nimmt ebenfalls eine `PrintStream`-Instanz entgegen.

8.4.5 Ausgabe von Umlauten auf der Windows-Kommandozeile

Versuchen Sie doch einmal Folgendes: Geben Sie aus Java heraus auf der Standard-Ausgabe (der Konsole) Umlaute aus – etwa so:

```
System.out.println("German Umlaute: ÄÖÜßäöü");
```

In einer Klasse ausgeführt, werden Sie diese Ausgabe erhalten:

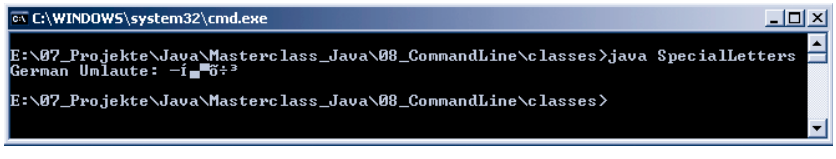


Abbildung 8.16
Umlaute auf der Kommando-Zeile? Nein, danke.

Das ist ein eher suboptimales Ergebnis. Das Problem liegt darin, dass das Encoding, das von der Standard-Ausgabe unter `System.out` verwendet wird, keine Umlaute unterstützt.

Wir müssen also nur das Encoding der Ausgabe ändern, um Umlaute ausgeben zu können. Das ist kein Problem, schließlich existiert doch eine Methode `setEncoding()`... Sparen Sie sich die Mühe! Diese Methode existiert nicht. Sun hat einfach nicht vorgesehen, dass jemand auf die Idee kommen könnte, Umlaute oder andere Nicht-Standard-Zeichen über die Kommandozeile auszugeben.

Allerdings gibt es eine Lösung: Wir können die Standard-Ausgabe über eine andere *PrintStream*-Instanz leiten, die das korrekte Encoding verwendet:

Listing 8.9
Setzen eines Encodings für die Kommandozeile, mit dem diese Umlaute ausgeben kann

```
import java.io.UnsupportedEncodingException;
import java.io.PrintStream;

/**
 * German Umlaute on the command line
 */
public class SpecialLetters {

   public static void main(String args[]) {
      System.out.println("German Umlaute: ÄÖÜßäöü");

      try {
         System.out.println("Switching output encoding!");

         PrintStream ps = new PrintStream(
            System.out, true, "cp850");
         System.setOut(ps);

         System.out.println("German Umlaute: ÄÖÜßäöü");
      } catch (UnsupportedEncodingException e) {
         e.printStackTrace();
      }
   }
}
```

Ein Encoding, mit dem die Kommandozeile zumindest auf Windows-Systemen Umlaute zuverlässig ausgeben kann, ist die *CodePage 850* – abgekürzt mit *cp850*. Dieses Encoding müssen wir einer neuen `PrintStream`-Instanz zuweisen und diese Instanz dann per `System.setOut()` als Standard-Ausgabe-Strom deklarieren.

Eine Überladung des Konstruktors der `PrintStream`-Klasse benötigt folgende Parameter:

- Stream-Instanz, die intern für die Ausgabe verwendet werden soll
- Angabe, ob ein automatisches Leeren des Ausgabe-Puffers vorgenommen werden soll
- Encoding, das für die Ausgabe verwendet werden soll

Das klingt doch exakt nach einer Überladung, mit der wir arbeiten können!

Definieren wir also unsere neue `PrintStream`-Instanz wie folgt: Als Basis-Stream soll `System.out` verwendet werden, schließlich könnte es sein, dass sich bereits Zeichen im Ausgabe-Puffer befinden. Außerdem wünschen wir ein automatisches Leeren des Ausgabe-Puffers und schließlich soll die *CodePage 850* als Encoding genutzt werden:

```
PrintStream ps = new PrintStream(System.out, true, "cp850");
```

Nun können wir die `PrintStream`-Instanz `ps` als Standard-Ausgabe-Strom festlegen. Dies erledigen wir mit Hilfe der Methode `System.setOut()`, der wir `ps` als Parameter übergeben:

```
System.setOut(ps);
```

Wenn wir nun Umlaute ausgeben würden, sollten diese nach dem Umstellen des Encodings auch korrekt dargestellt werden:

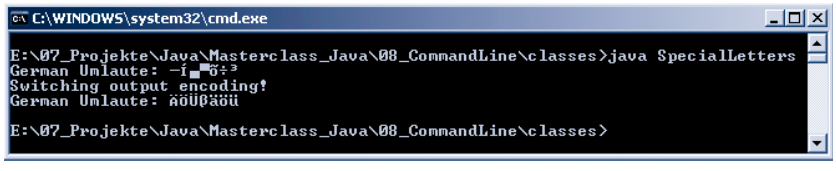


Abbildung 8.17
Nach dem Umstellen des Encodings werden auch Umlaute korrekt angezeigt

8.5 Zusammenfassung

Obwohl die Kommandozeile nicht den bevorzugten Kommunikationsweg von Java gegenüber einem Nutzer darstellt, ist sie doch recht mächtig und kann neben der reinen Ausgabe eine Menge mehr.

Interessant ist sicherlich, wie und mit welchem Aufwand man mit den unterschiedlichen Arten von Parametern arbeiten kann, ist dies doch eine der besten Möglichkeiten, Kommandozeilen-Applikationen (oder Applikationen, die von der Kommandozeile gestartet werden können) flexibler zu gestalten. Wenn Sie als Entwickler dabei den Sicherheitsgedanken nicht vernachlässigen und alles überprüfen, was man Ihnen als Parameter übergibt, dann können Sie eine Menge mit Java von der Kommandozeile aus anstellen.

Auch das Einlesen und Ausgeben von Daten über die Kommandozeile kann gemeistert werden. Ein wenig Übung und die richtigen Informationen vorausgesetzt, schreiben Sie schnell Applikationen, die die Kommandozeile effektiv für ihre Kommunikation mit Anwender und System nutzen. Sie sind ebenfalls in der Lage, Restriktionen der Kommandozeile zu umgehen und diese sogar zu anderen Ausgabe-Zielen umzuleiten.

9

Swing

Darauf haben Sie sicher schon die ganze Zeit gewartet: Swing ist das Paket, das uns hilft, grafische Benutzer-Oberflächen samt Maus- und Tastatur-Steuerung zu erstellen. Weg von der Kommando-Zeile, hin zu „richtigen“ Applikationen!

9.1 Was ist Swing?

Swing stellt eine Gruppe von Klassen dar, die auf dem so genannten Abstract Windowing Toolkit (AWT) aufbauen. Das AWT stellt Basis-Funktionalitäten für die Entwicklung grafischer Benutzer-Oberflächen bereit. Es ist schon seit Java 1.0 verfügbar und bildet heute die rudimentäre Basis von Swing.

Swing selbst ist ein fast komplettes Set zur Entwicklung grafischer Benutzer-Oberflächen. Es ist portabel und kann sich dem Stil des Betriebssystems, auf dem die Applikation läuft, anpassen. Es bringt auch ein eigenes Look & Feel – genannt „Metal“ – mit. Das Look & Feel einer Swing-Applikation kann sogar zur Laufzeit geändert werden.

Swing arbeitet komponentenorientiert. Dies bedeutet, dass eine Swing-Applikation nicht monolithisch ist, sondern aus einzelnen kleinen Elementen besteht. Diese Elemente werden innerhalb von so genannten Containern gruppiert und können Ereignisse werfen, auf die die Applikation reagieren kann und sollte.

Sämtliche Swing-Komponenten befinden sich im Paket javax.swing. Bestimmte Basis-Funktionalitäten und die Reaktion auf Events werden mit Hilfe der Pakete java.awt und java.awt.event umgesetzt.

Es existiert übrigens eine 1:n-Beziehung zwischen AWT-Komponenten und Swing-Komponenten. Dies bedeutet, dass sämtliche AWT-Komponenten eine Entsprechung in Form (mindestens) einer Swing-Komponente haben. Allerdings ist dies eine einseitige Angelegenheit, denn nicht zu allen Swing-Komponenten existieren Entsprechungen im AWT.

9.2 Swing oder JFC?

Wie soll man es nennen? *Swing* oder *Java Foundation Classes (JFC)*, was der offizielle Name ist? `Swing` war der Codename der `JFC` während der Entwicklung. Damit wäre eigentlich klar, dass wir zukünftig den offiziellen Namen verwenden.

Einerseits.

Andererseits ist der Name Swing so präsent in den Paket-Bezeichnungen und hat sich weltweit so eingebürgert, dass auch heute nur ganz Penible JFC sagen. Die große Masse der Entwickler nennt die JFC nach wie vor Swing.

Und genau aus diesem Grund werden wir dies auch tun.

9.3 Swing-Basics

Swing unterscheidet zwischen Komponenten und Containern. Komponenten dienen der Darstellung oder der Interaktion, Container gruppieren und organisieren Komponenten.

9.3.1 Komponenten

Alle Swing-Komponenten sind von einer Basis-Klasse abgeleitet: `javax.swing.JComponent`. Diese Klasse stellt einige Methoden bereit, mit deren Hilfe grundlegende Einstellungen vorgenommen werden können:

- `setBackground(Color)`: Setzt die Hintergrund-Farbe der Komponente.
- `setEnabled(boolean)`: Legt fest, ob die Komponente aktivierbar ist.
- `setFont(Font)`: Setzt die Schriftart für die Komponente.
- `setForeground(Color)`: Setzt die Vordergrund-Farbe der Komponente.
- `setToolTipText(String)`: Legt den Text fest, der angezeigt wird, wenn die Maus länger über der Komponente verweilt.
- `setVisible(boolean)`: Legt fest, ob die Komponente sichtbar ist.

Weitere Methoden werden von den Komponenten selbst bereitgestellt. Für die Initialisierung wird in der Regel der Standard-Konstruktor verwendet. Neue Instanzen von abgeleiteten Klassen werden wie gewohnt mit Hilfe des Schlüsselworts `new` erzeugt.

9.3.2 Root-Container

Eine Komponente alleine nutzt Ihnen nichts, wenn sie nicht einem Container zugeordnet ist. Container-Elemente erben direkt oder indirekt von `javax.awt.Container`. In der Praxis werden Sie selten mit einem `AWT`-Container arbeiten. Beim Einsatz von Swing verwenden Sie stattdessen viel eher die Klasse `javax.swing.JFrame` als Basis-Element, dem Sie alle anderen Elemente zuordnen.

Der JFrame verfügt über die Möglichkeit, eine Titel-Zeile mit den Icons zum Minimieren, Maximieren und Schließen der Applikation anzuzeigen. Sie können einen Titel-Text vergeben und ein Verhalten beim Schließen der Applikation festlegen.

Im Folgenden werden wir JFrame-Container in allen Beispielen als Basis- oder Root-Container verwenden.

9.3.3 Verwendung von JFrame

Die einfachste Möglichkeit, einen Container zu erzeugen, besteht darin, eine Klasse von JFrame erben zu lassen und im Konstruktor einige Eigenschaften zu setzen.

Zwei Eigenschaften sind dabei ganz besonders wichtig: die Sichtbarkeit und das Schließ-Verhalten.

JFrames sind per Default nicht sichtbar und müssen erst mit Hilfe der Methode setVisible() sichtbar gemacht werden, indem dieser Methode der Wert true übergeben wird.

Das Schließ-Verhalten sollte ebenfalls deklariert werden. Zu diesem Zweck genügt es, der Methode setDefaultCloseOperation() eine der folgenden Konstanten zu übergeben:

- WindowConstants.DO_NOTHING_ON_CLOSE: Keine Default-Operation vornehmen.
- WindowConstants.HIDE_ON_CLOSE: Verstecken des Containers – dies ist der Standard-Wert.
- WindowConstants.DISPOSE_ON_CLOSE: Verstecken des Containers, Entfernen aller Referenzen auf den Container, Setzen des Containers auf null.
- JFrame.EXIT_ON_CLOSE: Beenden der kompletten Java-Applikation beim Schließen des Containers.

Für eine Applikation, die aus einer JFrame-Instanz besteht, erscheint es ratsam, den Wert von JFrame.EXIT_ON_CLOSE als Argument zu übergeben, da so eine Möglichkeit gegeben ist, die komplette Applikation recht einfach zu beenden.

Weiterhin können innerhalb des Konstruktors untergeordnete Elemente angelegt und initialisiert werden. Ebenso können Position und Größe des Containers gesetzt werden.

Der Ablauf beim Erzeugen eines neuen JFrame-Containers ist also dieser:

- Deklarieren einer Klasse, die von JFrame erbt
- Titel setzen, der in der Titel-Zeile angezeigt werden soll (optional)
- Erzeugen untergeordneter Elemente (optional)
- Setzen der bevorzugten Breite und Höhe (optional)
- Anzeigen des Containers

Umgesetzt in einer kleinen Beispiel-Applikation könnte dies dann so aussehen:

Listing 9.1 Erzeugen eines Fensters mit einem kurzen Text

```
import javax.swing.JFrame;
import javax.swing.JLabel;
import java.awt.Dimension;

/**
 * Our first Swing application!
 */
public class SimpleFrame extends JFrame {

   /**
    * Constructs a new JFrame containing a JLabel
    */
   SimpleFrame() {
      super();

      setTitle("My first Swing component!");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      add(
         new JLabel("Welcome to Swing!", JLabel.CENTER));

      setPreferredSize(new Dimension(350, 100));
      pack();

      setVisible(true);
   }

   /**
    * Creates an instance of this class
    */
   public static void main(String[] args) {
      SimpleFrame sf = new SimpleFrame();
   }
}
```

Unsere Klasse `SimpleFrame` erbt von `JFrame`, was wir mit Hilfe des `extends`-Schlüsselworts deklarieren. Damit haben wir die Möglichkeit, innerhalb des Konstruktors verschiedene Einstellungen vorzunehmen. Zunächst legen wir den in der Titel-Zeile anzuzeigenden Text fest:

```
setTitle("My first Swing component!");
```

Nun definieren wir, welches Schließ-Verhalten das Fenster zeigen soll. Wir entscheiden uns hier für das Schließ-Verhalten `JFrame.EXIT_ON_CLOSE`, das dafür sorgt, dass sich die Applikation beim Klick auf die Schließen-Schaltfläche selbstständig beendet:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Jetzt fügen wir dem Container ein untergeordnetes `JLabel`-Element hinzu. Dieses erzeugen wir mit Hilfe einer Konstruktor-Überladung, die eine anzuzeigende Zeichenkette und die Position des Textes innerhalb des Labels entgegennimmt. Die Methode `add()` des Containers nimmt das `JLabel` dann entgegen:

```
add(new JLabel("Welcome to Swing!", JLabel.CENTER));
```

Mit Hilfe der Methode `setPreferredSize()` können wir die initiale Breite des Containers setzen. Wir übergeben ihr zu diesem Zweck eine `Dimension`-Instanz, die eine Breite von 350 Pixeln und eine Höhe von 100 Pixeln definiert:

```
setPreferredSize(new Dimension(350, 100));
```

Die Methode `pack()` des Containers sorgt für die optimale Anzeige der enthaltenen Elemente, indem sie deren Höhe und Breite setzt. Aus diesem Grund war es für uns nicht nötig, die Maße des `JLabels` anzugeben:

```
pack();
```

Die Methode `setVisible()` sorgt dafür, dass der Container sichtbar wird. Voraussetzung dafür ist, dass ihr als Argument `true` übergeben wird:

```
setVisible(true);
```

Wenn wir die Klasse ausführen, sehen wir eine kleine Java-Applikation erscheinen:

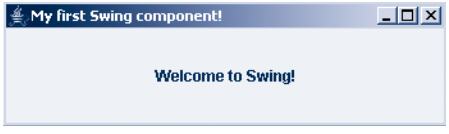


Abbildung 9.1
Unsere erste Swing-Komponente

Viel können wir nun mit diesem Fenster nicht anfangen – aber das Schließen funktioniert in jedem Fall!

Auf JFrame-Events reagieren

`JFrame` kann verschiedene Ereignisse werfen. Klassen können sich an diese Ereignisse binden, indem sie eine bestimmte Schnittstelle implementieren. Alle gebundenen Klassen und deren passende Ereignis-Behandlungsmethoden werden automatisch aufgerufen, wenn ein Ereignis eingetreten ist.

Folgende Ereignisse können von `JFrame` geworfen werden:

- `windowActivated(WindowEvent)`: Das Fenster wurde aktiviert.
- `windowClosed(WindowEvent)`: Das Fenster wurde geschlossen.
- `windowClosing(WindowEvent)`: Das Fenster wird gerade geschlossen.
- `windowDeactivated(WindowEvent)`: Das Fenster wurde deaktiviert.
- `windowDeiconified(WindowEvent)`: Die Fenstergröße wurde von minimiert auf normal geändert.
- `windowIconified(WindowEvent)`: Das Fenster wurde minimiert.
- `windowOpened(WindowEvent)`: Das Fenster wurde das erste Mal angezeigt.
- `windowStateChanged(WindowEvent)`: Der Zustand des Fensters hat sich geändert.
- `windowGainedFocus(WindowEvent)`: Das Fenster hat den Eingabe-Focus erhalten.
- `windowLostFocus(WindowEvent)`: Das Fenster hat den Eingabe-Focus verloren.

Die Behandlungsmethoden für diese sieben Ereignisse werden in drei Interfaces deklariert: `WindowListener`, `WindowFocusListener` (für die Ereignisse `windowGainedFocus` und `windowLostFocus`) und `WindowStateListener` (für das Ereignis `windowStateChanged`). Alle drei Interfaces gehören zum Package `java.awt.event`, das Ereignisse und deren Behandlungsmethoden (genannt `EventListener`) definiert.

Wenn wir also `EventListener` für eines der oben genannten Ereignisse schreiben wollten, müssten wir eines der drei Interfaces implementieren. Im schlechtesten Fall wären das dann sechs Methoden, die wir unnötigerweise umsetzen würden – nämlich dann, wenn wir uns nur für eines der Ereignisse aus dem Interface `WindowListener` interessierten.

Um dies zu vermeiden, werden wir in der Regel nicht direkt mit den Interfaces arbeiten, sondern stattdessen die abstrakte Klasse `WindowAdapter` als Basis verwenden. `WindowAdapter` implementiert alle Methoden der drei Interfaces – wir müssen also in Ableitungen von `WindowAdapter` nur noch die Methoden überschreiben, die uns interessieren.

Wie wäre es, wenn wir einfach eine `WindowAdapter`-Ableitung schreiben würden, die auf das `windowClosing`-Ereignis reagieren könnte? So könnte deren Code dann aussehen:

Listing 9.2
Binden eines EventListeners an eine JFrame-Instanz

```
import javax.swing.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

/**
 * EventListener for the windowClosing-event of a JFrame
 */
public class WindowAdapterImpl extends WindowAdapter {

   private int counter = 0;

   /**
    * Handling the windowClosing-event of a JFrame-instance
    */
   public void windowClosing(WindowEvent e) {
      JFrame frame = (JFrame)e.getComponent();
      frame.setLocation(
         frame.getLocation().x + 10,
         frame.getLocation().y + 10);

      counter++;
      if(counter > 10) {
         frame.setVisible(false);
         System.exit(0);
      }
   }

   /**
    * Invokes the Frame, sets some texts and adds the listener
    */
```

```
   public static void main(String args[]) {
      JFrame frame = new SimpleFrame();
      frame.setTitle("You won't close me!");
      frame.setDefaultCloseOperation(
         WindowConstants.DO_NOTHING_ON_CLOSE);

      JLabel label =
         (JLabel)frame.getContentPane().getComponent(0);
      label.setText("Try to close this window! :-)");

      frame.addWindowListener(new WindowAdapterImpl());
   }
}
```

Listing 9.2 *(Forts.)*
Binden eines EventListeners an eine JFrame-Instanz

Die hier deklarierte Klasse `WindowAdapterImpl` erbt von der abstrakten Klasse `WindowAdapter`, die Methoden-Deklarationen bereitstellt, die als `EventListener` für Ereignisse von `Swing`-Containern fungieren.

Beim Aufruf von `WindowAdapterImpl` wird deren statische Methode `main()` eingebunden. Diese erzeugt eine neue `JFrame`-Instanz. Der Einfachheit halber erzeugen wir hier eine Instanz der weiter oben in diesem Kapitel erläuterten `SimpleFrame`-Klasse, die ihrerseits schon ein untergeordnetes Element (eine `JLabel`-Instanz) erzeugt und sich selbst zugewiesen hat:

```
JFrame frame = new SimpleFrame();
```

Da die `JFrame`-Instanz bereits über einen Titel verfügt, überschreiben wir diesen mit einem Text, der mehr zu unserem Beispiel passt:

```
frame.setTitle("You won't close me!");
```

Viel wichtiger jedoch als der Fenster-Titel (der im Grunde nur Dekoration ist) ist die Änderung des Schließ-Verhaltens des Containers. Wir setzen es mit Hilfe der Methode `setDefaultCloseOperation()` auf den Wert von `WindowConstants.DO_NOTHING_ON_CLOSE`, da wir eine eigene Behandlung dieses Ereignisses implementieren wollen:

```
frame.setDefaultCloseOperation(
   WindowConstants.DO_NOTHING_ON_CLOSE);
```

Erinnern Sie sich, dass die `SimpleFrame`-Klasse im Konstruktor eine `JLabel`-Instanz erzeugt und sich selbst per `add()` zugewiesen hat? Auf die können wir nämlich auch aus dieser Klasse heraus zugreifen. Wenn man weiß, dass Komponenten dem vordefinierten Container `ContentPane` zugewiesen werden, kann man über dessen Methode `getComponent()` auf eines der enthaltenen Elemente zugreifen.

Wir greifen in diesem Fall auf das erste Element in diesem Container zu – und da dies das `JLabel` ist, das im Konstruktor von `SimpleFrame` erzeugt worden ist, können wir dessen Anzeigetext einfach ändern:

```
JLabel label =
   (JLabel)frame.getContentPane().getComponent(0);
label.setText("Try to close this window! :-)");
```

In der Praxis würden Sie vor dem Zugriff auf die Methode `setText()` sicherlich zunächst überprüfen, ob die `JLabel` Instanz `null` ist. Hier sei aus Platzgründen darauf verzichtet.

Nach der Initialisierung und Manipulation der `JFrame`-Instanz können wir jetzt unseren `EventListener` an deren Ereignisse binden. Dies geschieht, indem wir eine Instanz der von `WindowAdapter` abgeleiteten Klasse `WindowAdapterImpl` als Argument an die Methode `addWindowListener()` der `JFrame`-Instanz `frame` übergeben:

```
frame.addWindowListener(new WindowAdapterImpl());
```

Unsere `WindowAdapterImpl`-Instanz, die wir als `EventListener` an den `JFrame` übergeben haben, reagiert nur auf dessen Ereignis `windowClosing`. Dies ist durch das Überschreiben der Basis-Methode `windowClosing()` definiert.

Der EventListener windowClosing

Innerhalb von `windowClosing()` holen wir uns eine Referenz auf die `JFrame`-Instanz, die das Ereignis ausgelöst hat. Dies geschieht durch das Abrufen einer generischen `Component`-Instanz durch die Methode `getComponent()` der `WindowEvent`-Instanz `e`.

Da wir in diesem Beispiel natürlich wissen, dass das Ereignis `windowClosing` durch eine `JFrame`-Instanz erzeugt worden ist, können wir die `Component`-Instanz direkt in eine `JFrame`-Instanz casten:

```
JFrame frame = (JFrame)e.getComponent();
```

Nun erlauben wir uns einen kleinen (schlechten) Scherz mit dem ahnungslosen User unserer Applikation: Statt sie zu schließen, verschieben wir sie einfach um je zehn Pixel nach rechts und nach unten.

Die neue Position errechnet sich anhand der von `getLocation()` zurückgegebenen `Point`-Instanz, die uns mit Hilfe ihrer Member `x` und `y` Auskunft darüber gibt, wo genau sich die linke obere Ecke des Elements befindet. Den neuen Wert setzen wir mit Hilfe der Methode `setLocation()`, die als Argumente die neuen `x`- und `y`-Positionen des Containers erwartet:

```
frame.setLocation(
   frame.getLocation().x + 10, frame.getLocation().y + 10);
```

Um den schlechten Scherz nicht zu weit zu treiben und dadurch ein Schließen der Applikation unmöglich zu machen, haben wir eine private Variable `counter` deklariert, deren Wert bei jedem `windowClosing`-Ereignis erhöht wird. Sollte dieser Wert größer als 10 werden, schließen wir die `JFrame`-Instanz. Dies geschieht, indem wir sie zunächst auf unsichtbar setzen und anschließend die JVM terminieren:

```
frame.setVisible(false);
System.exit(0);
```

Ein Benutzer, der diese Applikation ausführen will, kann nach deren Aufruf versuchen, sie per Klick auf ihr Schließen-Icon zu beenden:

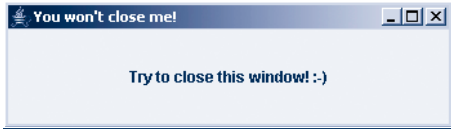


Abbildung 9.2
Dieses Fenster lässt sich nur nach mehrmaliger Aufforderung schließen

In der Praxis sollten Sie allerdings von solchen Spielereien Abstand nehmen. Das `windowClosing`-Ereignis bietet sich statt für Benutzer-Quälereien viel eher dafür an, Sicherheitsabfragen bei noch nicht gespeicherten Daten zu stellen oder bestimmte Aufräumarbeiten durchzuführen.

Einsatz anonymer Klassen für Events

Die bereits im Kapitel 3 angesprochenen abstrakten Klassen bieten sich gerade im Umfeld von Swing besonders an, da meist zielgerichtet auf ein bestimmtes Ereignis reagiert werden soll und die Wiederverwendbarkeit der Klassen keine maßgebliche Rolle spielt.

Lassen Sie uns das an obigem Beispiel verdeutlichen: Wir hatten die Methode `windowClosing()` als Bestandteil der Klasse `WindowAdapterImpl` implementiert, die ihrerseits von `WindowAdapter` erbte. Dies könnte stattdessen auch mit Hilfe einer anonymen Klasse umgesetzt werden:

Listing 9.3
Implementierung eines WindowListeners durch eine anonyme Klasse

```
import javax.swing.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class WindowAdapterImpl {

   private int counter = 0;

   public static void main(String args[]) {
      JFrame frame = new SimpleFrame();
      frame.setTitle("You won't close me!");
      frame.setDefaultCloseOperation(
         WindowConstants.DO_NOTHING_ON_CLOSE);

      JLabel label =
         (JLabel)frame.getContentPane().getComponent(0);
      label.setText("Try to close this window! :-)");

      frame.addWindowListener(
         new WindowAdapter{
            public void windowClosing(WindowEvent e) {
               JFrame frame = (JFrame)e.getComponent();
               frame.setLocation(
               frame.getLocation().x + 10,
               frame.getLocation().y + 10);
```

Listing 9.3 *(Forts.)* Implementierung eines WindowListeners durch eine anonyme Klasse

```
                counter++;
                if(counter > 10) {
                   frame.setVisible(false);
                   System.exit(0);
                }
             }
          }
       );
    }
}
```

Zu den Vor- und Nachteilen von anonymen Klassen ist an entsprechender Stelle schon einiges gesagt worden. Hier macht deren Einsatz durchaus Sinn, da der Code nicht wiederverwendbar sein muss und nur für diesen Frame zum Einsatz kommt.

9.3.4 Basis-Klasse für alle folgenden Beispiele

Für alle Beispiele in diesem Abschnitt werden wir eine Basis-Klasse verwenden, die eine `JFrame`-Instanz erzeugt, einen Titel setzt und die Abmessung des Containers definiert:

Listing 9.4 Die Klasse BaseFrame fungiert als Basis-Klasse für alle folgenden Beispiele

```
import javax.swing.*;
import java.awt.*;

/**
 * Base JFrame-implementation which defines some settings
 */
public class BaseFrame extends JFrame {

   BaseFrame() {
      super();


      setTitle("Swing Components");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setPreferredSize(new Dimension(350, 100));

      setVisible(true);
   }
}
```

9.4 Mehrzweck-Container

`JFrame` ist ein Container, der nur als Haupt-Container eingesetzt wird. Daneben existieren aber noch weitere Container, deren primäre Aufgabe darin besteht, Komponenten zu gruppieren und in bestimmten Darstellungsformen anzuzeigen.

9.4.1 JPanel

Das `JPanel` ist der einfachste und gleichzeitig der beliebteste Mehrzweck-Container. Per Default werden `JPanels` selber nicht dargestellt, man kann sie jedoch einfach mit Rahmen versehen. `JPanel`-Elemente können mit Layouts versehen

werden – diese Layout-Einstellungen beeinflussen, wie untergeordnete Elemente angeordnet werden. Standardmäßig verwendet das `JPanel` das `FlowLayout` – Elemente werden hintereinander angeordnet:

Listing 9.5
Mit Hilfe von JPanel-Elementen werden Komponenten organisiert

```
import javax.swing.*;
import java.awt.*;

/**
 * Displays assigned components
 */
public class Panel extends BaseFrame {

   Panel() {
      JPanel panel = new JPanel(new GridLayout(2, 1));
      panel.setBorder(
         BorderFactory.createEmptyBorder(10, 10, 10, 10));

      add(panel);

      JLabel message = new JLabel("This is some text");
      panel.add(message);

      JLabel secondLine = new JLabel(
         "This is a second line of text!");
      panel.add(secondLine);

      this.pack();
   }
}
```

Einer `JPanel`-Instanz kann im Konstruktor schon übergeben werden, welche Art von Layout verwendet werden soll. In diesem Fall ist es das so genannte `GridLayout`, das das Anordnen von Elementen in Zeilen und Spalten erlaubt. Wir definieren hier, dass zwei Zeilen mit jeweils einer Spalte verwendet werden sollen:

```
JPanel panel = new JPanel(new GridLayout(2, 1));
```

Nun definieren wir den Rahmen. In diesem Fall wollen wir keinen sichtbaren Rahmen haben, sondern stattdessen etwas, das mehr die Funktion eines Abstandhalters hat: Einen leeren oder unsichtbaren Rahmen, der oben, links, unten und rechts jeweils zehn Pixel Abstand erzwingt:

```
panel.setBorder(
   BorderFactory.createEmptyBorder(10, 10, 10, 10));
```

Das `JPanel` kann nun dem übergeordneten Container zugewiesen werden – dies muss nicht zwingend ein `JFrame` sein, sondern kann beispielsweise auch ein anderes `JPanel` sein. In diesem Fall ist es dann aber doch die `JFrame`-Instanz, die das Hauptfenster der Anwendung bildet und von der unsere Applikation indirekt erbt:

```
add(panel);
```

Untergeordnete Elemente können nun erzeugt und hinzugefügt werden. Wir entscheiden uns hier für zwei einfache Anzeige-Elemente in Form von `JLabel`-

Instanzen (auf deren spezifische Funktionalität wir weiter unten in diesem Kapitel eingehen werden), die jeweils eine Text-Zeile repräsentieren sollen. Nach dem Erzeugen der Label weisen wir sie dem Panel mit Hilfe von dessen `add()`-Methode zu:

```
JLabel message = new JLabel("This is some text");
panel.add(message);

JLabel secondLine = new JLabel(
"This is a second line of text!");
panel.add(secondLine);
```

Das Ergebnis der Mühen kann sich durchaus sehen lassen:

Abbildung 9.3
Das Panel enthält zwei JLabel-Instanzen, die untereinander angezeigt werden

9.4.2 JScrollPane

Das `JScrollPane` ist verantwortlich dafür, dass die enthaltene Komponente scrollen kann. Dies ist dann sinnvoll, wenn der Platz, den die Komponente einnehmen soll, größer ist als der Platz, der tatsächlich zur Verfügung steht. Typische Elemente, die in `JScrollPane`-Instanzen angezeigt werden, sind Text-Felder oder große Bilder. Die Arbeit mit dem `JScrollPane` gestaltet sich dabei sehr einfach:

- Zunächst wird eine neue Instanz erstellt, wobei das anzuzeigende Element zugewiesen wird.
- Anschließend sollte die darzustellende Größe festgelegt werden.
- Zuletzt wird das JScrollPane dem übergeordneten Container zur Anzeige zugewiesen.

Der Konstruktor der `JScrollPane`-Klasse nimmt dabei die darzustellende Komponente entgegen:

```
public JScrollPane(Component view)
```

Mehr Aufwand ist nicht nötig, wie auch das folgende Beispiel belegt:

Listing 9.6
Erzeugen einer JScrollPane-Instanz für das Scrolling von enthaltenen Komponenten

```
import javax.swing.*;
import java.awt.*;

/**
 * Displays a JTextArea in a JScrollPane
 */
public class ScrollPane extends BaseFrame {

   ScrollPane() {
      JPanel content = new JPanel(new BorderLayout());
      this.add(content);

      JTextArea textArea =
         new JTextArea("Enter your text here...", 20, 40);
```

```
        JScrollPane pane = new JScrollPane(textArea);
        pane.setPreferredSize(new Dimension(250, 150));
        content.add(pane, BorderLayout.CENTER);

        // ...


    }
}
```

Listing 9.6 *(Forts.)*
Erzeugen einer JScrollPane-Instanz für das Scrolling von enthaltenen Komponenten

In diesem Beispiel erzeugen wir als Inhalts-Container eine neue `JPanel`-Instanz. Diese wird das `JScrollPane` aufnehmen. Bevor wir aber die `JScrollPane`-Instanz erzeugen können, müssen wir zuerst die Komponente deklarieren, die sie beherbergen soll. In diesem Fall handelt es sich um eine `JTextArea`. Wir weisen dieser im Konstruktor den anzuzeigenden Text zu und geben danach an, dass sie eine Größe von 20 Zeilen und 40 Spalten haben soll:

```
JTextArea textArea =
   new JTextArea("Enter your text here...", 20, 40);
```

Offensichtlich wäre die `JTextArea`-Instanz in der Darstellung recht groß. Deshalb weisen wir sie einem `JScrollPane` zu:

```
JScrollPane pane = new JScrollPane(textArea);
```

Nun definieren wir die bevorzugte Anzeige-Größe der `JScrollPane`-Instanz. Zu diesem Zweck weisen wir ihrer Methode `setPreferredSize()` eine `Dimension`-Instanz zu, deren Konstruktor wir die Werte für Breite und Höhe in Pixeln übergeben haben:

```
pane.setPreferredSize(new Dimension(250, 150));
```

Nach dem Hinzufügen der `JScrollPane` zum Anzeige-Panel wird unsere Applikation so aussehen:

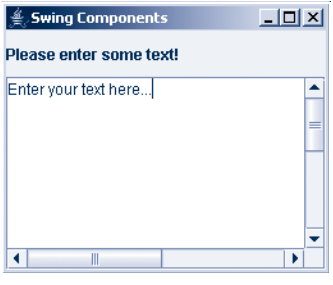


Abbildung 9.4
Anzeige einer JTextArea innerhalb einer JScrollPane-Instanz

9.4.3 JSplitPane

Wenn Sie zwei Elemente neben- oder übereinander anzeigen müssen und dabei den Nutzer entscheiden lassen wollen, welcher Komponente er welche Anzeige-Größe zugestehen möchte, empfiehlt sich der Einsatz des `JSplitPane`. Zwar könnten Sie diesem Container direkt Komponenten zuweisen – in der Praxis empfiehlt es sich jedoch, `JScrollPanes` mit den eigentlich anzuzeigenden Ele-

ment zuzuweisen, da Sie nicht absehen können, welches Element der Benutzer in welcher Größe darstellen möchte.

Der Konstruktor einer `JSplitPane` nimmt drei Argumente entgegen: den Anzeigetyp des Trenners (horizontal oder vertikal) und die beiden darzustellenden Komponenten:

```
public JSplitPane(int newOrientation,
   Component newLeftComponent,
   Component newRightComponent)
```

Daneben existieren noch weitere Überladungen. Erwähnenswert wäre noch diese Überladung, der wir mit Hilfe des booleschen Parameters `newContinuousLayout` mitgeben können, ob die enthaltenen Komponenten ständig neu gezeichnet werden (dann übergeben Sie *true* als Wert) oder dies nur erfolgt, nachdem ein Scrolling- oder Resize-Vorgang abgeschlossen worden ist. Letzteres entspricht dem Standard-Verhalten und ist performanter:

```
public JSplitPane(int newOrientation,
   boolean newContinuousLayout,
   Component newLeftComponent,
   Component newRightComponent)
```

Nach der Instanzierung kann das `JSplitPane` einem übergeordneten Container zugewiesen und somit zur Anzeige gebracht werden:

Listing 9.7
Anzeige zweier Elemente mit Hilfe einer JSplitPane

```
import javax.swing.*;
import java.awt.*;

/**
 * Uses a JSplitPane to display two elements
 */
public class SplitPane extends BaseFrame {

   SplitPane() {
      JLabel label = new JLabel(new ImageIcon("baby.jpg"));
      JTextArea text = new JTextArea(
         "Here you may enter some text", 20, 40);

      JScrollPane left = new JScrollPane(label);
      left.setPreferredSize(new Dimension(200, 200));

      JScrollPane right = new JScrollPane(text);
      right.setPreferredSize(new Dimension(200, 200));

      JSplitPane pane = new JSplitPane(
         JSplitPane.HORIZONTAL_SPLIT, left, right);

      this.add(pane);
      this.pack();
   }
}
```

In diesem Beispiel erzeugen wir zunächst zwei Elemente: ein `JLabel`, das ein Bild anzeigen soll, und eine `JTextArea`. Die Komponenten sollen nebeneinander angezeigt werden – und sind doch beide zu groß, um ohne Scrolling auskom-

men zu können. Deshalb werden sie jeweils einer `JScrollPane` zugewiesen, deren bevorzugte Größe auf 200 mal 200 Pixel festgelegt wird:

```
JScrollPane left = new JScrollPane(label);
left.setPreferredSize(new Dimension(200, 200));
```

Das `JSplitPane` kann nun beide Komponenten darstellen. Es benötigt aber auch noch die Information darüber, wie die Darstellung erfolgen soll: horizontal oder vertikal? Dies teilen wir ihm mit Hilfe zweier möglicher Konstanten mit:

- `JSplitPane.HORIZONTAL_SPLIT`: Beide Elemente werden nebeneinander dargestellt
- `JSplitPane.VERTICAL_SPLIT`: Die Elemente werden untereinander angeordnet

Nun können wir eine `JSplitPane`-Instanz, die die enthaltenen Komponenten nebeneinander anzeigt, erzeugen:

```
JSplitPane pane = new JSplitPane(
   JSplitPane.HORIZONTAL_SPLIT, left, right);
```

Nutzer, die diese Applikation ausführen, erhalten folgende Anzeige:

Abbildung 9.5
Anzeige zweier Komponenten per JSplitPane

Mit Hilfe des Splitters zwischen den beiden Komponenten können deren Breiten nun angepasst werden.

Das `JSplitPane` verfügt über eine Eigenschaft, die es erlaubt, das Verhalten des Splitters zu manipulieren: `setOneTouchExpandable()`:

```
public void setOneTouchExpandable(boolean newValue)
```

Weisen wir ihr den Wert `true` zu, wird der Splitter zusätzlich über zwei Pfeile verfügen, mit deren Hilfe die Anzeige der Elemente schnell angepasst werden kann:

Abbildung 9.6
Mit Hilfe der beiden Pfeile kann der Splitter schneller bewegt werden

9.4.4 JTabbedPane

Das `JTabbedPane` erlaubt es, eine Registerkarten-Ansicht zu definieren. Jedem Register (gerne auch Tab genannt) können dabei individuell untergeordnete Elemente zugewiesen werden. Typischerweise werden dies `JPanel`-Instanzen sein, die ihrerseits wieder untergeordnete Komponenten enthalten.

Das Hinzufügen von untergeordneten Elementen geschieht, indem wir sie der Methode `addTab()` der `JTabbedPane`-Klasse als Parameter übergeben:

```
public void addTab(String title,
   Component component)

public void addTab(String title,
   Icon icon,
   Component component)

public void addTab(String title,
   Icon icon,
   Component component,
   String tip)
```

Als zusätzlichen Parameter müssen wir den Titel des Tabs angeben. Optional können wir ein Icon und einen Tooltip-Text festlegen:

Listing 9.8
Hinzufügen von Elementen zu einem JTabbedPane

```
import javax.swing.*;
import java.awt.*;

/**
 * Displays components in a JTabbedPane
 */
public class TabbedPane extends BaseFrame {

   TabbedPane() {
      JTabbedPane tab = new JTabbedPane();
      tab.setPreferredSize(new Dimension(200, 50));

      JPanel panel = new JPanel();
      panel.add(new JLabel("First Tab"));
      tab.addTab("First", panel);

      JPanel second = new JPanel();
      second.add(new JLabel("Second Tab"));
      tab.addTab("Second", second);

      JPanel third = new JPanel();
      third.add(new JLabel("Third Tab"));
      tab.addTab("Third", null, third,
         "Click here to switch to the third tab!");

      add(tab);
      pack();
   }
}
```

Nach dem Erzeugen eines `JTabbedPane` empfiehlt es sich, dessen bevorzugte Anzeige-Maße zu definieren. Unser `JTabbedPane` soll 200 Pixel breit und 50 Pixel hoch sein, was wir mit Hilfe einer `Dimension`-Instanz angeben:

```
JTabbedPane tab = new JTabbedPane();
tab.setPreferredSize(new Dimension(200, 50));
```

Nun können die einzelnen Tabs definiert werden. Zu diesem Zweck wird der Inhalt des Tabs (meist ein `JPanel` oder ein `JScrollPane`) mit Hilfe der Methode `addTab()` unter Angabe des anzuzeigenden Namen des Tabs übergeben:

```
JPanel panel = new JPanel();
panel.add(new JLabel("First Tab"));
tab.addTab("First", panel);
```

Sie sind nicht darauf beschränkt, einen Anzeigenamen und den Inhalt deklarieren zu dürfen. Ebenfalls ist es möglich, ein Icon und einen Tooltip-Text zu übergeben. Wir machen genau dies beim Deklarieren des dritten Tabs und verwenden dabei die Überladung `addTab(Titel, Icon, Komponente, Tooltip-Text)`:

```
tab.addTab("Third", null, third,
   "Click here to switch to the third tab!");
```

Nach dem Zuweisen des Tabs zum übergeordneten Container können wir die Applikation anzeigen:

Abbildung 9.7
Elemente werden in einem JTabbedPane angezeigt

9.4.5 JToolBar

Die `JToolBar` stellt eine Gruppierung von `JButtons` dar, die andockbar und auf Wunsch frei beweglich ist. Zwischen den `JButton`-Instanzen können Trenner platziert werden, um sie besser voneinander unterscheiden zu können.

Der Konstruktor der `JToolBar`-Klasse ist mehrfach überladen:

```
public JToolBar()
public JToolBar(int orientation)
public JToolBar(String name)
public JToolBar(String name, int orientation)
```

Mit Hilfe des Parameters `name` wird der Anzeige-Name der Toolbar definiert, der angezeigt wird, wenn die Toolbar nicht angedockt ist. Der Parameter `orientation` gibt die Ausrichtung der Toolbar an – mögliche Werte sind `JToolBar.HORIZONTAL` (horizontale Ausrichtung, dies ist auch der Standard-Wert, falls der Parameter nicht explizit angegeben wird) und `JToolBar.VERTICAL` (vertikale Ausrichtung).

Der Einsatz der JToolBar-Klasse ist recht einfach, wie das folgende Beispiel demonstriert:

Listing 9.9
Erzeugen einer JToolBar-Instanz mit vier Buttons und einem Separator

```
import javax.swing.*;
import java.awt.*;

/**
 * Displays a JToolBar with four buttons
 */
public class Toolbar extends BaseFrame {

   Toolbar() {
      JPanel content = new JPanel(new BorderLayout());
      add(content);

      JLabel text = new JLabel("This is the content");
      text.setBorder(
         BorderFactory.createEmptyBorder(10, 10, 10, 10));
      content.add(text);

      JToolBar bar = new JToolBar(
         "My Toolbar", JToolBar.HORIZONTAL);
      bar.setRollover(true);
      content.add(bar, BorderLayout.NORTH);

      bar.add(new JButton("One"));
      bar.add(new JButton("Two"));
      bar.addSeparator();
      bar.add(new JButton("Three"));
      bar.add(new JButton("Four"));

      pack();
   }
}
```

Beim Erzeugen teilen wir der JToolBar-Instanz mit, welchen Titel sie tragen soll, wenn sie aus dem Haupt-Fenster herausgezogen wird, und wie sie die enthaltenen Elemente anzeigen soll:

```
JToolBar bar = new JToolBar(
   "My Toolbar", JToolBar.HORIZONTAL);
```

Um ein hübscheres optisches Erscheinungsbild zu erreichen, geben wir an, dass die enthaltenen Buttons Rollover-Effekte zeigen sollen:

```
bar.setRollover(true);
```

Mit Hilfe ihrer Methode `add()` fügen wir der JToolBar-Instanz neue Elemente hinzu:

```
bar.add(new JButton("One"));
```

Um einzelne Elemente voneinander optisch zu trennen, können wir einen Separator per `addSeparator()` einfügen:

```
bar.addSeparator();
```

Wenn wir die Applikation ausführen, werden wir diese Ausgabe erhalten:

Abbildung 9.8
Darstellung von Buttons in einer JToolBar

Nun können wir die `JToolBar` bewegen, indem wir das Element auf der linken Seite mit der Maus ziehen. Wir sind sogar in der Lage, sie aus dem aktuellen Hauptfenster herauszubewegen:

Abbildung 9.9
Die ToolBar wurde aus dem aktuellen Fenster herausbewegt

Um eine `JToolBar` zu fixieren und damit zu verhindern, dass sie bewegt wird, können Sie deren Eigenschaft `floatable` auf `false` setzen:

```
public void setFloatable(boolean b)
```

Damit hätten Sie dann zwar keine typische ToolBar mehr, aber eine hübsche Möglichkeit, Buttons optisch ansprechend zu präsentieren.

9.5 Komponenten

Eine Komponente haben Sie bereits kurz kennen gelernt: das `JLabel`. Dieses erlaubt es uns, einen Text oder ein Bild oder beides auszugeben. `Swing` stellt uns aber noch eine Menge weiterer Komponenten zur Verfügung. Deren Basisklasse ist übrigens `javax.swing.Component`, die ihrerseits indirekt von `java.awt.Component` erbt.

9.5.1 JLabel

Das `JLabel` ist eine einfache Anzeige-Komponente, die Text und/oder ein Bild darstellt. In einem `JLabel` können keine Daten neu erfasst werden, sondern es dient der reinen Ausgabe.

Ein `JLabel` kann sowohl Text als auch Bilder (in Form von `Icon`-Instanzen) anzeigen. Bereits im Konstruktor können wir angeben, was für Informationen dargestellt werden sollen:

```
public JLabel()

public JLabel(String text)

public JLabel(Icon image)

public JLabel(String text, int horizontalAlignment)
```

```
public JLabel(Icon image, int horizontalAlignment)

public JLabel(String text, Icon icon,
   int horizontalAlignment)
```

Der Parameter `horizontalAlignment` gibt an, wo die enthaltenen Elemente dargestellt werden sollen. Folgende Werte stehen dabei zur Auswahl:

- `SwingConstants.LEFT`: Linksbündige Ausrichtung
- `SwingConstants.CENTER`: Zentrierte Ausrichtung
- `SwingConstants.RIGHT`: Rechtsbündige Darstellung
- `SwingConstants.LEADING`: Am Anfang der Zeile
- `SwingConstants.TRAILING`: Am Ende der Zeile

Um ein `JLabel` für die Anzeige eines Textes zu verwenden, können Sie folgenden Code verwenden:

Listing 9.10
Erzeugen einer JLabel-Instanz und Setzen von Text und ToolTip

```
import javax.swing.*;

/**
 * Displays a simple JLabel-instance
 */
public class SimpleLabel extends BaseFrame {

   /**
    * Constructor which creates the JLabel instance and adds it to
    * a JFrame-instance
    */
   SimpleLabel() {
      super();
      JLabel lbl = new JLabel("This is Label.",
         JLabel.CENTER);
      lbl.setToolTipText("I am a JLabel instance");
      lbl.setPreferredSize(new Dimension(350, 100));

      super.add(lbl);
      super.pack();
   }

   public static void main(String[] args) {
      new SimpleLabel();
   }
}
```

Die `JLabel`-Instanz `lbl` wird erzeugt, indem wir dem Konstruktor der Klasse den anzuzeigenden Text und die Ausrichtung des Textes innerhalb des Labels zuweisen:

```
JLabel lbl = new JLabel("This is Label.", JLabel.CENTER);
```

Mit Hilfe der Eigenschaft `setToolTipText()` können wir einen so genannten ToolTip zuweisen, der erscheint, wenn die Maus einige Sekunden über der Komponente nicht bewegt wird.

Bevor wir das Label dem Container zuweisen, legen wir dessen bevorzugte Größe fest. Dies geschieht, indem wir der Eigenschaft `setPreferredSize()` eine neue `java.awt.Dimension`-Instanz mit den entsprechenden Maßen in Pixeln zuweisen:

```
lbl.setPreferredSize(new Dimension(350, 100));
```

Die Zuweisung der `JLabel`-Instanz zum Anzeige-Container erfolgt, indem wir dessen `add()`-Methode als Parameter übergeben:

```
super.add(lbl);
```

Achtung

Das Zuweisen von Komponenten zu Containern per `add()` funktioniert erst seit Java 1.5 so. Vorher mussten Sie die Zuweisung direkt im `ContentPane`-Container durchführen:

```
super.getContentPane().add(lbl);
```

Aufgrund eines Rewritings der entsprechenden Methoden kann nunmehr darauf verzichtet werden und Entwickler sparen ein wenig Schreibaufwand ein.

Wenn Sie die Klasse ausführen, werden Sie folgende Anzeige erhalten:

Abbildung 9.10
Erzeugen einer Label-Instanz samt eines entsprechenden ToolTips

Bild anzeigen

Neben einem Text kann ein `JLabel` auch ein Bild darstellen. Dieses Bild wird in der Regel in Form einer kleinen GIF- oder JPEG-Grafik eingebunden. Diese Grafik wird durch eine Instanz der `ImageIcon`-Klasse repräsentiert:

```
import javax.swing.*;
import java.awt.*;

// ...

Icon icon = new ImageIcon("baby.gif");

JLabel lbl = new JLabel("This is Johannes...", JLabel.CENTER);
lbl.setIcon(icon);
```

Listing 9.11
Anzeigen eines Bildes innerhalb einer JLabel-Instanz

Viel haben wir nicht ändern müssen, um ein Bild in Form einer `ImageIcon`-Instanz anzeigen zu können – wir mussten die Instanz lediglich erzeugen und ihr im Konstruktor den Pfad zum Bild übergeben:

```
Icon icon = new ImageIcon("baby.gif");
```

Achtung

Damit ein Bild wirklich angezeigt werden kann, muss der Pfad selbstverständlich stimmen. In diesem Beispiel haben wir einen relativen Pfad verwendet – dies setzt voraus, dass sich das Bild im Ausführungsverzeichnis der Applikation befindet. Sollte das Bild an einer anderen Stelle gespeichert sein, müssen Sie den Pfad entsprechend angeben, da sonst keine Anzeige erfolgen kann.

Die `JLabel`-Instanz wird wie gewohnt erzeugt. Anschließend weisen wir ihr mit Hilfe ihrer Methode `setIcon()` die `Icon`-Instanz zu:

```
lbl.setIcon(icon);
```

Mehr ist wirklich nicht nötig, um ein Bild anzuzeigen:

Abbildung 9.11
Das ist Johannes ...

Ist er nicht niedlich?

9.5.2 JButton

`JButtons` stellen die einfachsten Schaltflächen innerhalb des `Swing`-Frameworks dar. Sie können Text und/oder kleine Grafiken (`Icons`) darstellen und Events auslösen, an die `ActionListener`-Instanzen gebunden werden können. `JButtons` können aktiviert und deaktiviert werden und den Standard-Button in einem Formular darstellen. `JButtons` unterstützen den Zugriff per Shortcut (`Mnemonik`).

Der Konstruktor der `JButton`-Klasse ist mehrfach überladen und nimmt bei Bedarf sowohl den Anzeigetext als auch das darzustellende Bild entgegen:

```
public JButton()

public JButton(Icon icon)

public JButton(String text)

public JButton(Action a)

public JButton(String text, Icon icon)
```

Nach dem Erzeugen sollte ein `JButton` einem Container zugewiesen werden. Lassen Sie uns im folgenden Beispiel eine `JButton`-Instanz erstellen, die einen Text anzeigt und sonst nichts tut:

Listing 9.12
Erzeugen einer JButton-Instanz

```
import javax.swing.*;

// ...

JButton button = new JButton("Push me!");
button.setSize(100, 40);
button.setLocation(10, 10);

this.add(button);
```

Beim Erzeugen der `JButton`-Instanz haben wir ihr den anzuzeigenden Text als Parameter übergeben. Dieser wird dann sowohl horizontal als auch vertikal zentriert angezeigt:

```
JButton button = new JButton("Push me!");
```

Mit Hilfe seiner Methode `setSize()` können wir die Größe des Buttons in Pixeln festlegen. Um den Button nicht genau links oben anzuzeigen, sondern einen kleinen Rahmen außen herum zu lassen, setzen wir seine Position auf einen Punkt, der zehn Pixel von linken Rand und zehn Pixel vom oberen Rand des Containers liegt, dem der Button zugeordnet sein wird. Dies geschieht mit Hilfe der Methode `setLocation()`:

```
button.setSize(100, 40);
button.setLocation(10, 10);
```

Nun kann der Button zugewiesen und angezeigt werden:

Abbildung 9.12
Eine JButton-Instanz ohne weiterführende Funktionalität

Mnemonic definieren

Mit Hilfe eines `Mnemonic` (in anderen Kulturen auch gerne als Shortcut oder Tastenkürzel bezeichnet) können wir einen Button auch per Tastatur und ohne ⏎-Taste erreichbar machen.

Um ein Tastenkürzel zu setzen, verwenden wir die Methode `setMnemonic()`, die ein `char` oder eine Konstante aus der Klasse `java.awt.KeyEvent` als Parameter erwartet:

```
public void setMnemonic(int mnemonic)
public void setMnemonic(char mnemonic)
```

Die Überladung mit einem `char` als Parameter ist als `deprecated` (also veraltet) gekennzeichnet und sollte nicht mehr verwendet werden. Als Parameter kommen also alle Elemente von `java.awt.KeyEvent` in Frage, die mit dem Präfix *VK_* beginnen. Diese repräsentieren systemunabhängige Tastenkürzel.

Um obigem Button das Tastenkürzel Alt+P zuzuweisen, können wir folgenden Code verwenden:

```
JButton button = new JButton("Push me!");
button.setSize(100, 40);
button.setLocation(10, 10);
button.setMnemonic(KeyEvent.VK_P);

add(button);
```

Das erste Vorkommen des angegebenen Buchstabens wird nun durch einen Unterstrich gekennzeichnet. Mit Hilfe von Alt+P kann der Button betätigt werden:

Abbildung 9.13
Dieser Button kann per Alt+P aktiviert werden

Als Standard-Button festlegen

Wenn in einem Container mehrere Buttons definiert worden sind, kann einer der Buttons als Standard-Button festgelegt werden. Dies geschieht, indem man die Button-Instanz als Parameter an die Methode `setDefaultButton()` des `ContentPane`-Containers des Haupt-Containers übergibt.

```
public void setDefaultButton(JButton defaultButton)
```

Diese Zuweisung kann nach der Erzeugung der JButton-Instanz geschehen:

Listing 9.13
Erzeugen eines Default-Buttons

```
import javax.swing.*;
import java.awt.event.KeyEvent;

// ...

JButton button = new JButton("Push me!");
button.setMnemonic(KeyEvent.VK_P);
panel.add(button);

JButton secondButton = new JButton("Push me, too!");
secondButton.setMnemonic(KeyEvent.VK_M);
panel.add(secondButton);
getRootPane().setDefaultButton(secondButton);

JButton thirdButton = new JButton(
   "You can even push me...");
thirdButton.setMnemonic(KeyEvent.VK_Y);
panel.add(thirdButton);

// ...
```

Alle drei Buttons sind vom Typ `JButton` und bekommen einen Text und ein Tastaturkürzel zugewiesen. Sie werden nach dem Erzeugen dem Anzeige-Container `panel` zugewiesen:

```
JButton button = new JButton("Push me!");
button.setMnemonic(KeyEvent.VK_P);
panel.add(button);
```

Der zweite Button `secondButton` wird als Default-Button gesetzt. Dies geschieht, indem wir diesen Button der Methode `setDefaultButton()` eines `JRootPane`-Containers als Parameter übergeben. Über einen derartigen `JRootPane`-Container verfügt beispielsweise das `JPanel`:

```
getRootPane().setDefaultButton(secondButton);
```

Die `JPanel`-Instanz `panel` wird am Ende der Initialisierung dem Top-Level-Container zugewiesen. Damit können dann auch die drei `JButton`-Instanzen angezeigt werden:

Abbildung 9.14
Der zweite Button ist als Default-Button deklariert

Der zweite Button ist als Default-Button deklariert. Er kann nun durch einfaches Drücken der ↵-Taste ausgelöst werden.

Auf JButton-Events reagieren

`JButtons` können Ereignisse werfen. Sie werden dies spätestens dann tun, wenn sie angeklickt oder per ↵ ausgelöst worden sind. Um darauf reagieren zu können, müssen sich interessierte Klassen an die Events binden. Dies geschieht mit Hilfe von so genannten `ActionListener`-Instanzen, die das gleichnamige Interface implementieren:

```
void actionPerformed(ActionEvent e)
```

Die einzige Methode des `ActionListener`-Interface ist die Methode `actionPerformed()`, die als Parameter über eine `ActionEvent`-Instanz verfügt. Diese `ActionEvent`-Instanz erlaubt es uns, auf die Klasse zurückzugreifen, die das Event ursprünglich geworfen hat. Ebenfalls führt sie weitere Informationen mit sich, die wir auswerten können.

Um auf ein Event einer `JButton`-Instanz reagieren zu können, muss die `ActionListener`-Instanz an den Button mit Hilfe seiner Methode `addActionListener()` gebunden werden. Die so angemeldete `ActionListener`-Instanz wird dann eingebunden, wenn der Button aktiviert worden ist, und kann die Reaktion auf das Event vornehmen:

Listing 9.14
Reagieren auf ein Ereignis eines JButtons

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.KeyEvent;
import java.awt.event.ActionListener;
import java.awt.*;

/**
 * Catching the Click on a JButton
 */
public class CatchAnAction extends BaseFrame
   implements ActionListener {

   JButton button;
   JLabel label;

   /**
    * Creates a Button and a Label,
    * registers the actionPerformed-method with the Button
    */
   CatchAnAction() {

      // ...

      button = new JButton("Click me!");
      button.setMnemonic(KeyEvent.VK_C );
      button.addActionListener(this);
      panel.add(button, FlowLayout.CENTER);

      this.add(panel);
   }

   public void actionPerformed(ActionEvent e) {
      button.setEnabled(false);
      label.setText("Button was clicked!");
   }
}
```

Die Klasse `CatchAnAction` implementiert die Schnittstelle `ActionListener`. Dies erlaubt es ihr, sich als Event-Handler für die Ereignisse ihrer `JButton`-Instanz zu registrieren.

Die `JButton`-Instanz wird mit dem Text „Click me!“ und dem Shortcut Alt+C definiert:

```
button = new JButton("Click me!");
button.setMnemonic(KeyEvent.VK_C );
```

Nun wird die aktuelle Instanz der Klasse als `ActionListener` für den `JButton` registriert. Dies erlaubt es uns später, die Events innerhalb der Instanz zu behandeln und auf die hier deklarierten Komponenten zuzugreifen. Die Registrierung als `ActionListener` für den Button geschieht, indem wir die aktuelle Klasseninstanz, die durch das Schlüsselwort `this` repräsentiert wird, der Methode `addActionListener()` als Argument übergeben:

```
button.addActionListener(this);
```

Die Reaktion auf das `action`-Ereignis des Buttons findet in der Methode `actionPerformed()` statt, in der wir den Anzeige-Text des Labels ändern und den Button deaktivieren.

Das war es! Mehr ist nicht nötig, um auf ein `ActionEvent` eines Buttons reagieren zu können:

- Wir müssen die Methode `actionPerformed()` des `ActionListener`-Interfaces implementieren.
- Anschließend müssen wir die Klasse, die diese Methode implementiert, an das action-Ereignis des Buttons mit Hilfe von seiner Methode `addActionListener()` binden.

Wenn wir die Klasse kompilieren und ausführen, werden wir folgende Anzeige erhalten:

Abbildung 9.15
Der Button ist nicht angeklickt worden

Klicken Sie nun auf den Button oder drücken Sie die Tastenkombination Alt+C:

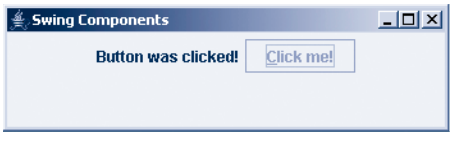


Abbildung 9.16
Der Button ist angeklickt worden

Ermitteln, welcher Button angeklickt worden ist

Wenn Sie mehrere Buttons auf einem Formular verwenden wollen, können Sie für diese jeweils eigene `ActionListener`-Instanzen verwenden. Oft reicht es aber auch aus, eine Instanz die Behandlung aller `action`-Ereignisse übernehmen zu lassen, denn deren `ActionEvent`-Parameter liefert uns alle Informationen, die wir benötigen:

```
public Object getSource()
public String getActionCommand()
```

Die Methode `getSource()` gibt uns die Quelle des Ereignisses zurück – also in der Regel die `JButton`-Instanz, die das Ereignis geworfen hat. Mit Hilfe von `getActionCommand()` können wir den Inhalt des `actionCommand`-Attributs des Buttons ermitteln und verarbeiten.

Sehen wir uns an, wie dies von sich gehen könnte:

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.KeyEvent;
import java.awt.event.ActionListener;
import java.awt.*;
```

Listing 9.15
Reaktion auf die Action-Events mehrerer Buttons

Listing 9.15 *(Forts.)*
Reaktion auf die Action-Events mehrerer Buttons

```
// ...

JButton button = new JButton("Click me!");
button.setMnemonic(KeyEvent.VK_C );
button.setActionCommand("first Button");
button.addActionListener(this);

// ...

public void actionPerformed(ActionEvent e) {
   if(e.getSource() instanceof JButton) {
      JButton button = (JButton)e.getSource();
      button.setEnabled(false);
      label.setText(
         String.format("The %s was clicked!",
            e.getActionCommand()));
   }
}
```

Das Zauberwort, mit dessen Hilfe wir mehrere Buttons auseinander halten und jedem dieser Buttons möglicherweise eigene Aktions-Parameter mitgeben können, nennt sich `actionCommand`.

Dieses `actionCommand` ist nichts anderes als eine Zeichenkette, die über die `ActionEvent`-Klasse an den Event-Handler übertragen wird. Das `actionCommand` eines Buttons setzen wir mit Hilfe von dessen `setActionCommand()`-Methode:

```
public void setActionCommand(String actionCommand)
```

Die übergebene Zeichenkette kann dabei völlig beliebig sein – sie dient nur dazu, die einzelnen Buttons mit deren Aktionen zu unterscheiden.

Wenn wir also eine `JButton`-Instanz erzeugen, können wir dieser wie gehabt einen Anzeigetext, ein `Mnemonik` und nunmehr auch ein `actionCommand` zuweisen:

```
JButton button = new JButton("Click me!");
button.setMnemonic(KeyEvent.VK_C);
button.setActionCommand("first Button");
```

Anschließend binden wir den `ActionListener` an den Button:

```
button.addActionListener(this);
```

Wenn nun ein action-Event ausgelöst worden ist, wird die `ActionListener`-Methode `actionPerformed()` aufgerufen, der als Parameter die `ActionEvent`-Instanz `e` übergeben wird.

Mit Hilfe der Methode `getSource()` der `ActionEvent`-Instanz können wir feststellen, welche Komponente das Event ausgelöst hat. Die Rückgabe von `getSource()` ist eine `Object`-Instanz. Diese müssen wir wieder in einen `JButton` casten, wobei wir zuvor überprüfen sollten, ob dies tatsächlich möglich ist:

```
if(e.getSource() instanceof JButton) {
   JButton button = (JButton)e.getSource();
   // ...
}
```

Nun kann ermittelt werden, welche `JButton`-Instanz eigentlich angeklickt worden ist. Dies geschieht, indem wir die `actionCommand`-Eigenschaft des `ActionEvents` auslesen und deren Inhalt ausgeben:

```
label.setText(
   String.format("The %s was clicked!",
      e.getActionCommand()));
```

Damit ist die Behandlung des Ereignisses abgeschlossen.

Führen wir nun die Klasse aus:

Abbildung 9.17
Noch wurde keiner der Buttons angeklickt

Wenn wir nun einen der Buttons aktivieren, werden wir sehen, dass er deaktiviert und sein Name im Label angezeigt werden wird:

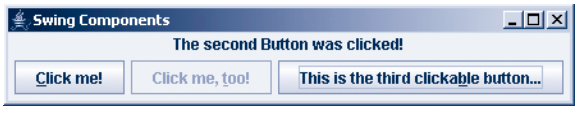


Abbildung 9.18
Ein Button wurde angeklickt

Zuweisen eines Icons zu einem Button

Statt eines Textes können Sie einen `JButton` ebenfalls ein kleines Bild anzeigen lassen. Sie können alternativ sogar Text und Bild kombinieren.

Die einfachste Möglichkeit, einem Button mitzuteilen, dass er ein Bild anzeigen soll, besteht darin, diese Information gleich im Konstruktor zu geben. Die Klasse `JButton` besitzt zwei Konstruktoren, die für uns in diesem Zusammenhang interessant sind:

```
public JButton(Icon icon)
public JButton(String text, Icon icon)
```

Die erste Überladung mit einer `Icon`-Instanz sorgt dafür, dass das durch die `Icon`-Instanz repräsentierte Bild innerhalb der Schaltfläche angezeigt wird. Die zweite Überladung nimmt zusätzlich noch einen String entgegen und zeigt diesen zusammen mit dem Bild an:

```
JPanel contents = new JPanel();
Icon icon = new ImageIcon("baby.gif");

JButton firstButton = new JButton(icon);
contents.add(firstButton);

JButton secondButton = new JButton("This is Johannes...", icon);
contents.add(secondButton);

this.add(contents);
```

Hier deklarieren wir eine `Icon`-Instanz vom Typ `ImageIcon`, die auf das Bild „baby.gif“ im Ausführungs-Verzeichnis der Klasse verweist. Diese `Icon`-Instanz kann nun den Konstruktoren der beiden Buttons als Argument übergeben werden. Der zweite Button erhält zusätzlich noch einen Anzeige-Text.

Beim Ausführen erzeugt dies folgende Ausgabe:

Abbildung 9.19
Zwei Buttons mit Bildern

Sie können das `Icon` auch nachträglich setzen – ein `JButton` verfügt über die Methode `setIcon()`, die eine `Icon`-Instanz entgegennimmt. Darüber hinaus haben Sie die Möglichkeit, weitere Icons zuzuweisen. Dies kann mit Hilfe folgender Methoden geschehen:

- `setDisabledIcon(Icon)`: Setzt das Icon, das angezeigt wird, wenn der Button deaktiviert ist.
- `setPressedIcon(Icon)`: Dieses Icon wird angezeigt, während gerade auf den Button geklickt wird.
- `setRolloverIcon(Icon)`: Wird angezeigt, wenn mit der Maus über den Button gefahren wird.
- `setSelectedIcon(Icon)`: Wird angezeigt, wenn der Button selektiert, aber nicht angeklickt ist.

9.5.3 JCheckBox

Die `JCheckBox`-Klasse baut auf der gleichen Basis-Klasse wie die `JButton`-Klasse auf. Aus diesem Grund haben beide recht ähnliche Methoden und der Einstieg in die Arbeit mit der `JCheckBox`-Klasse dürfte leichter fallen.

`JCheckBoxen` erlauben das An- und Abwählen von Optionen. Als Wert können die `JCheckBox`-Instanzen entweder `true` oder `false` haben – je nachdem, ob sie an- oder abgewählt waren.

Der Konstruktor der `JCheckBox`-Klasse ist mehrfach überladen:

```
public JCheckBox()
public JCheckBox(Icon icon)
public JCheckBox(String text)
public JCheckBox(Icon icon, boolean selected)
public JCheckBox(String text, boolean selected)
public JCheckBox(String text, Icon icon)
public JCheckBox(String text, Icon icon, boolean selected)
```

Wir können somit beim Erzeugen einer neuen `JCheckBox`-Instanz bereits den Anzeige-Text oder ein Icon festlegen. Ebenso haben wir die Möglichkeit, anzugeben, ob die CheckBox initial aktiviert worden ist:

Listing 9.16
Erstellen von JCheckBox-Instanzen

```
import javax.swing.*;
import java.awt.event.KeyEvent;

// ...

JCheckBox box = new JCheckBox("Check me!");
box.setMnemonic(KeyEvent.VK_C);
panel.add(box);

JCheckBox secondBox =
   new JCheckBox("Uncheck me!", true);
secondBox.setMnemonic(KeyEvent.VK_U);
panel.add(secondBox);
```

`JCheckBoxen` können ebenso wie `JButtons` mit einem Tastatur-Kürzel versehen werden. Unsere beiden CheckBoxen definieren als Kürzel Alt+C und Alt+U. Das Ausführen dieser Tasten-Kombinationen hat denselben Effekt, als ob die CheckBox aktiviert worden wäre.

Wenn Sie das Beispiel ausführen, werden Sie zwei CheckBoxen sehen, die Sie an- und abwählen können. Beim Start präsentiert sich Ihnen dieses Bild:

Abbildung 9.20
Zwei CheckBoxen, die von Ihnen an- und abgewählt werden können

Reaktion auf die Events der CheckBoxen

Wenn `JCheckBox`-Instanzen an- oder abgewählt werden, werfen Sie ein `item`-Ereignis, an das man sich mit Hilfe einer `ItemListener`-Instanz binden kann.

Zu diesem Zweck muss die Klasse, die das Ereignis behandeln soll, die Methode `itemStateChanged()` implementieren, der als Parameter eine Instanz der `ItemEvent`-Klasse übergeben wird:

```
void itemStateChanged(ItemEvent e)
```

Um auf das Aktivieren oder Deaktivieren einer `JCheckBox`-Instanz reagieren zu können, wäre beispielsweise folgender Code einsetzbar:

Listing 9.17
Reaktion auf das ItemEvent einer JCheckBox

```
import javax.swing.*;
import javax.swing.border.Border;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import java.awt.event.KeyEvent;
import java.awt.*;

/**
 * Catches the item-event of a CheckBox
 */
public class CatchAnItemEvent extends BaseFrame
   implements ItemListener {

   JLabel label;
```

Listing 9.17 *(Forts.)*
Reaktion auf das ItemEvent einer JCheckBox

```
    /**
     * Defines a JCheckBox and a JLabel, binds the ItemListener
     */
    CatchAnItemEvent() {
       // ...

       JCheckBox box = new JCheckBox("Check me, please!");
       box.setHorizontalAlignment(SwingConstants.CENTER);
       box.setMnemonic(KeyEvent.VK_C);
       box.addItemListener(this);

       // ...
    }

    /**
     * Catches the item-event of a JCheckBox
     */
    public void itemStateChanged(ItemEvent e) {
       switch(e.getStateChange()) {
          case(ItemEvent.SELECTED):
             label.setText("CheckBox is selected.");
             break;
          case(ItemEvent.DESELECTED):
             label.setText("CheckBox is not selected.");
             break;
          default:
             label.setText("Unknown state of CheckBox");
       }
    }
}
```

Die Klasse `CatchAnItemEvent` ist selbstständig in der Lage, `ItemEvents` zu verarbeiten, da sie das Interface `ItemListener` implementiert. Die Bindung an das `item`-Ereignis der CheckBox geschieht, indem wir eine Referenz auf die Klasse an deren Methode `addItemListener()` übergeben:

```
box.addItemListener(this);
```

Innerhalb der Methode `itemStateChanged()` ermitteln wir mit Hilfe der Methode `getStateChange()` der als Parameter übergebenen `ItemEvent`-Instanz den Status der CheckBox. Diese kann nur zwei Stati haben:

- `ItemEvent.SELECTED`: Die CheckBox ist angewählt.
- `ItemEvent.DESELECTED`: Die CheckBox ist nicht angewählt.

Diese Informationen können wir verarbeiten und beispielsweise einen entsprechenden Hinweistext ausgeben:

```
switch(e.getStateChange()) {
   case(ItemEvent.SELECTED):
      label.setText("CheckBox is selected.");
      break;
   case(ItemEvent.DESELECTED):
      label.setText("CheckBox is not selected.");
      break;
   default:
      label.setText("Unknown state of CheckBox");
}
```

Wenn wir die Applikation aufrufen, erhalten wir diese Ausgabe:

Abbildung 9.21
Initialer Status der CheckBox

Sobald die CheckBox angewählt worden ist, ändert sich der Text im Label:

Abbildung 9.22
CheckBox ist angewählt worden

CheckBox mit Icon

Einer `JCheckBox`-Instanz kann ebenso wie einem `JButton` ein `Icon` statt oder zusammen mit einem Text zugewiesen werden. Interessanter als dieser Fall ist aber, dass einer `JCheckBox` ebenfalls Icons für ihre Zustände zugewiesen werden können:

```
public void setIcon(Icon defaultIcon)
public void setSelectedIcon(Icon selectedIcon)
public void setPressedIcon(Icon pressedIcon)
public void setRolloverIcon(Icon rolloverIcon)
public void setRolloverSelectedIcon(
   Icon rolloverSelectedIcon)
public void setDisabledIcon(Icon disabledIcon)
public void setDisabledSelectedIcon(
   Icon disabledSelectedIcon)
```

Listing 9.18
Anzeigen der Stati einer CheckBox mit eigenen Icons

Die Icons sollten in Form von `ImageIcon`-Instanzen übergeben werden:

```
Icon defaultIcon = new ImageIcon("decline.gif");
Icon selectedIcon = new ImageIcon("accept.gif");

JCheckBox box = new JCheckBox("Do you accept it?");
box.setIcon(defaultIcon);
box.setSelectedIcon(selectedIcon);
```

Um den Status einer CheckBox ausdrücken zu können, müssen wir zwei `Icon`-Instanzen erzeugen, denn die CheckBox kennt zwei Zustände: Angewählt und nicht angewählt. Letzteres ist der Standard-Zustand. Die Icons werden durch `ImageIcon`-Instanzen repräsentiert, die in diesem Fall auf zwei Bilder verweisen, die im selben Verzeichnis wie die Applikation liegen:

```
Icon defaultIcon = new ImageIcon("decline.gif");
Icon selectedIcon = new ImageIcon("accept.gif");
```

Die beiden Icons setzen wir mit Hilfe der `setIcon()`- und `setSelectedIcon()`-Methoden der `JCheckBox`-Instanz:

```
box.setIcon(defaultIcon);
box.setSelectedIcon(selectedIcon);
```

Sonst ändert sich an der Funktionsweise der CheckBox nichts – es spielt also keine Rolle, welche Art der Darstellung Sie wünschen.

Wenn wir nun die Klasse aufrufen, sehen wir, dass die CheckBox nicht angewählt worden ist:

Abbildung 9.23
Die CheckBox ist nicht aktiviert

Wenn der Nutzer nun die CheckBox aktiviert, wechselt das Icon:

Abbildung 9.24
CheckBox ist aktiviert

9.5.4 JRadioButton

`JRadioButtons` sind `JButton`-Instanzen, die analog zu `JCheckBoxen` an- oder abgewählt sein können. `JRadioButtons` werden in Gruppen – den `ButtonGroups` – organisiert. Innerhalb einer Gruppe kann es nur einen angewählten `JRadioButton` geben.

`ButtonGroups` werden nicht sichtbar dargestellt. Sie dienen ausschließlich als logischer Container für `JRadioButton`-Instanzen.

Der Konstruktor einer `JRadioButton`-Instanz kann neben einer `Icon`-Instanz auch Text und einen booleschen Wert entgegennehmen:

```
public JRadioButton()
public JRadioButton(Icon icon)
public JRadioButton(Icon icon, boolean selected)
public JRadioButton(String text)
public JRadioButton(String text,boolean selected)
public JRadioButton(String text, Icon icon)
public JRadioButton(String text, Icon icon, boolean selected)
```

Der boolesche Parameter `selected` gibt Auskunft über den initialen Status des Buttons – ist der Wert `true`, dann ist der Button angewählt, anderenfalls ist er abgewählt:

Listing 9.19
Erzeugen von RadioButtons

```
import javax.swing.*;
import java.awt.*;

/**
 * Creates two RadioButtons
 */
public class SimpleRadioButtons extends BaseFrame {

   /**
    * Defines the RadioButtons and adds them to a ButtonGroup
    */
```

```
   SimpleRadioButtons() {
      ButtonGroup group = new ButtonGroup();

      // ...

      JRadioButton button = new JRadioButton(
         "First radio button", true);
      group.add(button);
      contents.add(button);

      JRadioButton secondButton = new JRadioButton(
         "Second radio button", true);
      group.add(secondButton);
      contents.add(secondButton);

      pack();
   }
}
```

Listing 9.19 *(Forts.)*
Erzeugen von RadioButtons

Da `JRadioButton`-Instanzen immer einer `ButtonGroup` zugewiesen sein sollen, erzeugen wir zunächst die `ButtonGroup`-Instanz:

```
ButtonGroup group = new ButtonGroup();
```

Nun können wir die Buttons erzeugen. Direkt nach ihrer Erzeugung wird jede `JRadioButton`-Instanz der Gruppe zugewiesen – diese sorgt später dafür, dass wirklich immer nur ein Button der Gruppe angewählt ist:

```
JRadioButton button = new JRadioButton(
   "First radio button", true);
group.add(button);
```

Nun können wir die Applikation ausführen:

Abbildung 9.25
RadioButtons im Einsatz

Fällt Ihnen etwas auf? Falls nicht, werfen Sie noch einmal einen Blick auf obigen Code – und dort insbesondere bei der Erzeugung des zweiten Buttons:

```
JRadioButton secondButton = new JRadioButton(
   "Second radio button", true);
```

Tatsächlich ist dieser Button – genau wie der erste Button – als angewählt gekennzeichnet worden. Bei der Anzeige ist er es aber nicht – dies ist der Verdienst der `ButtonGroup`.

Reaktion auf Events von RadioButtons

`JRadioButton`-Instanzen werfen ebenso wie CheckBoxen ein `item`-Event, auf das reagiert werden kann. Im Gegensatz zu CheckBoxen scheint dies aber bei RadioButtons nicht sinnvoll zu sein – das `item`-Ereignis, das die Zustandsän-

derung des JRadioButtons symbolisiert, würde zweimal geworfen werden: einmal für die JRadioButton-Instanz, die abgewählt worden ist, und einmal für den RadioButton, der neu angewählt wurde.

Deshalb ist es sinnvoller (und allgemein auch üblicher) auf das action-Ereignis des angeklickten Buttons zu reagieren. Dies bedeutet, dass wir nur noch ein Ereignis behandeln und dabei keine Fälle unterscheiden müssen. Sehr wohl aber müssen wir in der Lage sein, einzelne JRadioButtons mit ihrem Wert zu identifizieren. Dazu bietet sich die Methode setActionCommand() an, der wir eine Zeichenkette übergeben können, die den Button oder den durch ihn repräsentierten Wert identifiziert:

Listing 9.20
Reagieren auf die action-Ereignisse von JRadioButton-Instanzen

```
import javax.swing.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.*;

/**
 * Handles the action event of RadioButtons
 */
public class RadioButtonActionEvents extends BaseFrame
   implements ActionListener {

   JLabel label;

   /**
    * Creates the buttons, the group and a Label
    */
   RadioButtonActionEvents() {

      // ...

      ButtonGroup group = new ButtonGroup();

      JRadioButton male = new JRadioButton("Male");
      male.setActionCommand("Male");
      male.addActionListener(this);
      group.add(male);
      buttons.add(male);

      JRadioButton female = new JRadioButton("Female");
      female.setActionCommand("Female");
      female.addActionListener(this);
      group.add(female);
      buttons.add(female);

      JRadioButton dontknow = new JRadioButton("Don't know");
      dontknow.setActionCommand("Don't know");
      dontknow.addActionListener(this);
      group.add(dontknow);
      buttons.add(dontknow);

      // ...
   }
```

```
    /**
     * Handles the button's action events
     */
    public void actionPerformed(ActionEvent e) {
       String message = "Your selection: %s";
       label.setText(
          String.format(message,
             e.getActionCommand()));
    }
}
```

Listing 9.20 *(Forts.)*
Reagieren auf die action-Ereignisse von JRadioButton-Instanzen

Damit die Buttons das gewünschte Verhalten zeigen und nur ein Button angewählt sein kann, müssen wir sie alle derselben `ButtonGroup` zuordnen. Diese Gruppe wird deshalb noch vor den Buttons erzeugt:

```
ButtonGroup group = new ButtonGroup();
```

Nun können die einzelnen Buttons erzeugt werden. Zu diesem Zweck übergeben wir ihrem Konstruktor den anzuzeigenden Text und setzen mit Hilfe ihrer Methode `setActionCommand()` den Wert, den der Button repräsentieren soll:

```
JRadioButton male = new JRadioButton("Male");
male.setActionCommand("Male");
```

Jetzt kann der Event-Handler für das `action`-Ereignis zugewiesen werden. Dies ist die aktuelle Klasse – aus diesem Grund implementiert sie auch das Interface `ActionListener` und dessen Methode `actionPerformed()`. Mit Hilfe des Schlüsselworts `this` referenzieren wir die aktuelle Klasseninstanz und übergeben sie an die Methode `addActionListener()` der `JRadioButton`-Instanz:

```
male.addActionListener(this);
```

Es empfiehlt sich, zusammengehörige `JRadioButton`-Instanzen innerhalb eines `JPanel`-Containers zusammenzufassen. Dies erlaubt eine geordnete Darstellung der Buttons und sorgt für mehr Übersicht. Eine `JPanel`-Instanz mit zugeordneten `JRadioButtons` wird auch gerne `ButtonPanel` genannt.

Selbstverständlich dürfen wir nicht vergessen, den RadioButton seiner ButtonGruppe zuzuordnen:

```
group.add(male);
```

Sollte einer der Buttons angeklickt werden, wird sein `action`-Ereignis geworfen und von der Methode `actionPerformed()` behandelt. Hier wird der Wert des Buttons mit Hilfe der Methode `getActionCommand()` der `ActionEvent`-Instanz ermittelt und ausgegeben.

Wenn Sie die Applikation aufrufen, werden Sie feststellen können, dass noch keine Auswahl getroffen wurde:

Abbildung 9.26
Leider wurde noch keine Auswahl getroffen

Wenn einer der RadioButtons angeklickt worden ist, kann mit Hilfe der Methode `actionPerformed()` und des Wertes seines `actionCommands` der Wert des Buttons ausgelesen und angezeigt werden:

Abbildung 9.27
Jetzt wurde ein Wert ermittelt

9.5.5 JComboBox

Die `JComboBox` stellt ein Pulldown-Menü dar. Sie kann mehrere Werte platzsparend präsentieren und erlaubt eine intuitive Auswahl.

Der Konstruktor einer `JComboBox` ist mehrfach überladen:

```
public JComboBox()
public JComboBox(Object[] items)
public JComboBox(Vector items)
```

Sie können einer `JComboBox`-Instanz in ihrem Konstruktor unter anderem ein Array an Werten als Parameter übergeben – die Box wird dann mit diesen Werten befüllt und erlaubt eine Auswahl daraus:

Listing 9.21
Erzeugen und Anzeigen einer JComboBox-Instanz

```
import javax.swing.*;
import java.awt.*;

/**
 * Creates and displays a simple read-only ComboBox
 */
public class SimpleComboBox extends BaseFrame {

   /**
    * Creates the JComboBox-instance
    */
   SimpleComboBox() {

      // ...

      String[] values =
         new String[] {"One", "Two", "Three", "Four", "Five"};
      JComboBox box = new JComboBox(values);
      contents.add(box);

      // ...
   }
}
```

Nach dem Erzeugen der JComboxBox-Instanz und dem Zuweisen der Werte kann diese angezeigt werden und erlaubt eine platzsparende Auswahl aus mehreren Werten:

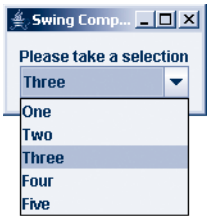


Abbildung 9.28
JComboBox in Aktion

JComboBox editierbar machen

Die JComboBox kann sowohl in einem schreibgeschützten als auch einem editierbaren Modus eingesetzt werden. Der Standard-Modus ist schreibgeschützt.

Der Sinn des editierbaren Modus einer ComboBox besteht darin, den Benutzer nicht nur auf die vorgegebenen Listen-Elemente zu beschränken.

Um den editierbaren Modus einzuschalten, müssen wir der Methode setEditable() einer JComboBox-Instanz den Wert true übergeben:

```
String[] values = new String[]
   {"One", "Two", "Three", "Four", "Five"} ;
JComboBox box = new JComboBox(values);
box.setEditable(true);
```

Listing 9.22
Einschalten des Editier-Modus einer ComboBox

Wenn mit Hilfe von setEditable() eine ComboBox in den Editier-Modus versetzt worden ist, kann der Nutzer einen eigenen Wert erfassen oder alternativ einen der vorgegebenen Werte nutzen:

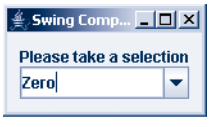


Abbildung 9.29
Hier können eigene Werte definiert werden

Reaktion auf Events der ComboBox

Die JComboBox unterstützt das action-Ereignis. Dieses Ereignis wird geworfen, wenn eine neue Auswahl in der ComboBox getroffen worden ist.

Das bedeutet, dass Klassen, die auf die Ereignisse einer JComboBox-Instanz reagieren wollen, das Interface ActionListener implementieren müssen. Die Reaktion auf das action-Event der JComboBox findet dann in der Methode actionPerformed() des ActionListeners statt:

```
import javax.swing.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.*;

/**
 * Creates a JComboBox whose action-event is handled
```

Listing 9.23
Reaktion auf das action-Event einer ComboBox

Listing 9.23 *(Forts.)* Reaktion auf das action-Event einer ComboBox

```
 */
public class ComboBoxActionEvent extends BaseFrame
   implements ActionListener {

   JLabel result;

   /**
    * Creates the JComboBox-instance and adds its event-handler
    */
   ComboBoxActionEvent() {

      // ...

      String[] values = new String[]
         {"One", "Two", "Three", "Four", "Five"};
      JComboBox box = new JComboBox(values);
      box.setEditable(true);
      box.addActionListener(this);

      // ...

   }

   /**
    * Handles the action event thrown by the JComboBox-instance
    */
   public void actionPerformed(ActionEvent e) {
      if(e.getSource() instanceof JComboBox) {
         JComboBox item = (JComboBox)e.getSource();
         String value = null;
         if(item.getSelectedIndex() > -1) {
            value = (String)item.getSelectedItem();
         } else {
            value = item.getEditor().getItem().toString();
         }

         this.result.setText(
            String.format("Your selection: %s", value));
      }
   }
}
```

Um auf das `action`-Ereignis unserer `JComboBox`-Instanz reagieren zu können, benötigen wir eine `ActionListener`-Instanz. Die aktuelle Klasse implementiert deshalb das Interface `ActionListener`.

Um der ComboBox, die übrigens editierbar ist, einen Handler für deren `action`-Ereignis zuweisen zu können, müssen wir ihrer `addActionListener()`-Methode eine `ActionListener`-Instanz als Parameter übergeben. Da dies die Instanz der aktuellen Klasse ist, referenzieren wir diese mit Hilfe des Schlüsselworts `this`:

```
box.addActionListener(this);
```

Das `action`-Event der ComboBox kann nun auf zwei Wegen ausgelöst werden:

- Ein vorhandener Eintrag aus der Liste wird angeklickt
- Ein neuer Wert wird eingegeben und die ↵-Taste wird gedrückt

Beide Aktionen resultieren in einem `action`-Ereignis, das in der Methode `actionPerformed()` abgefangen wird. Dort können wir zunächst überprüfen, ob der Auslöser des Ereignisses tatsächlich eine `JComboBox`-Instanz ist. Diese Prüfung nehmen wir mit Hilfe des Schlüsselworts `instanceof` vor:

```
if(e.getSource() instanceof JComboBox) { ... }
```

Nachdem wir so verifiziert haben, dass das `action`-Event von einer `JComboBox`-Instanz geworfen worden ist, können wir eine lokale Referenz auf den Auslöser des Ereignisses erzeugen. Logisch, dass diese Referenz vom Typ `JComboBox` sein sollte:

```
JComboBox item = (JComboBox)e.getSource();
```

Nun geraten wir in schwierigeres Fahrwasser: Wir müssen unterscheiden, ob ein vorhandenes Element der Liste ausgewählt worden ist oder ob in einer editierbaren `JComboBox`-Instanz ein neuer Wert eingegeben und anschließend ↵ gedrückt worden ist.

Aber eigentlich ist diese Unterscheidung nicht schwierig. Sie geschieht anhand des Wertes, den uns die Methode `getSelectedIndex()` der `JComboBox`-Instanz liefert:

- Ist der Wert größer oder gleich null, dann wurde ein Element aus der Liste ausgewählt
- Ist der Wert gleich -1, dann wurde ein neuer Begriff in das Feld eingegeben und ↵ gedrückt

Sollte also die Rückgabe von `getSelectedIndex()` größer als -1 sein, können wir davon ausgehen, dass ein Element aus der zugrunde liegenden Liste ausgewählt worden ist. Den Wert dieses Elements ermitteln wir mit Hilfe der Methode `getSelectedItem()`, deren Rückgabe wir allerdings noch explizit in einen `String` casten müssen:

```
if(item.getSelectedIndex() > -1) {
   value = (String)item.getSelectedItem();
}
```

Sollte der Auswahlindex den Wert -1 haben, dann ist ein Text eingegeben worden. In diesem Fall können wir mit Hilfe von `getEditor().getItem().toString()` den im Editor eingegebenen Wert abrufen:

```
else {
   value = item.getEditor().getItem().toString();
}
```

Wenn Sie die Klasse ausführen, könnten Sie zunächst einen Wert aus der Liste auswählen:

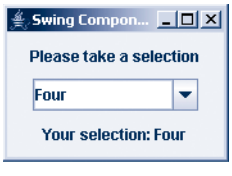
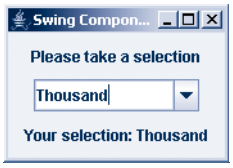


Abbildung 9.30
Ein Wert aus der Liste ist ausgewählt worden

Wenn Ihnen die Einträge in der Auswahlliste nicht zusagen, können Sie selbstverständlich auch einen eigenen Wert eingeben:

Abbildung 9.31
Eingabe und Anzeige eines eigenen Werts



9.5.6 JTextField

Das `JTextField` hat die Aufgabe, ein Eingabefeld für Texteingaben bereitzustellen. Der Nutzer kann hier einen Text eingeben und editieren.

Der Konstruktor der `JTextField`-Klasse ist überladen:

```
public JTextField()
public JTextField(String text)
public JTextField(int columns)
public JTextField(String text, int columns)
public JTextField(Document doc, String text, int columns)
```

Einer `JTextField`-Instanz können somit schon beim Erzeugen ein initialer Text und/oder eine maximale Breite zugewiesen werden:

```
JTextField text = new JTextField("Here you may enter some text!");
```

Eine derartige erzeugte `JTextField`-Instanz erlaubt die Eingabe eines Textes in einer standardisierten Anzeige-Breite:

Abbildung 9.32
Eingabe von Text in einem JTextField

Um die Breite einer `JTextField`-Instanz zu setzen, können Sie deren Eigenschaft `setColumns()` verwenden oder ihr im Konstruktor die Anzahl der darzustellenden Zeichen als Parameter übergeben:

```
JTextField text =
   new JTextField("Here you may enter some text!", 10);
```

Achtung

Die Angabe der Breite eines `JTextFields` erfolgt dabei nicht in Pixeln, sondern in Zeichen. Das Text-Feld rechnet diese dann in einer Breite basierend auf der aktuellen Schriftart und deren Schriftgröße um. Die Berechnung der Breite basiert darauf, wie breit der Buchstabe `m` dargestellt werden würde.

Um die dargestellte Schriftart zu ändern, können Sie die Methode `setFont()` der `JTextField`-Klasse verwenden. Diese erwartet eine Instanz der `Font`-Klasse als Parameter, die Auskunft darüber gibt, welche Schriftart mit welchem Stil und welcher Zeichengröße verwendet werden soll:

```
JTextField text = new JTextField("Here you may enter some text!");
text.setFont(new Font("Sans-Serif", Font.BOLD, 20));
```

Sollten wir also die Schriftart auf einen serifenlosen Zeichensatz mit einer Größe von 20 Pixeln in fettem Druck gesetzt haben, würde sich das Text-Feld etwa so darstellen:

Abbildung 9.33
Hier wurde die Schriftgröße deutlich erhöht

Achtung

Sie sollten Schrift-Angaben immer nur unter Angabe von generischen Schrifttypen machen – also „Serif“ für eine Schrift im Stile von Times und „Sans-Serif“ für eine serifenlose Schrift, wie etwa Arial, Helvetica oder Verdana. Verzichten Sie auf die Angabe der konkreten Schriftart, denn Sie können nicht wissen, welche Schriften auf dem System des Benutzers vorhanden sind.

Reaktion auf das action-Ereignis einer JTextField-Instanz

Eine `JTextField`-Instanz kann beim Drücken der ↵-Taste ein `action`-Ereignis werfen. Dieses kann mit Hilfe einer `ActionListener`-Instanz abgefangen und verarbeitet werden:

Listing 9.24
Behandeln des action-Ereignisses einer JTextField-Instanz

```
import javax.swing.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.*;

/**
 * Reacting on a JTextField's action event
 */
public class TextFieldEvents extends BaseFrame
   implements ActionListener {

   JLabel resultText;

   /**
    * Creates the JTextField-instance and two Labels,
    * binds the ActionListener to the JTextField-instance
    */
   TextFieldEvents() {

      // ...
```

Listing 9.24 *(Forts.)*
Behandeln des action-Ereignisses einer JTextField-Instanz

```
        JTextField text = new JTextField(
           "Your text goes here...", 20);
        text.addActionListener(this);

        // ...
    }

    /**
     * Handles the action-event of a JTextField-instance
     */
    public void actionPerformed(ActionEvent e) {
        JTextField source = (JTextField)e.getSource();
        resultText.setText("Your text: " +
           source.getText());
    }
}
```

Damit das `action`-Ereignis überhaupt behandelt werden kann, muss eine `ActionListener`-Implementierung gebunden sein. Diese Bindung geschieht mit Hilfe der Methode `addActionListener()` der `JTextField`-Instanz. Der in diesem Fall gebundene Listener ist die Instanz der aktuellen Klasse selbst – wir referenzieren sie mit Hilfe des Schlüsselwortes `this`:

```
JTextField text = new JTextField(
   "Your text goes here...", 20);
text.addActionListener(this);
```

Nun kann beim Druck der ↵-Taste das `action`-Ereignis geworfen werden. Die Behandlung findet in der Methode `actionPerformed()` statt.

Hier casten wir den Auslöser des Ereignisses von der generischen Klasse `Object` wieder zurück in eine `JTextField`-Instanz:

```
JTextField source = (JTextField)e.getSource();
```

Den im Textfeld enthaltenen Text können wir nun mit Hilfe seiner Methode `getText()` auslesen und anzeigen.

Wenn Sie die Klasse ausführen, erhalten Sie zunächst die initiale Ansicht mit der Aufforderung, doch bitte einen Text einzugeben:

Abbildung 9.34
Noch ist die ↵-Taste nicht gedrückt worden ...

Wenn Sie – dieser Aufforderung folgend – einen Text eingeben und die ↵-Taste drücken, wird der von Ihnen eingegebene Text ausgegeben:

Abbildung 9.35
Jetzt schon ...

Reaktion auf die focus-Ereignisse

Neben dem `action`-Event können wir auch auf das `focus`-Ereignis reagieren. Dies erlaubt es uns beispielsweise, auf Änderungen des in dem `JTextField` enthaltenen Texts zu reagieren, sobald der Benutzer den Eingabefokus auf ein anderes Element legt (etwa durch Klicken oder die ⇆-Taste). Die Behandlung des `focus`-Events erfordert die Implementierung des Interfaces `FocusListener()`, das die beiden Methoden `focusGained()` und `focusLost()` deklariert:

```
void focusGained(FocusEvent e)
void focusLost(FocusEvent e)
```

Wenn wir auf die focus-Ereignisse der `JTextField`-Instanz reagieren wollen, könnte dies dann so umgesetzt werden:

Listing 9.25
Reaktion auf Setzen und Verlieren des Eingabe-Fokus

```
import javax.swing.*;
import java.awt.event.FocusListener;
import java.awt.event.FocusEvent;
import java.awt.*;

/**
 * Reacting on a JTextField's focus events
 */
public class TextFieldFocus extends BaseFrame
   implements FocusListener {

   JLabel resultText;

   /**
    * Creates the JTextField-instance and two Labels,
    * binds the FocusListener to the JTextField-instance
    */
   TextFieldFocus() {
      JPanel content = new JPanel(new BorderLayout());
      this.add(content);

      // ...

      JTextField text = new JTextField(
         "Your text goes here...", 20);
      text.addFocusListener(this);
      center.add(text);

      // ...
   }

   /**
    * Is invoked, when the JTextField gets the focus
    * --> selects the whole text contained in the field
    */
   public void focusGained(FocusEvent e) {
      JTextField source = (JTextField)e.getSource();
      source.setSelectionStart(0);
      source.setSelectionEnd(source.getText().length());
   }
```

Listing 9.25 *(Forts.)*
Reaktion auf Setzen und Verlieren des Eingabe-Fokus

```
    /**
     * Is invoked, when the JTextField looses the focus
     * --> reads out the whole text contained in the field
     */
    public void focusLost(FocusEvent e) {
       JTextField source = (JTextField)e.getSource();
       resultText.setText("Your text: " + source.getText());
    }
}
```

Die `JTextField`-Instanz wird wie gewohnt deklariert. Anschließend können wir den `FocusListener` an sie binden. Dies geschieht, indem wir der Methode `addFocusListener()` des Text-Feldes die Instanz des FocusListeners übergeben, die die Ereignisbehandlung übernehmen soll:

```
public void addFocusListener(FocusListener l)
```

Da die aktuelle Klasse das Interface `FocusListener` implementiert, kann eine Referenz auf sie als Parameter übergeben werden. Die Referenz auf die aktuelle Klassen-Instanz erhalten wir, indem wir das Schlüsselwort `this` verwenden:

```
JTextField text = new JTextField(
   "Your text goes here...", 20);
text.addFocusListener(this);
```

Nun kann das Text-Feld einem Container hinzugefügt werden:

Abbildung 9.36
Die Applikation ist gestartet, es wurde noch kein Element aktiviert

Sobald der Nutzer den Eingabe-Fokus auf die `JTextField`-Instanz legt, wird das `focusGained`-Ereignis geworfen und die gleichnamige Methode aller gebundenen Event-Handler aufgerufen. Hier casten wir den als `Object`-Instanz übergebenen Auslöser, den wir mit Hilfe der Methode `getSource()` der `FocusEvent`-Instanz `e` ermitteln können, wieder in seinen eigentlichen Typ – eine `JTextField`-Instanz:

```
JTextField source = (JTextField)e.getSource();
```

Nun können wir innerhalb des Ereignis-Handlers den schon eingegebenen Text selektieren. Dies geschieht, indem wir mit Hilfe der Methode `setSelectionStart()` die Zeichen-Position des im `JTextField` enthaltenen Texts festlegen, an der der Auswahlbereich starten soll. Das Ende des Auswahlbereichs legen wir mit Hilfe von `setSelectionEnd()` fest – in unserem Fall soll der komplette Text markiert werden und deshalb übergeben wir die Textlänge als Parameter:

```
source.setSelectionStart(0);
source.setSelectionEnd(source.getText().length());
```

Der Sinn der ganzen Aktion liegt darin, dass der Nutzer nun bereits eingegebenen Text nicht mehr löschen muss, sondern einfach schreiben kann – der zuvor vorhandene Text wird einfach überschrieben:

Abbildung 9.37
Das Text-Feld hat den Eingabefokus erhalten

Sobald der Nutzer nun den Eingabe-Fokus auf ein anderes Element bewegt – sei es durch die ⇆-Taste oder durch einen Maus-Klick –, wird das Ereignis `focusLost` geworfen und kann von `FocusListenern` in deren gleichnamiger Methode behandelt werden.

In unserem Fall casten wir hier wieder den Auslöser in eine `JTextField`-Instanz, lesen deren Wert aus und zeigen ihn in einem `JLabel` an:

```
JTextField source = (JTextField)e.getSource();
resultText.setText("Your text: " + source.getText());
```

Um die Applikation abzurunden, könnten Sie nun noch einen `ActionListener` für den Klick auf den Button schreiben.

Texteingabe-Ereignisse behandeln

Manchmal ist es nicht ausreichend oder einfach nicht erwünscht, erst bei einem `focus-` oder `action`-Ereignis benachrichtigt zu werden und die Reaktion soll direkt bei jedem Tastendruck erfolgen. Eine direkte Reaktion auf Tastendrücke ist jedoch bei `JTextField`-Instanzen nicht möglich – es existiert schlicht keine Möglichkeit, einen wie auch immer gearteten Handler zu registrieren.

Wir müssen deshalb eine Ebene tiefer ansetzen: `JTextField`-Instanzen verfügen über ein `Document`-Objekt. Dieses `Document` verwaltet den enthaltenen Text und stellt intern die Basis von `JTextField`-Objekten dar.

Das `Document`-Objekt ist mit Hilfe der Methode `getDocument()` auch von außen erreichbar und kann Events werfen, sobald der enthaltene Text geändert oder anderweitig bearbeitet wird. Diese Ereignisse können von der `DocumentListener`-Implementierung aufgefangen und behandelt werden.

Das Interface `DocumentListener` erwartet, dass drei Methoden implementiert werden:

- `insertUpdate(DocumentEvent)`: Informiert darüber, dass ein Zeichen hinzugefügt wurde
- `removeUpdate(DocumentEvent)`: Informiert darüber, dass ein Zeichen entfernt wurde
- `changedUpdate(DocumentEvent)`: Informiert darüber, dass sich Eigenschaften der Document-Instanz geändert haben

Uns interessieren bei der weiteren Verarbeitung nur die beiden Methoden `insertUpdate()` und `removeUpdate()`, da diese sich auf den im Document enthaltenen Text beziehen:

Listing 9.26
Behandeln von Document-Ereignissen

```
import javax.swing.*;
import javax.swing.text.Document;
import javax.swing.text.BadLocationException;
import javax.swing.event.DocumentListener;
import javax.swing.event.DocumentEvent;
import java.awt.*;

/**
 * Capturing key-press-events of a JTextField-instance
 */
public class TextFieldInput extends BaseFrame
   implements DocumentListener {

   JLabel resultText;

   /**
    * Creates the JTextField-instance and two Labels,
    * binds a DocumentListener to the JTextField's Document-instance
    */
   TextFieldInput() {

      /..

      JTextField text = new JTextField(
         "Your text goes here...", 20);
      text.getDocument().addDocumentListener(this);
      center.add(text);

      / ..

   }

   /**
    * Handles both insert- and remove-events
    * from the DocumentListener
    */
   private void handle(DocumentEvent e) {
      Document d = e.getDocument();
      try {
         resultText.setText(
            "Your text: " +
               d.getText(0, d.getLength()));
      } catch (BadLocationException ignored) {}
   }

   /**
    * Handles the insert-event
    */
   public void insertUpdate(DocumentEvent e) {
      handle(e);
   }

   /**
    * Handles the remove-event
    */
   public void removeUpdate(DocumentEvent e) {
      handle(e);
   }
```

```
    /**
     * Is ignored and not used
     */
    public void changedUpdate(DocumentEvent e) {}
}
```

Listing 9.26 *(Forts.)*
Behandeln von Document-Ereignissen

Das Binden eines `DocumentListeners` an das `Document`-Objekt einer `JTextBox`-Instanz geschieht, indem zunächst mit Hilfe von `getDocument()` eine Referenz auf die `Document`-Instanz erzeugt und der Listener mit Hilfe der Methode `addDocumentListener()` registriert wird:

```
JTextField text = new JTextField(
   "Your text goes here...", 20);
text.getDocument().addDocumentListener(this);
center.add(text);
```

Sobald der Nutzer Daten in das `JTextField` eingibt, wird das `insert`-Ereignis der `Document`-Instanz geworfen. Löscht er dagegen Zeichen, erfolgt das Werfen des `remove`-Events. Beide haben zwar getrennte Entsprechungen im `DocumentListener`-Interface, werden aber von uns identisch in der Methode `handle()` behandelt.

Innerhalb der Methode `handle()` erzeugen wir zunächst eine Referenz auf die `Document`-Instanz, die das Ereignis geworfen hatte. Dies geschieht, indem wir die Methode `getDocument()` der `DocumentEvent`-Instanz `e` auswerten:

```
Document d = e.getDocument();
```

Nun rufen wir den in der `Document`-Instanz hinterlegten Text ab. Wir verwenden zu diesem Zweck die Methode `getText()`, die zwei Parameter erwartet: den Index des ersten und den Index des letzten Zeichens:

```
String getText(int offset, int length)
   throws BadLocationException
```

Ersterer Wert ist in diesem Fall Null, da uns der Text ab dem ersten Zeichen interessiert. Den Index des letzten Zeichens ermitteln wir mit Hilfe der Methode `getLength()`. Den so ermittelten Text können wir dann ausgeben:

```
resultText.setText(
   "Your text: " +
      d.getText(0, d.getLength()));
```

Beachten Sie, dass es auf dieser recht niedrigen Ebene keine Abfrage mehr gibt, ob die von Ihnen angegebenen Werte plausibel sind. Sind sie es nicht, dann wird eine `BadLocationException` geworfen, die wir deshalb mit Hilfe eines `try-catch`-Blocks abfangen.

Der Lohn der Mühe: Der vom Nutzer eingegebene Text wird nunmehr direkt bei einem Tastendruck verarbeitet – im Falle unserer Applikation erscheint er also fast in Echtzeit unter dem Text-Feld in einem dort platzierten Label:

Abbildung 9.38
Der eingegebene Text erscheint fast in Echtzeit am unteren Rand

9.5.7 JPasswordField

Die `JPasswordField`-Komponente ist eine Ableitung der `JTextField`-Komponente, die statt der eingegebenen Zeichen so genannte `Echo`-Zeichen (in der Standard-Einstellung sind das Sterne) anzeigt. Bei der Erzeugung einer `JPasswordField`-Instanz können wir dieser unter anderem ihre darzustellende Breite und einen initialen (mit Hilfe der Echo-Zeichen dargestellten) Text als Parameter übergeben:

```
public JPasswordField()
public JPasswordField(String text)
public JPasswordField(int columns)
public JPasswordField(String text, int columns)
public JPasswordField(Document doc, String txt, int columns)
```

Anschließend kann das Echo-Zeichen mit Hilfe der Methode `setEchoChar()` gesetzt werden:

```
JPasswordField pass = new JPasswordField(10);
pass.setEchoChar('#');
content.add(pass);
```

Wenn Sie nun ein Zeichen in das Password-Feld eingeben, wird statt des Zeichens eine Raute angezeigt:

Abbildung 9.39
Eingabe eines Kennwortes in ein JPasswordField

Da das `JPasswordField` direkt von `JTextField` erbt, haben Sie die Möglichkeit, Ereignisse auf die gleiche Art wie beim `JTextField` zu behandeln und `Listener` zu binden.

9.5.8 JFormattedTextField

Das `JFormattedTextField` stellt ebenfalls eine Ableitung von `JTextField` dar. Mit Hilfe dieser Komponente sind wir in der Lage, schon bei der Eingabe Validierungen des Inhalts vorzunehmen. Der Einsatz von `JFormattedTextField` bietet sich immer dann an, wenn wir ein Muster der erwarteten Eingabe definieren können.

Dem `JFormattedTextField` kann im Konstruktor unter anderem eine Instanz der abstrakten Klasse `Format` übergeben werden. Damit haben wir die Möglichkeit, beispielsweise Telefonnummer-Formate zu deklarieren – und der Nutzer muss diese korrekt ausfüllen, sonst wird der Inhalt des Feldes nicht als valide betrachtet:

```
public JFormattedTextField()
public JFormattedTextField(Object value)
public JFormattedTextField(Format format)
public JFormattedTextField(
   JFormattedTextField.AbstractFormatter formatter)
```

```
public JFormattedTextField(
   JFormattedTextField.AbstractFormatterFactory factory)
public JFormattedTextField(
   JFormattedTextField.AbstractFormatterFactory factory,
   Object currentValue)
```

Der Einsatz von `JFormattedTextField`-Instanzen ist immer in Zusammenhang mit `action`- oder `focus`-Events sinnvoll:

Listing 9.27
Einsatz einer JFormatted TextField-Instanz

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.text.SimpleDateFormat;

/**
 * Forces the user to enter a valid value
 */
public class FormattedTextField extends BaseFrame
   implements ActionListener {

   JFormattedTextField field;
   JLabel result;

   /**
    * Creates some Label, a Button and a JFormattedTextField
    * which only allows to user to enter a date in the following
    * format: dd.MM.yyyy
    */
   FormattedTextField() {

      // ...

      field = new JFormattedTextField(
         new SimpleDateFormat("dd.MM.yyyy"));
      field.addActionListener(this);
      center.add(field);

      JButton button = new JButton("Validate");
      button.addActionListener(this);
      center.add(button);

      // ...

   }

   /**
    * Handles the Button's and the
    * JFormattedTextField's action event
    */
   public void actionPerformed(ActionEvent e) {
      if(field.getText() != null &&
         field.getText().length() > 0 &&
         field.isEditValid()) {
```

Listing 9.27 *(Forts.)*
Einsatz einer JFormatted TextField-Instanz

```
            result.setText(
               String.format("You entered: %s",
                  field.getText()));
         } else {
            result.setText("Please enter a valid date!");
         }
      }
   }
```

Einer `JFormattedTextField`-Instanz können wir das zu verwendende `Format` oder den zu verwendenden Formatierer schon im Konstruktor übergeben. Fügen wir dann noch einen `ActionListener` hinzu, können wir später reagieren, wenn Daten eingegeben worden sind und validiert werden sollen:

```
field = new JFormattedTextField(
   new SimpleDateFormat("dd.MM.yyyy"));
field.addActionListener(this);
```

Die hier verwendete `SimpleDateFormat`-Instanz legt fest, dass nur Datumswerte im deutschen Datumsformat (Tag, Monat, Jahr) eingegeben werden dürfen.

Wenn das Format nicht korrekt ist, können wir zwar die ↵-Taste innerhalb der `JFormattedTextField`-Instanz drücken, es wird jedoch kein `action`-Ereignis geworfen. Sollten wir den Eingabe-Fokus auf ein anderes Element der Applikation (etwa den daneben liegenden Button) setzen, löscht die `JFormattedTextField`-Instanz automatisch ihren Inhalt, falls dieser nicht dem geforderten Format entspricht.

Wenn das `action`-Event geworfen wird, können wir recht einfach feststellen, ob ein gültiger Wert eingegeben worden ist. In diesem Fall müsste nämlich die Methode `isEditValid()` der `JFormattedTextField`-Instanz den Wert `true` zurückgeben. Leider gilt dies nicht für eine Länge von 0 Zeichen, weshalb wir diesen Sonderfall ebenfalls mit überprüfen müssen, wenn wir wissen wollen, ob die Eingabe korrekt war. Folglich müssen diese Bedingungen zutreffen, um eine Eingabe als gültig betrachten zu können:

- Die Eingabe muss ungleich `null` sein
- Die Länge der Eingabe muss mindestens ein Zeichen sein
- Die Methode `isEditValid()` der `JFormattedTextField`-Instanz muss `true` zurückgeben

Wenn alle diese Bedingungen zutreffen, können wir eine entsprechende Meldung ausgeben:

```
if(field.getText() != null &&
   field.getText().length() > 0 &&
   field.isEditValid()) {

   result.setText(
      String.format("You entered: %s",
         field.getText()));

} else {
   result.setText("Please enter a valid date!");
}
```

Wenn wir nun die Klasse ausführen und einen nicht gültigen Wert eingeben, wird dies sofort mit einer entsprechenden Ausgabe quittiert:

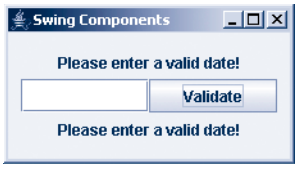


Abbildung 9.40
Hier wurde kein gültiger Wert eingegeben

Geben wir also lieber einen gültigen Wert ein:

Abbildung 9.41
Diese Eingabe ist gültig

Anzeige des Patterns

Die Klasse `JFormatTextField` deklariert eine untergeordnete Klasse `AbstractFormatter`, die – wie der Name bereits ausdrückt – eine Basis-Implementierung einer Formatierungsklasse für einzugebenden Text darstellt.

Klassen, die von `AbstractFormatter` erben, können das durch sie definierte Muster sichtbar machen. Sie verfügen zu diesem Zweck über eine Methode `setPlaceholderCharacter()`, der wir ein Zeichen übergeben können, das als Platzhalter für die tatsächlich einzugebenden Zeichen fungiert:

Listing 9.28
Deklaration und Anzeige eines Eingabe-Patterns

```
import javax.swing.*;
import javax.swing.text.MaskFormatter;
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.text.ParseException;

/**
 * Forces the user to enter a valid value
 */
public class FormattedTextFieldDecimal extends BaseFrame
   implements ActionListener {

   JFormattedTextField field;
   JLabel result;

   /**
    * Creates some Label, a Button and a JFormattedTextField
    * which only allows to user to enter a date in the following
    * fornat: #.###
    */
   FormattedTextFieldDecimal() throws ParseException {

      // ...
```

Listing 9.28 *(Forts.)* Deklaration und Anzeige eines Eingabe-Patterns

```
            MaskFormatter mf = new MaskFormatter("#.###");
            mf.setPlaceholderCharacter('#');

            field = new JFormattedTextField(mf);
            field.setFocusLostBehavior(JFormattedTextField.PERSIST);
            field.addActionListener(this);
            center.add(field);

            // ...

        }

        /**
         * Handles the Button's and the
         * JFormattedTextField's action event
         */
        public void actionPerformed(ActionEvent e) {
            boolean isValid = field.isEditValid();
            if(isValid) {
                try {
                    field.commitEdit();
                } catch (ParseException ignored) {
                    isValid = false;
                }
            }

            if(isValid) {
                result.setText(
                    String.format("You entered: %s",
                        field.getText()));
            } else {
                result.setText("Please enter a valid number!");
            }
        }
    }
```

Das von uns deklarierte Eingabe-Pattern *#.###* lässt nur Ziffern zu. Wir weisen es einer `MaskFormatter`-Instanz im Konstruktor zu und legen anschließend die Raute als Platzhalter fest. Die so instanzierte `MaskFormatter`-Instanz kann nun dem Konstruktor des `JFormattedFields` als Parameter übergeben werden:

```
MaskFormatter mf = new MaskFormatter("#.###");
mf.setPlaceholderCharacter('#');

field = new JFormattedTextField(mf);
```

Nun treffen wir eine folgenreiche Festlegung. Wir definieren das Verhalten der `JFormattedTextField`-Instanz beim Verlust des Eingabe-Fokus mit Hilfe der Methode `setFocusLostBehavior()` um.

Das Standard-Verhalten ist `JFormattedTextField.COMMIT_OR_REVERT` – das Text-Feld legt selbst fest, wann es Werte wirklich übernimmt (wenn sie gültig sind) oder sie verwirft.

Das von uns neu definierte Verhalten wird durch `JFormattedTextField.PERSIST` repräsentiert und legt fest, dass der eingegebene Wert auch dann angezeigt wer-

den soll, wenn er ungültig ist – die Übernahme in die `JFormattedTextField`-Instanz muss später also manuell erfolgen:

```
field.setFocusLostBehavior(JFormattedTextField.PERSIST);
```

Da diese Einstellung den eingegebenen Wert nicht automatisch aus dem Eingabefeld in die `JFormattedTextField`-Instanz übernimmt, müssen wir dies beim Werfen des `action`-Events selbst ausführen:

```
boolean isValid = field.isEditValid();
if(isValid) {
   try {
      field.commitEdit();
   } catch (ParseException ignored) {
      isValid = false;
   }
}
```

Die Methode `isEditValid()` gibt uns zurück, ob der eingegebene Wert gültig ist. Sollte dies der Fall sein, können wir das Text-Feld anweisen, den Wert zu übernehmen. Dies geschieht durch Aufruf der Methode `commitEdit()`.

Ist der Wert in diesem Moment jedoch ungültig, wird eine `ParseException` geworfen – aus diesem Grund ist die Übernahme der Eingabe auch in einem `try-catch`-Block gekapselt. Innerhalb des `catch`-Blocks wird die Kontrollvariable `isValid` auf `false` gesetzt, so dass wir bei der weiteren Auswertung wissen, dass ein ungültiger Wert eingegeben worden ist.

Wenn die Kontrollvariable `isValid` den Wert `false` hat, wird der Nutzer darauf hingewiesen, dass der Wert ungültig ist:

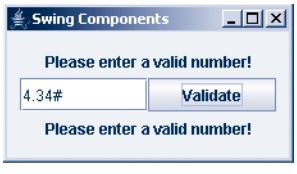


Abbildung 9.42
Der eingegebene Wert ist (noch) nicht gültig

Ist die Eingabe korrekt, wird sie übernommen und wir können ihren Wert weiter verarbeiten:

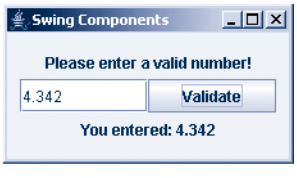


Abbildung 9.43
Diese Eingabe war gültig

9.5.9 JTextArea

Die `JTextArea` kann verwendet werden, um mehrzeiligen Text anzuzeigen. Der Text kann schreibgeschützt oder vom User editierbar sein. Die `JTextArea` kann keinen speziell formatierten Text anzeigen – dafür ist ihr Gebrauch sehr eingängig, da die Komponente von `JTextField` erbt.

Beim Erzeugen von JTextArea-Instanzen können wir unter anderem deren Ausdehnung in Spalten und Breiten oder einen initial enthaltenen Text angeben:

```
public JTextArea()
public JTextArea(String text)
public JTextArea(int rows, int columns)
public JTextArea(String text, int rows, int columns)
public JTextArea(Document doc)
public JTextArea(Document doc, String text, int rows,
   int columns)
```

Da der Text in JTextArea-Instanzen nicht automatisch scrollt, müssen wir dafür sorgen, dass diese scroll-fähig werden. Zu diesem Zweck setzen wir eine JScrollPane-Instanz ein, der wir die Text-Area zuweisen:

Listing 9.29 Erzeugen einer JTextArea

```
import javax.swing.*;
import java.awt.*;

/**
 * Displays a scollable JTextArea
 */
public class TextArea extends BaseFrame {

   TextArea() {
      JPanel content = new JPanel();
      this.add(content);

      JTextArea text = new JTextArea(
         "Please enter your text here!", 20, 40);
      JScrollPane scroll = new JScrollPane(text);
      scroll.setPreferredSize(new Dimension(200, 100));

      content.add(scroll);
      this.pack();
   }
}
```

Beim Erzeugen der JTextArea übergeben wir ihrem Konstruktor den initial anzuzeigenden Text sowie die Anzahl der Zeilen und Spalten, die die JTextArea groß sein soll. In diesem Fall soll sie zwanzig Zeilen hoch und vierzig Spalten breit sein:

```
JTextArea text = new JTextArea(
   "Please enter your text here!", 20, 40);
```

Eine JScrollPane-Instanz nimmt die so erzeugte JTextArea auf, damit diese scrollen kann. Wir vermeiden somit, dass die JTextArea, die eine beträchtliche Größe hat, komplett angezeigt werden muss:

```
JScrollPane scroll = new JScrollPane(text);
```

Die anzuzeigende Fläche der ScrollPane wird definiert, indem ihrer Methode setPreferredSize() eine neue Dimension-Instanz übergeben wird, die eine Breite von 200 Pixeln und eine Höhe von 100 Pixeln definiert:

```
scroll.setPreferredSize(new Dimension(200, 100));
```

Wenn Sie die Applikation ausführen, werden Sie diese Darstellung erhalten:

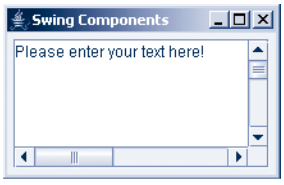


Abbildung 9.44
Eine scrollbare JTextArea

Die Ereignisse der `JTextArea` entsprechen denen des `JTextFields`. Sie können also auf das Verlieren des Fokus der Komponente ebenso reagieren, wie dies bei der `JTextField`-Klasse der Fall ist.

Virtuellen Zeilenumbruch aktivieren

Der virtuelle Zeilenumbruch sorgt dafür, dass der Text am Rand der Komponente in die nächste Zeile umbricht, ohne dass ein Scrolling erfolgt. Dies ist das Verhalten, dass die meisten Anwender bei einer derartigen Komponente erwarten.

Glücklicherweise gibt es Abhilfe durch die Methode `setLineWrap()`:

```
public void setLineWrap(boolean wrap)
```

Bei Zuweisung des Wertes `true` an die Methode verschwinden die unnötig gewordenen Scrollbalken am unteren Rand und die `JTextArea` scrollt nur noch vertikal.

Leider gibt es nun noch ein Problem mit den Umbrüchen: Diese finden sogar innerhalb eines Wortes statt – das zu erwartende Verhalten wäre aber, dass der Zeilenumbruch bei einem Leerzeichen erfolgen würde. Dieses Verhalten lässt sich erzwingen:

```
public void setWrapStyleWord(boolean word)
```

Übergeben Sie den Wert `true` an die Methode `setWrapStyleWord()`, wird der korrekte Wort-Umbruch aktiviert.

Länge der Eingabe beschränken

Die Text-Komponenten von `Swing/JFC` bieten zwar viele Einstellmöglichkeiten, nirgendwo aber findet sich eine Methode, die das Setzen der maximalen Eingabelänge zuließe. Eine Möglichkeit wäre sicherlich, auf ein `JFormattedField` zurückzugreifen und dort ein entsprechendes Muster zu definieren. In der Praxis dürfte dies aber spätestens bei Eingabelängen ab 100 Zeichen mehr als nervend werden.

Dabei gibt es einen anderen Ansatz, der nebenbei auch besonders elegant ist – packt er doch das Problem an der Wurzel und löst es auf einer recht niedrigen Ebene. Dieser Ansatz besteht darin, das von jedem Text-Eingabefeld verwendete `Document`-Objekt, das alle Einfüge- und Bearbeitungsoperationen auf dem Text durchführt, durch eine eigene, abgeleitete Version zu ersetzen:

Listing 9.30
Das LengthLimitedDocument erlaubt es, die Länge des eingegebenen Texts zu limitieren

```
import javax.swing.text.PlainDocument;
import javax.swing.text.AttributeSet;
import javax.swing.text.BadLocationException;

public class LengthLimitedDocument extends PlainDocument {

   private int maxLength = 255;

   LengthLimitedDocument() {}

   LengthLimitedDocument(int maxLength) {
      this.maxLength = maxLength;
   }

   LengthLimitedDocument(int maxLength, Document base)  {
      this.maxLength = maxLength;

      String baseContent = null;
      try {
         baseContent = base.getText(0, base.getLength());
         this.insertString(0, baseContent, null);
      } catch (BadLocationException e) {
         e.printStackTrace();
      }
   }

   public void insertString (
      int offset, String  str, AttributeSet attr)
      throws BadLocationException {

      if (str == null) {
         return;
      }

      if (getLength() + str.length() <= maxLength) {
         super.insertString(offset, str, attr);
      }
   }
}
```

Die Klasse `LengthLimitedDocument` erweitert die Standard-Klasse `PlainDocument`. Sie verfügt über drei Konstruktoren:

- `LengthLimitedDocument()`: Verwendet die Standard-Länge von 255 Zeichen
- `LengthLimitedDocument(int)`: Setzt die maximale Länge der Zeichenkette
- `LengthLimitedDocument(int, Document)`: Setzt die maximale Länge der Zeichenkette, übernimmt eine eventuell bereits vorhandene Zeichenkette

Interessant ist für uns die Methode `insertString()`, die für das Einfügen einer Zeichenkette (etwa nach dem Eingeben eines Zeichens) zuständig ist. Hier überprüfen wir, ob die Gesamtlänge aller Zeichen und die Länge der einzufügenden Zeichenkette größer als die erlaubte Maximallänge der Zeichen ist. Sollte dies nicht so sein, kann die Zeichenkette über die Funktionalität der Basis-Klasse eingefügt werden.

Die Klasse `LengthLimitedDocument` kann nun in allen Swing-Text-Komponenten eingesetzt werden. Ergänzen wir zu diesem Zweck eine `JTextArea`-Instanz mit dieser Funktionalität:

Listing 9.31
Erzeugen einer JTextArea, die nur 45 Zeichen zulässt

```
import javax.swing.*;
import java.awt.*;

/**
 * Displays a scrollable JTextArea with a maximum input length
 */
public class LengthLimitedTextArea extends BaseFrame {

   LengthLimitedTextArea() {

      // ...

      JTextArea text = new JTextArea(
         "Please enter your text here!", 20, 40);
      text.setDocument(
         new LengthLimitedDocument(45), this.getDocument());
      JScrollPane scroll = new JScrollPane(text);
      scroll.setPreferredSize(new Dimension(200, 100));

      // ...

   }
}
```

Die einzige Änderung, die vorzunehmen ist, besteht darin, der `JTextArea` eine neue `Document`-Instanz mit Hilfe von deren Methode `setDocument()` zuzuweisen. Diese nimmt in unserem Fall eine neue Instanz der `LengthLimitedDocument`-Klasse auf, die maximal 45 Zeichen zulässt und die Zugriff auf die Document-Instanz der `JTextArea` bekommt, um dort enthaltenen Text auszulesen.

Nun kann der Nutzer zwar eigenen Text eingeben, mehr als 45 Zeichen sind aber nicht möglich.

9.5.10 JEditorPane

Das `JEditorPane` ist eine der mächtigsten textverarbeitenden Komponenten innerhalb des `Swing`-Frameworks. Sie erlaubt es, formatierten RTF-Text und sogar HTML-Code zu verarbeiten.

Der Konstruktor des JEditorPanes ist mehrfach überladen:

```
public JEditorPane()
public JEditorPane(URL initialPage)
   throws IOException
public JEditorPane(String url)
   throws IOException
public JEditorPane(String type, String text)
```

Um Text in einem JEditorPane anzuzeigen, können Sie auf diesen mit Hilfe einer URL-Instanz verweisen und dem JEditorPane im Konstruktor übergeben:

Listing 9.32 Anzeigen einer HTML-Datei in einem JEditorPane

```
import javax.swing.*;
import java.io.IOException;
import java.awt.*;

/**
 * Displays a HTML-file in a JEditorPane
 */
public class EditorPane extends BaseFrame {

   EditorPane() {
      JPanel content = new JPanel();
      add(content);

      JEditorPane pane = null;
      try {
         pane = new JEditorPane(
            EditorPane.class.getResource("text.html"));
      } catch (IOException e) {
         e.printStackTrace();
      }

      JScrollPane scroll = new JScrollPane(pane);
      scroll.setPreferredSize(new Dimension(300, 150));
      content.add(scroll);

      pack();
   }
}
```

Der Konstruktor des JEditorPane nimmt in einer Überladung eine URL-Instanz entgegen, die auf eine Datei verweist, welche sowohl auf dem lokalen System als auch im Internet liegen kann. Diese URL-Instanz bezieht sich in diesem Fall auf eine Datei text.html, die im selben Verzeichnis wie die kompilierte Java-Klasse des Beispiels liegt:

```
pane = new JEditorPane(
   EditorPane.class.getResource("text.html"));
```

Dabei kann es zu einer IOException kommen, falls die angegebene Ressource nicht gefunden oder aus einem anderen Grund nicht geladen werden konnte.

Das so erzeugte JEditorPane weisen wir nun einer neuen JScrollPane-Instanz zu – schließlich können wir Breite und Höhe der darzustellenden Seite nicht im Voraus wissen und überlassen somit dem JScrollPane, eventuell nötige Scroll-Balken anzuzeigen:

```
JScrollPane scroll = new JScrollPane(pane);
```

Nach dem Setzen der bevorzugten Anzeige-Größe des `JScrollPane` können wir es dem Basis-Frame zuweisen und anzeigen lassen:

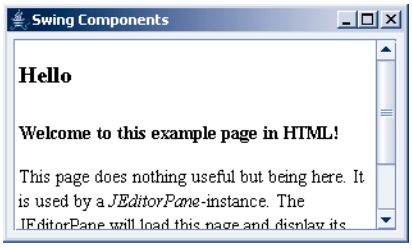


Abbildung 9.45
Anzeigen einer HTML-Seite in einem JEditorPane

setPage()

Sie sind selbstverständlich nicht darauf beschränkt, das `JEditorPane` mit einem `URL` zu instanzieren. Dies kann auch nachträglich geschehen, wie das folgende Beispiel demonstriert:

Listing 9.33
Ein Browser auf Basis des JEditorPanes

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.io.IOException;

/**
 * Using a JEditorPane as a Browser
 */
public class EditorPaneBrowser extends BaseFrame
   implements ActionListener {

   JTextField location;
   JEditorPane browser;

   /**
    * Constructs the layout
    */
   EditorPaneBrowser() {
      JPanel main = new JPanel(new BorderLayout());
      main.setPreferredSize(new Dimension(400, 200));

      /*
       * Constructing the location-JTextField
       */
      JPanel locationPanel = new JPanel();
      location = new JTextField(20);
      location.addActionListener(this);
      locationPanel.add(location);

      /*
       * a "Go"-Button
       */
      JButton locationButton = new JButton("Go!");
      locationButton.addActionListener(this);
      locationPanel.add(locationButton);
```

Listing 9.33 *(Forts.)* Ein Browser auf Basis des JEditorPanes

```
        /*
         * Displays the loaded content
         */
        browser = new JEditorPane();
        browser.setEditable(false);


        /*
         * Allows the browser-component to scroll
         */
        JScrollPane scroll = new JScrollPane(browser);

        main.add(scroll);
        main.add(locationPanel, BorderLayout.NORTH);
        add(main);

        setTitle("EditorPaneBrowser");
        pack();
    }

    /**
     * Reacts on the action-event thrown by the
     * location- and the locationButton-components
     */
    public void actionPerformed(ActionEvent e) {
        try {
            browser.setPage(this.location.getText());
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```

Der Mini-Browser besteht im Grunde aus drei Komponenten: einer `JTextField` zur Eingabe einer Web-Adresse, einem `JButton` zum Aufruf der angegebenen Seite und einem `JEditorPane` zur Darstellung der Seite. Diese werden im Konstruktor zusammen mit zwei `JPanel`-Instanzen und einer `JScrollPane` angelegt.

Beim Anlegen von `JTextField` und `JButton` werden deren `action`-Events mit einer `ActionListener`-Instanz verknüpft. Diese `ActionListener`-Instanz ist die aktuelle Klasseninstanz, weshalb wir den `addActionListener()`-Methoden der beiden Objekte jeweils eine Referenz per `this` übergeben:

```
location = new JTextField(20);
location.addActionListener(this);
```

Das `JEditorPane browser` wird hier ohne Parameter instanziert. Anschließend wird es in einen schreibgeschützten Zustand versetzt, so dass ein Benutzer keine Änderungen an der Darstellung vornehmen kann. Dies geschieht, indem seiner Methode `setEditable()` der Wert `false` übergeben wird:

```
browser = new JEditorPane();
browser.setEditable(false);
```

Jetzt kann die Zuweisung des `JEditorPanes` zu einem `JScrollPane` erfolgen. Dies stellt sicher, dass Webseiten, die größer als der aktuelle Anzeige-Bereich der Komponente sind, tatsächlich auch vollständig dargestellt werden können:

```
JScrollPane scroll = new JScrollPane(browser);
```

Nach dem Zuweisen der Elemente zum übergeordneten Container kann die Applikation angezeigt werden:

Abbildung 9.46
Anzeige einer Webseite im JEditorPane

Wenn nun der Benutzer einen gültigen URL im Adressfeld eingibt und anschließend ↵ drückt oder auf den Button klickt, wird jeweils ein `action`-Event gefeuert. Dieses wird von der aktuellen Klasseninstanz in deren Methode `actionPerformed()` behandelt. Hier wird nun dem `JEditorPane browser` mit Hilfe seiner Methode `setPage()` mitgeteilt, dass es den Inhalt des im Adressfeld `location` angegebenen URL laden und anzeigen soll:

```
browser.setPage(this.location.getText());
```

Dabei kann eine `IOException` auftreten – etwa wenn der URL ungültig ist oder aus anderen Gründen auf dessen Inhalt nicht zugegriffen werden kann. Aus diesem Grund ist das Zuweisen des URL in einem `try-catch`-Block gekapselt.

Zuweisen und Abrufen von Inhalten

Neben der Verwendung der Methode `setPage()` und des Konstruktors können Sie dem `JEditorPane` auch mit Hilfe seiner Methode `setText()` einen darzustellenden Inhalt zuweisen. Bei diesem Inhalt kann es sich entweder um reinen Text

oder HTML-Code handeln. Welche Art von Inhalt zugewiesen werden soll, legen Sie mit Hilfe der Methode `setContentType()` fest – hier stehen entweder `text/plain` (reiner Text) oder `text/html` (HTML-Code) zur Auswahl:

```
JEditorPane editor = new JEditorPane();
editor.setContentType("text/html");
editor.setText("<h2>Hello World!</h2>");
```

Die Darstellung dieses Inhalts kann vom Benutzer editiert werden – und der editierte Text kann selbstverständlich wieder ausgelesen werden. Das Auslesen des Inhaltes geschieht mit Hilfe der Methode `getText()` des `JEditorPanes`:

```
String result = editor.getText();
```

9.5.11 JFileChooser

Mit Hilfe der `JFileChooser`-Komponente können Sie auf einfachstem Wege eine Datei-Auswahl für das Laden und Speichern von Daten vornehmen. Der `JFileChooser` selbst beschränkt sich darauf, dem Nutzer eine Oberfläche für die Auswahl von Dateien oder Verzeichnissen zu geben – das Laden und Speichern der Daten müssen Sie selbst implementieren. Der `JFileChooser` zeigt sich als ein modales Fenster an – solange er aktiv ist, kann im aufrufenden Fenster nicht gearbeitet werden.

Um einen `JFileChooser` einsetzen zu können, müssen wir eine neue Instanz der `JFileChooser`-Klasse erzeugen und deren `openXXX`-Methode aufrufen. `XXX` steht hier für eine der folgenden Methoden:

- `showOpenDialog()`: Zeigt einen Öffnen-Dialog an.
- `showSaveDialog()`: Zeigt einen Speichern-Dialog an.
- `showDialog()`: Zeigt einen anpassbaren Dialog an.

Die Rückgabe dieser Methoden ist jeweils ein Integer-Wert, der eine der folgenden Aktionen kennzeichnet:

- `JFileChooser.APPROVE_OPTION`: Die Bestätigungs-Schaltfläche („Öffnen“, „Okay“, „Speichern“ etc.) wurde betätigt und eine Auswahl getroffen.
- `JFileChooser.CANCEL_OPTION`: Die Abbrechen-Schaltfläche wurde aktiviert.
- `JFileChooser.ERROR_OPTION`: Ein Fehler ist aufgetreten.

Dem `JFileChooser` kann eine Einschränkung der anzuzeigenden Dateien zugewiesen werden. So haben wir die Möglichkeit, beispielsweise nur GIF-Dateien für eine Auswahl zuzulassen. Das Zuweisen dieser Einschränkung geschieht mit Hilfe seiner Methode `setFileFilter()`, die eine Ableitung der abstrakten Basisklasse `FileFilter` als Parameter erwartet.

Weiterhin ist es möglich, dass der `JFileChooser` die mögliche Auswahl auf Dateien, Verzeichnisse oder beides beschränkt. Die Standard-Einstellung lässt eine Datei-Auswahl zu.

Damit noch nicht genug: Der `JFileChooser` ist so weit an unsere Bedürfnisse anpassbar, dass wir unter anderem diese Dinge ebenfalls manipulieren können:

- Art der Darstellung von Dateien (Liste, Vorschaubilder etc.)
- Titel des Fensters
- Start-Verzeichnis für die Auswahl
- Beschriftungen der Buttons
- Einfach- oder Mehrfachauswahl möglich

Dennoch ist der Umgang mit dem `JFileChooser` einfach geblieben. Um ihn einzusetzen, reicht es oftmals aus, eine neue Instanz zu erzeugen, anzuzeigen, die Rückgabe auszuwerten und zu ermitteln, welche Datei oder welches Verzeichnis angezeigt worden ist. Dieses Verhalten ist in folgendem Beispiel umgesetzt worden, das einen einfachen Datei-Viewer für eine Plain-Text-Ausgabe von Datei-Inhalten implementiert:

Listing 9.34
Ein File-Viewer, der die anzuzeigenden Dateien mit Hilfe eines JFileChoosers auswählt

```
import javax.swing.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.*;
import java.io.*;

/**
 * A basic file viewer based on a JTextArea
 * and a JFileChooser-component
 */
public class BasicFileChooser extends BaseFrame
   implements ActionListener {

   JLabel fileName;
   JScrollPane scroll;

   /**
    * Constructs the elements of the application
    */
   BasicFileChooser() {
      JPanel content = new JPanel(new BorderLayout());
      add(content);

      JPanel clickPanel = new JPanel();
      JButton click = new JButton("SELECT A FILE!");
      click.addActionListener(this);
      clickPanel.add(click);
      content.add(clickPanel, BorderLayout.NORTH);

      JTextArea fileContents = new JTextArea();
      fileContents.setPreferredSize(new Dimension(300, 200));
      scroll = new JScrollPane(fileContents);
      content.add(scroll);

      fileName = new JLabel("No file selected...");
      content.add(fileName, BorderLayout.SOUTH);

      pack();
   }
```

Listing 9.34 *(Forts.)*
Ein File-Viewer, der die anzuzeigenden Dateien mit Hilfe eines JFileChoosers auswählt

```
    /**
     * Handles the action-event which forces the
     * application to load and display a file
     */
    public void actionPerformed(ActionEvent e) {
       JFileChooser choose = new JFileChooser();
       int result = choose.showOpenDialog(this);
       if(result == JFileChooser.APPROVE_OPTION) {
          File f = choose.getSelectedFile();
          try {
             BufferedReader rdr = new BufferedReader(
                new FileReader(f));
             StringBuilder sb = new StringBuilder();

             String line = null;
             while(null != (line = rdr.readLine())) {
                sb.append(line + "\n");
             }
             rdr.close();

             fileName.setText(f.getName());
             JTextArea fileContents =
                new JTextArea(sb.toString());
             scroll.setViewportView(fileContents);
          } catch (FileNotFoundException e1) {
             e1.printStackTrace();
          } catch (IOException ignored) {
             ignored.printStackTrace();
          }
       }
    }
}
```

Unser File-Viewer besteht aus drei Komponenten: einem `JButton`, der ein `action`-Event auslösen wird, einer `JTextArea`, die die Anzeige des Datei-Inhalts übernimmt, und einem `JLabel`, das den Dateinamen der ausgewählten Datei anzeigen wird.

An den `JButton` wird die aktuelle Klasse – referenziert durch das Schlüsselwort `this` – als `ActionListener` gebunden:

```
JButton click = new JButton("SELECT A FILE!");
click.addActionListener(this);
```

Wenn nach Anzeige der Applikation auf den Button geklickt wird, wird dessen `action`-Ereignis geworfen und kann von der Methode `actionPerformed()` der `ActionListener`-Implementierung behandelt werden.

Innerhalb dieser Methode erfolgt die Auswahl der anzuzeigenden Datei mit Hilfe einer `JFileChooser`-Instanz. Nach deren Deklaration können wir ihre Methode `shopOpenDialog()` aufrufen und damit eine Datei-Auswahl des Benutzers erzwingen:

```
JFileChooser choose = new JFileChooser();
int result = choose.showOpenDialog(this);
```

Der Parameter der `showOpenDialog()`-Methode entspricht dem aufrufenden Objekt. Sollte dies nicht übergeben werden, kann auch `null` als Parameter übergeben werden.

Die Rückgabe von `showOpenDialog()` ist ein `int`-Wert, der einen der drei möglichen Zustände kennzeichnen kann. Sollte dieser Wert dem Wert der Konstanten `JFileChooser.APPROVE_OPTION` entsprechen, ist eine Datei ausgewählt worden.

Eine Referenz auf ein `File`-Objekt, das die ausgewählte Datei repräsentiert, erhalten wir durch die Methode `getSelectedFile()` des `JFileChoosers`:

```
File f = choose.getSelectedFile();
```

Der Inhalt der durch diese `File`-Instanz repräsentierten Datei kann nun mit Hilfe eines `BufferedReaders` eingelesen werden. Zur Erzeugung des `BufferedReaders` übergeben wir diesem eine `FileReader`-Instanz, deren Konstruktor seinerseits die `File`-Instanz als Parameter zugewiesen bekommt:

```
BufferedReader rdr = new BufferedReader(new FileReader(f));
```

Der Datei-Inhalt wird in eine `StringBuilder`-Instanz zeilenweise eingelesen – und zwar so lange, bis das Datei-Ende ereicht worden ist. Anschließend weisen wir den Dateinamen der eingelesenen Datei dem `JLabel fileName` zu:

```
fileName.setText(f.getName());
```

Eine neue `JTextArea`-Instanz nimmt den Inhalt der Datei auf. Sie wird dem `JScrollPane` mit Hilfe seiner `setViewportView()`-Methode zugewiesen. Dies sorgt dafür, dass entsprechend der Größe des eingelesenen Inhalts Scrollbalken angezeigt werden, falls dies nötig wird:

```
JTextArea fileContents = new JTextArea(sb.toString());
scroll.setViewportView(fileContents);
```

Führt der Nutzer die Applikation aus, kann er auf den Button im oberen Bereich klicken. Dann öffnet sich das Datei-Auswahlfenster:

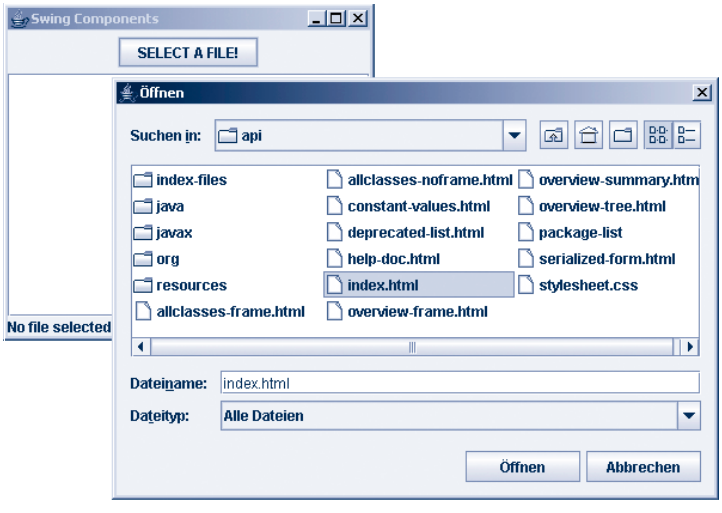
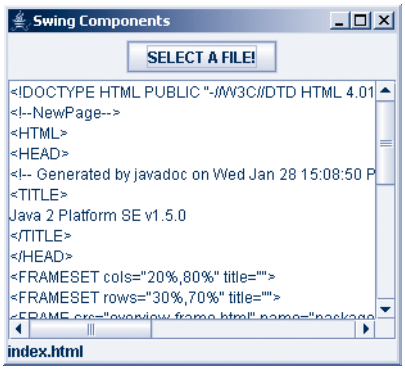


Abbildung 9.47
Auswahl einer Datei mit Hilfe eines JFileChoosers

Nach der Auswahl einer Datei wird deren Inhalt dann im mittleren Bereich der Applikation angezeigt:

Abbildung 9.48
Anzeige einer Datei im File-Viewer



9.5.12 JSlider

Mit Hilfe eines `JSliders` können Nutzer numerische Werte zwischen einem Minimum- und einem Maximum-Wert auswählen. Eigene Eingaben sind nicht möglich, dafür kann auf optisch ansprechende und intuitive Art eine Auswahl getroffen werden.

Die `JSlider`-Klasse besitzt einen mehrfach überladenen Konstruktor, der unter anderem eine Zuweisung von Ausrichtung, Minimal- und Maximal-Wert sowie des Startwerts zulässt:

```
public JSlider()
public JSlider(int orientation)
public JSlider(int min, int max)
public JSlider(int min, int max, int value)
public JSlider(int orientation, int min, int max, int value)
```

Auf das Ändern der Position des Sliders kann mit Hilfe von `ChangeListener`-Implementierungen reagiert werden. Das Auslesen des Wertes einer `JSlider`-Instanz erfolgt mit Hilfe ihrer Methode `getValue()`, die einen `int`-Wert zurückgibt.

Die Klasse erlaubt es uns, festzulegen, ob eine Anzeige einer Skala und der zugehörigen Werte erwünscht ist. Ebenfalls sind wir in der Lage, anzugeben, in welchen Schritten eine Erhöhung beziehungsweise Verringerung des Slider-Werts erfolgt.

Sehen wir uns an, wie wir einen funktionsfähigen `JSlider` implementieren könnten:

Listing 9.35
Auswahl von numerischen Werten mit Hilfe einer JSlider-Instanz

```
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;
import javax.swing.*;
import java.awt.*;

/**
 * Displays a JSlider for the selection of numeric values
 */
```

Listing 9.35 *(Forts.)*
Auswahl von numerischen Werten mit Hilfe einer JSlider-Instanz

```
public class Slider extends BaseFrame implements ChangeListener {

   JLabel amount;
   JSlider slider;

   Slider() {
      JPanel content = new JPanel(new GridLayout(2, 1));
      content.setBorder(
         BorderFactory.createEmptyBorder(10, 10, 10, 10));
      add(content);

      amount = new JLabel("", SwingConstants.CENTER);
      amount.setFont(new Font("Sans-Serif", Font.ITALIC, 24));
      content.add(amount);

      slider = new JSlider(
         SwingConstants.HORIZONTAL, 0, 100, 1);
      slider.setBorder(
         BorderFactory.createEmptyBorder(10, 0, 0, 0));

      slider.setMajorTickSpacing(10);
      slider.setMinorTickSpacing(2);
      slider.setSnapToTicks(true);
      slider.setPaintLabels(true);
      slider.setPaintTicks(true);
      slider.addChangeListener(this);

      content.add(slider);

      pack();
   }

   /**
    * Handles the changed-event of a JSlider-instance
    */
   public void stateChanged(ChangeEvent e) {
      amount.setText(String.valueOf(slider.getValue()));
   }
}
```

Im Konstruktor übergeben wir der `JSlider`-Instanz Informationen über dessen Ausrichtung sowie seine Minimal-, Maximal- und Initial-Werte:

```
slider = new JSlider(SwingConstants.HORIZONTAL, 0, 100, 1);
```

Um eine vertikale Ausrichtung der `JSlider`-Instanz zu erreichen, können Sie ihr entweder im Konstruktor den Wert von `SwingConstants.VERTICAL` übergeben oder diesen Wert alternativ seiner Methode `setOrientation()` zuweisen.

Mit Hilfe der beiden Methoden `setMajorTickSpacing()` und `setMinorTickSpacing()` können Sie festlegen, in welchem numerischen Werte-Intervall die großen und kleinen Werte-Anzeigen dargestellt werden:

```
slider.setMajorTickSpacing(10);
slider.setMinorTickSpacing(2);
```

Hier wurde festgelegt, dass alle zehn Einheiten ein großer Strich und alle zwei Einheiten ein kleiner Strich gesetzt wird. Dies bedeutet: Bei 2, 4, 6 und 8 wird eine kleine und bei 0, 10, 20, 30 oder 40 wird eine große Markierung angezeigt.

Wenn wir der Methode `setSnapToTicks()` den Wert `true` übergeben, kann der Slider nur Werte annehmen, die einem Vielfachen der von `setMajorTickSpacing()` und `setMinorTickSpacing()` festgelegten Werte entsprechen:

```
slider.setSnapToTicks(true);
```

Die Anzeige der Markierungen und der durch die Haupt-Markierungen repräsentierten Werte erfolgt nur, wenn wir den Methoden `setPaintTicks()` (für die Markierungen) und `setPaintLabels()` (für die Werte) jeweils `true` übergeben:

```
slider.setPaintLabels(true);
slider.setPaintTicks(true);
```

Um auf Änderungen des Wertes des Sliders zu reagieren, kann ein `ChangeListener` an die `JSlider`-Instanz gebunden werden. Das Interface `ChangeListener` definiert dabei eine Methode `stateChanged()`:

```
void stateChanged(ChangeEvent e)
```

Klassen, die dieses Interface implementieren, können mit Hilfe von `addChangeListener()` an die `JSlider`-Instanz gebunden werden:

```
slider.addChangeListener(this);
```

Innerhalb der Methode `stateChanged()` der `ChangeListener`-Implementierung kann nun der Wert des `JSliders` ausgelesen und verarbeitet werden. Dies geschieht mit Hilfe seiner Methode `getValue()`, die einen `int`-Wert zurückgibt:

```
amount.setText(String.valueOf(slider.getValue()));
```

Nutzer, die die Applikation ausführen, werden diese Anzeige erhalten:

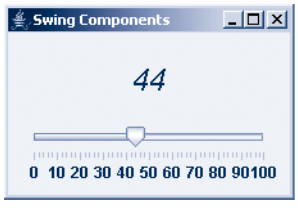


Abbildung 9.49
Auswahl eines numerischen Wertes mit Hilfe einer JSlider-Instanz

9.5.13 JProgressBar

Die `JProgressBar` ist eine Komponente, die zwei Aufgaben haben kann: Sie kann einen numerischen Fortschritt (etwa eine Prozent-Angabe) darstellen oder einen lang laufenden Prozess kennzeichnen, indem sie in den so genannten indeterminanten Modus versetzt wird.

Der Konstruktor der JProgressBar ist mehrfach überladen:

```
public JProgressBar()
public JProgressBar(int orient)
public JProgressBar(int min, int max)
public JProgressBar(int orient, int min, int max)
```

Der derart überladene Konstruktor erlaubt die Zuweisung von Start- und End-Wert sowie der Ausrichtung der Komponente.

Wenn wir mit der Standard-Form einer Progress-Bar arbeiten wollen, müssen wir Start- und Endwert definieren. Dies geschieht in der Regel gleich bei deren Instanzierung. Anschließend sind wir in der Lage, mit Hilfe der Methode `setValue()` den jeweils aktuellen Wert zuzuweisen:

Listing 9.36
Einsatz einer JProgressBar

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

/**
 * Displays a JProgressBar-instance
 */
public class ProgressBar extends BaseFrame
   implements ActionListener {

   JProgressBar bar;
   JButton button;
   Timer timer;

   /**
    * Creates a JLabel, a JButton and a JProgressBar
    */
   ProgressBar() {
      JPanel content = new JPanel(new GridLayout(3, 1));
      add(content);

      content.add(new JLabel("Current progress:"),
         SwingConstants.CENTER);

      bar = new JProgressBar(0, 200);
      bar.setValue(bar.getMinimum());
      bar.setStringPainted(true);
      content.add(bar);

      button = new JButton("Start");
      button.addActionListener(this);
      content.add(button);

      pack();
   }

   /**
    * Performs the action-events of a JButton and a Timer
    */
   public void actionPerformed(ActionEvent e) {
      if(e.getSource() instanceof JButton) {
         button.setEnabled(false);
         timer = new Timer(100, this);
         timer.start();
      } else {
         int current = bar.getValue();
```

Listing 9.36 *(Forts.)*
Einsatz einer JProgressBar

```
            if(current < bar.getMaximum()) {
               bar.setValue(current + 1);
            } else {
               timer.stop();
            }
         }
      }
   }
```

In diesem Beispiel deklarieren wir einen `JButton` und eine `JProgressBar`-Instanz. Der Button wird einen `Timer` starten, der der Progress-Bar alle hundert Millisekunden einen neuen Wert zuweisen lassen wird, denn ebenso wie ein Button wirft auch ein Timer ein `action`-Ereignis – nur eben auf regelmäßiger Basis.

Die Erzeugung der `JProgressBar`-Instanz beinhaltet die Übergabe von Minimal- und Maximalwert:

```
bar = new JProgressBar(0, 200);
```

Um einen definierten Zustand zu erreichen, wird zunächst der aktuelle Wert auf den Minimal-Wert gesetzt, den wir mit Hilfe der Methode `getMinimum()` der `JProgressBar`-Instanz ermitteln können:

```
bar.setValue(bar.getMinimum());
```

Eine `JProgressBar` zeigt in der Default-Einstellung einzig den Fortschrittsbalken an. Um auch eine Anzeige des Prozent-Werts zu erreichen, weisen wir der Methode `setStringPainted()` den Wert `true` zu:

```
bar.setStringPainted(true);
```

Nun kann die Applikation angezeigt werden. Klickt der Nutzer auf die Schaltfläche zum Starten der Progress-Bar, wird die Methode `actionPerformed()` des `ActionListener`-Interfaces eingebunden. Hier erfolgt zuerst eine Prüfung, ob der Auslöser des Ereignisses eine `JButton`-Instanz oder der `Timer` war. Sollte es sich um die Schaltfläche handeln, wird diese deaktiviert und der `Timer` gestartet.

Nun wird alle einhundert Millisekunden erneut ein `action`-Ereignis geworfen – diesmal aber ist der `Timer` der Auslöser. Also ermitteln wir jetzt zunächst den angezeigten Wert der Progress-Bar mit Hilfe ihrer Methode `getValue()`:

```
int current = bar.getValue();
```

Sollte dieser Wert kleiner als der Maximalwert sein, wird er der Progress-Bar mit ihrer Methode `setValue()` um eins erhöht wieder zugewiesen:

```
int current = bar.getValue();
if(current < bar.getMaximum()) {
   bar.setValue(current + 1);
} else {
   timer.stop();
}
```

Wenn der Maximalwert, der bei der Instanzierung zugewiesen worden ist, erreicht wurde, wird die Ausführung des Timers beendet und die `JProgressBar`-Instanz bekommt keinen neuen Wert mehr zugewiesen.

Nutzer, die die Applikation ausführen, werden nach einem Klick auf die Schaltfläche „Start“ diese Anzeige zu sehen bekommen:

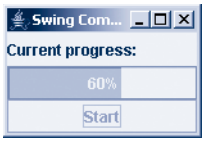


Abbildung 9.50
JProgressBar in Aktion

Unendliche Progress-Bar

Die zweite Variante der `JProgressBar` ist die so genannte indeterminante Progress-Bar. Sie wird dann eingesetzt, wenn ein Maximal-Wert nicht bekannt ist. Aus diesem Grund zeigt sie auch keinen Fortschrittsbalken, sondern eine wandernde Markierung an.

Den indeterminanten Modus einer `JProgressBar`-Instanz schalten wir ein, indem wir ihrer Methode `setIndeterminate()` den Wert `true` übergeben. Weitere Konfigurationen sind nicht nötig – insofern ist der Einsatz dieser Variante der `JProgressBar` sogar noch einfacher als der Einsatz ihrer endlichen Schwester:

```
JProgressBar bar = new JProgressBar();
bar.setIndeterminate(true);
```

Eine weitere Beeinflussung der Progress-Bar ist nicht sinnvoll. Nutzer, die eine Applikation mit dieser Art der `JProgressBar` ausführen, werden eine Ausgabe analog zu dieser erhalten:

Abbildung 9.51
Unendliche JProgressBar

9.6 Menüs mit JMenu

`JMenus` sind Komponenten, die per Definition innerhalb einer `JMenuBar` angezeigt werden. Diese `JMenuBar` wird in der Regel am Kopf der Applikation angezeigt, wobei diese Anzeigeposition Betriebssystem-abhängig ist. Eine `JMenu`-Komponente kann ihrerseits wieder andere Komponenten aufnehmen – dies bedeutet, dass wir in der Lage sind, innerhalb von Menüs auch RadioButtons oder Check-Boxen anzuzeigen.

Um ein `JMenuBar`-Element zu befüllen, müssen wir `JMenu`-Instanzen erzeugen. Eine `JMenu`-Instanz repräsentiert einen Menü-Eintrag, der untergeordnete Menüpunkte enthält. Die einzelnen Menüpunkte werden ihrerseits durch `JMenuItem`-Instanzen repräsentiert.

Ein Klick auf den Menü-Eintrag sorgt meist für das Werfen eines `action`-Events – wobei dies von der Art der enthaltenen Komponente abhängig ist. Mit Hilfe ihrer Methode `addActionListener()` können wir Event-Handler an das `action`-Ereignis binden. Die Methode `setActionCommand()` legt fest, was als Parameter an den Event-Handler übergeben werden soll.

Für JMenu- und JMenuItem-Instanzen können Mnemonics definiert werden. Ebenfalls sind wir in der Lage, Separatoren zu erzeugen und innerhalb von Menüs anzuzeigen.

9.6.1 Erzeugen von Menüs

Sehen wir uns im Folgenden an, wie wir Menüs erzeugen können. Zum Einsatz kommen hier JMenu- und JMenuItem-Instanzen, deren Konstruktoren stets auch über eine Überladung verfügen, die einen String für den Anzeigetext des Menüs als Parameter akzeptieren:

```
public JMenu()
public JMenu(String s)
public JMenuItem()
public JMenuItem(Icon icon)
public JMenuItem(String text)
public JMenuItem(String text, Icon icon)
public JMenuItem(String text, int mnemonic)
```

Dem Konstruktor der JMenuItem können wir neben dem Anzeige-Text auch ein Icon oder einen Mnemonic (ein Tastenkürzel) übergeben.

Wir werden hier nun einen Haupt-Menüpunkt mit mehreren untergeordneten Menüpunkten und einem Sub-Menü erzeugen. Bei Klick auf die untergeordneten Menüpunkte werden action-Events geworfen, die wir auffangen und protokollieren werden:

Listing 9.37
Erzeugen einer Menü-Struktur und Behandeln ihrer Events

```
import javax.swing.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.KeyEvent;
import java.awt.*;

/**
 * Displays a JMenu
 */
public class Menu extends BaseFrame
   implements ActionListener {

   JScrollPane scroll;
   JTextArea log;

   /**
    * Creates a Menu and adds ActionListener to the menu's entries
    */
   Menu() {
      log = new JTextArea(10, 30);
      scroll = new JScrollPane(log);
      scroll.setPreferredSize(new Dimension(200, 100));

      add(scroll);

      JMenuBar menu = new JMenuBar();
      setJMenuBar(menu);

      JMenu fileMenu = new JMenu("File");
```

Listing 9.37 *(Forts.)*
Erzeugen einer Menü-Struktur und Behandeln ihrer Events

```
        fileMenu.setMnemonic(KeyEvent.VK_F);
        menu.add(fileMenu);

        JMenuItem menuItem = new JMenuItem("Open");
        menuItem.addActionListener(this);
        menuItem.setMnemonic(KeyEvent.VK_O);
        fileMenu.add(menuItem);

        JMenu subMenu = new JMenu("Recent files...");
        subMenu.setMnemonic(KeyEvent.VK_R);
        fileMenu.add(subMenu);

        menuItem = new JMenuItem("file.txt");
        menuItem.setMnemonic(KeyEvent.VK_1);
        menuItem.addActionListener(this);
        subMenu.add(menuItem);

        menuItem = new JMenuItem("other.txt");
        menuItem.setMnemonic(KeyEvent.VK_2);
        menuItem.addActionListener(this);
        subMenu.add(menuItem);

        menuItem = new JMenuItem("third.txt");
        menuItem.setMnemonic(KeyEvent.VK_3);
        menuItem.addActionListener(this);
        subMenu.add(menuItem);

        menuItem = new JMenuItem("Create New");
        menuItem.addActionListener(this);
        menuItem.setMnemonic(KeyEvent.VK_N);
        fileMenu.add(menuItem);

        fileMenu.addSeparator();

        menuItem = new JMenuItem("Exit");
        menuItem.addActionListener(this);
        menuItem.setMnemonic(KeyEvent.VK_X);
        fileMenu.add(menuItem);

        pack();
    }

    /**
     * Handles action-events for JMenuItem-instances
     */
    public void actionPerformed(ActionEvent e) {
        JMenuItem item = (JMenuItem)e.getSource();
        log.append("Clicked: " + item.getText() + "\n");

        String text = item.getText();
        if(text == "Exit") {
            setVisible(false);
            System.exit(0);
        }

        scroll.setViewportView(log);
        log.revalidate();
    }
}
```

Der erste Schritt beim Anlegen einer Menü-Struktur ist das Erzeugen des Wurzelpunktes. Dieser Wurzelpunkt wird durch eine Instanz der `JMenuBar`-Klasse repräsentiert, die dem übergeordneten `JFrame` mit Hilfe seiner Methode `setJMenuBar()` zugewiesen werden muss:

```
JMenuBar menu = new JMenuBar();
this.setJMenuBar(menu);
```

Nun können die Menü-Einträge auf der obersten Ebene erzeugt werden. Diese werden in der Regel durch `JMenu`-Instanzen repräsentiert. Wir legen hier einen Haupt-Menü-Eintrag „File“ an, der in einer realen Applikation Zugriff auf verschiedene Datei-Operationen erlauben würde und auch per Tastatur-Kürzel Alt+F aktiviert werden kann:

```
JMenu fileMenu = new JMenu("File");
fileMenu.setMnemonic(KeyEvent.VK_F);
menu.add(fileMenu);
```

Nun können die untergeordneten Menüpunkte angelegt werden. Soweit diese nicht selbst wieder untergeordnete Menüpunkte enthalten sollen, werden sie durch `JMenuItem`-Instanzen repräsentiert:

```
JMenuItem menuItem = new JMenuItem("Open");
menuItem.addActionListener(this);
menuItem.setMnemonic(KeyEvent.VK_O);
fileMenu.add(menuItem);
```

Da auf das Auslösen der einzelnen Menüpunkte reagiert werden soll, müssen wir einen `ActionListener` an sie binden. Da die aktuelle Klasse dieses Interface implementiert, können wir der Methode `addActionListener()` der jeweiligen `JMenuItem`-Instanz eine Referenz auf die aktuelle Klasseninstanz mit Hilfe des Schlüsselwortes *this* übergeben.

Dem Haupt-Menü-Punkt können nun weitere Untermenüs zugeordnet werden. Zu diesem Zweck erzeugen wir eine neue `JMenu`-Instanz, der wir deren Menüpunkte in Form von `JMenuItem`-Instanzen zuweisen. Die `JMenu`-Instanz wird – ebenso wie ein normaler Menüpunkt – an die übergeordnete Instanz gebunden, indem sie deren `add()`-Methode als Parameter übergeben wird:

```
JMenu subMenu = new JMenu("Recent files...");
subMenu.setMnemonic(KeyEvent.VK_R);
fileMenu.add(subMenu);
```

Vor dem letzten Menüpunkt, der für das Schließen der Applikation zuständig sein soll, fügen wir einen optischen Trenner ein. Dies geschieht durch einen einfachen Aufruf der Methode `addSeparator()` der übergeordneten `JMenu`-Instanz:

```
fileMenu.addSeparator();
```

Nachdem wir so alle Menüpunkte erfasst und unsere Menüstruktur erzeugt haben, wird diese innerhalb der Applikation angezeigt:

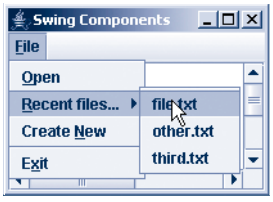


Abbildung 9.52
Anzeige einer Menüstruktur

Wenn der Benutzer nun einen der Menüpunkte anklickt, wird dessen `action`-Ereignis geworfen und die Methode `actionPerformed()` eingebunden. Hier können wir den als generische `Object`-Instanz übergebenen Auslöser wieder zurück in eine `JMenuItem`-Instanz casten:

```
JMenuItem item = (JMenuItem)e.getSource();
```

Den Inhalt der `JTextArea log` ergänzen wir um die Meldung, welche Schaltfläche angeklickt worden ist.

9.6.2 Beenden einer Swing-Applikation

Wenn die Beschriftung der Schaltfläche „Exit“ lautet, werden wir die Applikation beenden. Zu diesem Zweck verbergen wir zuerst deren Hauptfenster und beenden die Java-Instanz dann per `System.exit()`:

```
setVisible(false);
System.exit(0);
```

Leider existiert kein feinerer Weg, eine `Swing`-Applikation zu beenden – nach dem Erkennen, dass die Applikation geschlossen werden soll, können wir zwar noch in Ruhe aufräumen und eventuell andere Bestandteile der Applikation davon in Kenntnis setzen, dass diese sich ebenfalls beenden sollen, ein weniger brutaler Weg als das Beenden der kompletten Java-Instanz existiert aber leider nicht.

Achtung

Bevor Sie eine `Swing`-Applikation per `System.exit()` beenden, sollten Sie sicherstellen, dass eventuell von Ihnen verwendete Ressourcen freigegeben und aufgeräumt sind. Diese bleiben sonst unter Umständen als Leichen im Speicher hängen und können somit die Speicher- und Systemlast unnötig hoch halten.

9.6.3 Anzeigen anderer Menü-Elemente

Statt „normaler“ Buttons können Sie `JMenuItem`-Instanzen auch andere Komponenten zur Anzeige und Auswahl zuweisen. Wenn wir beispielsweise eine Auswahl analog zu `JRadioButtons` innerhalb einer Menü-Struktur erlauben wollen, können wir dafür `JRadioButtonMenuItem`-Instanzen verwenden:

Listing 9.38
Verwenden von JRadioButton-MenuItem-Instanzen

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
 * Displays a JMenu with JRadioButtonMenuItem-instances
 */
public class MenuRadioButtons extends BaseFrame
   implements ActionListener, ItemListener {

   JScrollPane scroll;
   JTextArea log;

   /**
    * Constructs the JMenuBar
    */
   MenuRadioButtons() {

      // ...

      JMenuBar menu = new JMenuBar();
      setJMenuBar(menu);

      JMenu fileMenu = new JMenu("File");
      fileMenu.setMnemonic(KeyEvent.VK_F);
      menu.add(fileMenu);

      // ...

      ButtonGroup bg = new ButtonGroup();
      JRadioButtonMenuItem button = new JRadioButtonMenuItem(
         "Ask before opening new file", true);
      button.addItemListener(this);
      button.setMnemonic(KeyEvent.VK_A);
      bg.add(button);
      fileMenu.add(button);

      button = new JRadioButtonMenuItem("Do not ask");
      button.addItemListener(this);
      button.setMnemonic(KeyEvent.VK_D);
      bg.add(button);
      fileMenu.add(button);

      // ...
   }

   /**
    * Reacts on the action-events thrown by menu items
    */
   public void actionPerformed(ActionEvent e) {
      // ...
   }

   /**
    * Reacts on the item-event of a JRadioButtonMenuItem-instance
    */
```

```
    public void itemStateChanged(ItemEvent e) {
        JRadioButtonMenuItem item =
            (JRadioButtonMenuItem)e.getItem();
        log.append("State changed: \"" +
            item.getText() + "\" is now " +
            item.isSelected() + "\n");
        scroll.setViewportView(log);
        log.revalidate();
    }
}
```

Listing 9.38 *(Forts.)*
Verwenden von JRadioButtonMenuItem-Instanzen

Die Arbeit mit den `JRadioButtonMenuItem`-Instanzen gestaltet sich analog zu der mit `JRadioButtons`: Wir müssen eine `ButtonGroup`-Instanz deklarieren und können danach die einzelnen Buttons erzeugen. Diese werden der `ButtonGroup` zugewiesen – und die `ButtonGroup` sorgt dann dafür, dass auch tatsächlich nur jeweils eine Instanz aktiviert ist:

```
ButtonGroup bg = new ButtonGroup();
JRadioButtonMenuItem button = new JRadioButtonMenuItem(
    "Ask before opening new file", true);
// ...
bg.add(button);
```

Analog zu `JRadioButtons` werfen `JRadioButtonMenuItem`-Instanzen ebenfalls ein `item`-Ereignis, wenn sie aktiviert oder deaktiviert werden. Dieses kann von `ItemListener`-Implementierungen aufgefangen und behandelt werden. Zuvor muss der `ItemListener` (in diesem Fall die aktuelle Klasse) aber an das `JRadioButtonMenuItem`-Element gebunden werden:

```
button.addItemListener(this);
```

Nutzer, die die Applikation aufrufen, können nun im Menü eine Auswahl mit Hilfe von RadioButtons treffen:

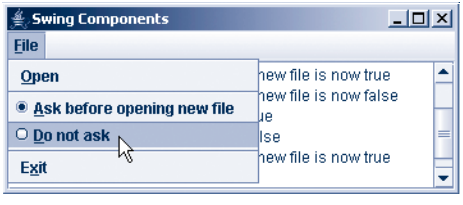


Abbildung 9.53
Auswahl mit Hilfe von JRadioButtonMenuItem-Instanzen

Wird eine der Optionen angeklickt, werfen gleich zwei `JRadioButtonMenuItem`-Instanzen das `item`-Ereignis: Die Instanz, die deaktiviert worden ist, und die neu aktivierte Instanz. Die Events werden in der vom Interface `ItemListener` definierten Methode `itemStateChanged()` behandelt.

9.6.4 CheckBoxen in Menüs verwenden

Der Einsatz von CheckBoxen in Menüs gestaltet sich ähnlich wie die Verwendung von `JRadioButtonMenuItems`. Die dafür zu verwendende Komponente ist `JCheckBoxMenuItem`, deren Konstruktor so überladen ist, dass wir ihr schon beim Erzeugen einen Text oder ein Icon zuweisen können:

```
public JCheckBoxMenuItem()
public JCheckBoxMenuItem(Icon icon)
public JCheckBoxMenuItem(String text)
public JCheckBoxMenuItem(String text, Icon icon)
public JCheckBoxMenuItem(String text, boolean b)
public JCheckBoxMenuItem(String text, Icon icon, boolean b)
```

Besonders praktisch ist der Einsatz der Konstruktoren, denen wir gleich beim Erzeugen einen boolschen Parameter übergeben können, denn damit können wir festlegen, ob die CheckBox aktiviert ist oder nicht.

Ebenfalls können wir per `setMnemonic()` Tastenkürzel mit Hilfe der in der Klasse `KeyEvent` definierten Konstanten festlegen und per `addItemListener()` Event-Handler für das `itemChanged`-Ereignis binden.

Die Klasse `JCheckBoxMenuItem` ist somit von der Handhabung her eine Mischung aus `JCheckBox` und `JMenuItem`:

Listing 9.39 Verwendung von JCheckBox-MenuItem-Instanzen für eine Auswahl in einem Menü

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
 * Displays a JMenu with JCheckBoxMenuItem-instances
 */
public class MenuCheckBoxes extends BaseFrame
   implements ActionListener, ItemListener {

   JScrollPane scroll;
   JTextArea log;

   /**
    * Constructs the JMenuBar
    */
   MenuCheckBoxes() {

      // ...

      JMenuBar menu = new JMenuBar();
      setJMenuBar(menu);

      JMenu fileMenu = new JMenu("File");
      fileMenu.setMnemonic(KeyEvent.VK_F);
      menu.add(fileMenu);

      // ...

      JCheckBoxMenuItem check = new JCheckBoxMenuItem(
         "Ask before opening a new file", true);
      check.setMnemonic(KeyEvent.VK_A);
      check.addItemListener(this);
      fileMenu.add(check);

      check = new JCheckBoxMenuItem(
         "Ask before closing an edited file");
      check.setMnemonic(KeyEvent.VK_C);
      check.addItemListener(this);
      fileMenu.add(check);
```

Listing 9.39 *(Forts.)*
Verwendung von JCheckBox-MenuItem-Instanzen für eine Auswahl in einem Menü

```
      // ...
   }

   /**
    * Reacts on the action-events thrown by menu items
    */
   public void actionPerformed(ActionEvent e) {
      // ...
   }

   /**
    * Reacts on the item-event of a JCheckBoxMenuItem-instance
    */
   public void itemStateChanged(ItemEvent e) {
      JCheckBoxMenuItem item = (JCheckBoxMenuItem)e.getItem();
      log.append("State changed: \"" +
         item.getText() + "\" is now " +
         item.isSelected() + "\n");
      scroll.setViewportView(log);
      log.revalidate();
   }
}
```

Das Erzeugen und Zuweisen von `JCheckBoxMenuItem`-Instanzen gestaltet sich ähnlich zu dem von gewöhnlichen `JCheckBox`-Objekten und weist ebenfalls starke Ähnlichkeiten mit der Verwendung von `JRadioButtonMenuItem`-Instanzen auf. Der einzige wesentliche Unterschied beim Erzeugen von `JCheckBoxMenuItem`-Instanzen gegenüber `JRadioButtonMenuItems` besteht darin, dass auf die für die Organisation von RadioButtons benötigte `ButtonGroup` verzichtet werden kann:

```
JCheckBoxMenuItem check = new JCheckBoxMenuItem(
   "Ask before opening a new file", true);
check.setMnemonic(KeyEvent.VK_A);
check.addItemListener(this);
fileMenu.add(check);
```

Durch das Binden eines `ItemListeners` können wir auf Zustandsänderungen der CheckBox reagieren. Wichtig ist, dass wir innerhalb der Methode `itemStateChanged()` zunächst die Quelle des Ereignisses wieder in eine `JCheckBoxMenuItem`-Instanz zurückcasten:

```
JCheckBoxMenuItem item = (JCheckBoxMenuItem)e.getItem();
```

In diesem Fall beschränkt sich unsere Reaktion darauf, deren Anzeige-Text und Auswahl-Wert auszulesen und zur Anzeige zu bringen:

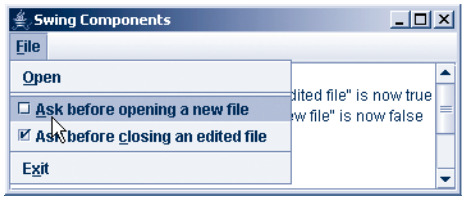


Abbildung 9.54
Verwenden von JCheckBox-MenuItem-Instanzen

9.6.5 JPopupMenu

Im Grunde sind Popup-Menüs auch nichts anderes als gewöhnliche Menü-Instanzen. Einzig die Art der Darstellung ändert sich – und, dass wir selbst für die Anzeige des Menüs verantwortlich sind. Ein Popup-Menü (und damit auch eine `JPopupMenu`-Instanz) erscheint in der Regel, wenn der Nutzer die rechte Maustaste innerhalb des Applikations-Fensters betätigt. Für uns bedeutet dies, dass wir nun auch Maus-Ereignisse abfangen müssen. Dies kann auf zwei Arten erfolgen:

- Implementieren des Interface `MouseListener`
- Ableiten von der abstrakten Klasse `MouseAdapter`

Wenn wir – wie in unserem Fall – nicht auf alle Maus-Events reagieren und deshalb auch nicht alle vom Interface `MouseListener` definierten Methoden implementieren wollen, sollten wir dem zweiten Ansatz nachgehen und eine Klasse verwenden, die von `MouseAdapter` erbt. Dadurch sind wir in der komfortableren Situation, nur die Methoden überschreiben zu müssen, die uns tatsächlich auch interessieren:

- `mousePressed()`: Behandelt das Ereignis des Drucks auf eine Maustaste
- `mouseReleased()`: Wird eingebunden, wenn die Maustaste wieder losgelassen worden ist

Hier können wir nun überprüfen, ob es sich beim Maus-Event um ein Event handelt, das für die Darstellung eines Popup-Menüs verantwortlich ist. Diese Überprüfung hat Java natürlich schon vorgenommen – wir können deren Ergebnis der Eigenschaft `isPopupTrigger()` der `MouseEvent`-Instanz, die den beiden Event-Handlern als Parameter übergeben wird, entnehmen.

Sehen wir uns an, wie eine Umsetzung eines Popup-Menüs aussehen könnte:

Listing 9.40
Anzeige eines Popup-Menüs

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

/**
 * Displays a JPopupMenu
 */
public class PopupMenu extends BaseFrame
   implements ActionListener {

   JScrollPane scroll;
   JTextArea log;
   JPopupMenu menu;

   /**
    * Constructs the application
    */
   PopupMenu() {
      JPanel content = new JPanel(new BorderLayout());
      content.setBorder(
         BorderFactory.createEmptyBorder(10, 10, 10, 10));

      add(content);
      addMouseListener(new PopupListener());
```

Listing 9.40 *(Forts.)*
Anzeige eines Popup-Menüs

```
    log = new JTextArea(10, 30);
    scroll = new JScrollPane(log);
    scroll.setPreferredSize(new Dimension(200, 100));
    content.add(scroll);

    menu = new JPopupMenu();
    JMenuItem menuItem = new JMenuItem("Open");
    menuItem.addActionListener(this);
    menuItem.setMnemonic(KeyEvent.VK_O);
    menu.add(menuItem);

    JMenu subMenu = new JMenu("Recent files...");
    subMenu.setMnemonic(KeyEvent.VK_R);
    menu.add(subMenu);

    menuItem = new JMenuItem("file.txt");
    menuItem.setMnemonic(KeyEvent.VK_1);
    menuItem.addActionListener(this);
    subMenu.add(menuItem);

    menuItem = new JMenuItem("other.txt");
    menuItem.setMnemonic(KeyEvent.VK_2);
    menuItem.addActionListener(this);
    subMenu.add(menuItem);

    menuItem = new JMenuItem("third.txt");
    menuItem.setMnemonic(KeyEvent.VK_3);
    menuItem.addActionListener(this);
    subMenu.add(menuItem);

    menuItem = new JMenuItem("Create New");
    menuItem.addActionListener(this);
    menuItem.setMnemonic(KeyEvent.VK_N);
    menu.add(menuItem);

    menu.addSeparator();

    menuItem = new JMenuItem("Exit");
    menuItem.addActionListener(this);
    menuItem.setMnemonic(KeyEvent.VK_X);
    menu.add(menuItem);

    pack();
}

/**
 * Handles the action-event of the menu items
 */
public void actionPerformed(ActionEvent e) {
    JMenuItem item = (JMenuItem)e.getSource();
    log.append("Clicked: " + item.getText() + "\n");

    String text = item.getText();
    if(text == "Exit") {
        setVisible(false);
        System.exit(0);
    }
```

Listing 9.40 *(Forts.)*
Anzeige eines Popup-Menüs

```
        scroll.setViewportView(log);
        log.revalidate();
    }

    /**
     * Decides when to display the popup-menu
     */
    private class PopupListener extends MouseAdapter {

        public void mousePressed(MouseEvent e) {
            displayPopup(e);
        }

        public void mouseReleased(MouseEvent e) {
            displayPopup(e);
        }

        /**
         * Displays the popup
         */

        private void displayPopup(MouseEvent e) {
            if(e.isPopupTrigger()) {
                menu.show(
                    e.getComponent(), e.getX(), e.getY());
            }
        }
    }
}
```

Innerhalb des Konstruktors der Klasse wird das Popup-Menü deklariert und außerdem die Bindung des `MouseAdapters` an die Anzeige-Elemente vorgenommen. Wir binden eine `MouseAdapter`-Instanz an das Basis-`JFrame`-Element, so dass das Popup-Menü später beim Klick auf eine freie Stelle der Applikation erscheinen wird:

```
addMouseListener(new PopupListener());
```

Die Klasse `PopupListener`, von der hier eine neue Instanz als Parameter übergeben wird, erbt von der abstrakten Klasse `MouseAdapter`. Diese stellt ihrerseits alle vom Interface `MouseListener` definierten Methoden in Form von Basis-Implementierungen zur Verfügung:

```
public void mouseClicked(MouseEvent e)
public void mousePressed(MouseEvent e)
public void mouseReleased(MouseEvent e)
public void mouseEntered(MouseEvent e)
public void mouseExited(MouseEvent e)
```

Die Methode `mouseClicked()` wird eingebunden, wenn ein Maus-Klick mit der linken Maustaste stattgefunden hat. Sobald eine der Maus-Tasten gedrückt worden ist, wird `mousePressed()` aufgerufen und `mouseReleased()` wird eingebunden, wenn die Taste wieder losgelassen wird. Bewegt sich die Maus über eine Komponente, wird das Ereignis `mouseEntered()` geworfen – beim Verlassen dagegen `mouseExited()`.

Wir müssen in ableitenden Klassen nun nur die Methoden überschreiben, die uns wirklich interessieren. Zur Umsetzung der `PopupListener`-Klasse kommen wir gleich – zuvor allerdings sollten wir uns ansehen, wie das Popup-Menü selbst deklariert wird:

```
menu = new JPopupMenu();
```

Tatsächlich ist dies bereits alles! Wir müssen die `JPopupMenu`-Instanz nirgendwo zuweisen oder hinzufügen – sie muss nur global genug verfügbar sein, dass sie später von der untergeordneten Instanz der `PopupListener`-Klasse erreicht und angezeigt werden kann.

Einzelne Menü-Elemente weisen wir wie gewohnt in Form von `JMenuItem`- und `JMenu`-Instanzen zu. Selbiges gilt für den Einsatz von Trennern, die per `addSeparator()` erzeugt werden:

```
JMenuItem menuItem = new JMenuItem("Open");
// ...
menu.add(menuItem);
// ...
JMenu subMenu = new JMenu("Recent files...");
// ...
menu.add(subMenu);
// ...
menu.addSeparator();
```

Wenn die Applikation nun angezeigt wird und der Nutzer mit der rechten Maustaste in einen freien Bereich des Fensters klickt, wird die bereits angesprochene `PopupListener`-Instanz mit ihren überschriebenen Methoden `mousePressed()` und `mouseReleased()` eingebunden:

```
public void mousePressed(MouseEvent e) {
   displayPopup(e);
}

public void mouseReleased(MouseEvent e) {
   displayPopup(e);
}
```

Beide Methoden rufen ihrerseits die Methode `displayPopup()` auf und übergeben dieser dabei die `MouseEvent`-Instanz `e`, deren Informationen nun ausgewertet werden.

Interessant ist hier die Eigenschaft `isPopupTrigger()`. Sollte diese den Wert `true` haben, können wir das Popup-Menü anzeigen lassen. Dies geschieht, indem wir dessen Methode `show()` aufrufen. Dabei werden folgende Informationen übergeben:

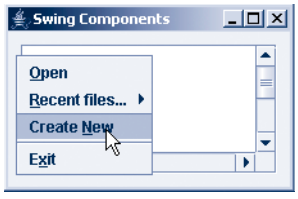
- Komponente, in deren Kontext das Popup-Menü angezeigt werden soll (in der Regel die Komponente, auf die der Nutzer den Rechtsklick gesetzt hat)
- X- und Y-Position der linken oberen Ecke des Popup-Menüs. Diese Werte entsprechen den X- und Y-Positionen des Mauszeigers zum Zeitpunkt des Klicks

Auf die rechtsgeklickte Komponente greifen wir mit Hilfe der Methode `getComponent()` der `MouseEvent`-Instanz zu. Die Methoden `getX()` und `getY()` liefern die benötigten Positions-Angaben:

```
menu.show(e.getComponent(), e.getX(), e.getY());
```

Wenn der Nutzer die Applikation ausführt und einen Rechtsklick auf den freien Bereich am Rand des Fensters macht, wird er unser Popup-Menü eingeblendet bekommen:

Abbildung 9.55
Anzeige eines JPopupMenus



9.7 Layout-Manager

Die Container-Elemente im `Swing`-/`JFC`-Framework – allen voran das `JPanel` – unterstützen verschiedene Layout-Manager, die festlegen, auf welche Art der enthaltene Inhalt angezeigt werden soll. Werfen wir im Folgenden einen Blick auf die gebräuchlichsten Swing-Layout-Manager – die im Übrigen alle von einer Basisklasse `LayoutManager` erben und deshalb recht einfach gegeneinander ausgetauscht werden können.

Das Zuweisen eines Layout-Managers zu einem Container sollte immer vor dem Zuweisen von Elementen erfolgen – berücksichtigen Sie dies nicht, erzeugen Sie im besten Fall unnötigen Overhead für die Applikation und im schlimmsten Fall ist die Darstellung nicht korrekt. Die eigentliche Zuweisung der `LayoutManager`-Instanz erfolgt mit Hilfe der Methode `setLayout` des jeweiligen Containers.

9.7.1 FlowLayout

Das `FlowLayout` ist das Standard-Layout aller Container. In ihm werden die einzelnen Elemente hintereinander angeordnet und nur in eine neue Zeile umgebrochen, wenn die maximale Breite des Containers erreicht ist.

Wenn Sie das `FlowLayout` einsetzen wollen, müssen Sie einem Container nichts zusätzlich übergeben – weisen Sie diesem einfach die darzustellenden Komponenten mit Hilfe seiner `add()`-Methode zu:

```
JPanel content = new JPanel();
add(content);

for(int i=1; i<6; i++) {
   JButton button = new JButton("Button #" +i);
   content.add(button);

}
```

Die so zugeordneten `JButton`-Instanzen werden nebeneinander dargestellt:

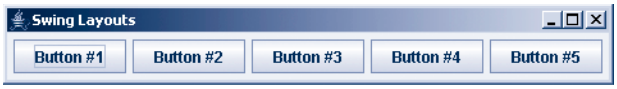


Abbildung 9.56
Verwenden des FlowLayouts

9.7.2 GridLayout

Das `GridLayout` erlaubt die Definition eines Layouts, das an eine Tabelle erinnert: Es besteht aus Zeilen und Spalten. Das `GridLayout` wird sequentiell befüllt, wobei die Reihenfolge von links nach rechts und dann von oben nach unten ist:

```
JPanel content = new JPanel(new GridLayout(3, 2));
add(content);

for(int i=1; i<7; i++) {
   JButton button = new JButton("Button #" + i);
   content.add(button);
}
```

Die dem `JPanel` zugewiesene `java.awt.GridLayout`-Instanz wurde mit drei Zeilen und zwei Spalten definiert. Wenn die Elemente zugewiesen werden, erfolgt die Zuweisung zunächst in der obersten Zeile und erst, wenn dort alle Spalten befüllt worden sind, wird in die nächste Zeile gewechselt:

Abbildung 9.57
Die Buttons wurden per GridLayout angeordnet

9.7.3 BorderLayout

Das `BorderLayout` eignet sich sehr gut, wenn Elemente an den vier Seiten und/oder in der Mitte angeordnet werden sollen. Die `BorderLayout`-Klasse definiert vier Himmelsrichtungen und einen zentralen Bereich:

- `BorderLayout.NORTH` (Oben)
- `BorderLayout.SOUTH` (Unten)
- `BorderLayout.WEST` (Links)
- `BorderLayout.EAST` (Rechts)
- `BorderLayout.CENTER` (Mitte)

Der mittlere Bereich, der als `BorderLayout.CENTER` definiert ist, wird dabei immer so groß wie möglich dargestellt, während die vier Randbereiche so klein als möglich angezeigt werden.

Die Zuweisung von Elementen zu den vier Himmelsrichtungen erfolgt wie üblich mit Hilfe der Methode `add()` des Containers, allerdings unter Angabe der Konstante, deren Wert die Himmelsrichtung repräsentiert, als zweiten Parameter:

In der Regel werden Sie dem zentralen Bereich eines `BorderLayouts` keine Komponente direkt zuweisen. Dies macht nur Sinn, wenn Sie in der Mitte tatsächlich nur ein Element anzeigen wollen. Normalerweise erfolgt die Zuweisung einer `JPanel`-Instanz, die die untergeordneten Komponenten aufnehmen wird.

```
import java.awt.*;
import javax.swing.*;

JPanel content = new JPanel(new BorderLayout());
add(content);

JButton button = new JButton("North");
content.add(button, BorderLayout.NORTH);

button = new JButton("South");
content.add(button, BorderLayout.SOUTH);

button = new JButton("West");
content.add(button, BorderLayout.WEST);

button = new JButton("East");
content.add(button, BorderLayout.EAST);

button = new JButton("Center");
content.add(button);
```

Sämtliche Buttons, die nicht im zentralen Bereich des Layouts liegen, müssen unter Angabe der Region zugewiesen werden, der sie zugeordnet werden sollen. Einzig bei der Zuweisung zum zentralen Bereich kann darauf verzichtet werden.

Das `BorderLayout` erzeugt eine in fünf Bereiche unterteilte Anzeige:

Abbildung 9.58
Anzeige von Elementen mit Hilfe des BorderLayouts

Wenn ein Nutzer nun die Applikation vergrößert, wird das zentrale Element in Breite und Höhe angepasst. Die anderen Elemente ändern lediglich ihre Breite (`NORTH`, `SOUTH`) oder Höhe (`WEST`, `EAST`), um eine ausgeglichene Darstellung zu ermöglichen:

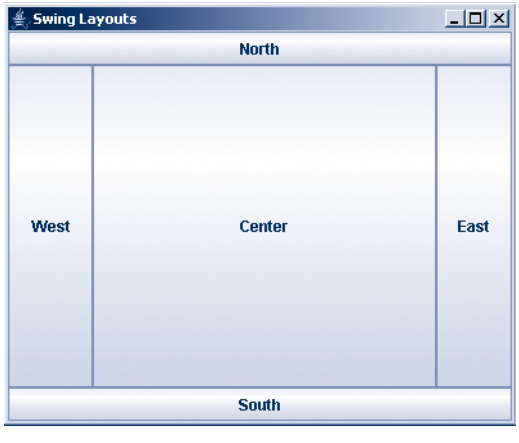


Abbildung 9.59
Bei einer Größenänderung ändern sich auch die enthaltenen Elemente

9.7.4 GridBagLayout

Das GridBagLayout ist eines der flexibelsten Layouts, die Swing uns zur Verfügung stellt. Gleichzeitig kann die Arbeit mit dieser Art von Layout sehr komplex werden – es soll Entwickler geben, die beim Entwerfen von Oberflächen mit dem GridBagLayout stets Papier, Stift und ein Lineal neben sich liegen haben, damit sie den Überblick nicht verlieren.

Das Funktionsprinzip des GridBagLayouts besteht darin, dass Komponenten oder enthaltene Container in Zeilen und Spalten angeordnet werden können. Im Gegensatz zum gewöhnlichen GridLayout können sich Komponenten jedoch auch über mehrere Spalten oder Zeilen erstrecken – und Zeilen müssen auch nicht gleich groß sein.

Um diese Abhängigkeiten und Bemaßungen darstellen zu können, wird eine Hilfsklasse verwendet, die die Informationen mit sich führt – die GridBagConstraints. Diese bietet uns folgende Eigenschaften an, mit deren Hilfe wir die Position der zugehörigen Komponente genau definieren können:

- gridx: Gibt die Zeile innerhalb des GridBagLayouts an (Zählung beginnt bei Null)
- gridy: Gibt die Spalte innerhalb des GridBagLayouts an (Zählung beginnt bei Null)
- gridwidth: Gibt die Breite in Spalten (nicht in Pixeln!) an
- gridheight: Gibt die Höhe in Spalten (nicht in Pixeln!) an
- fill: Gibt an, wie sich die Komponente verhalten soll, wenn sie weniger Platz einnimmt, als ihr zur Verfügung steht – mögliche Werte sind:
  - GridBagConstraints.NONE (Standard-Wert): Keine Größen-Änderung
  - GridBagConstraints.HORIZONTAL: Komponente skaliert horizontal
  - GridBagConstraints.VERTICAL: Komponente skaliert vertikal
  - GridBagConstraints.BOTH: Komponente skaliert in beide Richtungen
- ipadx: Gibt die Anzahl in Pixeln an, die die Komponente auf der x-Achse (horizontal) Abstand zur nächsten Komponente hat. Wird der per setMinimumSize() festgelegten Breite hinzugefügt – der mindestens benötigte Platz für die Komponente ist also 2*ipadx + minimale Breite, da die durch ipadx angegebene Anzahl an Pixeln auf beiden Seiten angetragen wird.
- ipady: Gibt die Anzahl an Pixeln an, die die Komponente auf der y-Achse (vertikal) Abstand zur nächsten Komponente hat. Wird der per setMinimumSize() festgelegten Höhe doppelt hinzugefügt (oben und unten).
- insets: Gibt den externen Abstand der Komponente zu anderen Komponenten an. Wird als Insets-Instanz erwartet. Per Default ist der Wert null.
- anchor: Wird genutzt, falls die Komponente kleiner als der ihr zur Verfügung stehende Platz ist. Gibt an, wo die Komponente innerhalb ihres Platzes angezeigt wird. Folgende Werte stehen zur Verfügung:
  - GridBagConstraints.FIRST_LINE_START: Links oben
  - GridBagConstraints.PAGE_START: Mitte oben

- `GridBagConstraints.FIRST_LINE_END`: Rechts oben
- `GridBagConstraints.LINE_START`: Mitte links
- `GridBagConstraints.CENTER` (Standard-Wert): Zentriert in der Mitte
- `GridBagConstraints.LAST_LINE_START`: Links unten
- `GridBagConstraints.PAGE_END`: Mitte unten
- `GridBagConstraints.LAST_LINE_END`: Rechts unten

- `weightx`: Gewichtung der Komponente. Beeinflusst, wie der zur Verfügung stehende Platz verteilt wird. Die Gewichtung wird als Zahl zwischen 0.0 und 1.0 angegeben. Wenn die Gewichtung 0.0 (Standard-Wert) zugewiesen ist, wird die Komponente bei einer Vergrößerung des zur Verfügung stehenden Platzes nicht vergrößert – lediglich ihr Abstand zu den umgebenden Elementen wächst. Genau das Gegenteil bewirkt die Gewichtung 1.0: Die Komponente füllt den zur Verfügung stehenden Platz komplett aus. `weightx` beeinflusst dieses Verhalten in horizontaler Richtung
- `weighty`: Gewichtung der Komponente in vertikaler Richtung. Funktioniert analog zu `weightx`.

Die Hauptschwierigkeit beim Arbeiten mit dem `GridBagLayout` liegt in der richtigen Gewichtung der Komponenten und dem korrekten Anordnen der Elemente. Sehen wir uns an, wie eine Applikation mit Hilfe des `GridBagLayouts` umgesetzt werden kann:

Listing 9.41 Erzeugen von Elementen in einem GridBagLayout

```
import javax.swing.*;
import java.awt.*;

// ...

JPanel content = new JPanel(new GridBagLayout());
add(content);

// Button, row #1, column #1
GridBagConstraints c = new GridBagConstraints();
c.gridx = 0;
c.gridy = 0;
content.add(new JButton("Button #1"), c);

// Button, row #1, column #2
c = new GridBagConstraints();
c.gridx = 1;
content.add(new JButton("Button #2"), c);

// Button, row #2, column #1, width 3 columns,
// auto-resizing itself in vertical and horizontal
// direction
c = new GridBagConstraints();
c.gridy = 1;
c.gridwidth = 3;
c.weightx = 1.0;
c.fill = GridBagConstraints.HORIZONTAL;
content.add(new JButton("Button #3"), c);
```

Listing 9.41 *(Forts.)*
Erzeugen von Elementen in einem GridBagLayout

```
// Button, row #3, column #3,
// auto-resizing itself in horizontal & vertical direction
c = new GridBagConstraints();
c.gridx = 2;
c.gridy = 2;
c.weighty = 1.0;
c.weightx = 1.0;
c.fill = GridBagConstraints.BOTH;
content.add(new JButton("Button #4"), c);
```

Die Erzeugung von Elementen für ein `GridBagLayout` wird durch Verwendung der `GridBagConstraints` um einiges aufwändiger. Dabei ist der Anfang sehr simpel: Wir übergeben einem Container (in diesem Fall ein `JPanel`) bei dessen Erzeugung eine neue `GridBagLayout`-Instanz als Parameter:

```
JPanel content = new JPanel(new GridBagLayout());
```

Für den ersten Button erzeugen wir zunächst eine neue Instanz der `GridBagConstraints`-Klasse. Der Button wird in der ersten Reihe an der ersten Position eingefügt, was wir mit Hilfe der Eigenschaften `gridx` und `gridy` festlegen. Beim Zuweisen des Buttons zum Container, der das `GridBagLayout` verwendet, übergeben wir als zweiten Parameter die `GridBagConstraints`-Instanz:

```
GridBagConstraints c = new GridBagConstraints();
c.gridx = 0;
c.gridy = 0;
content.add(new JButton("Button #1"), c);
```

Nun können wir den zweiten Button in derselben Zeile (`gridy` bleibt ohne Zuweisung, deshalb enthält diese Eigenschaft den Standard-Wert 0, der die erste Zeile repräsentiert) anzeigen – wir weisen deshalb der Eigenschaft `gridx` den Wert 1 zu. Da die Indizes bei 0 anfangen, steht der Button somit an der zweiten Stelle:

```
c = new GridBagConstraints();
c.gridx = 1;
content.add(new JButton("Button #2"), c);
```

Der dritte Button kommt in die erste Spalte der zweiten Reihe. Da wir der Eigenschaft `gridwidth` den Wert Drei zuweisen, erstreckt er sich über drei Spalten. Durch die Zuweisung des Werts von `GridBagConstraints.HORIZONTAL` zur Eigenschaft `fill` erreichen wir, dass die Größe der Schaltfläche in horizontaler Richtung angepasst wird, wenn sich die Größe des Containers ändert. Dies allein würde aber noch keinen Effekt haben – erst die Zuweisung eines hohen Wertes zur Eigenschaft `weightx` wird dafür sorgen, dass der Button skaliert:

```
c = new GridBagConstraints();
c.gridy = 1;
c.gridwidth = 3;
c.weightx = 1.0;
c.fill = GridBagConstraints.HORIZONTAL;
content.add(new JButton("Button #3"), c);
```

Der vierte Button wird der dritten Spalte der dritten Zeile zugeordnet. Beiden `weight`-Eigenschaften wird der Wert 1.0 zugewiesen – der Button soll sowohl horizontal als auch vertikal skalieren. Damit dies geschieht, weisen wir seiner `fill`-Eigenschaft zuletzt den Wert von `GridBagConstraints.BOTH` zu:

```
c = new GridBagConstraints();
c.gridx = 2;
c.gridy = 2;
c.weighty = 1.0;
c.weightx = 1.0;
c.fill = GridBagConstraints.BOTH;
content.add(new JButton("Button #4"), c);
```

Nun können wir die Applikation anzeigen lassen:

Abbildung 9.60
Anzeige von Schaltflächen im GridBagLayout

Wenn ein Benutzer nun das Anzeigefenster vergrößert, werden die Buttons in der zweiten und dritten Reihe skalieren. Die Buttons am oberen Rand hingegen bleiben unverändert:

Abbildung 9.61
Buttons können skalieren – müssen aber nicht

9.8 Standard-Dialoge

Standard-Dialoge informieren den Nutzer, erfragen Werte von ihm oder präsentieren ihm bestimmte Informationen. Wie der Name bereits andeutet, sind sie weitestgehend standardisiert, was es uns sehr einfach macht, mit ihnen zu arbeiten.

9.8.1 Message-Dialog

Mit Hilfe des Message-Dialogs können Sie Fehlermeldungen, Hinweise oder andere Informationen ausgeben. Message-Dialoge verfügen über ein Icon, einen Titel und eine Meldung – und alle diese Werte können Sie beeinflussen.

Message-Dialoge werden über die statische Methode `showMessageDialog()` der `JOptionPane`-Klasse angezeigt. Die Methode ist mehrfach überladen und erfordert zumindest die Angabe eines Ausgabe-Textes und des `JFrames`, in dessen Kontext die Anzeige erfolgen soll:

```
JOptionPane.showMessageDialog(instance, "This is a simple message");
```

Dies erzeugt diesen Message-Dialog:

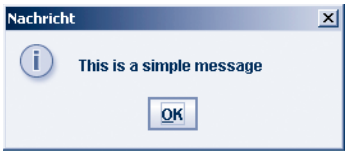


Abbildung 9.62
Einfacher MessageDialog

Eine zweite Überladung nimmt zusätzlich zu Nachricht und Komponente Werte für den Titel und die Art des Message-Dialogs entgegen. Folgende Werte können für die Art des Dialogs verwendet werden:

- `JOptionPane.ERROR_MESSAGE`: Icon für eine Fehlermeldung wird angezeigt
- `JOptionPane.INFORMATION_MESSAGE` (Standard-Wert): Informations-Nachricht wird angezeigt
- `JOptionPane.WARNING_MESSAGE`: Anzeige des Icons für eine Warnung
- `JOptionPane.QUESTION_MESSAGE`: Fragezeichen wird als Icon angezeigt
- `JOptionPane.PLAIN_MESSAGE`: Es wird kein Icon angezeigt

Wenn wir eine Fehlermeldung anzeigen wollen, können wir dies mit diesem Code erledigen:

```
JOptionPane.showMessageDialog(
   instance, "An error has occured", "Error!",
   JOptionPane.ERROR_MESSAGE);
```

Dies führt zur Anzeige eines Message-Dialogs mit einem Icon für eine Fehlermeldung:

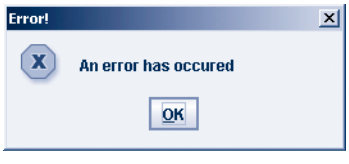


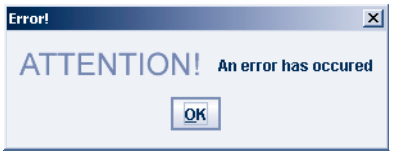
Abbildung 9.63
Anzeige einer Fehlermeldung

Die dritte verfügbare Überladung der `showMessageDialog()`-Methode nimmt als letzten Parameter eine `Icon`-Instanz entgegen und zeigt das dadurch repräsentierte Bild als `Icon` innerhalb der Dialog-Box an:

```
JOptionPane.showMessageDialog(
   instance, "An error has occured", "Error!",
   JOptionPane.WARNING_MESSAGE, new ImageIcon("warning.gif"));
```

Bei Übergabe einer `Icon`-Instanz wird das vom System vorgegebene Bild durch ein benutzerdefiniertes Bild ersetzt:

Abbildung 9.64
Anzeige eines eigenen Icons im Dialog



Message-Dialoge geben niemals eine Rückgabe – sie werden einfach nur angezeigt und vom Nutzer bestätigt. Dies ändert sich, wenn wir etwa Option-Dialoge anzeigen lassen.

9.8.2 Confirm-Dialog

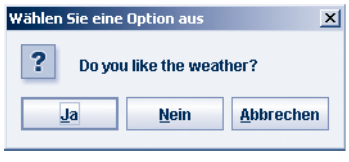
Der Confirm-Dialog erlaubt dem User die Auswahl zwischen zwei oder drei verschiedenen Reaktions-Möglichkeiten. Die Anzeige eines Confirm-Dialogs erfolgt, indem wir die Methode `showConfirmDialog()` der `JOptionPane`-Klasse aufrufen. Deren Rückgabe ist ein `Integer`-Wert, der die getroffene Auswahl des Nutzers repräsentiert.

Die einfachste Variante des Confirm-Dialogs nimmt eine Referenz auf die `JFrame`-Instanz, in deren Kontext der Dialog angezeigt werden soll, und die dem Nutzer zu stellende Frage als Parameter entgegen:

```
int result = JOptionPane.showConfirmDialog(
   instance, "Do you like the weather?");
```

Dies zeigt folgenden Confirm-Dialog an:

Abbildung 9.65
Ein einfacher Confirm-Dialog



Eine zweite Überladung erlaubt es uns, den Titel des Fensters und die Antwort-Möglichkeiten zu definieren. Folgende Werte stehen zur Auswahl, um die Antwort-Möglichkeiten festzulegen:

- `JOptionPane.YES_NO_OPTION`: Erlaubt dem Nutzer, mit *Ja* oder *Nein* zu antworten
- `JOptionPane.YES_NO_CANCEL_OPTION`: Der Nutzer darf zwischen *Ja*, *Nein* und *Abbrechen* auswählen

Um dem Nutzer nur *Ja* oder *Nein* als Antwortmöglichkeiten zu lassen, können wir folgenden Code verwenden:

```
int result = JOptionPane.showConfirmDialog(
   instance, "Do you like the weather?", "Question",
   JOptionPane.YES_NO_OPTION);
```

Nun kann der Benutzer nur noch zwischen den Werten *Ja* oder *Nein* wählen:

Abbildung 9.66
Anzeige eines OptionDialogs, der die Auswahl zwischen Ja oder Nein zulässt

Um zu überprüfen, welchen Button der Nutzer angeklickt hat, prüfen Sie die Rückgabe darauf, ob sie einer der folgenden Konstanten entspricht:

- `JOptionPane.YES_OPTION`: Der Nutzer hat *Ja* angeklickt
- `JOptionPane.NO_OPTION`: Die Auswahl des Nutzers war *Nein*
- `JOptionPane.CANCEL_OPTION`: Der Nutzer hat die *Abbrechen*-Option ausgewählt

Wenn Sie die dritte Überladung der `showConfirmDialog()`-Methode einsetzen, können Sie neben Titel und Schaltflächen auch das angezeigte `Icon` bestimmen. Dieses `Icon` wird durch den Typ der Frage bestimmt und muss einen der für die Art des Dialogs möglichen Werte repräsentieren. Mögliche Werte sind:

- `JOptionPane.ERROR_MESSAGE`
- `JOptionPane.INFORMATION_MESSAGE`
- `JOptionPane.WARNING_MESSAGE`
- `JOptionPane.QUESTION_MESSAGE`
- `JOptionPane.PLAIN_MESSAGE`

Die sinnvollsten und dem Nutzer auch am gängigsten erscheinenden Message-Typen sind in diesem Kontext sicherlich `QUESTION_MESSAGE` (wird mit einem Fragezeichen gekennzeichnet) und `PLAIN_MESSAGE` (ohne Icon):

```
int result = JOptionPane.showConfirmDialog(
   instance, "Do you like the weather?", "Question",
   JOptionPane.YES_NO_CANCEL_OPTION,
   JOptionPane.PLAIN_MESSAGE);
```

Diese Anweisung führt zu folgender Ausgabe (die sich natürlich systembedingt optisch unterscheiden kann):

Abbildung 9.67
PLAIN_MESSAGE mit YES_NO_CANCEL-Option

Die Sprache der Schaltflächen-Beschriftungen ist übrigens von der eingestellten Sprache des Benutzers abhängig – bei englischer Systemsprache werden auch die Ausgaben in Englisch erscheinen.

Die letzte Überladung der `showConfirmDialog()`-Methode akzeptiert zusätzlich zur eben besprochenen dritten Variante eine `Icon`-Instanz als Parameter. Wir haben somit die Möglichkeit, ein eigenes Symbol in der Dialog-Box anzuzeigen:

```
int result = JOptionPane.showConfirmDialog(
   instance, "Do you like the weather?", "Question",
   JOptionPane.YES_NO_OPTION,
   JOptionPane.QUESTION_MESSAGE,
   new ImageIcon("question.gif"));
```

Sicherlich werden Sie bessere grafische Fähigkeiten als der Autor haben – Ihr Icon sollte also an Stelle dieses sehr simpel gehaltenen Fragezeichens erscheinen können:

Abbildung 9.68
Darstellung eines eigenen Icons im Confirm-Dialog

9.8.3 Input-Dialog

Wir sind beim Einsatz von `JFC/Swing` nicht darauf beschränkt, Eingaben im Frage-Antwort-Stil zu ermitteln. Mit Hilfe des Input-Dialogs können wir dem Nutzer die Möglichkeit geben, Einträge aus Listen auszuwählen oder selber Informationen im Freitext anzugeben.

Input-Dialoge werden mit Hilfe der statischen Methode `showInputDialog()` der `JOptionPane`-Klasse erzeugt und angezeigt. Von ihr existieren mehrere Überladungen, die es uns erlauben, unterschiedlichste Informationen zu erfragen und verschiedene Darstellungen zu verwenden.

Die Rückgabe der `showInputDialog()`-Methode ist stets ein `String`, der die eingegebene oder ausgewählte Antwort beinhaltet. Sie können im zweiten Parameter des Methoden-Aufrufs eine Antwort-Vorgabe übergeben:

```
String result = JOptionPane.showInputDialog(
   "Tell me your gender!", "Don't know");
```

Hier wird nun ein Eingabe-Dialog angezeigt werden, der nach dem Geschlecht des Benutzers fragt und als Antwort-Vorgabe „Don't know“ gibt:

Abbildung 9.69
Anzeige eines Input-Dialogs

Wollen Sie keine Antwort-Vorgabe haben, übergeben Sie einfach `null` als zweiten Parameter an die Methode `showInputDialog()` oder lassen diesen ganz weg.

Wenn Sie den Titel oder das Icon der Eingabe-Box beeinflussen wollen, müssen Sie mehrere Parameter übergeben:

- Instanz, die den Dialog aufruft
- Nachricht
- Titel
- Art des Dialogs

```
String result = JOptionPane.showInputDialog(
   instance, "Tell me your gender!",
   "Answer my question!",
   JOptionPane.ERROR_MESSAGE);
```

Wir verfügen hier nun leider nicht mehr über die Möglichkeit, eine Antwort-Vorgabe zu verwenden – eine für den Autor leider etwas unlogische Entscheidung der Entwickler. Wie dem auch sei – die Darstellung unserer eben gestellten Frage nach dem Geschlecht des Nutzers sieht doch etwas anders aus, als dies beim letzten Beispiel der Fall war:

Abbildung 9.70
Auch das Icon und der Titel des Input-Dialogs können manipuliert werden

Achtung

Hüten Sie sich davor, dem Nutzer Dialog-Elemente mit Icons zu präsentieren, die gegen seine Gewohnheiten verstoßen. Sie sollten davon absehen, Fragen mit Icons für Fehlermeldungen oder Fehlermeldungen ohne Icons zu präsentieren – die Verwirrung der Benutzer nähme rapide zu und die Nutzbarkeit Ihrer Applikation im gleichen Maße ab.

Die Königsdisziplin beim Einsatz des Input-Dialogs besteht darin, den Nutzer aus einer Liste von Optionen wählen zu lassen. Hier ist am meisten Schreibarbeit nötig, müssen doch insgesamt sieben Parameter zugewiesen werden. Und ein Array muss eventuell auch noch erzeugt werden:

```
String[] options = new String[] { "Female", "Male", "Don't know" };
String result = (String)JOptionPane.showInputDialog(
   instance, "Tell me your gender!", "Please answer my question!",
   JOptionPane.QUESTION_MESSAGE, new ImageIcon("question.gif"),
   options, "Don't know");
```

Bevor wir einen Input-Dialog mit Listen-Auswahl anzeigen können, müssen wir die Werte definieren, die zur Auswahl stehen. Da es sich (so lautet die Definition einer Liste) um mehrere Werte handelt, erfassen wir die Optionen als Array. Dieses muss nicht zwingend vom Typ String sein, sondern kann beispielsweise auch in Form von Integer-Werten deklariert werden.

Der Methode `showInputDialog()` werden nun folgende Parameter übergeben:

- Instanz, in deren Kontext der Dialog angezeigt wird
- Frage
- Titel-Zeilen-Text
- Typ der Frage
- Eventuell Icon (stattdessen kann auch null als Wert übergeben werden)
- Optionen
- Ausgewählter Wert

Mit Hilfe des letzten Parameters können Sie festlegen, welcher Wert selektiert sein soll. Der Inhalt dieses Parameters muss einem der Parameter aus dem Werte-Array entsprechen – anderenfalls wird die Selektion auf den ersten Listen-Eintrag gesetzt.

Die Rückgabe der Methode ist ein Element aus der als Parameter übergebenen Liste oder `null`, falls keine Auswahl getroffen worden ist.

Die Anzeige lohnt alle Mühen:

Abbildung 9.71
Auswahl aus einer Liste

Bevorzugen Sie statt einer Listen-Auswahl die freie Eingabe eines Wertes durch den Benutzer, können Sie auch `null` statt eines Arrays als vorletzten Parameter übergeben. Verwenden Sie auch `null` als letzten Parameter, bekommt der Nutzer ein Eingabefeld ohne initialen Text angezeigt, anderenfalls steht dort der im letzten Parameter definierte Inhalt.

9.8.4 Schließ-Verhalten von Input-Dialogen beeinflussen

An dieser Stelle schließt sich ein wenig der Bogen dieses – zugegebenermaßen recht großen – Kapitels: Auch das Schließ-Verhalten von Input-Dialogen – ebenso wie das von JFrame-Instanzen, mit denen wir dieses Kapitel begonnen haben – kann beeinflusst und für unsere Zwecke geändert werden. Nützlich ist dies, wenn wir die Eingaben des Nutzers validieren wollen – etwa, weil Zahlen erwartet worden sind.

Nun haben wir leider keine Möglichkeit, im Input-Dialog ein `JFormattedTextField` einzusetzen, das uns spielend einfach die Angabe einer Maske für die Eingabe erlaubt hätte. Also müssen wir etwas weiter unten anfassen und auf Ereignis-Ebene agieren.

Leider wirft der Input-Dialog kein Ereignis, falls auf einen seiner Buttons geklickt wird oder der Eingabe-Bereich den Fokus verliert. Stattdessen verwenden wir einen `PropertyChangeListener` um das `propertyChange`-Ereignis einer `JOptionPane`-Instanz abfangen zu können. Dieses Ereignis wird immer dann geworfen, wenn sich eine Eigenschaft des `JOptionPane` geändert hat. Bleibt uns die Aufgabe, festzustellen, welche Eigenschaft mit welchem Wert geändert wurde, und dann entsprechend darauf zu reagieren.

Um das Schließ-Verhalten beeinflussen zu können, müssen wir eine Instanz des `JOptionPane` erzeugen, der wir die Informationen zu Nachricht und erlaubten Schaltflächen übergeben. Weiterhin weisen wir die Instanz an, eine Eingabe entgegenzunehmen, binden den `PropertyChangeListener` und fügen es einem `JDialog`-Container-Element hinzu, damit dieser unser `JOptionPane` anzeigen kann. Sollte nun eine Schaltfläche betätigt worden sein, ändert sich der Wert der Property `value`. Dadurch wird das `propertyChange`-Ereignis geworfen und der Event-Handler eingebunden. Hier findet dann die weitere Verarbeitung der eingegebenen Informationen statt:

Listing 9.42
Anzeigen eines Dialogs, bis die korrekten Werte eingegeben worden sind

```
import javax.swing.*;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeEvent;

/**
 * Displays an input dialog as long as the user does not enter
 * the correct word(s)
 */
public class OptionSimple extends BaseFrame
   implements PropertyChangeListener {

   JOptionPane pane;
   JDialog dialog;

   /**
    * Displays the dialog
    */
   OptionSimple() {
      setVisible(false);

      pane = new JOptionPane(
         "Please enter your gender",
         JOptionPane.QUESTION_MESSAGE,
         JOptionPane.OK_CANCEL_OPTION);
      pane.setWantsInput(true);
      pane.addPropertyChangeListener(
         JOptionPane.VALUE_PROPERTY, this);

      dialog = new JDialog(this, "Question", true);
      dialog.setContentPane(pane);

      dialog.pack();
      dialog.setVisible(true);

      System.exit(0);
   }
```

Listing 9.42 *(Forts.)*
Anzeigen eines Dialogs, bis die korrekten Werte eingegeben worden sind

```
    /**
     * Handles the propertyChange-event
     */
    public void propertyChange(PropertyChangeEvent evt) {
       String value = (String)pane.getInputValue();
       if(value != null && (value.toLowerCase() == "male" ||
          value.toLowerCase() =="female")) {
          dialog.setVisible(false);
       }
       if(dialog.isVisible()) {
          JOptionPane.showMessageDialog(this,
             "Please enter either 'male' or 'female' as values!");
       }
    }
}
```

Wie bereits erwähnt, müssen wir uns hier der Anzeige einer Dialogbox so nähern, wie dies intern auch geschieht – also müssen wir ein `JOptionPane` erzeugen und diesem dabei übergeben, was es anzeigen soll, welche Art von Dialog wir anzeigen wollen und welche Schaltflächen zulässig sind:

```
pane = new JOptionPane(
   "Please enter your gender",
   JOptionPane.QUESTION_MESSAGE,
   JOptionPane.OK_CANCEL_OPTION);
```

Nun sollten wir auch noch deklarieren, dass eine Eingabe des Nutzers erwünscht ist, was wir durch die Übergabe von `true` an die Methode `setWantsInput()` erledigen können:

```
pane.setWantsInput(true);
```

Beim Binden des `PropertyChangeListeners` an das `JOptionPane` können wir mit Hilfe seiner Methode `addPropertyChangeListener()` angeben, welche Eigenschaft überwacht werden soll – die Konstante `VALUE_PROPERTY` drückt aus, dass uns die Eigenschaft `value` interessiert:

```
pane.addPropertyChangeListener(
   JOptionPane.VALUE_PROPERTY, this);
```

Jetzt muss das `JOptionPane` einem `JDialog`-Container zugewiesen werden, damit es korrekt angezeigt werden kann. Diesem übergeben wir im Konstruktor, dass er im Kontext der aktuellen Klasse angezeigt werden soll, dass sein Titel „Question“ lauten muss und dass die Anzeige modal erfolgen wird:

```
dialog = new JDialog(this, "Question", true);
```

Nun können wir das `JContentPane` zuweisen:

```
dialog.setContentPane(pane);
```

Da die Anzeige modal erfolgen soll, reicht es aus, die `JDialog`-Instanz sichtbar zu machen, um eine Reaktion vom Nutzer zu erzwingen – irgendetwas muss er nun tun, da das Fenster über allen anderen Elementen der aktuellen Applikation liegt und sich erst schließen wird, wenn der Nutzer entweder die richtige Eingabe vorgenommen oder auf dessen Schließen-Symbol geklickt hat.

Sollte er einen Wert in das Eingabefeld eingeben und anschließend einen der Buttons oder die Taste ↵ betätigen, wird das `propertyChanged`-Ereignis geworfen und der Event-Handler eingebunden. Hier lesen wir zunächst den eingegebenen Text aus, bevor eine Überprüfung stattfinden wird:

```
String value = (String)pane.getInputValue();
```

Nur, wenn die Eingabe des Nutzers male oder female (wobei die Groß-/Kleinschreibung keine Rolle spielt) lautet, blenden wir das Dialog-Element aus. Anderenfalls wird die Methode `getVisible()` zurückgeben, dass das Dialog-Element noch sichtbar ist, und wir zeigen einen entsprechenden Hinweis an:

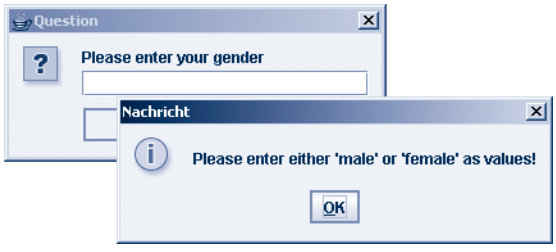


Abbildung 9.72
Abfangen einer Eingabe des Benutzers

9.9 Fazit

Sie haben nun das `Swing/JFC`-Framework annäherungsweise kennen gelernt. Der hier zur Verfügung stehende Platz reicht bei weitem nicht aus, Ihnen alle Informationen zukommen zu lassen, die es zu diesem Thema gibt. Oder Ihnen auch nur annäherungsweise alle Komponenten und Container zu zeigen – wir mussten uns hier schon auf die wichtigsten und am häufigsten gebrauchten Elemente beschränken. Dennoch ist es die Hoffnung des Autors, Ihnen den Einsatz und die Wirkprinzipien von Swing etwas näher gebracht zu haben.

In jedem Fall sollten Sie nun in der Lage sein, eigene Swing-Applikationen zu entwerfen, ihnen Elemente zuzuweisen oder deren Ereignisse zu behandeln. Den Rest werden die Übung und die Praxis mit sich bringen. Lassen Sie sich nicht davon abschrecken, dass `Swing/JFC` recht umfangreich und komplex wird – die grundlegenden Prinzipien sollten Ihnen klar sein und den Rest erledigen die Zeit und Ihr wachsendes Können.

10

Applets

Applets sind der Teil der Java-Technologie, der vielen Nutzern aus dem Internet gebräuchlich ist. Dort werden Applets in der Regel als Seiten-Bestandteile eingesetzt, die die Funktionalität einer Web-Seite um Dinge ergänzen, die mit einfachem HTML nicht machbar sind. Applets müssen aber nicht zwingend im Web-Browser ausgeführt werden, werden es in der Regel allerdings.

Applets gibt es seit Java 1.0. Leider ist die Entwicklung dieser Technologie faktisch auch auf diesem Level stehen geblieben, was aber nicht an Sun, sondern an Microsoft liegt, da deren Java VM, die mittlerweile nicht mehr vertrieben werden darf, auf dem Stand von Java 1.1 eingefroren wurde. Das wäre nicht weiter schlimm – aber da die Microsoft-VM auf vielen Rechnern vorinstalliert ist, bedeutet das für uns, dass wir nicht davon ausgehen können, dass Nutzer die aktuelle Virtual Machine installiert haben. Ergo müssen wir auf alle angenehmen Dinge (etwa Swing) verzichten und für Applets auf das `Abstract Windowing Toolkit` (`AWT`) zurückgreifen. Oder in Kauf nehmen, dass unsere Applets möglicherweise nicht auf allen Systemen laufen können.

Etwas erleichtert wird dies nur dadurch, dass wir nicht ganz von vorn anfangen müssen – Applets erben von der Basisklasse `java.applet.Applet`, und die stellt uns einige Basis-Methoden zur Verfügung, mit deren Hilfe wir Applets recht einfach erstellen können.

10.1 Ein einfaches Applet

Applets sind Ableitungen der Klasse `java.applet.Applet` und diese wiederum leitet sich von `java.awt.Container` ab. Ein Container erlaubt es uns, Layouts zu definieren – analog zu denen, die wir auch bei `Swing/JFC` eingesetzt haben. Eigentlich stimmt der Begriff „analog“ in diesem Zusammenhang nicht: Es sind die gleichen Layouts, wie wir sie bei Swing verwendet haben. Ebenso können wir in Applets verschiedene Komponenten einsetzen, so dass wir nicht komplett von vorne beginnen müssen.

Sehen wir uns nach diesen einführenden Worten also an, wie wir ein einfaches Applet deklarieren können:

Listing 10.1 Deklaration eines Applets, das eine „Hello World"-Nachricht ausgibt

```
import java.applet.Applet;
import java.awt.*;

/**
 * Displays a simple Applet drawing a "Hello World"-message
 */
public class SimpleApplet extends Applet {

   public void start() {
      setLayout(new BorderLayout());
      add(new Label("Hello World from Applet!",
         Label.CENTER));
   }
}
```

So kompliziert war das nun wirklich nicht, oder? Im Kopf der Datei haben wir die `import`-Anweisungen für den `java.awt`-Namensraum und die `java.applet.Applet`-Klasse eingefügt.

Unsere Klasse `SimpleApplet` erbt von der Basis-Klasse `Applet` und überschreibt gleich einmal deren Methode `start()`. Dort wird festgelegt, dass als Anzeige-Layout das schon vom letzten Kapitel bekannte `BorderLayout` (vier Seiten und eine große Fläche in der Mitte, die mitskaliert) verwendet wird:

```
setLayout(new BorderLayout());
```

Jetzt können wir ein neues `Label` zentriert einfügen. Das `Label` zeigt den Text „Hello World from Applet!" an – eben genauso, wie wir es im Konstruktor definiert haben:

```
add(new Label("Hello World from Applet!",
   Label.CENTER));
```

Nun müssen wir das Applet nur noch anzeigen. Hier gibt es zwei Möglichkeiten: über den Browser oder über den Applet-Viewer.

10.1.1 Anzeigen eines Applets über den Browser

Wenn Sie ein Applet über den Browser anzeigen wollen, müssen Sie eine HTML-Seite anlegen und in dieser ein `<applet>`-Tag einfügen:

```
<applet code="SimpleApplet.class" width="250" height="150"></applet>
```

HTML-Puristen werden gequält aufschreien: Hier fehlt ja alles, was HTML ausmacht! Dennoch reicht dies aus, um ein Applet im Browser ausführen zu lassen.

Achtung

Applet-Klassen und HTML-Dateien zur Anzeige müssen so liegen, dass die Angaben in der HTML-Datei auf die Applet-Klasse verweisen. In der Regel (und wenn keine Package verwendet wird) liegen also Applet und HTML-Seite im gleichen Verzeichnis. Wenn Sie mit Packages arbeiten, muss die HTML-Datei im Wurzel-Verzeichnis des Pakets abgelegt werden.

Speichern wir eine HTML-Datei mit obigem Code also im gleichen Verzeichnis wie das kompilierte Applet, können wir sie im Browser öffnen:

Abbildung 10.1
Anzeige eines Applets im Browser

Das Problem beim Einsatz von Applets im Browser (insbesondere beim Entwickeln): Browser cachen Java-Applets. Dies bedeutet für uns, dass wir eigentlich nach jedem Kompilieren den Browser-Cache löschen müssten. Die Lösung: Der Einsatz des Applet-Viewers!

10.1.2 Anzeige eines Applets per Applet-Viewer

Der Applet-Viewer verhält sich genauso, wie es auch ein Browser tun würde. Tatsächlich verarbeitet er sogar HTML-Dateien – der Aufruf von Applets als Java-Klassen funktioniert erst gar nicht!

Also benötigen wir auch für den Einsatz des Applet-Viewers eine HTML-Datei. Wir können mit dem Applet-Viewer sogar die gleiche HTML-Datei verwenden, wie mit einem Browser. Der Applet-Viewer befindet sich im gleichen Verzeichnis des JDK wie auch Java selbst.

Der Applet-Viewer ist ein kleines Kommandozeilen-Tool. Der Aufruf gestaltet sich sehr einfach:

```
appletviewer <URL>
```

Der Platzhalter `<URL>` ist dabei durch den URL zur HTML-Datei, die das `<applet>`-Tag deklariert, zu ersetzen. Dieser URL kann entweder auf das lokale Datei-System, oder eine Ressource im Internet verweisen. Um unser Applet aus einer Datei *SimpleApplet.html* heraus anzuzeigen, können wir folgenden Aufruf verwenden:

```
appletviewer file:/c:/applets/SimpleApplet.html
```

Dies setzt voraus, dass die Datei „c:\applets\SimpleApplet.html“ auf das Java-Applet verweist. Sollte sich die HTML-Datei auf Ihrem System woanders befinden, müssen Sie den Pfad natürlich entsprechend anpassen.

Lange Rede, kurzer Sinn: Rufen wir die Datei einfach per Applet-Viewer auf!

Abbildung 10.2
Anzeige eines Applets im Applet-Viewer

10.2 Lebenszyklus eines Applets

Der Lebenszyklus eines Applets umfasst vier Stati: `Init`, `Start`, `Stop` und `Destroy`. Genau diese Stati spiegeln sich auch in der Benennung der zugehörigen Methoden wider:

- `init()`: Wird zur Initialisierung aufgerufen
- `start()`: Startet das Applet – kann mehrfach aufgerufen werden
- `stop()`: Stoppt das Applet – kann mehrfach aufgerufen werden
- `destroy()`: Vorbereitung für das Zerstören – sinnvoll, um Ressourcen wieder freizugeben

10.2.1 init()

Die `init()`-Methode entspricht dem Konstruktor einer normalen Klasse. Sie wird aufgerufen, nachdem die Applet-Instanz erzeugt worden ist. Da manche Browser aufgrund von Implementierungsfehlern die `init()`-Methode mehrfach aufrufen, sollte man bei zeit- oder ressourcenkritischen Abläufen eine boolesche Variable auf Klassen-Ebene setzen und diese Anweisungen nur ausführen, wenn deren Wert `false` ist. Nach dem Durchlaufen wird der booleschen Variable der Wert `true` zugewiesen:

```
bool inited = false;

public void init() {
   if(!inited) {
      // ...
      inited = true;
   }
}
```

10.2.2 start()

Fast alle Applets, die nach der Initialisierung Aktionen vornehmen, erledigen diese in der `start()`-Methode. Diese wird ausgeführt, wenn das Applet geladen worden ist oder der User die Seite erneut aufruft, weil er den Zurück-Button seines Browsers betätigt. Meistens werden innerhalb der `start()`-Methode zeitaufwändigere Vorgänge ausgeführt oder Threads gestartet, die weitere Aktionen durchführen.

10.2.3 stop()

Die Methode `stop()` ist das Gegenstück zur Methode `start()`. Hier sollten Threads wieder beendet oder so weit aufgeräumt werden, dass System-Ressourcen nicht unnötig belastet werden, wenn das Applet gerade nicht ausgeführt wird. Diese Methode wird stets nach der `start()`-Methode ausgeführt.

10.2.4 destroy()

Hier sollten Sie nicht mehr benötigte Ressourcen endgültig wieder freigeben. Das Überschreiben von `destroy()` ist aber in der Regel nicht nötig, da die meisten Entwickler dazu übergegangen sind, schon in `stop()` gründlich aufzuräumen – eine Unterscheidung in normales und finales Aufräumen erscheint in der Praxis meist als zu aufwändig. Stattdessen wird einmal aufgeräumt – und das eben nicht in `destroy()`, sondern in `stop()`.

10.3 Elemente des Benutzer-Interfaces

Folgende Elemente können wir einem Container in Applets zuweisen:

- Buttons (java.awt.Button)
- CheckBoxen (java.awt.CheckBox)
- Textfelder (java.awt.TextField und java.awt.TextArea)
- Labels (java.awt.Label)
- Listen (java.awt.List)
- Choice (java.awt.Choice)
- Menüs (java.awt.Menu, java.awt.MenuItem, java.awt.CheckBoxMenuItem)
- Container (java.awt.Panel und java.awt.Window sowie abgeleitete Klassen)

Die Verwendung der Komponenten geschieht analog zur Arbeit mit ihren `Swing`-Pendants, auf die wir im *Kapitel 9* eingegangen sind. Selbst die Methoden- und Eigenschaftsbenennungen stimmen zum größten Teil überein.

10.3.1 java.awt.Button

Der `Button` ist eine Schaltfläche, die ein User anklicken kann. Bei einem Klick auf einen Button wird ein `action`-Event geworfen, das von `ActionListener`-Instanzen behandelt werden kann:

Listing 10.2 Deklaration eines AWT-Buttons und Reaktion auf dessen action-Ereignis

```
import java.applet.Applet;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.*;

/**
 * Displays a button and reacts to its action-event
 */
public class Button extends Applet
   implements ActionListener {

   Label label;

   /**
    * Creates the components
    */
   public void init() {
      this.setLayout(new BorderLayout());

      java.awt.Button b = new java.awt.Button("Click me!");
      b.addActionListener(this);
      add(b, BorderLayout.SOUTH);

      label = new Label("Button was not clicked!", Label.CENTER);
      add(label);
   }

   /**
    * Handles the button's action event
    */
   public void actionPerformed(ActionEvent e) {
      label.setText("Button clicked!");
   }
}
```

Das Applet implementiert das Interface `ActionListener`, das eine Reaktion auf das `action`-Ereignis des `Buttons` erlaubt. Dieses Ereignis wird geworfen, wenn ein User auf den Button klickt.

Nach der Erzeugung einer `Button`-Instanz fügen wir diesem deshalb mit Hilfe seiner Methode `addActionListener()` eine Referenz auf die aktuelle Klasse als `ActionListener` hinzu:

```
java.awt.Button b = new java.awt.Button("Click me!");
b.addActionListener(this);
```

Klickt der Nutzer auf den Button, wird die in der Klasse deklarierte Methode `actionPerformed()` aufgerufen und eine entsprechende Meldung ausgegeben.

Wenn wir das Applet im Applet-Viewer betrachten, können wir sehen, dass nach einem Klick auf den Button ein entsprechender Hinweistext ausgegeben wird:

Abbildung 10.3
Anzeige eines Buttons und Reaktion auf dessen action-Ereignis

10.3.2 java.awt.CheckBox

Die `CheckBox` aus dem `AWT` funktioniert weitestgehend analog zu ihrem uns bereits bekannten Gegenstück aus der `Swing`-Welt.

Sie wirft ein `itemChanged`-Event, wenn ein Wert an- oder abgewählt worden ist:

Listing 10.3
Deklaration einer CheckBox und Behandeln ihres item-Ereignisses

```
import java.applet.Applet;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import java.awt.*;

/**
 * Displays a CheckBox and reacts to its events
 */
public class AwtCheckBox extends Applet
   implements ItemListener {

   Label label;

   /**
    * Constructs the UI
    */
   public void init() {
      setLayout(new BorderLayout());

      CheckBox cb = new CheckBox(
         "I accept this as a CheckBox!");
      cb.addItemListener(this);
      add(cb);

      label = new Label("State: inited", Label.CENTER);
      add(label, BorderLayout.SOUTH);
   }

   /**
    * Handles the CheckBox's item-event
    * @param e
    */
   public void itemStateChanged(ItemEvent e) {
      String state = "";
      if(e.getStateChange() == ItemEvent.DESELECTED) {
         state = "not ";
      }

      state += "selected";
      label.setText("State: " + state);
   }
}
```

Die Klasse AwtCheckBox implementiert das Interface ItemListener, was sie dafür qualifiziert, das itemChanged-Ereignis einer CheckBox zu behandeln. Genau diese CheckBox wird innerhalb der init()-Methode erzeugt. Anschließend wird ihr die aktuelle Klassen-Instanz als ItemListener zugewiesen:

```
CheckBox cb = new CheckBox(
   "I accept this as a CheckBox!");
cb.addItemListener(this);
```

Aktiviert oder deaktiviert der Nutzer nun die CheckBox, wird deren itemChanged-Ereignis geworfen und eine entsprechende Nachricht ausgegeben. Anhand des Status der Quelle des Ereignisses (der CheckBox) kann dabei bestimmt werden, welcher Text genau angezeigt werden soll.

Der Nutzer, der das Applet aufruft, wird diese Ausgabe erhalten:

Abbildung 10.4
Eine CheckBox und die Reaktion auf ihr item-Ereignis

Der Schritt von CheckBoxen zu RadioButtons ist nun nicht mehr groß. Im Gegensatz zu Swing gibt es im AWT keine explizite RadioButton-Implementierung. Stattdessen kann die CheckBox je nach Einsatzzweck auch zu einem RadioButton werden.

Um RadioButtons einsetzen zu können, müssen wir nur zwei Dinge erledigen:

- Erzeugen einer neuen CheckBoxGroup-Instanz
- Zuweisen der CheckBoxGroup-Instanz zum Konstruktor der CheckBox

Im Code kann das dann so aussehen:

```
CheckBoxGroup g = new CheckBoxGroup();

CheckBox cb = new CheckBox(
   "I accept this as a RadioButton!", true, g);
cb.addItemListener(this);
p.add(cb);

cb = new CheckBox(
   "I do not accept this as a RadioButton!", false, g);
cb.addItemListener(this);
p.add(cb);
```

Der hier verwendete Konstruktor der CheckBox-Klasse nimmt drei Parameter entgegen: Den anzuzeigenden Text, einen booleschen Wert, der angibt, ob die CheckBox aktiviert ist, und die CheckBoxGroup-Instanz, der die CheckBox angehört.

Wenn alle drei Informationen übergeben worden sind, erfolgt die Darstellung der Elemente als RadioButtons:

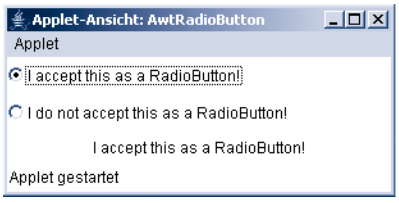


Abbildung 10.5
Anzeige von CheckBoxen als RadioButtons

10.3.3 java.awt.TextArea

Die Eingabe von Texten erfolgt innerhalb von Applets mit Hilfe von `TextField`- oder `TextArea`-Objekten. Erstere repräsentieren einzeilige Eingabefelder, letztere erlauben eine mehrzeilige Eingabe. Auf das Drücken der ↵-Taste innerhalb des `TextFields` können wir mit Hilfe eines `ActionListeners` reagieren – andere Reaktionen sind auf `focus`- oder `itemChanged`-Events möglich.

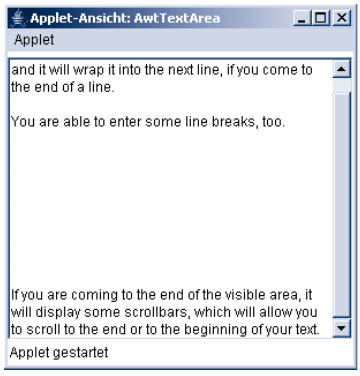
`TextField`- und `TextArea`-Objekte unterscheiden sich im Wesentlichen nur in der Größe der Eingabe-Felder. Wir wollen uns hier beispielhaft die Instanzierung und Anzeige einer `TextArea` anschauen:

Listing 10.4
Deklaration und Anzeige einer TextArea

```
import java.applet.Applet;
import java.awt.*;

/**
 * Presents a TextArea to the user
 */
public class AwtTextArea extends Applet {

   public void init() {
      Panel content = new Panel(new BorderLayout());
      add(content);

      TextArea text = new TextArea(
         "Welcome to my Applet's TextArea!",
         15, 40, TextArea.SCROLLBARS_VERTICAL_ONLY);

      content.add(text);
   }
}
```

Die Anzeige einer `TextArea` ist offensichtlich recht einfach. Im Konstruktor können wir die benötigten Informationen zu Anzeigetext, Höhe in Zeilen, Breite in Spalten und Scrollbars übergeben. Anschließend kann die Komponente angezeigt werden:

Abbildung 10.6
Anzeige einer TextArea



Wichtig für das Verhalten der TextArea ist, wie Sie die Anzeige der Scrollbars regeln:

- `TextArea.SCROLLBARS_NONE`: Keine Scrollbalken werden angezeigt, der eingegebene Text bricht am Ende einer Zeile automatisch in die nächste Zeile um.
- `TextArea.SCROLLBARS_BOTH`: Scrollbalken werden rechts und unten angezeigt – eingegebener Text bricht nicht automatisch um.
- `TextArea.SCROLLBARS_VERTICAL_ONLY`: Scrollbalken werden nur rechts angezeigt. Eingegebener Text bricht automatisch um.
- `TextArea.SCROLLBARS_HORIZONTAL_ONLY`: Scrollbalken werden nur horizontal dargestellt, der eingegebene Text bricht nicht automatisch um.

Das Abrufen von eingegebenen Inhalten erledigen Sie analog zum Vorgehen bei `JTextField`- oder `JTextArea`-Instanzen mit Hilfe der Methode `getText()`.

10.3.4 java.awt.TextField

Das `TextField` ist der kleine Bruder der `TextArea`. Tatsächlich erbt die `TextArea` vom `TextField`, so dass die Methoden der beiden Komponenten einander weitestgehend gleichen:

Listing 10.5
TextField-Instanzen im Einsatz

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

/**
 * Creates two TextFields and reacts to their action-events
 */
public class AwtTextField extends Applet
   implements ActionListener{

   TextField username;
   TextField password;
   Label result;

   /**
    * Building the UI
    */
```

Listing 10.5 *(Forts.)*
TextField-Instanzen im Einsatz

```
    public void init() {
        Panel content = new Panel(new GridBagLayout());
        add(content);

        GridBagConstraints c = new GridBagConstraints();

        Label label = new Label("Username");
        c.gridx = 0;
        c.gridy = 0;
        c.weightx = 1;
        c.fill = GridBagConstraints.BOTH;
        content.add(label, c);

        username = new TextField(20);
        username.addActionListener(this);
        c.gridx = 1;
        content.add(username, c);

        label = new Label("Password");
        c.gridx = 0;
        c.gridy = 1;
        content.add(label, c);

        password = new TextField(20);
        password.setEchoChar('*');
        password.addActionListener(this);
        c.gridx = 1;
        content.add(password, c);

        java.awt.Button b = new java.awt.Button();
        b.setLabel("Log in");
        b.addActionListener(this);
        c.gridx = 1;
        c.gridy = 2;
        content.add(b, c);

        result = new Label("");
        c.gridx = 1;
        c.gridy = 3;
        content.add(result, c);
    }

    /**
     * Reacting to the action-events of username, password and Button
     */
    public void actionPerformed(ActionEvent e) {
        String login = username.getText();
        String pwd = password.getText();

        if(login != null && login.equals("username") &&
           pwd != null && pwd.equals("secret")) {
           result.setText("Login successful!");

        } else {
           result.setText("Login failed!");
        }
    }
}
```

Beim Erzeugen von `TextField`-Instanzen können wir diesen einen initialen Text und eine Breite (nicht in Pixeln, sondern in Zeichen – die Berechnung erfolgt anhand der Breite des Buchstabens `m`) zuweisen. Hier machen wir nur von der zweiten Möglichkeit Gebrauch – unsere `TextField`-Instanzen haben eine Breite von jeweils 20 Zeichen:

```
username = new TextField(20);
```

Sollte der Benutzer in den Eingabefeldern die ↵-Taste drücken, wird deren `action`-Event geworfen. Um dieses behandeln zu können, implementiert unsere Klasse das Interface `ActionListener`.

Das Eingabefeld für das Password soll die eingegebenen Zeichen nicht im Klartext anzeigen, sondern stattdessen einen Platzhalter ausgeben. Wir legen dieses Platzhalterzeichen für die `TextField`-Instanz `password` mit Hilfe ihrer Methode `setEchoChar()` fest:

```
password.setEchoChar('*');
```

Werden nun Zeichen in das Feld eingegeben, dann erscheinen statt der eingegebenen Zeichen Sterne – eben so, wie wir es von gewöhnlichen Password-Feldern auch gewohnt sind.

Ein Klick auf den `Button` „Log in“ oder das Drücken der ↵-Taste in den Eingabefeldern löst jeweils ein `action`-Ereignis aus. Dieses wird innerhalb der Methode `actionPerformed()` behandelt – hier werden die Werte der `TextField`-Instanzen mit Hilfe ihrer `getText()`-Methoden ausgelesen und ausgewertet. Sollte die korrekte Benutzernamen-/Password-Kombination eingegeben worden sein, wird dies mit einer entsprechenden Ausgabe bestätigt:

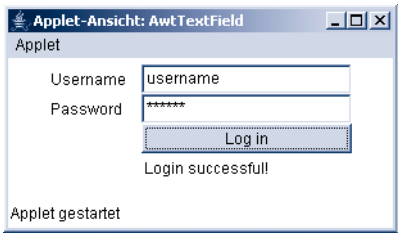


Abbildung 10.7
Eingabe von Benutzername und Password in einem Applet

10.3.5 Menu

Menüs in Applets sind in der Regel Popup-Menüs. Diese bestehen aus einer Instanz der `PopupMenu`-Klasse als Wurzel und untergeordneten `Menu`- und `MenuItem`-Instanzen. Eine `Menu`-Instanz stellt ein Unter-Menü dar, `MenuItem`-Instanzen bilden die eigentlichen Menü-Punkte. Mit Hilfe der Methode `addSeparator()` der Menu-Instanz fügen sie innerhalb von Unter-Menüs einen Trenner ein.

Die Reaktion auf Klicks auf Menü-Punkte findet mit Hilfe von `ActionListenern` statt, die sie den `MenuItem`-Instanzen per `addActionListener()` hinzufügen. Der

Umgang mit Menüs gleicht weitestgehend dem bei Swing – primär unterscheiden sich die Klassen-Bezeichnungen:

Listing 10.6
Anzeige von Popup-Menüs und Behandeln ihrer Ereignissen

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

/**
 * Displays Popup-Menus and handles their action-events
 */
public class AwtMenu extends Applet {

   PopupMenu menu;
   Label label;

   /**
    * Handles the action- and mouse-events for a Menu
    */
   class EventHandler extends MouseAdapter
      implements ActionListener {

      /**
       * Decides, whether to show the menu or not
       */
      private void display(MouseEvent e) {
         if(e.isPopupTrigger()) {
            menu.show(e.getComponent(), e.getX(), e.getY());
         }
      }

      /**
       * Invoked when mouse key was pressed
       */
      public void mousePressed(MouseEvent e) {
         display(e);
      }

      /**
       * Invoked when mouse key was released
       */
      public void mouseReleased(MouseEvent e) {
         display(e);
      }

      /**
       * Handles a MenuItem's action event
       */
      public void actionPerformed(ActionEvent e) {
         MenuItem m = (MenuItem)e.getSource();

         if(m != null) {
            label.setText("Clicked: " + m.getLabel());
         }
      }
   }
```

Listing 10.6 *(Forts.)*
Anzeige von Popup-Menüs und Behandeln ihrer Ereignissen

```
    /**
     * Builds the UI
     */
    public void init() {
       EventHandler handler = new EventHandler();

       Panel content = new Panel(new BorderLayout());
       add(content);

       label = new Label("No action taken!", Label.CENTER);
       content.add(label);

       menu = new PopupMenu("Menü");
       add(menu);

       Menu subMenu = new Menu("File");
       menu.add(subMenu);

       MenuItem item = new MenuItem("Open");
       item.addActionListener(handler);
       subMenu.add(item);

       item = new MenuItem("Close");
       item.addActionListener(handler);
       subMenu.add(item);

       subMenu.addSeparator();

       item = new MenuItem("Create");
       item.addActionListener(handler);
       subMenu.add(item);

       addMouseListener(handler);
    }
}
```

Die Anzeige von Popup-Menüs innerhalb von Applets erfolgt mit Hilfe der `PopupMenu`-Klasse aus dem `java.awt`-Package. Instanzen von `PopupMenu` können sowohl `Menu`- als auch `MenuItem`-Klassen als untergeordnete Elemente besitzen und werden dem Haupt-Panel eines Applets direkt zugewiesen:

```
menu = new PopupMenu("Menü");
add(menu);
```

Unter-Menüs repräsentiert die Klasse `Menu`, der ihrerseits `MenuItem`- (Menüpunkte) und weitere `Menu`-Instanzen, die wiederum untergeordnete Menüs repräsentieren, zugeordnet werden können. `Menu`- und `MenuItem`-Instanzen verfügen über einen überladenen Konstruktor, der unter anderem den anzuzeigenden Namen des Menüpunktes als Parameter entgegennimmt:

```
Menu subMenu = new Menu("File");
menu.add(subMenu);
```

Einzelnen `MenuItem`-Instanzen können `ActionListener` zugewiesen werden, die eine Reaktion auf das Auswählen des Menü-Eintrags erlauben:

```
MenuItem item = new MenuItem("Open");
item.addActionListener(handler);
subMenu.add(item);
```

Für dieses Beispiel wird das Interface `ActionListener` durch die verschachtelte Klasse `EventHandler` repräsentiert. Diese implementiert aber nicht nur die Schnittstelle, sondern erbt ebenfalls von der abstrakten Klasse `MouseAdapter`:

```
class EventHandler extends MouseAdapter
   implements ActionListener {
   ...
}
```

Das abstrakte Interface `MouseAdapter` stellt leere Implementierungen bereit, die eine Reaktion auf die verschiedenen Maus-Ereignisse (Klicken, etc.) erlauben. Für uns interessant sind die beiden Methoden `mousePressed()` und `mouseReleased()`, die eingebunden werden, wenn die Maustaste gedrückt- oder losgelassen worden ist. Die Reaktion auf beide Ereignisse sieht identisch aus – deshalb ist sie auch in die Methode `display()` ausgelagert worden.

Diese nimmt die `MouseEvent`-Instanzen der beiden auslösenden Methoden entgegen und wertet mit Hilfe ihrer Methode `isPopupTrigger()` aus, ob das Popup-Menü angezeigt werden soll:

```
if(e.isPopupTrigger()) {
   menu.show(e.getComponent(), e.getX(), e.getY());
}
```

Die Anzeige des Popup-Menüs erfolgt, indem wir dessen Methode `show()` übergeben, in wessen Kontext und wo die Anzeige stattfinden soll – auch diese Informationen werden uns von der übergebenen `MouseEvent`-Instanz geliefert.

Wenn der Nutzer nun einen Menüpunkt auswählt, wird dessen `action`-Event geworfen. Gut, dass wir eine Instanz der Klasse `EventHandler` an das Ereignis gebunden haben – so können wir reagieren und auswerten, welcher Menüpunkt angeklickt worden ist:

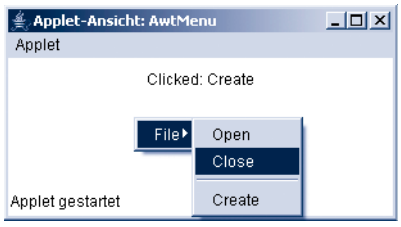
```
public void actionPerformed(ActionEvent e) {
   MenuItem m = (MenuItem)e.getSource();
   /// ...
}
```

Sollte es sich bei dem auslösenden Element um eine `MenuItem`-Instanz handeln (etwas anderes kann es in diesem Beispiel eigentlich nicht sein), dann rufen wir deren Anzeigetext mit Hilfe der Methode `getLabel()` ab und zeigen eine entsprechende Nachricht im Applet an:

```
if(m != null) {
   label.setText("Clicked: " + m.getLabel());
}
```

Wenn wir das Applet im Applet-Viewer aufrufen, werden wir eine Anzeige ähnlich dieser erhalten:

Abbildung 10.8
Anzeige eines Popup-Menüs



10.3.6 java.awt.List

Die Klasse `List` dient der Visualisierung von mehreren Datensätzen, aus denen eine Auswahl getroffen werden kann. Sie ist zwar optisch gewiss nicht das schönste Element zur Darstellung von Werten, kommt aber insbesondere dann zum Einsatz, wenn viele Elemente dargestellt werden sollen und der Platz beschränkt ist. Der Nutzer kann durch einen Doppelklick auf eines der Elemente der Liste ein `action`-Ereignis auslösen:

Listing 10.7
Anzeige einer List und Reaktion auf deren action-Ereignis

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

/**
 * Displays a List and reacts to its action-event
 */
public class AwtList extends Applet implements ActionListener {

   Label label;

   /**
    * Constructs the UI and binds the Listener
    */
   public void init() {
      Panel content = new Panel(new BorderLayout());
      add(content);

      List list = new List(5);
      list.addActionListener(this);
      content.add(list);

      label = new Label("Nothing selected", Label.CENTER);
      content.add(label, BorderLayout.SOUTH);

      String[] items = new String[] {
         "One", "Two", "Three", "Four", "Five", "Six",
         "Seven", "Eight", "Nine", "Ten", "Eleven", "Twelve" };

      for(int i=0; i<items.length; i++) {
         list.add(items[i]);
      }
   }
```

```
    /**
     * Reacts to the list's action-event
     */
    public void actionPerformed(ActionEvent e) {
       List source = (List)e.getSource();
       label.setText(source.getSelectedItem());
    }
}
```

Listing 10.7 *(Forts.)*
Anzeige einer List und Reaktion auf deren action-Ereignis

Beim Erzeugen einer neuen `List`-Instanz können wir dieser ihre darzustellende Größe in Zeilen übergeben. Dies beeinflusst die Darstellung, nicht jedoch die Anzahl der zuweisbaren Elemente:

```
List list = new List(5);
```

Nach Zuweisung eines `ActionListeners` – in diesem Fall die aktuelle Klasse, was durch das Schlüsselwort `this` ausgedrückt wird – können wir auf das `action`-Ereignis der Liste reagieren, das geworfen wird, wenn der Nutzer einen Doppelklick auf ein Element ausführt:

```
list.addActionListener(this);
```

Die Zuweisung der Listen-Elemente erfolgt nicht, wie man annehmen könnte, indem man ein Array an die Liste übergibt. So bequem geht es leider nicht – wir müssen fein säuberlich die einzelnen Elemente an die Methode `add()` der `List`-Instanz übergeben:

```
for(int i=0; i<items.length; i++) {
   list.add(items[i]);
}
```

Nun ist die Zuweisung der Daten beendet und die Anzeige der Informationen erfolgt.

Führt der Nutzer einen Doppelklick auf eines der Elemente aus, wird das `action`-Ereignis der Liste geworfen. Dies bedeutet in der Konsequenz, dass die Methode `actionPerformed()` des `ActionListeners` aufgerufen wird.

Hier kann der Auslöser des Ereignisses von einer generischen `Object`- wieder in die spezifische `List`-Instanz zurückgecastet werden. Das ermöglicht uns, den Text des ausgewählten Elements abzurufen – die Methode `getSelectedItem()` liefert uns die gewünschte Information:

```
List source = (List)e.getSource();
label.setText(source.getSelectedItem());
```

Nutzer, die das Applet ausführen, werden folgende Ausgabe erhalten:

Abbildung 10.9
Auswahl aus einer List

Mehrfache Auswahl ermöglichen

Die List ist nicht darauf beschränkt, nur ein Element auswählen zu lassen. Wenn Sie der Methode `setMultipleMode()` den Wert `true` übergeben, können mehrere Elemente zugleich durch Halten der Strg-Taste ausgewählt werden:

```
list.setMultipleMode(true);
```

Die Auswertung, welche Elemente ausgewählt worden sind, erfolgt mit Hilfe der Methode `getSelectedItems()`, die ein Array der ausgewählten Elemente zurückgibt:

```
String[] selected = list.getSelectedItems();
for(int i=0; i<selected.length; i++) {
   System.out.println(selected[i]);
}
```

Index-Werte verwenden

Mit Hilfe der Index-Werte von Elementen in der Liste können Sie diese vorauswählen oder auswerten, welches Element vom Nutzer aktiviert worden ist. Sie können somit Listen als Repräsentation komplexerer Datenstrukturen oder eigener Elemente verwenden.

Wenn Sie ein Element aus Bequemlichkeitsgründen vorher auswählen wollen, können Sie den Index-Wert des Elements an die Methode `select()` übergeben:

```
list.select(1);
```

Die Methode `getSelectedIndex()` gibt uns zurück, welches Element ausgewählt worden ist:

```
int index = list.getSelectedIndex();
```

Wenn Sie eine Mehrfach-Auswahl aktiviert haben, rufen Sie die ausgewählten Indizes mit Hilfe der Methode `getSelectedIndexes()` ab:

```
int[] selected = list.getSelectedIndexes();
for(int i=0; i<selected.length; i++) {
   System.out.println(String.valueOf(selected[i]));
}
```

10.3.7 java.awt.Choice

Eine Auswahl per Pulldown können Sie mit Hilfe der `Choice`-Komponente ermöglichen. Die Reaktion auf die Auswahl eines Elements erfolgt mit Hilfe eines `ItemListeners`, der die Methode `itemStateChanged()` bereitstellen muss:

Listing 10.8
Mit Hilfe des choice-Elements können Sie eine Auswahl via Pulldown ermöglichen

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;

/**
 * Displays a choice and reacts to its
 * itemStateChanged-event
 */
```

Listing 10.8 *(Forts.)*
Mit Hilfe des choice-Elements können Sie eine Auswahl via Pulldown ermöglichen

```
public class AwtChoice extends Applet
   implements ItemListener {

   Label label;

   /**
    * Constructs the UI
    */
   public void init() {
      Panel content = new Panel(new BorderLayout());
      add(content);

      Choice c = new Choice();
      c.addItemListener(this);
      c.add("One");
      c.add("Two");
      c.add("Three");
      c.add("Four");
      c.add("Five");
      content.add(c);

      label = new Label(
         "Nothing selected - Please select an item!");
      content.add(label, BorderLayout.SOUTH);
   }

   /**
    * Reacts to the itemStateChanged-event
    */
   public void itemStateChanged(ItemEvent e) {
      String item = (String)e.getItem();
      label.setText("Selected item: " + item);
   }
}
```

Nach dem Erzeugen einer neuen Choice-Instanz können wir ihr die aktuelle Klasseninstanz, die das Interface implementiert, als ItemListener zuweisen. Die Bindung an das Event geschieht durch die Methode `addItemListener()`:

```
Choice c = new Choice();
c.addItemListener(this);
```

Das Zuweisen von Elementen erfolgt durch die Methode `add()` – leider existiert auch hier keine Möglichkeit, ein Array oder eine Liste direkt zu übergeben:

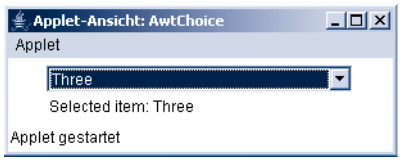
```
c.add("One");
```

Jedes Mal, wenn sich die Auswahl ändert, wird nun ein `itemStateChanged`-Ereignis geworfen – und von der gleichnamigen Methode aufgefangen. Hier können wir nun das ausgewählte Element ermitteln – `getItem()` liefert uns eine `Object`-Instanz, die in einen `String` gecastet den Wert des ausgewählten Elements repräsentiert:

```
String item = (String)e.getItem();
```

Nutzer, die das Applet ausführen, können bei Änderung des ausgewählten Elements feststellen, dass diese Information auch in einem darunter stehenden Label angezeigt wird:

Abbildung 10.10
Auswahl eines Elements per Choice



Sollten Sie direkt auf die `Choice`-Instanz zugreifen können, müssen Sie sich nicht auf ein `itemStateChanged`-Ereignis verlassen, um zu ermitteln, welches Element der Liste derzeit ausgewählt ist. Nutzen Sie stattdessen die Methode `getSelectedItem()`, um den Wert des ausgewählten Elements zu ermitteln:

```
System.out.println(c.getSelectedItem());
```

Ebenso wie bei der List ist es möglich, ein Element vorauszuwählen. Übergeben Sie zu diesem Zweck den Index des auszuwählenden Elements an die Methode `select()`:

```
c.select(1);
```

Wenn es Ihnen zu unbequem erscheint, einen Index-Wert an die `Choice`-Instanz übergeben zu müssen, können Sie hier (anders als bei der `List`-Klasse) auch den Wert des zu selektierenden Elements übergeben:

```
c.select("Two");
```

Den Index des aktuell ausgewählten Elements ermitteln Sie dann wieder analog zur `List`-Klasse mit Hilfe der Methode `getSelectedIndex()`:

```
int index = c.getSelectedIndex();
```

10.3.8 Komponenten, LayoutManager, Applets ...?

Innerhalb von Applets können Sie eigentlich alle Komponten des `AWT`-Frameworks einsetzen. Dies ist umso leichter möglich, als das Applet selbst ein `Panel` ist und sich auch so verhält – so lassen sich Komponenten mit Hilfe seiner Methode `add()` hinzufügen und die Methode `addMouseListener()` erlaubt es, auf Ereignisse zu reagieren, die von der Maus ausgelöst werden.

Die Darstellung kann strukturierter erfolgen, wenn wir dem Basis-Panel einen Layout-Manager zuweisen – statt des gewöhnlichen `FlowLayouts` können wir beispielsweise auf das `BorderLayout`, das `GridLayout` oder sogar das `GridBagLayout` zurückgreifen.

Wenn wir Applets als Container-Elemente analog zu `JFrames` im `Swing`-Framework betrachten, können wir mit ihnen fast genauso effektiv arbeiten und Daten visualisieren. Ein anderer Ansatz ist mehr zeichnerischer Natur – auf den gehen wir nun etwas genauer ein.

10.4 paint() und update() – Grafik mit dem AWT

Beim Einsatz von Applets sind Sie nicht darauf beschränkt, mit Layouts oder Komponenten zu arbeiten, die Sie dem Applet-Container zuweisen. Sie können ebenso direkt auf die Applet-Fläche zeichnen und auf ihr Text ausgeben. Sie greifen zu diesem Zweck auf ein Koordinaten-System zurück, dessen Nullpunkte in x- und y-Richtung im linken oberen Bereich des Applets liegen.

Um mit Grafik zu arbeiten, überschreiben Sie die Methode `paint()` des Applets. Hier können Sie nun Ihre Ausgabe selbst erstellen und somit auf Wunsch (und mit entsprechender Geduld und dem nötigen Können) auch recht anspruchsvolle Zeichnungen darstellen.

10.4.1 Grafische Elemente zeichnen

Das Zeichnen grafischer Elemente findet mit Hilfe der `Graphics`-Instanz statt, die der Methode `paint()` vom System als Parameter übergeben worden ist. Rechtecke können wir mit Hilfe der Methode `drawRect()` zeichnen und durch `fillRect()` auch ausgefüllt darstellen. Ovale werden durch `drawOval()` gezeichnet und per `fillOval()` ausgegeben:

Listing 10.9
Zeichnen einiger grafischer Elemente

```
import java.applet.Applet;
import java.awt.*;

/**
 * Displays some graphical elements
 */
public class AwtPaint extends Applet {

   public void paint(Graphics g) {
      super.paint(g);

      setBackground(Color.lightGray);

      g.setColor(Color.white);
      g.drawRect(10, 10,
         getWidth() - 20, getHeight() - 20);
      g.setColor(Color.blue);
      g.fillRect(11, 11,
         getWidth() - 21, getHeight() - 21);

      g.setColor(Color.red);
      g.fillRoundRect(5, 5, 200, 30, 5, 10);
      g.setColor(Color.white);
      g.drawRoundRect(5, 5, 200, 30, 5, 10);

      g.setColor(Color.yellow);
      g.fillOval(205, 15, 30, 30);
      g.setColor(Color.white);
      g.drawOval(205, 15, 30, 30);

      g.setColor(Color.white);
      g.drawString("Hello!", 30, 30);
   }
}
```

Bevor wir grafische Elemente ausgeben, wollen wir die Hintergrund-Farbe des Applets anpassen. Dies geschieht, indem wir der Methode `setBackgroundColor()` des Applets die neue Hintergrundfarbe als Parameter übergeben. Diese Farbe wird durch eine Instanz der `Color`-Klasse repräsentiert, die uns einige vordefinierte Farben in Form von Konstanten zur Verfügung stellt und daneben mehrere Konstruktoren, der wir die gewünschten Farbwerte in Form von RGB-Tripeln oder als System-Bezeichnungen übergeben können. Hier beschränken wir uns auf den Einsatz von vordefinierten Konstanten:

```
setBackground(Color.lightGray);
```

Wenn wir Elemente zeichnen wollen, können wir jedes Mal direkt vor dem Zeichnen die zu verwendende Farbe mit Hilfe der Methode `setColor()` festlegen:

```
g.setColor(Color.white);
```

Solange wir keine andere Farbe definieren, werden nunmehr alle folgenden Ausgaben mit der so angegebenen Farbe vorgenommen.

Für alle darstellbaren geometrischen Objekte gibt es mindestens zwei Zeichen-Methoden: `draw` und `fill`. Während die `draw`-Methoden einen Rahmen von einem Pixel in den angegebenen Maßen darstellen, füllen die `fill`-Methoden den angebenenen Bereich in der mit Hilfe von `setColor()` definierten Farbe.

Um ein Rechteck mit einem weißen Rahmen und blauer Füll-Farbe darzustellen, können wir die Methoden `fillRect()` und `drawRect()` verwenden:

```
g.setColor(Color.white);
g.drawRect(10, 10,
   getWidth() - 20, getHeight() - 20);
g.setColor(Color.blue);
g.fillRect(11, 11,
   getWidth() - 21, getHeight() - 21);
```

Wenn Sie die Methoden `getHeight()` und `getWidth()` für Höhen- und Breitenangaben verwenden, werden die so deklarierten Komponenten skalieren – wird das Applet größer dargestellt, dann wachsen die Komponenten auch in x- und y-Richtung mit. Dies ist bei Zuweisung von absoluten Angaben nicht der Fall.

Die `draw`- und `fillRect`-Methoden nehmen als Parameter den x- und y-Punkt der linken oberen Ecke sowie relative Breite und relative Höhe entgegen. Die Breiten- und Höhen-Angaben ermitteln wir anhand der Ausmaße des Applets – dessen Methoden `getWidth()` und `getHeight()` liefern uns die benötigten Informationen.

Anschließend fügen wir ein Rechteck mit gerundeten Ecken ein – auch dies geschieht mit Hilfe der `draw`- und `fill`-Methoden. Erstere zeichnet den Rahmen, letztere füllt den Inhalt. Um nicht ständig die Maße ändern zu müssen, zeichnen wir zuerst den Inhalt und danach mit den gleichen Maßen den Rahmen:

```
g.setColor(Color.red);
g.fillRoundRect(5, 5, 200, 30, 5, 10);
g.setColor(Color.white);
g.drawRoundRect(5, 5, 200, 30, 5, 10);
```

Die Methoden `fillRoundRect()` und `drawRoundRect()` nehmen sechs Parameter entgegen:

- x-Position der linken oberen Ecke
- y-Position der linken oberen Ecke
- Breite des Elements
- Höhe des Elements
- Breite des Bogens
- Höhe des Bogens

Nun fügen wir noch ein Oval ein. Dieses ist durch folgende Parameter gekennzeichnet:

- x-Position des höchsten Punktes
- y-Position des am weitesten links befindlichen Punktes
- Breite des Ovals
- Höhe des Ovals

Wenn Breite und Höhe des Ovals gleich sind, wird dieses als Kreis dargestellt.

Zuletzt geben wir noch einen Text aus. Wir verwenden zu diesem Zweck die Methode `setString()`, der wir wie beim Zeichnen von grafischen Elementen die Koordinaten der linken oberen Ecke übergeben:

```
g.setColor(Color.white);
g.drawString("Hello!", 30, 30);
```

Das Ergebnis unserer Mühen würde zwar Picasso nicht übertreffen, dennoch sieht es doch schon recht ansehnlich aus:

Abbildung 10.11
Ausgabe von geometrischen Figuren und Texten

Beachten Sie, wie sich die Größe des blauen Rechtecks im Vergleich zu den anderen Elementen ändert, wenn wir die Größe des Applets ändern:

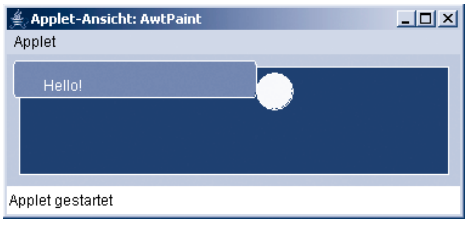


Abbildung 10.12
Bei einer Größenänderung skaliert das blaue Rechteck mit

Dieses Skalieren ist dem Setzen der Größenangaben mit relativen Werten ähnlich – beim Ändern der Applet-Größe wird die Methode `paint()` erneut eingebunden und die Ausgabe erfolgt unter Verwendung der dann aktuellen Breiten- und Höhenangaben.

10.4.2 Text direkt ausgeben

Wir haben es beim letzten Beispiel schon praktiziert: Die Ausgabe des Textes mit Hilfe der Methode `setText()`. Dass wir dabei nicht nur auf den Standard-Font beschränkt sind, demonstriert das folgende Beispiel:

Listing 10.10
Anzeige eines Textes mit einem benutzerdefinierten Font

```
import java.applet.Applet;
import java.awt.*;
import java.awt.font.LineMetrics;

/**
 * Displays a text with a custom font and a
 * gradient-color
 */
public class AwtText extends Applet {

   public void paint(Graphics g) {
      super.paint(g);
      String output = "Hello World!";
      setBackground(Color.lightGray);

      Graphics2D g2 = (Graphics2D)g;

      Font f = new Font("Sans serif", Font.ITALIC, 24);
      FontMetrics fm = getFontMetrics(f);
      LineMetrics lm =fm.getLineMetrics(output, g);

      int width = fm.stringWidth(output);
      float height = lm.getHeight();

      int displayX = (getWidth() / 2) - (width / 2);
      float displayY = (getHeight() / 2) + (height / 2);

      Paint p = new GradientPaint(
         displayX, displayY, Color.red,
         displayX + width, displayY + height, Color.white);

      g2.setPaint(p);
      g2.setFont(f);

      g2.drawString(output, displayX, displayY);
   }
}
```

Bevor ganz am Ende des Beispiels die Ausgabe des Textes mit Hilfe von `setText()` stattfinden kann, haben wir uns einiges an Arbeit gemacht. Ganz am Anfang der Methode `paint()` haben wir den generischen Grafik-Kontext in eine Instanz der `Graphics2D`-Klasse gecastet, was uns erweiterte Möglichkeiten der Ausgabe von grafischen Elementen erschließt:

```
Graphics2D g2 = (Graphics2D)g;
```

Nach der Definition der zu verwendenden Schriftart mit Hilfe einer neuen `Font`-Instanz erzeugen wir Referenzen auf zwei Objekte, die uns bei der Bestimmung der Maße des auszugebenden Textes behilflich sein werden: `FontMetrics` (damit

werden wir die Breite des Textes bestimmen) und `LineMetrics` (hilft uns, die tatsächliche Höhe des Textes zu erfassen):

```
Font f = new Font("Sans serif", Font.ITALIC, 24);
FontMetrics fm = getFontMetrics(f);
LineMetrics lm = fm.getLineMetrics(output, g);
```

Um eine `LineMetrics`-Instanz zu erhalten, müssen wir uns der `FontMetrics`-Instanz (die wir durch die Methode `getFontMetrics()` der `Applet`-Instanz erhalten haben) bedienen und deren Methode `getLineMetrics()` sowohl auszugebende Zeichenkette als auch den Grafik-Kontext als Parameter übergeben.

Haben wir beide Referenzen erzeugt, können wir Breite und Höhe der auszugebenden Zeichenkette bestimmen. Erstere Information liefert uns die Methode `stringWidth()` der `FontMetrics`-Klasse, wenn wir dieser die zugrunde liegende Zeichenkette übergeben und die Höhe gibt uns die Methode `getHeight()` der `LineMetrics`-Instanz zurück:

```
int width = fm.stringWidth(output);
float height = lm.getHeight();
```

Da wir perfektionistisch veranlagt sind, möchten wir den Text zentriert und in der Mitte des Applets ausgeben. Kein Problem – alle benötigten Informationen haben wir eigentlich schon zur Verfügung: Die Höhe des Applet-Fensters liefert uns die Methode `getHeight()` und die Breite erfahren wir durch `getWidth()`. Diese Werte müssen wir nur noch halbieren und auch noch die Text-Maße anteilig abziehen:

```
int displayX = (getWidth() / 2) - (width / 2);
float displayY = (getHeight() / 2) + (height / 2);
```

Zuletzt versehen wir den Ausgabe-Text noch mit einem Farbverlauf. Dies ist übrigens auch der Grund, warum wir eine `Graphics2D`-Instanz statt des gewöhnlichen `Graphics`-Objekts verwenden, denn dessen Methoden `setPaint()` können wir eine `Paint`-Instanz übergeben, die einen Farbverlauf definiert.

Farbverläufe werden durch `GradientPaint`-Instanzen repräsentiert. Deren überladener Konstruktor nimmt x- und y-Koordinaten des Farbverlauf-Starts in Bezug auf das Applet-Fenster (und nicht etwa auf das darzustellende Objekt) und die Start-Farbe entgegen. Als Parameter vier, fünf und sechs übergeben wir x- und y-Koordinaten sowie die Farbe des Endpunktes des Farbverlaufs. An dieser Stelle erweist es sich als nützlich, dass wir über genau diese Informationen bereits verfügen.

```
Paint p = new GradientPaint(
   displayX, displayY, Color.red,
   displayX + width, displayY + height, Color.white);
```

Jetzt weisen wir nur noch die `Font`- und `Paint`-Instanzen zu und können dann den Text ausgeben:

```
g2.setPaint(p);
g2.setFont(f);

g2.drawString(output, displayX, displayY);
```

Das war eine Menge Aufwand, um einen einfachen Text anzuzeigen – aber dafür ist die Ausgabe auch ganz besonders gelungen:

Abbildung 10.13
Text mit Farbverlauf

10.4.3 Bilder laden

Natürlich müssen Sie beim Einsatz von Applets auch nicht auf die Möglichkeit verzichten, ein Bild zu laden und anzuzeigen. Das Applet stellt uns die Methode `getImage()` bereit, der wir `URL` und Name des zu ladenden Bildes übergeben können und die uns die `Image`-Instanz zurückgibt, die wir dann anzeigen können:

Listing 10.11
Anzeige eines Bildes im Applet

```
import java.applet.Applet;
import java.awt.*;
import java.awt.image.ImageObserver;

/**
 * Loads and displays an image
 */
public class AwtImage extends Applet {

   Image image;

   public void init() {
      image = getImage(getCodeBase(), "baby.jpg");
   }

   public void paint(Graphics g) {
      ImageObserver io = (ImageObserver)this;
      g.drawImage(image, 10, 10, io);
   }
}
```

Das Laden eines externen Bildes in ein Applet erfolgt durch dessen Methode `getImage()`, der wir als Parameter den URL zum Bild und den eigentlichen Dateinamen des Bildes übergeben müssen. Den URL zum Verzeichnis, in dem sich das Applet befindet, ermitteln wir mit Hilfe der Methode `getCodeBase()` des Applets:

```
image = getImage(this.getCodeBase(), "baby.jpg");
```

Nachdem wir so eine `Image`-Instanz erzeugt haben, können wir diese mit Hilfe der Methode `drawImage()` der `Graphics`-Instanz innerhalb von `paint()` ausgeben. Dies setzt allerdings voraus, dass wir einen `ImageObserver` erzeugen, der von der Methode `drawImage()` als letzter Parameter erwartet wird.

Glücklicherweise implementiert die `Applet`-Klasse das Interface `ImageObserver`, das verwendet wird, um Informationen zu Status und Maßen des Bildes zu erhalten – wir müssen also lediglich die aktuelle `Applet`-Instanz in eine `ImageObserver`-Instanz casten und können dann das Bild ausgeben:

```
ImageObserver io = (ImageObserver)this;
```

Da wir uns in einem Koordinaten-System für das Zeichnen von grafischen Elementen bewegen, müssen wir der Methode `drawImage()` auch die x- und y-Positionen der linken oberen Ecke des Bildes als Parameter übergeben.

Insgesamt übergeben wir also vier Parameter an `drawImage()`: Die `Image`-Instanz, die das Bild repräsentiert, die Koordinaten der linken oberen Ecke und die `ImageObserver`-Instanz:

```
g.drawImage(image, 10, 10, io);
```

Nun wird das Bild angezeigt:

Abbildung 10.14
Anzeige eines Bildes in einem Applet

Wenn Sie Bilder anzeigen wollen, die an einer unveränderlichen Ressource liegen, können Sie statt der Überladung von `getImage()` mit zwei Parametern (`URL` und `String`) die Variante mit nur einem Parameter verwenden:

```
try {
   URL url = new URL("http://localhost/data/baby.jpg");
   image = getImage(url);
} catch (MalformedURLException e) {}
```

Beachten müssen Sie, dass die Zeichenkette, die Sie dem Konstruktor der `URL`-Klasse übergeben, einen syntaktisch gültigen URL darstellt, da sonst eine `MalFormedURLException` geworfen wird.

10.5 Threading

Threading ist insbesondere bei Applets ein oftmals bestimmendes Thema. Threads werden innerhalb der `init()`- oder `start()`-Methoden gestartet und laufen so lange, bis die `stop()`- oder `destroy()`-Methoden eingebunden werden.

Eine Klasse, die als `Thread` gestartet werden soll, muss das Interface `Runnable` implementieren und kann dann ausgeführt werden:

Listing 10.12
Handling von Threads innerhalb von Applets

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

/**
 * Demonstrates the usage of threads
 */
public class AppletThread extends Applet
   implements Runnable, ActionListener {

   private final static String BUTTON_START = "Start thread";
   private final static String BUTTON_STOP = "Stop thread";

   private TextArea output;
   private Thread t;
   private java.awt.Button button;
   private boolean stop = false;

   /**
    * Inites the applet and generates the UI
    */
   public void init() {
      Panel content = new Panel(new BorderLayout());
      add(content);

      output = new TextArea(null, 2, 30,
         TextArea.SCROLLBARS_VERTICAL_ONLY);
      output.setEditable(false);
      content.add(output);

      button = new java.awt.Button(BUTTON_START);
      button.addActionListener(this);
      content.add(button, BorderLayout.SOUTH);
   }

   /**
    * Forces the thread to stop
    */
   public void stop() {
      stop = true;
   }

   /**
    * The thread's main method
    */
```

```
    public void run() {
       output.setText("Running...");
       stop = false;

       while(!stop) {
          output.setText(output.getText() + ".");
          try {
             Thread.sleep(100);
          } catch (InterruptedException e) { }
       }
    }

    /**
     * Handles the button's action-event
     */
    public void actionPerformed(ActionEvent e) {
       if(t == null) {
          t = new Thread(this);
          t.start();
          button.setLabel(BUTTON_STOP);
       } else {
          stop = true;
          t = null;
          button.setLabel(BUTTON_START);
       }
    }
}
```

Listing 10.12 *(Forts.)*
Handling von Threads innerhalb von Applets

init()

Innerhalb der Methode `init()` des `Applets` deklarieren wir eine `TextArea` und einen `Button`. Erstere dient der Anzeige des Lauf-Fortschrittes eines `Threads`, letzterer startet oder stoppt einen `Thread`. Zu diesem Zweck wird das `Applet`, das das Interface `ActionListener` implementiert, an den Button als Handler für das `action`-Ereignis gebunden und verfügt über eine Methode `actionPerformed()`, in der die Behandlung des Ereignisses stattfindet.

actionPerformed()

Innerhalb von `actionPerformed()` können wir bezüglich des `Threads` zwei Fälle unterscheiden: Entweder ist die `Thread`-Instanz `null`, dann läuft kein `Thread` und wir erzeugen einen neuen `Thread`. Oder die `Thread`-Instanz `t` ist ungleich `null`, dann stoppen wir den `Thread`.

Das Starten eines `Threads` geschieht, indem wir eine neue Instanz der `Thread`-Klasse erzeugen, deren Konstruktor eine `Runnable`-Instanz als Parameter erwartet. Diese `Thread`-Instanz wird dann mit Hilfe ihrer Methode `start()` aktiviert – diese bindet übrigens die im Interface `Runnable` definierte Methode `run()` ein:

```
t = new Thread(this);
t.start();
```

Zuletzt setzen wir den Anzeige-Text des `Buttons` auf den durch die Konstante `BUTTON_STOP` definierten Wert und lassen den `Thread` laufen, bis der Button erneut betätigt worden ist:

```
button.setLabel(BUTTON_STOP);
```

Wenn der `Button` erneut betätigt wurde, wird der `Thread` diesmal noch laufen. Aus diesem Grund weisen wir der als Stopp-Signal verwendeten Variablen `stop` den Wert `false` zu und signalisieren dem `Thread` damit, dass er sich beenden soll. Anschließend setzen wir die `Thread`-Instanz auf `null`, damit wir bei einer erneuten Betätigung des `Buttons` eine neue `Thread`-Instanz erzeugen können:

```
stop = true;
t = null;
```

Zuletzt weisen wir dem `Button` wieder den Anzeige-Text zum Starten eines neuen `Threads` zu:

```
button.setLabel(BUTTON_START);
```

run()

Die Methode `run()` des Applets ist die Steuer-Methode des `Threads`. Sie wird eingebunden, sobald der `Thread` gestartet worden ist, und läuft so lange, bis die Variable `stop` den Wert `true` hat und damit signalisiert, dass eine Beendigung des `Threads` erwünscht ist:

```
public void run() {

   // ...

   while(!stop) {
      // ...
   }
}
```

Innerhalb der `while`-Schleife geben wir in der `TextArea output` einen Hinweis aus, der den Nutzer darüber aufklärt, dass der Thread tatsächlich noch läuft. Um eine zu hohe Prozessorlast und unnötige Verschwendung von System-Ressourcen zu verhindern, lassen wir den `Thread` bei jedem Durchlauf der Schleife für 100 Millisekunden pausieren:

```
try {
   Thread.sleep(100);
} catch (InterruptedException e) { }
```

stop()

Zuletzt sei noch die Methode `stop()` erwähnt: Diese wird aufgerufen, wenn das Applet pausieren soll oder endgültig beendet wird. Hier weisen wir der Instanzvariablen `stop` den Wert `false` zu – genauso, als ob der `Button`, der das Starten oder Stoppen der `Threads` steuert, betätigt worden ist. Der `Thread` wird daraufhin nach spätestens 100 Millisekunden beendet.

Wenn wir das Applet ausführen, können wir auf den `Button` klicken und damit `Threads` starten und stoppen:

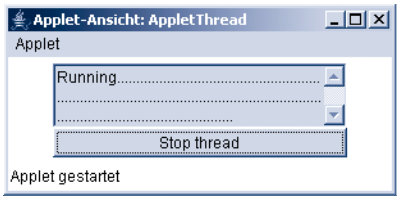


Abbildung 10.15
Ausgabe eines laufenden Threads

Mehr zum Thema Threading finden Sie im *Kapitel 6*.

10.6 Was Applets dürfen und was nicht

Applets sind ungleich Applets. Es ist beim Einsatz von Applets wichtig, dass Sie zwischen einem lokal geladenen Applet (das darf nämlich alles, was lokale Applikationen auch dürfen) und einem über das Netzwerk geladenen Applet unterscheiden. Letztere Applet-Form darf aus Sicherheitsgründen nur wenige Dinge. Zu den nicht erlaubten Dingen zählen unter anderem:

- Ein Applet darf keine anderen Libraries (eigene Packages) nachladen. Lediglich die Bibliotheken des `java.*`-Namensraumes dürfen benutzt werden.
- Ein Applet darf keine Dateien vom lokalen Rechner öffnen, lesen, ändern oder speichern. Dateien, auf die per URL-Instanz zugegriffen worden ist, können dagegen geöffnet und gelesen werden.
- Ein Applet darf keine Netzwerk-Verbindungen aufbauen. Die einzige Ausnahme sind Netzwerk-Verbindungen zu dem Host, von dem das Applet geladen worden ist.
- Ein Applet darf kein Programm auf dem lokalen Rechner starten.
- Ein Applet darf nur bestimmte System-Properties auslesen.

Der Einsatz von Applets ist also mit sehr starken Restriktionen verbunden. Aus diesem Grund werden Sie häufig Applets antreffen, die mit ihrem Host-System Client-Server-Verbindungen aufbauen, um somit einige der genannten Beschränkungen zu umgehen.

Im letzten Aufzählungspunkt ist die Rede davon, dass nur bestimmte System-Properties erreichbar sind. Dabei handelt es sich um die folgenden Schlüssel:

- `"file.separator"`: Gibt das Trennzeichen zwischen Pfad und Datei an
- `"java.class.version"`: Java-Class-Version
- `"java.vendor"`: Java-Anbieter-Kennung
- `"java.vendor.url"`: Homepage-Adresse des Anbieters
- `"java.version"`: Java-Version
- `"line.separator"`: Zeilen-Umbruchs-Token

- `"os.arch"`: Hardware-Architektur
- `"os.name"`: Betriebssystem-Name
- `"path.separator"`: Trenner zwischen Laufwerks- und Pfad-Angaben (soweit vorhanden)

Auf andere System-Properties kann aus Applets heraus möglicherweise zugegriffen werden, jedoch ist der Zugriff nicht garantiert, sondern von der Implementierung der Java-Runtime auf dem Client-System abhängig.

Auf folgende Properties kann in jedem Fall nicht zugegriffen werden:

- `"java.class.path"`: Java-Klassenpfad
- `"java.home"`: Installationsverzeichnis der Java-Instanz
- `"user.dir"`: Aktuelles Arbeitsverzeichnis
- `"user.home"`: Home-Verzeichnis des Users
- `"user.name"`: Anmeldename des Users

Der einzige Weg, um all diese Beschränkungen zu umgehen, ist die Signierung des Applets.

10.7 Interaktion mit umgebenden Elementen

Die Interaktion mit den umgebenden Elementen eines Applets kann auf mehrere Arten stattfinden. Wir gehen hier von einem Applet aus, das in einem Browser-Fenster läuft und durch eine HTML-Webseite per `<applet>`-Tag eingebunden worden ist.

10.7.1 Übergabe von Parametern an das Applet

Das wesentliche Element der Interaktion in Richtung zum Applet ist die Übergabe von Parametern. Dem Entwickler erleichtert das Entgegennehmen von Parametern die Entscheidungsfindung, denn schließlich muss er bestimmte Dinge nicht fest verdrahten, sondern kann sie flexibel gestalten.

Für den Nutzer wiederum ist auch die Flexibilität das ausschlaggebende Kriterium, denn so kann er das Applet und dessen Erscheinungsweise an seine persönlichen Vorlieben anpassen.

Die Übergabe von Parametern an ein Java-Applet erfolgt innerhalb des `<applet>`-Tags in Form von eingebundenen `<param>`-Tags. Diese Tags definieren Name-Wert-Paare. Innerhalb des Applets können die in den `<param>`-Tags definierten Werte mit Hilfe der Methode `getParameter()` eingelesen werden:

Listing 10.13
Einlesen von Parametern

```
import javax.swing.*;
import java.awt.*;

/**
 * Displays a JApplet in the defined fore-
 * and back-colors
 */

public class Parameters extends JApplet {

   Color backColor;
   Color foreColor;

   /**
    * Reads the given parameters
    */
   public void init() {
      String fc = getParameter("foreColor");
      if(null != fc) {
         foreColor = Color.decode(fc);
      } else {
         foreColor = Color.blue;
      }

      String bc = getParameter("backColor");
      if(null != bc) {
         backColor = Color.decode(bc);
      } else {
         backColor = Color.lightGray;
      }
   }

   /**
    * Displays the contents
    */
   public void start() {
      JPanel panel = new JPanel(new BorderLayout());
      panel.setBackground(backColor);
      add(panel);

      JLabel out = new JLabel(
         "This text is displayed in the foreColor",
         SwingConstants.CENTER);
      out.setForeground(foreColor);
      panel.add(out);
   }
}
```

Das Applet – hier übrigens eine `JApplet`-Instanz, die statt `AWT` das `Swing`-/`JFC`-Framework für die Darstellung verwendet – liest in seiner `init()`-Methode die übergebenen Parameter ein. Dies geschieht, indem es den Namen des Parameters an die Methode `getParameter()` übergibt und deren Rückgabe auf `null` überprüft:

```
String fc = getParameter("foreColor");
if(null != fc) {
   // ...
}
```

Da wir in diesem Fall Farbwerte einlesen, müssen wir die übergebenen Informationen in `Color`-Instanzen umwandeln. Nun könnten wir sicherlich den Wert des Parameters selbstständig analysieren, doch warum sollten wir uns so viel Arbeit machen? Die Methode `decode()` der `Color`-Klasse erledigt genau dies und gibt uns eine `Color`-Instanz, die den gewünschten Farbwert repräsentiert, zurück:

```
foreColor = Color.decode(fc);
```

Ist der Wert des Parameters dagegen `null`, ist dieser gar nicht übergeben worden. Wir greifen dann auf einen Standard-Wert zurück:

```
foreColor = Color.blue;
```

Hinweis

Das Vorgehen beim Einlesen von Parametern in Applets muss immer das folgende sein:

- Einlesen des Wertes
- Überprüfen, ob der Wert ungleich `null` ist
- Falls der Wert gleich `null` ist, dann Zurückgreifen auf einen Standard-Wert

Nach dem Einlesen der Parameter kann das Applet seine Komponenten visualisieren.

Wenn das Applet in einer HTML-Seite eingebunden werden soll, können Sie folgenden HTML-Code verwenden:

Listing 10.14
HTML-Code für die Anzeige eines Applets und das Setzen von Parametern

```
<html>
   <head>
      <title>Parameters and applets</title>
   </head>
   <body bgColor="#efefef">
      <applet code="Parameters.class"
         width="300" height="100">
         <param name="foreColor" value="#cc0000"></param>
         <param name="backColor" value="#cccccc"></param>
      </applet>
   </body>
</html>
```

Innerhalb des `<applet>`-Tags werden die Werte für die beiden Parameter deklariert. Wir übergeben hier für die Vordergrund-Farbe den hexadezimalen Wert für Rot und definieren die Hintergrund-Farbe als einen mittleren Grau-Ton.

Wenn wir dann die HTML-Seite im Browser aufrufen, sehen wir das Ergebnis unserer Anpassungen:

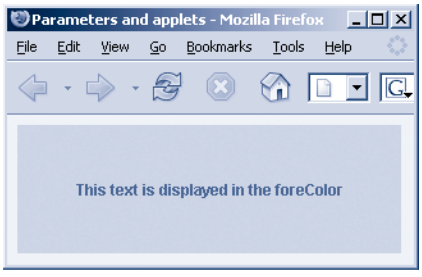


Abbildung 10.16
Die Farben des Applets wurden als Parameter übergeben

Bevor Sie allerdings Parameter verwenden, sollten Sie sich durchaus intensiv Gedanken über die folgenden Punkte machen:

- Welche Parameter werden benötigt?
- Wie lauten kurze, prägnante und beschreibende Namen für diese Parameter?
- Welche Werte-Typen werden erwartet?
- Welche Standard-Werte können gesetzt werden?

Erst, nachdem Sie sich über diese Fragen klar geworden sind, sollten Sie die Unterstützung für Parameter in Ihren Applets implementieren. Bedenken Sie auch, dass ein Zuviel an Parametrisierbarkeit die Nutzer Ihrer Applikationen möglicherweise mehr verwirren kann, als dass es ein Mehr an Nutzen brächte.

10.7.2 Das <applet>-Tag und seine Parameter

Mit Hilfe des `<applet>`-Tags können Sie noch weitaus mehr an Informationen übergeben oder Einstellungen setzen, als wir dies bisher gemacht haben.

Die Syntax beim Einsatz des `<applet>`-Tags innerhalb von HTML-Seiten sieht so aus:

```
<applet
   code="class-Datei"
   [width="Pixel"]
   [height="Pixel"]
   [archive="JAR-Datei"]
   [codebase="CodeBase-URL"]
   [alt="Alternativer Text"]
   [name="Instanz Name"]
   [align="Ausrichtung"]
   [vspace="Pixel"]
   [hspace="Pixel"]
>
   [<param name="name" value="value"></param>]...
   [alternatives HTML]
</applet>
```

Der Parameter `code` ist der einzige Parameter, der obligatorisch beim Einsatz des Applets ist. Alle anderen Parameter können weggelassen werden.

Die einzelnen Parameter haben diese Bedeutung:

- `code`: class-Datei des Applets, das angezeigt werden soll
- `width`: Breite der Anzeige im Browser
- `height`: Höhe der Anzeige im Browser
- `archive`: *JAR*-Datei, in der das Applet und seine Ressourcen sowie weitere verwendete Libraries enthalten sind. Mehrere *JAR*-Dateien können durch Kommata voneinander getrennt werden
- `codebase`: CodeBase-Angabe, an der das Applet externe nachzuladende Ressourcen erwartet. Beachten Sie, dass diese Angabe auf den Host verweisen muss, von dem das Applet geladen worden ist
- `alt`: Alternativer Text, der vom Browser angezeigt wird, wenn der Mauszeiger über dem Applet verharrt. Die Anzeige ist Browser-abhängig
- `name`: Name des Applets
- `align`: Ausrichtung des Applets innerhalb der umgebenden HTML-Elemente. Mögliche Werte sind left, center und right
- `vspace`: Abstand der benachbarten HTML-Elemente in vertikaler Richtung
- `hspace`: Abstand der benachbarten HTML-Elemente in horizontaler Richtung

Wie bereits demonstriert, können Sie dem Applet Parameter übergeben. Dies erfolgt, indem Sie in das `<applet>`-Tag `<param>`-Tags einfügen. Das `<param>`-Tag hat genau zwei Attribute:

- `name`: Name des Parameters
- `value`: Wert des Parameters

Sie können in einem `<applet>`-Tag auch einen alternativen HTML-Code einfügen. Dieser wird angezeigt, falls der Browser das Applet aus Sicherheitsgründen nicht anzeigen kann oder auf dem System keine Java-Unterstützung existiert.

10.7.3 Beeinflussung des Status-Textes des Fensters

Ein Applet kann mit dem Browser in Grenzen kommunizieren. Eine Art dieser Kommunikation besteht darin, einen Text in der Status-Zeile auszugeben. Wird ein Applet im Applet-Viewer ausgeführt, erfolgt die Ausgabe des Textes in dessen Status-Zeile. Beim Einsatz in einem Browser werden die Informationen in der Status-Zeile des Browsers angezeigt – es kann also durchaus einen gewissen räumlichen Abstand zwischen Applet und Status-Zeile geben.

Der Einsatz der Statusleiste zur Ausgabe von Informationen unterliegt allerdings einigen Restriktionen:

- Die Statuszeile muss nicht zwingend angezeigt werden
- Der Text in der Statuszeile ist nicht prominent sichtbar
- Der Text in der Statuszeile kann von anderen Applets, dem Browser oder per JavaScript geändert werden

Somit disqualifiziert sich der Einsatz der Statusleiste für alles andere als optionale Status-Informationen.

Um nun aber einen Text in der Statuszeile auszugeben, verwenden Sie die Methode `showStatus()` des Applets, der Sie den anzuzeigenden Text als Parameter übergeben:

```
showStatus("This is a short text which will be displayed " +
   "in the browser's status line");
```

In einem Applet eingebaut und per Browser angeschaut, wird dieser Text in der Statuszeile des Browsers angezeigt – allerdings nur, wenn diese sichtbar ist und keine anderen Ausgaben des Applets oder anderer Seiten-Komponenten erfolgt sind.

10.7.4 Kommunikation mit anderen Applets in derselben Seite

Java-Applets können mit anderen Applets auf der gleichen HTML-Seite kommunizieren und diese wie normale Java-Komponenten ansprechen. Bevor dies möglich ist, müssen wir eine Referenz auf das andere Applet herstellen – damit das funktioniert, müssen wir es aber erst einmal finden.

Zum Finden eines uns namentlich bekannten Applets können wir die Methode `getApplet()` verwenden, der wir den Namen des gesuchten Applets übergeben müssen. Wenn wir alle Applets einer Seite ermitteln wollen, nutzen wir die Methode `getApplets()`.

Lassen Sie uns dies an einem Beispiel verdeutlichen: Auf einer HTML-Seite werden wir zwei Applets einsetzen – eines zum Eingeben eines Textes und ein zweites, das den Text anzeigt. Das sendende Applet wird versuchen, eine Referenz auf das darstellende Applet zu ermitteln – das Vorgehen ist dabei, das Darstellungsapplet zu finden, indem wir alle auf der Seite vorhandenen Applets ermitteln und durchlaufen.

Werfen wir zunächst einen Blick auf das Darstellungsapplet:

Listing 10.15
Das Applet CommunicateReceiver stellt empfangene Nachrichten dar

```
import javax.swing.*;
import java.awt.*;

/**
 * Displays a message sent by another Applet
 */
public class CommunicateReceiver extends JApplet {

   private JLabel label;

   public void init() {
      JPanel content = new JPanel(new GridBagLayout());
      add(content);
```

Listing 10.15 *(Forts.)*
Das Applet Communicate-Receiver stellt empfangene Nachrichten dar

```
        label = new JLabel("", SwingConstants.CENTER);
        GridBagConstraints c = new GridBagConstraints();
        c.gridx = 0;
        c.gridy = 0;
        c.weightx = 1;
        c.weighty = 1;
        c.fill = GridBagConstraints.BOTH;

        content.add(label, c);
    }

    public void displayMessage(String message) {
        label.setText(message);
    }
}
```

An dem Applet `CommunicationReceiver` fällt einem nichts Besonderes auf, außer dass es eine Methode `displayMessage()` besitzt, die den übergebenen Text in einem Label ausgibt. Dieses Applet wird – wenn es sich alleine in einer Webseite befindet – nichts Sinnvolles machen. Eigentlich macht es überhaupt nichts – bis auf das Ausgeben der Nachricht.

Anders dagegen das Applet `CommunicationSender`: Es stellt dem Nutzer ein Textfeld zum Eingeben und einen `Button` zum Senden der Nachricht zur Verfügung. An den Button ist ein `ActionListener` gebunden – die Methode `actionPerformed()` wird also ausgeführt, wenn der `Button` betätigt worden ist. In ihr findet das Übermitteln der Nachricht an das Empfänger-Applet statt:

Listing 10.16
Das Applet Communication-Sender übermittelt Nachrichten an andere Applets

```
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.applet.Applet;
import java.util.Enumeration;

/**
 * Takes a message and sends it to the user
 */
public class CommunicateSender extends Applet
    implements ActionListener {

    TextField input;
    CommunicateReceiver receiver;

    public void init() {
        Panel content = new Panel(new GridBagLayout());
        add(content);

        GridBagConstraints c = new GridBagConstraints();
        c.gridx = 0;
        c.gridy = 0;
        c.gridwidth = 3;

        input = new TextField(20);
        content.add(input, c);
```

Listing 10.16 *(Forts.)*
Das Applet Communication-Sender übermittelt Nachrichten an andere Applets

```
      java.awt.Button submit = new java.awt.Button("Submit!");
      submit.addActionListener(this);
      c.weightx = 0;
      c.gridwidth = 1;
      c.gridx = 3;
      content.add(submit, c);

      Enumeration applets =
         getAppletContext().getApplets();
      while(applets.hasMoreElements()) {
         Applet applet = (Applet)applets.nextElement();
         if(applet instanceof CommunicateReceiver) {
            receiver = (CommunicateReceiver)applet;
         }
      }
   }

   public void actionPerformed(ActionEvent e) {
      if(receiver != null) {
         receiver.displayMessage(input.getText());
      }
   }
}
```

In der Methode `init()` des Applets werden nicht nur die Anzeige- und Eingabe-Elemente erzeugt, sondern es werden auch alle anderen Applets des aktuellen Applet-Kontextes ermittelt und in Form einer Enumeration durchlaufen:

```
Enumeration applets = getAppletContext().getApplets();
while(applets.hasMoreElements()) {
   // ...
}
```

Alle Elemente der Auflistung sind vom Typ `Applet`. Uns interessieren aber nicht generell alle Applets, sondern nur die Applets, die `CommunicationReceiver`-Instanzen darstellen – also prüfen wir die einzelnen Elemente darauf, ob sie dieser Bedingung entsprechen. Dies geschieht mit Hilfe des Schlüsselworts `instanceof`. Wenn das Applet vom Typ `CommunicationReceiver` ist, dann weisen wir es der Instanz-Variablen `receiver` zu, auf die wir in der Methode `actionPerformed()` später wieder zurückgreifen werden:

```
if(applet instanceof CommunicateReceiver) {
   receiver = (CommunicateReceiver)applet;
}
```

Die bereits angesprochene Methode `actionPerformed()` wird eingebunden, wenn der User den `Button` zum Senden der Nachricht anklickt. In ihr wird – falls die Instanzvariable `receiver` ungleich `null` ist und wir also ein `CommunicationReceiver`-Applet finden konnten – dem Empfänger die vom User eingegebene Nachricht übergeben:

```
if(receiver != null) {
   receiver.displayMessage(input.getText());
}
```

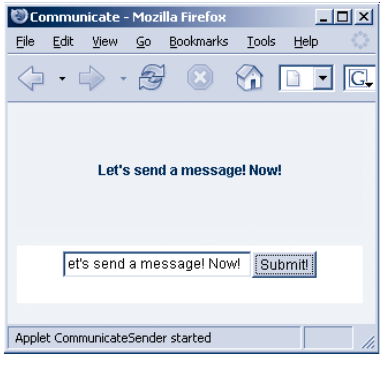
Nun können wir beide Applets innerhalb von einer HTML-Seite anzeigen lassen:

Listing 10.17
Anzeige der beiden Applets in einer HTML-Seite

```
<html>
   <head>
      <title>Communicate</title>
   </head>
   <body bgColor="#efefef">
      <p>
         <applet code="CommunicateReceiver.class"
            width="300" height="100"></applet>
      </p>
      <p>
         <applet code="CommunicateSender.class"
            width="300" height="50"></applet>
      </p>
   </body>
</html>
```

Ruft der User diese Seite nun auf, werden ihm zwei Applets angezeigt – das obere der beiden Applets gibt die im unteren Applet eingegebenen Nachrichten aus:

Abbildung 10.17
Das obere Applet stellt die Nachrichten den unteren Applets dar



Damit die Kommunikation zwischen Applets funktioniert, muss das Ziel-Applet der Kommunikation mindestens eine als `public` gekennzeichnete Methode bereitstellen, die vom Sender aufgerufen werden kann.

10.7.5 Redirects im Browser

Mit Hilfe der Methode `showDocument()` können wir den Browser zwingen, einen anderen URL aufzurufen. Dabei muss die neue Adresse nicht im aktuellen Fenster geöffnet werden, sondern der Browser kann angewiesen werden, bestimmte bereits existierende Fenster oder neue Fenster zu verwenden:

Listing 10.18
Redirect zu einer vom Nutzer angegebenen Adresse

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.net.URL;
import java.net.MalformedURLException;

/**
 * Lets the user enter an URL and redirects to it
```

Listing 10.18 *(Forts.)*
Redirect zu einer vom Nutzer angegebenen Adresse

```
 */
public class Redirect extends JApplet
   implements ActionListener {

   JTextField input;
   JComboBox box;

   /**
    * Builds the UI
    */
   public void init() {
      JPanel content = new JPanel(new GridLayout(3, 1));
      add(content);

      input = new JTextField(20);
      content.add(input);

      String[] values = {
         "Let the browser decide", "_blank", "_top", "_self"};
      box = new JComboBox(values);
      content.add(box);

      JButton submit = new JButton("Go!");
      submit.addActionListener(this);
      content.add(submit);
   }

   /**
    * Processes the action-event and redirects to the given target
    */
   public void actionPerformed(ActionEvent e) {
      URL address = null;
      try {
         address = new URL(input.getText());
      } catch (MalformedURLException e1) {
         e1.printStackTrace();
      }

      if(address != null) {
         if(box.getSelectedIndex() == 0) {
            getAppletContext().showDocument(address);
         } else {
            getAppletContext().showDocument(
               address, (String)box.getSelectedItem());
         }
      }
   }
}
```

Die Methode `showDocument()` stellt uns zwei Überladungen zur Verfügung: eine Variante, die eine `URL`-Instanz als Parameter entgegennimmt, und eine zweite, die daneben noch einen `String` erwartet, der das zu verwendende Browser-Fenster angibt.

Aus diesem Grund stellen wir es dem Nutzer frei, festzulegen, welches Browser-Fenster für die Weiterleitung auf die neue Adresse verwendet werden soll, und geben ihm die Möglichkeit, dies in einer `JComboBox` auszuwählen:

```
String[] values = {
   "Let the browser decide", "_blank", "_top", "_self"};
box = new JComboBox(values);
```

Die einzelnen Werte haben dabei folgende Bedeutung:

- `"Let the browser decide"`: Es wird keine Angabe über den Namen des zu verwendenden Fensters an den Browser gegeben.
- `"_blank"`: Der Browser soll die neue Seite in einem neuen Fenster anzeigen.
- `"_top"`: Der Browser soll die neue Seite im obersten Fenster des aktuellen Frames anzeigen.
- `"_self"`: Der Browser zeigt die neue Seite im aktuellen Fenster an.

Sollte der Benutzer eine neue Adresse angeben und den Button für das Weiterleiten anklicken, wird die Methode `actionPerformed()` eingebunden. Hier erzeugen wir aus der eingegebenen Web-Adresse eine neue `URL`-Instanz:

```
address = new URL(input.getText());
```

Sollte der Nutzer den Standard-Wert der ComboBox ausgewählt haben, liefert deren Eigenschaft `getSelectedIndex()` den Wert 0 zurück. In diesem Fall soll der Browser die eingegebene Adresse entsprechend seiner Implementierung öffnen – in der Regel wird dies also im aktuellen Browser-Fenster sein:

```
if(box.getSelectedIndex() == 0) {
   getAppletContext().showDocument(address);
}
```

Jeder andere Wert der ComboBox wird als zweiter Parameter an die Methode `showDocument()` übergeben:

```
getAppletContext().showDocument(
   address, (String)box.getSelectedItem());
```

Nutzer, die das Applet in einem Browser aufrufen, können unter anderem diese Ausgabe erhalten:

Abbildung 10.18
Per showDocument() wurde ein neues Fenster geöffnet

Beachten Sie, dass die Funktionalität des Öffnens neuer Browser-Fenster nicht mit jedem Browser funktioniert – dies ist stark von der Implementierung abhängig. Was in jedem Fall funktioniert, ist das Weiterleiten auf eine Adresse per `showDocument()` ohne explizite Angabe eines Browser-Fensters.

10.8 Applet – JApplet?

Wenn Sie sich die letzten Beispiele intensiver angeschaut haben, werden Sie festgestellt haben, dass einige Beispiele nicht von der Basis-Klasse `Applet`, sondern von `JApplet` erben. Wie das vorangestellte „J" schon andeutet, handelt es sich um eine Klasse aus dem `JFC`-Framework, besser bekannt als `Swing`.

Der Unterschied zwischen `Applet` und `JApplet` als Basisklasse besteht neben einigen Implementierungs-Details darin, dass Sie `Applets` `AWT`-Komponenten zuweisen können und `JApplets` `Swing`-Komponenten erwarten.

Weiterhin gibt es einen wesentlichen Unterschied im Sinne der erwarteten Java-Version: Während das klassische `Applet` bereits seit Version 1.0 zur Verfügung steht, funktioniert das `JApplet` erst ab Java-Version 1.2 (die auch gerne als Java 2.0 bezeichnet wird). Mittlerweile kann man Nutzer ermutigen, zumindest auf Java 1.4 umzusteigen – auch Windows-Nutzer können sich eine aktuelle Version installieren.

Generell empfiehlt es sich, `Swing`-Komponenten statt der `AWT`-Geschwister einzusetzen und die Klassen immer von `JApplet` erben zu lassen. Der Rückgriff auf `Applet` statt `JApplet` als Basisklasse ist zwar möglich und auch zulässig, jedoch verschenkt man viele Möglichkeiten, wenn man auf `JFC`/`Swing` verzichtet. Alle Beispiele dieses Kapitels sind so geschrieben, dass sie recht einfach und schnell von `AWT` auf `JFC` geändert werden können.

Wenn Sie also auf das letzte Quäntchen an Kompatibilität verzichten können, ermutigen Sie Ihre User, auf eine aktuellere Java-Version umzusteigen, und verwenden Sie das `JApplet` als Basis-Klasse.

10.9 Fazit

`Applets` sind kleine Programme, die innerhalb des Applet-Viewers oder eines Browsers angezeigt werden. Sie können über Netzwerke verteilt werden, unterliegen aber einigen Sicherheits-Restriktionen. Wenn Sie auf das allerletzte Quäntchen an Abwärtskompatibilität Ihrer Applikation verzichten können, verwenden Sie statt der `Applet`- die `JApplet`-Klasse als Basis-Klasse und setzen statt `AWT`- lieber `Swing`/`JFC`-Komponenten ein.

Eine intensive Beschäftigung mit Applets kann durchaus lohnen – auch abseits von verspielten Applikationen, wie man sie aus dem Internet kennt, denn die Funktionalität, die Applets bereitstellen, ist beachtlich. Insbesondere im Zusammenspiel mit einem Server können hier beeindruckende Ergebnisse erzielt werden.

11

Streams und Datei-Behandlung

Die wenigsten Applikationen kommen ohne Datei-Handling aus. Und noch seltener werden wir Applikationen antreffen, die auf den Einsatz von Streams verzichten können. Grund genug, sich dieses Themas anzunehmen.

11.1 Streams und zugrunde liegende Prinzipien

Streams sind Klassen, die es erlauben, Daten von einer Informationsquelle einzulesen. Diese Informationsquelle kann beispielsweise eine Datei oder ein Socket oder ein Speicher-Segment sein. Streams kapseln den Zugriff auf die Quelle, so dass wir über die Art der Quelle nicht mehr als nötig wissen müssen.

Das Stream-Prinzip funktioniert auch in entgegengesetzter Richtung – Streams erlauben es uns, Daten zu einem Ziel zu senden. Dabei ist es für uns relativ unwichtig, ob es sich bei diesem Ziel um eine Datei, ein Ziel im Netzwerk oder ein Arbeitsspeicher-Segment handelt.

Egal, welche Art von Daten wir laden wollen, der grundlegende Ablauf der Arbeit mit Streams bleibt in jedem Fall gleich:

1. Stream öffnen
2. Solange der Stream Informationen zurückgibt: Daten einlesen
3. Stream schließen

Ebenso gleicht sich der Ablauf, wenn in einen Stream geschrieben werden soll:

1. Stream öffnen
2. Solange Informationen vorhanden sind: Informationen in Stream schreiben
3. Stream schließen

Sämtliche Streams befinden sich im Paket `java.io`. Um also mit Streams arbeiten zu können, müssen wir das Package importieren. Allerdings werden Sie schnell

feststellen, dass es eine Menge an verschiedenen Streams gibt, denen wir uns später etwas genauer widmen werden.

Lassen Sie uns nun einen Blick auf eine ganz wichtige und wesentliche Unterteilung von Streams werfen, über die Sie früher oder später stolpern werden und die die Arbeit mit Streams deutlich vereinfachen oder erschweren kann – je nachdem, von welchem Standpunkt aus wir es betrachten.

11.1.1 Byte-Streams

Byte-Streams repräsentieren eine Art des Datenzugriffs, bei dem jede Daten-Einheit 8 Bit (1 Byte) lang ist. Dies ist ideal für binäre Daten oder textuelle Daten, die in einer einfachen Codierung vorliegen. Problematisch wird dieser Ansatz, wenn wir mit Inhalten in `Unicode`-Codierung (also Texten und nicht binären Daten) arbeiten wollen – die sind so einfach nicht darstellbar.

Byte-Streams erben (je nach Einsatzzweck) von den Basisklassen `InputStream` (Einlesen von Informationen) und `OutputStream` (Ausgabe). Diese Basisklassen stellen die grundlegende API für Streams, die Daten lesen oder Daten schreiben sollen, bereit. In der Regel werden sie für den Zugriff auf binäre Daten verwendet.

InputStream

Die (abstrakte) Basisklasse für den lesenden Zugriff auf Streams ist `InputStream`. Diese stellt uns folgende Methoden zur Verfügung:

```
public abstract int read()
   throws IOException

public int read(byte[] b)
   throws IOException

public int read(byte[] b, int off, int len)
   throws IOException

public long skip(long n)
   throws IOException

public int available()
   throws IOException

public void close()
   throws IOException

public void reset()
   throws IOException

public void mark(int readlimit)

public boolean markSupported()
```

Die drei `read()`-Methoden lesen Bytes ein – wobei diese entweder als einzelner Wert oder als Byte-Array zurückgegeben werden.

Die Methode `skip()` dient dazu, die übergebene Anzahl an Bytes zu überspringen. Wenn Sie `available()` aufrufen, können Sie die Anzahl der Bytes erhalten, die derzeit zum Abruf bereitstehen – wobei der Wert der Rückgabe stets von der tatsächlichen Implementierung der Klasse abhängig ist.

Den Stream schließen Sie, indem Sie dessen Methode `close()` aufrufen. Wenn Sie ohne Aufwand zu einer vorher definierten Position zurückkehren wollen, rufen Sie die Methoden `reset()` auf.

Sollte der Stream das Markieren von bestimmten Stellen unterstützen, gibt die Methode `markSupported()` den Wert `true` zurück. Sie können dann mit Hilfe von `mark()` die aktuelle Position markieren, so dass wir durch `reset()` an diese Stelle zurückkehren können. Das Argument der Methode gibt an, wie viele Bytes der Stream sich merken soll.

OutputStream

Das Gegenstück zu `InputStream` repräsentiert `OutputStream`. Diese abstrakte Klasse ist Basis aller ableitenden Byte-Output-Stream-Implementierungen und stellt uns folgende Methoden zur Verfügung:

```
public abstract void write(int b)
   throws IOException

public void write(byte[] b)
   throws IOException

public void write(byte[] b, int off, int len)
   throws IOException

public void flush()
   throws IOException

public void close()
   throws IOException
```

Die drei `write()`-Methoden nehmen einzelne Bytes oder Byte-Arrays als Parameter entgegen. Die bei der dritten `write()`-Überladung angegebenen Parameter `off` und `len` geben Offset und Anzahl der zu schreibenden Bytes aus dem übergebenen Byte-Array an. Sie haben dadurch die Möglichkeit, nicht das komplette Byte-Array (wie in der zweiten Überladung) zu speichern, sondern einen Ausschnitt daraus.

Das Schreiben des Streams findet je nach Ableitung nicht sofort statt. Stattdessen werden die Daten in einem Puffer zwischengespeichert – die Methode `flush()` löscht diesen Puffer und sorgt dafür, dass sein Inhalt zuvor auf das Ausgabe-Medium übertragen wird.

Mit Hilfe von `close()` wird der Stream geschlossen.

11.1.2 Character-Streams

Character-Streams verwenden im Gegensatz zu Byte-Streams grundsätzlich 16 Bit, um die Informationen zu repräsentieren. Dies ermöglicht eine deutlich bessere Anpassung an unterschiedliche Zeichensätze. Dies prädestiniert sie für den Zugriff auf Textdateien.

Reader

Der `Reader` ist die Basisklasse aller Eingabe-Ströme im Umfeld von Character-Streams. Die Klasse stellt uns folgende Methoden zur Verfügung, die sich auch in allen ableitenden Klassen wiederfinden lassen:

```
public int read(CharBuffer target)
   throws IOException

public int read()
   throws IOException

public int read(char[] cbuf)
   throws IOException

public abstract int read(char[] cbuf, int off, int len)
   throws IOException

public long skip(long n)
   throws IOException

public boolean ready()
   throws IOException

public void reset()
   throws IOException

public abstract void close()
   throws IOException

public boolean markSupported()

public void mark(int readAheadLimit)
   throws IOException
```

Der Reader stellt einige `read()`-Methoden zum Lesen von Daten zur Verfügung. Die `read()`-Version ohne Argumente liest das nächste Zeichen des Streams ein und gibt seine Entsprechung als int zurück. Dessen Wert kann zwischen 0 und 65535 liegen. Sollte der Rückgabewert -1 sein, dann kennzeichnet dies das Ende des Streams.

Die anderen drei `read()`-Überladungen übertragen Zeichen in den übergebenen `CharBuffer` (etwa eine `String`-Instanz) oder das übergebene `Char`-Array. Die Rückgabe dieser Methoden stellt die Anzahl der gelesenen Zeichen dar. Eine Rückgabe von -1 gibt auch hier an, dass das Ende des Streams erreicht worden ist.

Um Zeichen zu überspringen, verwenden Sie die Methode `skip()`. Dieser übergeben Sie die Anzahl der zu überspringenden Zeichen als Parameter.

Falls der Stream nicht bereit ist, weitere Zeichen zurückzugeben, liefert seine Methode `ready()` den Wert `false` zurück. Den Stream schließen Sie mit Hilfe seiner Methode `close()`. Um auf eine gemerkte Lese-Position zurückzugehen, verwenden wir die Methode `reset()`. Sollte keine Position gemerkt worden sein, springt der Stream auf seine Start-Position zurück.

Um festzustellen, ob der Stream das Merken von Positionen unterstützt, werten wir die Rückgabe der Methode `markSupported()` aus. Falls deren Rückgabe `true` ist, können wir den Stream die aktuelle Position mit Hilfe von `mark()` merken lasseb. Als Parameter übergeben Sie der Methode die Anzahl der vorzumerkenden Zeichen.

Writer

Zum Schreiben in Character-Streams verwenden wir `Writer`-Ableitungen. Diese Basisklasse stellt uns folgende Methoden zur Verfügung, die wir auch in allen ableitenden Klassen finden können:

```
public void write(int c)
   throws IOException

public void write(char[] cbuf)
   throws IOException

public abstract void write(char[] cbuf, int off, int len)
   throws IOException

public void write(String str)
   throws IOException

public void write(String str, int off, int len)
   throws IOException

public Writer append(CharSequence csq)
   throws IOException

public Writer append(char c)
   throws IOException

public abstract void flush()
   throws IOException

public abstract void close()
   throws IOException
```

Die sechs `write()`-Überladungen speichern die übergebenen Zeichen, Zeichen-Arrays (oder Teile davon), Strings (oder Teile davon) oder `CharSequence`-Instanzen in den Stream. Dabei werden die Daten – ebenso wie bei den `append()`-Methoden, die an das Stream-Ende ein Zeichen oder eine `CharSequence`-Instanz anfügen – nicht sofort geschrieben. Dies geschieht, sobald der interne Schreib-Puffer voll ist oder die Methode `flush()` aufgerufen wurde. Diese leert den Schreib-Puffer, nicht ohne allerdings die Daten zu speichern.

Wenn der Schreibvorgang abgeschlossen ist, kann der Stream mit Hilfe seiner Methode `close()` geschlossen werden. Auch hier wird zuvor der Schreib-Puffer geleert, so dass keine Informationen verloren gehen.

11.1.3 Character- oder Byte-Streams?

Die Antwort auf diese Frage ist einfach: Wenn Sie nicht zwingend eine Kompatibilität zu Java 1.0 benötigen sowie keine binären Daten lesen oder schreiben wollen, verwenden Sie Character-Streams. Diese sind einfacher zu handhaben und unterstützen auch exotische Zeichensätze.

Haben Sie dagegen binäre Dateien zu verarbeiten, wollen Ihre Applikation Java 1.0-kompatibel halten oder müssen Informationen schreiben, die zwingend als 8-Bit-Daten vorliegen sollen, dann verwenden Sie die älteren und etwas komplexeren Byte-Streams.

11.1.4 Brückenklassen

Es gibt zwei Klassen, die eine Art Brücke zwischen beiden Stream-Arten darstellen. Mit deren Hilfe haben wir die Möglichkeit, Streams als Byte-Stream zu öffnen und anschließend wie einen Character-Stream zu verarbeiten.

InputStreamReader

Die Klasse `InputStreamReader` stellt eine Brücke zwischen Byte-Streams und Character-Streams dar. Im Konstruktor nimmt sie einen Byte-Stream entgegen und kann anschließend wie ein Character-Stream agieren. Sie erbt von der abstrakten Basisklasse `Reader` und implementiert deren abstrakte Methoden.

Der Konstruktor der Klasse ist mehrfach überladen:

```
public InputStreamReader(InputStream in)

public InputStreamReader(InputStream in, String charsetName)
   throws UnsupportedEncodingException

public InputStreamReader(InputStream in, Charset cs)

public InputStreamReader(InputStream in, CharsetDecoder dec)
```

Neben einer `InputStream`-Instanz können wir in verschiedenen Versionen ein `Charset` angeben. Wir haben damit die Möglichkeit, auch Daten mit Umlauten korrekt einzulesen.

Der Umgang mit dem `InputStreamReader` ist recht einfach, da Sie selten direkt mit dieser Klasse arbeiten werden. Stattdessen wird sie gerne zum Casten von Stream-Instanzen in spezialisiertere Reader verwendet – wie in diesem Beispiel, wo wir mit Hilfe eines `InputStreamReaders` einen `BufferedReader` erzeugen, der einen `InputStream` konsumiert:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

BufferedReader rdr =
   new BufferedReader(new InputStreamReader(System.in));
```

Eine derartige Schachtelung von Reader-Instanzen werden Sie in der Praxis häufig antreffen. Sie ist auf die Brückenfunktion der `InputStreamReader`-Klasse zurückzuführen und erlaubt es den spezialisierten Reader-Klassen, ohne großen Overhead mit Byte-InputStreams zu arbeiten.

OutputStreamWriter

Das Gegenstück zum `InputStreamReader` ist der `OutputStreamWriter`. Diese Klasse erlaubt es `Writer`-Ableitungen, mit Byte-Output-Stream-Instanzen zu arbeiten. Intern wandelt der `OutputStreamWriter` die übergebenen Zeichen in Bytes um. Der dabei zu verwendende Zeichensatz kann auf Wunsch explizit angegeben werden.

Die `OutputStreamWriter`-Klasse erbt von der Basis-Klasse `Writer` und implementiert deren abstrakte Methoden. Zusätzlich stellt sie einen mehrfach überladenen Konstruktor bereit, der auf Wunsch den zu verwendenen Zeichensatz als Parameter entgegennimmt:

```
public OutputStreamWriter(
   OutputStream out, String charsetName)
   throws UnsupportedEncodingException

public OutputStreamWriter(OutputStream out)

public OutputStreamWriter(OutputStream out, Charset cs)

public OutputStreamWriter(
   OutputStream out, CharsetEncoder enc)
```

Ebenso wie der Umgang mit dem `InputStreamReader` gestaltet sich die Handhabung des `OutputStreamWriters` recht einfach, da Sie in der Praxis selten direkt mit ihr arbeiten werden. Stattdessen verwenden Sie den `OutputStreamWriter` als Brücke zwischen einem Output-Stream und spezialisierteren Writer-Klassen, so wie im folgenden Beispiel.

Hier verwenden wir die `OutputStreamWriter`-Klasse als Brücke zwischen einem `BufferedWriter` und der per `System.out` erreichbaren `PrintStream`-Instanz:

```
import java.io.BufferedWriter;
import java.io.OutputStreamWriter;

BufferedWriter writer =
   new BufferedWriter(new OutputStreamWriter(System.out));
```

Sinn macht eine derartige Schachtelung von Writern immer dann, wenn Sie in einen Stream schreiben wollen und dabei einen spezialisierten Writer verwenden möchten. Sie werden relativ häufig darauf in der Praxis stoßen.

11.2 Input- und OutputStreams

Es existieren einige Input- und OutputStream-Ableitungen im `java.io`-Package. Werfen wir hier einen Blick auf die wichtigsten Vertreter:

11.2.1 ByteArrayInputStream und ByteArrayOutputStream

Diese beiden Klassen erlauben das Lesen und Schreiben von Bytes in und vom Arbeitsspeicher.

ByteArrayOutputStream

Die Klasse `ByteArrayOutputStream` schreibt die auszugebenden Daten in ein Byte-Array, dessen Größe nicht limitiert ist, sondern mit der Anzahl der hinzuzufügenden Daten wächst. Die enthaltenen Daten können mit Hilfe der Methoden `toString()` und `toByteArray()` abgerufen werden. Die Klasse verfügt über zwei Konstruktoren:

```
public ByteArrayOutputStream()
public ByteArrayOutputStream(int size)
```

Der Konstruktor ohne Parameter erzeugt eine `ByteArrayOutputStream`-Instanz, die einen internen Puffer von 32 Bytes besitzt. Der zweite Konstruktor nimmt die Größe des internen Puffers entgegen.

Um die enthaltenen Bytes als String zu verarbeiten, können Sie die überladene Methode `toString()` verwenden. Ohne Parameter aufgerufen, wandelt sie die im Stream enthaltenen Daten in eine Zeichenkette unter Verwendung des Standard-Encodings des Systems um. Alternativ können Sie der Methode auch ein Encoding übergeben, das dann verwendet wird.

ByteArrayInputStream

Das Gegenstück zum `ByteArrayOutputStream` stellt der `ByteArrayInputStream` dar, der das Lesen von Daten aus einem übergebenen Byte-Array erlaubt. Sie besitzt zwei Konstruktoren, die das zu verwendende Byte-Array entgegennehmen:

```
public ByteArrayInputStream(byte[] buf)
public ByteArrayInputStream(
   byte[] buf, int offset, int length)
```

Mit Hilfe der überladenen Methode `read()` kann aus dem Byte-Array gelesen werden. Das Schließen eines `ByteArrayInputStreams` (und auch eines `ByteArrayOutputStreams`) hat übrigens keinen Effekt – anschließend kann weiterhin mit dem Stream gearbeitet werden.

11.2.2 FileInputStream und FileOutputStream

Diese beiden Streams werden für den Zugriff auf Dateien verwendet. Mit Hilfe von `FileInputStreams` können Sie Dateien einlesen, wogegen ein `FileOutputStream` das Speichern von Informationen in Dateien erlaubt.

FileInputStream

Die Klasse `FileInputStream` dient dem Einlesen von Informationen aus einer Datei. Sie verfügt über einen mehrfach überladenen Konstruktor, der Repräsentationen der einzulesenden Datei entgegennimmt:

```
public FileInputStream(String name)
   throws FileNotFoundException

public FileInputStream(File file)
   throws FileNotFoundException

public FileInputStream(java.io.FileDescriptor fdObj)
```

Die sicherste Methode, um einen `FileInputStream` zu erzeugen, besteht darin, dessen Konstruktor eine `FileDescriptor`-Instanz zu übergeben. Aber auch die Übergabe des gewünschten Datei-Namens ist möglich:

Listing 11.1
Einlesen einer Datei mit Hilfe eines FileInputStreams

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

/**
 * Reads the contents of a file and prints it to System.out
 */
public class SimpleFileInputStream {

   public static void main(String[] args) {
      try {
         FileInputStream fis = new FileInputStream(args[0]);
         byte[] buff = new byte[128];
         int length;

         while((length = fis.read(buff)) > 0) {
            System.out.write(buff, 0, length);
         }

         fis.close();
      } catch (FileNotFoundException e) {
         System.out.println(e.getStackTrace().toString());
      } catch (IOException e) {
         System.out.println(e.getStackTrace().toString());
      }
   }
}
```

In diesem Beispiel, das so in der Praxis nicht eingesetzt werden sollte, erzeugen wir einen neuen `FileInputStream`, dessen Konstruktor wir den Datei-Namen als Parameter übergeben. Dieser Datei-Name ist vom Benutzer bei Aufruf der Klasse

anzugeben. Eine Prüfung, ob die angegebene Datei vorhanden ist, findet hier nicht statt, weshalb Sie das Beispiel so nicht einsetzen sollten.

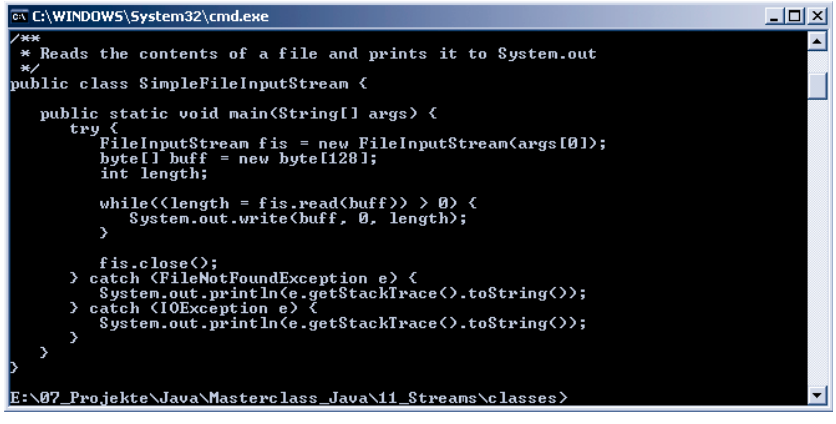
Die Daten werden vom `FileInputStream` in ein Byte-Array eingelesen. Deshalb deklarieren wir ein Byte-Array mit einer Größe von 128 Elementen. Dieses Array wird der Methode `read()` der `FileInputStream`-Instanz `fis` übergeben und von ihr befüllt. Gleichzeitig gibt `read()` die Anzahl der gelesenen Bytes zurück – sollte diese größer als 0 Zeichen sein, ist der Stream noch nicht am Ende angekommen und die eingegebenen Daten können ausgegeben werden:

```
while((length = fis.read(buff)) > 0) {
   System.out.write(buff, 0, length);
}
```

Zuletzt wird der Stream geschlossen.

Wenn Sie die Klasse `SimpleFileInputStream` kompilieren und beispielsweise den Pfad zur Java-Quelltext-Datei `SimpleFileInputStream.java` als Parameter übergeben, werden Sie eine Ausgabe ähnlich dieser erhalten:

Abbildung 11.1
Der Datei-Inhalt wurde eingelesen und ausgegeben



Sollte die angegebene Datei nicht existieren, wird eine `FileNotFoundException` geworfen. Wenn ein Fehler beim Einlesen der Daten auftritt, wird Java eine `IOException` werfen. Diese Exceptions können wir dann entweder auffangen oder in der Signatur der entsprechenden Methode deklarieren.

`FileInputStream` eignet sich insbesondere, um binäre Daten in ein Byte-Array einzulesen. Um Textdateien zu verarbeiten, ist die `FileReader`-Klasse besser geeignet. Wir gehen auf diese Klasse weiter unten in diesem Kapitel genauer ein.

FileOutputStream

Um in Dateien zu schreiben, wird ein `FileOutputStream` verwendet. Dessen Konstruktor ist mehrfach überladen, was es uns erlaubt, die Datei, in die geschrieben werden soll, auf verschiedene Arten zu repräsentieren:

```
public FileOutputStream(String name)
   throws FileNotFoundException

public FileOutputStream(String name, boolean append)
   throws FileNotFoundException

public FileOutputStream(File file)
   throws FileNotFoundException

public FileOutputStream(File file, boolean append)
   throws FileNotFoundException

public FileOutputStream(java.io.FileDescriptor fdObj)
```

Die Varianten des Konstruktors, die eine `File`-Instanz oder einen String als Parameter entgegennehmen, können eine `FileNotFoundException` werfen. Dies kann bei dem Konstruktor, der ein `FileDescriptor`-Objekt als Parameter entgegennimmt, nicht geschehen – allerdings erlaubt dieser auch keine explizite Angabe des Verhaltens, falls die angegebene Datei existiert. Dieses Verhalten wird durch die boolesche Variable `append` repräsentiert – der Wert `true` gibt an, dass der neue Inhalt an den existierenden Inhalt angehängt werden soll, während `false` dafür sorgt, dass der vorhandene Datei-Inhalt überschrieben wird.

Die `FileOutputStream`-Klasse wird in der Regel verwendet, um binäre Daten zu speichern. Für den Einsatz mit Textdateien besser geeignet ist die `FileWriter`-Klasse, auf die wir weiter unten eingehen werden.

11.2.3 FilterInputStream und FilterOutputStream

Die `FilterInputStream`- und `FilterOutputStream`-Klassen stellen Basisklassen dar. Ableitende Klassen arbeiten tatsächlich wie Filter – sie fangen Lese- oder Schreibzugriffe ab, verarbeiten sie und geben die Daten an die aufrufende Methode weiter. Beide Klassen sind abstrakt und werden ausschließlich als Basisklassen verwendet. Ableitende Klassen definieren stets mindestens einen Konstruktor, der eine zugrunde liegende `InputStream`- oder `OutputStream`-Instanz erwartet.

11.2.4 BufferedInputStream und BufferedOutputStream

`BufferedInputStream` und `BufferedOutputStream` dienen – wie die Namen bereits andeuten – der Pufferung von Daten. Ihr Einsatz sorgt in der Regel für eine merkliche Performance-Steigerung beim Zugriff auf Streams.

BufferedInputStream

Die `BufferedInputStream`-Klasse erlaubt einen gepufferten Zugriff auf die Daten des zugrunde liegenden Streams. Intern verfügt diese Klasse über ein `Byte`-Array, das die Zwischenspeicherung der Daten übernimmt. Dies ist für uns in der Anwendung völlig transparent.

Sie verfügt über einen überladenen Konstruktor, der den zu puffernden Stream entgegennimmt:

```
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int size)
```

Der zusätzliche Parameter der zweiten Überladung gibt die Größe des zu verwendenden Puffers an.

Sinnvoll ist der `BufferedInputStream` beispielsweise, wenn Sie auf Dateien zugreifen wollen:

```
BufferedInputStream bis =
   new BufferedInputStream(new FileInputStream("example.txt"))
```

BufferedOutputStream

Wenn Sie viele einzelne Schreibzugriffe auf einen Stream haben, sollten Sie über den Einsatz des `BufferedOutputStreams` nachdenken, denn diese Klasse puffert die zu schreibenden Daten und übergibt sie dem zugrunde liegenden Stream in größeren Blöcken. Dies kann die Performance Ihrer Applikationen deutlich steigern.

Die `BufferedOutputStream`-Klasse verfügt über einen überladenen Konstruktor, der den zugrunde liegenden `OutputStream` als Parameter entgegennimmt:

```
public BufferedOutputStream(OutputStream out)
public BufferedOutputStream(OutputStream out, int size)
```

Der Parameter `size` der zweiten Überladung gibt die Größe des internen Puffers in Byte an. Sollte dieser Puffer mit Inhalten gefüllt sein, wird sein Inhalt automatisch in den zugrunde liegenden Stream geschrieben. Sie können dies auch manuell mit Hilfe der Methode `flush()` anstoßen.

Insbesondere beim Zugriff auf Dateien oder andere externe Datenquellen kann ein `BufferedOutputStream` sinnvoll eingesetzt werden:

```
BufferedOutputStream out =
   new BufferedOutputStream(
      new FileOutputStream("example.txt", true))
```

11.2.5 DataInputStream und DataOutputStream

Die Klassen `DataInputStream` und `DataOutputStream` erlauben es, primitive Datentypen in definierter und plattformunabhängiger Art und Weise zu verarbeiten.

DataOutputStream

Die `DataOutputStream`-Klasse dient dem Speichern von primitiven Datentypen in strukturierter Form. Die Daten werden dabei so abgelegt, dass sie von einer `DataInputStream`-Instanz wieder eingelesen und verarbeitet werden können.

Die DataOutputStream-Klasse implementiert das Interface DataOutput, das folgende Methoden definiert:

```
void write(int b)
   throws IOException

void write(byte[] b)
   throws IOException

void write(byte[] b, int off, int len)
   throws IOException

void writeBoolean(boolean v)
   throws IOException

void writeByte(int v)
   throws IOException

void writeShort(int v)
   throws IOException

void writeChar(int v)
   throws IOException

void writeInt(int v)
   throws IOException

void writeLong(long v)
   throws IOException

void writeFloat(float v)
   throws IOException

void writeDouble(double v)
   throws IOException

void writeBytes(String s)
   throws IOException

void writeChars(String s)
   throws IOException

void writeUTF(String str)
   throws IOException
```

Jede dieser Methoden schreibt ein Element des im Konstruktor verwendeten Typs in so definierter Form, dass Dateien auf jedem System, für das es eine Java Runtime gibt, problemlos wieder eingelesen werden können. Die Vorgaben sind dabei so streng, dass das erzeugte Datei-Format aufgrund der JDK-Dokumentation auch von Nicht-Java-Applikationen wieder eingelesen werden können.

Während die meisten Methoden recht selbsterklärend sind, noch ein Wort zur Methode writeUTF(): Diese speichert den übergebenen String nicht – wie sonst bei Java üblich – im 2-Byte-Unicode-Format, sondern im flexibleren UTF-8-Format. Dieses Format reserviert den verwendeten Speicherplatz dynamisch anhand des Zeichens und kann zwischen einem und vier Byte je Zeichen bean-

spruchen. Bei Zeichen, die beispielsweise aus dem ASCII-Zeichensatz stammen, kann somit effektiv Speicherplatz gespart werden – wenn Zeichen allerdings aus einem exotischeren Zeichensatz (etwa traditionellem Chinesisch) stammen, erhöht sich der Speicherplatz-Bedarf. Der Hauptzweck des Einsatzes von `UTF-8` ist allerdings nicht die Reduktion des verwendeten Speicherplatzes, sondern der Wunsch, alle Zeichen in allen Zeichensätzen speichern zu können.

Achtung

Verzichten Sie auf den Einsatz der Methode `writeChars()`. Diese scheint zwar in den verschiedenen Implementierungen zu funktionieren, allerdings funktioniert ihr Gegenstück beim Einlesen (die Methode `readLine()`) nicht korrekt. Wenn Sie Zeichenketten speichern wollen, sollten Sie die Methode `writeUTF()` verwenden.

Die Klasse `DataOutputStream` implementiert das Interface `DataOutput`. Sie stellt uns daneben einen Konstruktor zur Verfügung, dem wir den zugrunde liegenden Stream (typischerweise eine `BufferedOutputStream`- oder `FileOutputStream`-Instanz) übergeben können:

```
public DataOutputStream(OutputStream out)
```

Werfen wir nun einen kurzen Blick auf ein Einsatz-Beispiel für die `DataOutputStream`-Instanz:

Listing 11.2 Speichern von strukturierten Daten durch einen DataOutputStream

```
import java.io.*;

/**
 * Saves data using a DataOutputStream
 */
public class DataOutputStreamExample {

   public static void main(String[] args) {
      try {
         DataOutputStream out = new DataOutputStream(
            new BufferedOutputStream(
               new FileOutputStream("example.txt")));
         out.writeBoolean(true);
         out.writeUTF("Hello world!");
         out.writeInt(2004);
         out.writeUTF(
            "This is an UTF-8 encoded String " +
            "containing German Umlauts: äöüß");

         out.flush();
         out.close();
      } catch (FileNotFoundException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Bei der Erzeugung des `DataOutputStreams` übergeben wir dessen Konstruktor eine Instanz der `BufferedOutputStream`-Klasse, die ihrerseits einen `FileOutputStream` kapselt. Insbesondere die Kapselung des `FileOutputStreams` durch einen `BufferedOutputStream` ist aus Performance-Gründen sinnvoll.

Anschließend können die Daten geschrieben werden – zuerst wird ein boolescher Wert gespeichert, dann erfolgt die Ausgabe eines `Strings` und einer `int`-Zahl. Zuletzt speichern wir eine Zeichenkette mit Umlauten ab.

Nach dem Schließen des Streams können wir uns die erzeugte Datei *example.txt* ansehen:

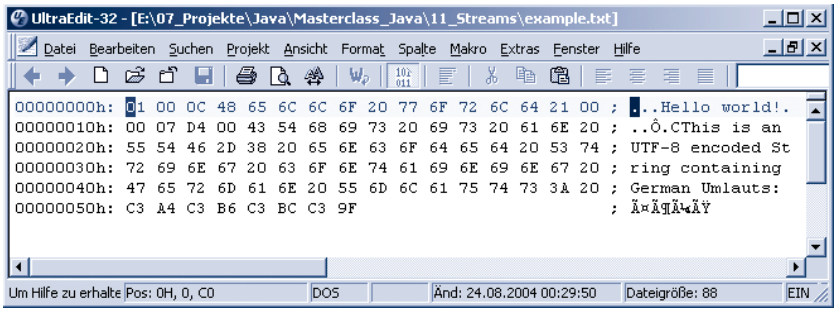


Abbildung 11.2
Ansicht der per DataOutputStream gespeicherten Datei

DataInputStream

Diese Klasse `DataInputStream` erlaubt das Einlesen von Daten, die mit Hilfe eines `DataOutputStreams` geschrieben worden sind. Sie implementiert das Interface `DataInput`, das folgende Methoden definiert:

```
void readFully(byte[] b)
   throws IOException

void readFully(byte[] b, int off, int len)
   throws IOException

int skipBytes(int n)
   throws IOException

boolean readBoolean()
   throws IOException

byte readByte()
   throws IOException

int readUnsignedByte()
   throws IOException

short readShort()
   throws IOException

int readUnsignedShort()
   throws IOException
```

```
char readChar()
   throws IOException

int readInt()
   throws IOException

long readLong()
   throws IOException

float readFloat()
   throws IOException

double readDouble()
   throws IOException

String readLine()
   throws IOException

String readUTF()
   throws IOException
```

Die einzelnen Methoden des Interfaces lesen stets ein Element des durch sie repräsentierten Datentyps ein.

Die Methode `readFully()` liest nicht etwa Elemente eines neuen Datentyps ein, sondern dient dazu, Bytes in das übergebene Byte-Array einzulesen – und dies unabhängig von der Art des repräsentierten Datentyps.

Die Methode `readLine()` liest eine Zeile in einen String ein. Voraussetzung für das einwandfreie Funktionieren dieser Methode ist, dass die zugrunde liegenden Daten `Unicode`-codiert sind. Dies ist bei Zeichenketten, die von Java erzeugt worden sind, der Fall. Wenn Sie `UTF-8`-codierte Strings einlesen wollen, können Sie die Methode `readUTF()` verwenden.

Achtung

Verzichten Sie am besten komplett auf den Einsatz von `readLine()`. Diese Methode ist einerseits als `deprecated` gekennzeichnet und funktioniert andererseits schlichtweg oftmals nicht. Der sichere Weg, Strings zu speichern und auch wieder einzulesen, führt über die Methoden `writeUTF()` der `DataOutputStream`-Klasse und `readUTF()` des `DataInputStreams`.

Ein `DataInputStream` implementiert das Interface. Er wird stets den Endpunkt von internen Stream-Instanzen darstellen. Wenn Sie Daten aus der durch einen `DataOutputStream` geschriebenen Text-Datei *example.txt* lesen wollten, könnten Sie dies beispielsweise so erledigen:

Listing 11.3
Einlesen von strukturierten Daten mit Hilfe eines DataInputStreams

```
import java.io.*;

/**
 * Reads data using a DataInputStream
 */
public class DataInputStreamExample {

   public static void main(String[] args) {
      try {
         DataInputStream in = new DataInputStream(
            new BufferedInputStream(
               new FileInputStream("example.txt")));

         System.out.println(in.readBoolean());
         System.out.println(in.readUTF());
         System.out.println(in.readInt());
         System.out.println(in.readUTF());

         in.close();
      } catch (FileNotFoundException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Wenn Sie mit Hilfe eines `DataInputStreams` auf Dateien zugreifen wollen (so wie dies in diesem Beispiel demonstriert wird), dann sollten Sie die `DataInputStream`-Instanz auf eine `BufferedInputStream`-Instanz zugreifen lassen, die ihrerseits den für das Einlesen einer Datei verwendeten `FileInputStream` kapselt. Sie können so die Performance Ihrer Applikation deutlich erhöhen.

Nach dem Öffnen des `DataInputStreams` können wir nun dessen Daten einlesen. Wir verwenden zu diesem Zweck die verschiedenen `read`-Methoden, die vom Interface `DataInput` definiert worden sind. In unserem Beispiel geben wir die eingelesenen Daten direkt wieder aus – in der Praxis werden wir sie sicherlich intern weiterverarbeiten.

Nach dem Schließen des `DataInputStreams` sollte ein Nutzer, der dieses Beispiel ausführt, folgende Ausgabe erhalten:

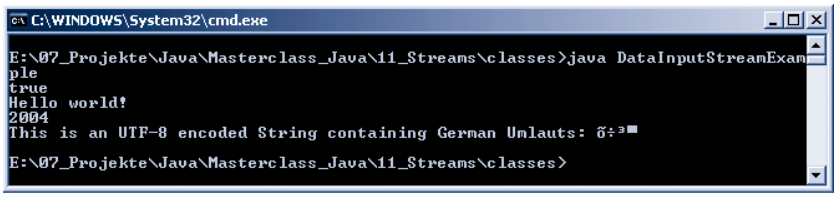


Abbildung 11.3
Per DataInputStream geladene und ausgegebene Informationen

11.2.6 PrintStream

Die Klasse `PrintStream` dient dem Zweck, primitive Datentypen im Textformat auszugeben. Sie bietet deshalb eine Vielzahl an `print()`- und `println()`-Methoden, die in ihren Überladungen die verschiedenen Datentypen entgegennehmen:

```
public void print(boolean b)
public void print(char c)
public void print(int i)
public void print(long l)
public void print(float f)
public void print(double d)
public void print(char[] s)
public void print(String s)
public void print(Object obj)

public void println()
public void println(boolean x)
public void println(char x)
public void println(int x)
public void println(long x)
public void println(float x)
public void println(double x)
public void println(char[] x)
public void println(String x)
public void println(Object x)
```

Der Unterschied zwischen den `print()`- und `println()`-Methoden besteht darin, dass beim Ausgeben von Daten per `println()` am Ende stets ein Zeilenumbruch ausgegeben wird. Dieser Zeilenumbruch entspricht dem in der Property `line.separator` definierten Token und ist systemspezifisch.

Neu in Java 5 sind die `printf()`- und `format()`-Methoden. Letztere nehmen einen `String` entgegen und ersetzen die in der Zeichenkette enthaltenen Platzhalter durch die als weitere Parameter übergebenen Parameter. Die `printf()`-Methoden liefern eine `PrintStream`-Instanz zurück, in der die entsprechenden Ersetzungen bereits vorgenommen worden sind.

```
public PrintStream printf(String format, Object... args)
public PrintStream printf(
   Locale l, String format, Object... args)

public PrintStream format(String format, Object... args)
public PrintStream format(
   Locale l, String format, Object... args)
```

Ebenfalls neu in Java 5 ist die Methode `append()` des `PrintStreams`. Sie nimmt eine Zeichenkette oder ein einfaches Zeichen als Parameter entgegen und fügt es an die intern enthaltene Ausgabe an. Praktisch verhält sich diese Methode wie die Methode `print()`.

```
public PrintStream append(CharSequence csq)
   throws IOException
public PrintStream append(char c)
   throws IOException
```

11.3 Writer und Reader

Die Ableitungen der abstrakten Basisklassen `Writer` und `Reader` bilden die Gegenstücke zu den Byte-Streams. Werfen wir nun einen Blick auf die gebräuchlichsten `Reader` und `Writer`.

11.3.1 FileReader und FileWriter

Diese Klassen erlauben den Zugriff auf Dateien mit textuellem Inhalt. Sie bilden somit die Gegenstücke zu den Klassen `FileInputStream` und `FileOutputStream`.

FileWriter

Um Daten in einer Text-Datei zu speichern, können wir eine Instanz der `FileWriter`-Klasse verwenden. Der mehrfach überladene Konstruktor dieser Klasse nimmt sowohl Dateinamen als auch `File`- und `FileDescriptor`-Instanzen als Parameter entgegen:

```
public FileWriter(String fileName)
   throws IOException

public FileWriter(String fileName, boolean append)
   throws IOException

public FileWriter(File file)
   throws IOException

public FileWriter(File file, boolean append)
   throws IOException

public FileWriter(java.io.FileDescriptor fd)
```

Fast alle `FileWriter`-Konstruktoren können eine `IOException` werfen, falls der Zugriff auf die angegebene Datei nicht möglich ist. Dieses Verhalten ist allerdings systemabhängig – manche Systeme erlauben ein paralleles Schreiben mehrerer `FileWriter`-Instanzen in eine Datei, andere nicht.

Einzig der Konstruktor, der eine `FileDescriptor`-Instanz als Parameter entgegennimmt, wird in keinem Fall eine derartige Exception werfen – allerdings können wir auch nicht mit Hilfe eines booeschen Parameters angeben, ob die zu schreibenden Daten an den existierenden Datei-Inhalt angehängt werden (dann wäre der zu übergebende Wert `true`) sollen oder nicht (in diesem Fall müsste `false` übergeben werden).

Sehen wir uns an, wie wir mit Hilfe eines `FileWriters` einen Text speichern können:

Listing 11.4
Speichern eines Textes mit Hilfe einer FileWriter-Instanz

```
import java.io.FileWriter;
import java.io.IOException;

/**
 * Writes a String by the help of a FileWriter-instance
 */
```

Listing 11.4 *(Forts.)* Speichern eines Textes mit Hilfe einer FileWriter-Instanz

```
public class SimpleFileWriter {

   public static void main(String[] args) {
      String output =
         "Lorem ipsum dolor sit amet, consectetuer adipiscing " +
         "elit. Donec sit amet justo at velit convallis " +
         "dignissim. Mauris magna eros, mattis id, iaculis " +
         "nec, fermentum non.";

      try {
         FileWriter fw =
            new FileWriter("c:/temp/textfile.txt");
         fw.write(output);
         fw.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

FileReader

Das Einlesen von Text-Dateien geschieht mit Hilfe einer `FileReader`-Instanz. Diese verfügt über einen mehrfach überladenen Konstruktor, der die Freiheit lässt, die einzulesende Text-Datei auf verschiedene Arten zu spezifizieren:

```
public FileReader(String fileName)
   throws FileNotFoundException

public FileReader(File file)
   throws FileNotFoundException

public FileReader(java.io.FileDescriptor fd)
```

Wenn die angegebene Datei nicht existiert, wird eine `FileNotFoundException` geworfen. Dies ist bei Übergabe einer `FileDescriptor`-Instanz als Parameter nicht nötig, da Instanzen dieser Klasse existierende Dateien im Dateisystem repräsentieren.

Nach dem Erzeugen der `FileReader`-Instanz können wir mit Hilfe ihrer `read()`-Methode den Datei-Inhalt einlesen:

Listing 11.5 Einlesen einer Textdatei mit Hilfe eines FileReaders

```
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;

/**
 * Reads the content of a text file by using a FileReader
 */
public class SimpleFileReader {

   public static void main(String[] args) {

      try {
         FileReader fr =
            new FileReader("c:/temp/textfile.txt");
         StringBuilder content = new StringBuilder();
```

```
            int charCode;
            while((charCode = fr.read()) != -1) {
               content.append((char)charCode);
            }

            fr.close();
            System.out.println(content.toString());
         } catch (FileNotFoundException e) {
            e.printStackTrace();
         } catch (IOException e) {
            e.printStackTrace();
         }
      }
}
```

Listing 11.5 *(Forts.)*
Einlesen einer Textdatei mit Hilfe eines FileReaders

Die `FileReader`-Klasse wird nur ungern zum direkten Laden von Datei-Inhalten eingesetzt. Den Grund dafür können wir deutlich erkennen: Die Methode `read()` liest immer nur zeichenweise ein. Als Steigerung werden die eingelesenen Daten als `char`-Codes – also numerische Repräsentationen eines Zeichens – zurückgegeben, so dass wir zwar einerseits sehr einfach auf das Ende des Streams prüfen können (dann müsste die von `read()` zurückgegebene Zahl -1 sein), andererseits aber jeden einzelnen `char`-Code noch in das zugehörige Zeichen casten müssen:

```
while((charCode = fr.read()) != -1) {
   content.append((char)charCode);
}
```

Zwar besteht auch die Möglichkeit, die Überladung der `read()`-Methode zu nutzen, die ein `char`-Array befüllt, aber auch der Inhalt dieses Arrays müsste noch manuell in einen String gecastet werden.

11.3.2 BufferedReader und BufferedWriter

Diese Klassen erlauben es, Zugriffe auf die zugrunde iegenden Daten zu puffern. Dies kann insbesondere beim Lesen oder Schreiben von kleinen, strukturierten Daten deutliche Performance-Vorteile bringen.

BufferedReader

Die Klasse `BufferedReader` nimmt im Konstruktor die `Reader`-Instanz entgegen, deren Datenzugriffe gepuffert werden sollen. Die Überladung des Konstruktors nimmt als zweiten Parameter eine benutzerdefinierte Puffer-Größe in Bytes entgegen:

```
public BufferedReader(Reader in)
public BufferedReader(Reader in, int sz)
```

Eine Änderung der Puffer-Größe ist allerdings in der Praxis selten nötig – die Standard-Größe reicht in der Regel mehr als aus, um eine Applikation spürbar zu beschleunigen.

Meistens wird der `BufferedReader` zusammen mit einem `FileReader` eingesetzt. Hier spielt die Klasse ihre Stärke aus – insbesondere deshalb, da sie zusätzlich zu

den von der Basisklasse definierten Methoden über eine Methode `readLine()` verfügt, mit deren Hilfe ein zeilenweises Einlesen der Daten möglich wird:

```
public String readLine()
   throws IOException
```

Sehen wir uns ein kleines Beispiel für das zeilenweise Einlesen einer Textdatei mit Hilfe eines `BufferedReaders` an:

Listing 11.6
Einlesen einer Datei mit Hilfe eines BufferedReaders

```
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.BufferedReader;

/**
 * Uses a BufferedReader to read the content of a text file
 */
public class SimpleBufferedReader {

   public static void main(String[] args) {

      try {
         FileReader fr =
            new FileReader("c:/temp/textfile.txt");

         BufferedReader br = new BufferedReader(fr);

         String line;
         while((line = br.readLine()) != null) {
            System.out.println(line);
         }

         br.close();
      } catch (FileNotFoundException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Dieser Code erledigt dieselbe Aufgabe wie beim weiter oben gezeigten Beispiel zum `FileReader`. Intern wird sogar auf eine `FileReader`-Instanz zurückgegriffen, die dem `BufferedReader` im Konstruktor übergeben worden ist:

```
FileReader fr =
   new FileReader("c:/temp/textfile.txt");
BufferedReader br = new BufferedReader(fr);
```

Statt nun den Datei-Inhalt zeichenweise oder in Form eines `char`-Arrays einzulesen, verwenden wir hier den intuitiveren Ansatz des zeilenweisen Einlesens. Dieses Erfassen der Daten geschieht so lange, bis die Methode `readLine()` der `BufferedReader`-Instanz `null` zurückgibt und somit kein weiterer Inhalt einzulesen ist:

```
while((line = br.readLine()) != null) {
   System.out.println(line);
}
```

BufferedWriter

Das Gegenstück zum `BufferedReader` ist der `BufferedWriter`, dessen Aufgabe darin besteht, die zu schreibenden Daten zu puffern und somit die Performance insbesondere bei Schreibzugriffen stark zu steigern.

Die Klasse `BufferedWriter` verfügt über einen mehrfach überladenen Konstruktor, der den zugrunde liegenden `Writer` als Parameter entgegennimmt:

```
public BufferedWriter(Writer out)
public BufferedWriter(Writer out, int sz)
```

Der zweite Parameter des Konstruktors der `BufferedWriter`-Klasse nimmt die Größe des internen Puffers entgegen. In der Regel muss diese aber nicht gesetzt werden, da die Pufferung auch schon mit der Standard-Einstellung sehr effektiv funktioniert.

Der `BufferedWriter` verfügt über die Methode `newLine()`, mit deren Hilfe ein Zeilenumbruch in der Ausgabe eingefügt werden kann:

```
public void newLine()
   throws IOException
```

Der Einsatz der Klasse `BufferedWriter` gestaltet sich sehr einfach und transparent:

Listing 11.7
Speichern eines Textes mit Hilfe eines BufferedWriters

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.BufferedWriter;

/**
 * Uses a BufferedWriter to write a String into a file
 */
public class SimpleBufferedWriter {

   public static void main(String[] args) {
      String output =
         "Lorem ipsum dolor sit amet, consectetuer adipiscing " +
         "elit. Donec sit amet justo at velit convallis " +
         "dignissim. Mauris magna eros, mattis id, iaculis " +
         "nec, fermentum non.";

      try {
         FileWriter fw =
            new FileWriter("c:/temp/textfile.txt");
         BufferedWriter bw = new BufferedWriter(fw);

         bw.write(output);
         bw.newLine();
         bw.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

11.4 RandomAccessFile

Bisher waren wir nur in der Lage, Dateien per Stream zum Lesen oder Schreiben zu öffnen. Dies ändert sich, wenn wir die Klasse `java.io.RandomAccessFile` einsetzen. Diese verlässt allerdings das angenehme Schema der Streams und erlaubt ausschließlich einen Zugriff auf physikalisch existierende Dateien – Arrays oder andere Konstrukte werden nicht unterstützt.

Weiterhin gibt es eine Einschränkung hinsichtlich von Filter-Klassen – so funktionieren `BufferedReader`, `-Writer`, `-InputStream` oder `-OutputStream` nicht mit `RandomAccessFile`-Instanzen.

Wenn Sie mit diesen Einschränkungen leben können, finden Sie hier eine äußerst flexible Lösung für den Datei-Zugriff. So implementiert die Klasse `RandomAccessFile` beispielsweise die bei den Klassen `DataInputStream` und `DataOutputStream` besprochenen Interfaces `DataInput` und `DataOutput`.

Der Konstruktor der Klasse nimmt eine Repräsentation der uns interessierenden Datei und die Art des Datei-Zugriffs als Parameter entgegen:

```
public RandomAccessFile(String name, String mode)
   throws FileNotFoundException
public RandomAccessFile(File file, String mode)
   throws FileNotFoundException
```

Der zweite Parameter für den Datei-Zugriffsmodus verdient einen genaueren Blick. Folgende Werte sind hier zulässig:

- `"r"`: Schreibgeschütztes Öffnen der Datei. Das Aufrufen einer der `write`-Methoden wird mit dem Werfen einer `IOException` quittiert.
- `"rw"`: Öffnen zum Lesen und Schreiben. Falls die angegebene Datei nicht existiert, wird versucht, sie zu erzeugen.
- `"rws"`: Datei wird zum Lesen und Schreiben geöffnet. Jede Änderung am Datei-Inhalt oder den Meta-Daten wird sofort und direkt gesichert.
- `"rwd"`: Datei wird zum Lesen und Schreiben geöffnet. Jede Änderung am Datei-Inhalt wird sofort und direkt gesichert.

Positions-Zeiger

Wenn Sie eine `RandomAccessFile`-Instanz öffnen, müssen Sie selbstständig in der Datei navigieren. Sie verwenden dazu einen Positionszeiger, was Sie sich ähnlich wie die Handhabung eines überdimensionalen Arrays vorstellen können. Dessen Positionszeiger war der aktuelle Index. Ganz so einfach ist die Datei-Handhabung zwar nicht, dennoch unterstützen Sie einige Methoden bei der Positionierung Ihres Zeigers:

```
public long getFilePointer()
   throws IOException

public void seek(long pos)
   throws IOException

public int skipBytes(int n)
   throws IOException
```

Die Methode `getFilePointer()` liefert die aktuelle Position des Zeigers. Weil die Rückgabe vom Typ `long` ist, kann Java auch mit Dateien umgehen, die größer als 2 GByte sind. Tendenziell werden Sie beim Umgang mit derartig großen Dateien eher an Betriebssystem- oder Hardware-Grenzen stoßen.

Mit Hilfe von `seek()` setzen Sie den Positionszeiger an die angegebene Position. Diese wird vom Beginn der Datei gezählt, wobei die Zählung bei Position 0 beginnt. Um sich nicht absolut (wie das bei Verwendung von `seek()` der Fall ist) zum Datei-Anfang, sondern relativ zur aktuellen Position zu bewegen, verwenden Sie die Methode `skipBytes()`, die die angegebene Anzahl an Bytes von der aktuellen Position an überspringt.

Datei-Länge bestimmen und festlegen

Beim Einsatz der Klasse `RandomAccessFile` können Sie sowohl die Länge bestimmen als auch (im schreibenden Modus) festlegen. Dies geschieht mit Hilfe zweier Methoden:

```
public long length()
   throws IOException

public void setLength(long newLength)
   throws IOException
```

Die Methode `length()` bestimmt die Größe der Datei in Bytes zum derzeitigen Zeitpunkt – ändert sich die Größe, ändert sich auch die Rückgabe von `length()`.

Die Methode `setLength()` kann genutzt werden, um die Datei-Größe festzulegen. Ein kleinerer Wert als die derzeitige Größe verringert die Länge, ein größerer Wert macht genau das Gegenteil und erhöht die mögliche Länge, wobei der zusätzliche Platz mit leeren Bytes gefüllt wird.

Inhalte lesen

Die Inhalte geöffneter Dateien können mit Hilfe einiger – im Interface `DataInput` definierten – Methoden eingelesen werden:

```
boolean readBoolean()
   throws IOException

int skipBytes(int n)
   throws IOException

byte readByte()
   throws IOException
```

```
char readChar()
   throws IOException

double readDouble()
   throws IOException

float readFloat()
   throws IOException

void readFully(byte[] b)
   throws IOException

void readFully(byte[] b, int off, int len)
   throws IOException

int readInt()
   throws IOException

String readLine()
   throws IOException

long readLong()
   throws IOException

short readShort()
   throws IOException

int readUnsignedByte()
   throws IOException

int readUnsignedShort()
   throws IOException

String readUTF()
   throws IOException
```

Die einzelnen Methoden sind recht selbsterklärend – sie lesen immer ein Element des angegebenen Typs. Die Methoden setzen voraus, dass die einzulesenden Informationen mit Hilfe der korrespondierenden `write`-Methoden geschrieben worden sind. Sollte dies nicht der Fall sein, können Sie stark damit rechnen, dass eine `IOException` geworfen wird.

Neben den durch das Interface `DataInput` definierten Methoden stehen noch weitere Methoden zum Einlesen von Byte-Werten zur Verfügung. Sie entsprechen den `read()`-Methoden der `InputStream`-Klasse:

```
public int read()
   throws IOException

public int read(byte[] b, int off, int len)
   throws IOException

public int read(byte[] b)
   throws IOException
```

Inhalte schreiben

Die Gegenstücke zu den verschiedenen `read`-Methoden stellen diverse `write`-Methoden dar, die zum größten Teil im Interface `DataOutput` definiert sind:

```
void write(int b)
   throws IOException

void write(byte[] b)
   throws IOException

void write(byte[] b, int off, int len)
   throws IOException

void writeBoolean(boolean v)
   throws IOException

void writeByte(int v)
   throws IOException

void writeBytes(String s)
   throws IOException

void writeChar(int v)
   throws IOException

void writeChars(String s)
   throws IOException

void writeDouble(double v)
   throws IOException

void writeFloat(float v)
   throws IOException

void writeInt(int v)
   throws IOException

void writeLong(long v)
   throws IOException

void writeShort(int v)
   throws IOException

void writeUTF(String str)
   throws IOException
```

Auch wenn mehrere Methoden existieren, um Strings zu speichern, sollten Sie im Sinne der Betriebssicherheit Ihrer Applikation Zeichenketten immer nur mit Hilfe der Methode `writeUTF()` speichern. Nebenbei sorgen Sie so dafür, dass eventuelle Sonderzeichen in Ihren Strings erhalten bleiben und auch auf Systemen mit anderen Spracheinstellungen gelesen werden können.

Ein Einsatz-Beispiel

Am besten nähert man sich dem Umgang mit `RandomAccessFiles`, indem man sie einsetzt. Sehen wir uns also an, wie wir mit Hilfe dieses Datei-Typs eine Art Gästebuch schreiben können:

Listing 11.8
Lesen und Schreiben mit Hilfe einer RandomAccessFile-Instanz

```
import java.io.RandomAccessFile;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.File;

/**
 * A very basic guestbook using a RandomAccessFile
 */
public class RandomAccessFileExample {

   public static void main(String[] args) {

      long entries = 0;

      try {
         File f = new File("c:/temp/guestbook.txt");
         boolean exists = f.exists();
         if(!exists) {
            f.createNewFile();
         }

         RandomAccessFile rf = new RandomAccessFile(f, "rw");
         if(exists && rf.length() > 0) {
            entries = rf.readLong();

            System.out.println(
               String.format("%s Entries in your guestbook",
                  String.valueOf(entries)));

            while(rf.getFilePointer() < rf.length()) {
               String line = rf.readUTF();
               System.out.println(line);
            }
         }

         if(args.length > 0 && null != args[0]) {
            rf.seek(0);
            rf.writeLong(++entries);

            rf.seek(rf.length());
            rf.writeUTF(args[0]);
            System.out.println(
               "\nAdded your entry to our guestbook!");
         }

         rf.close();
      } catch (FileNotFoundException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Bevor wir auf die Datei zugreifen, überprüfen wir zunächst, ob diese überhaupt existiert – denn schließlich können wir keine Daten aus einer nicht existierenden Datei auslesen. Also erzeugen wir eine `File`-Instanz, deren Konstruktor wir den Namen der Daten-Datei übergeben. Diese `File`-Instanz verfügt über eine Methode `exists()`, die `true` zurückgibt, wenn die angegebene Datei existiert. Ist die Rückgabe `false`, dann existiert die Datei nicht und wird mit Hilfe der Methode `createNewFile()` der `File`-Instanz neu angelegt:

```
File f = new File("c:/temp/guestbook.txt");
boolean exists = f.exists();
if(!exists) {
   f.createNewFile();
}
```

Nun kann die `RandomAccessFile`-Instanz erzeugt werden. Ihr Zugriffsmodus ist „rw“, so dass ein Lesen und Schreiben möglich wird.

Wenn die Datei nicht neu erzeugt worden ist und eine Länge größer Null hat, dann können wir im ersten Schritt die Anzahl der enthaltenen Einträge auslesen – dieser Wert wird am Anfang der Datei abgelegt:

```
RandomAccessFile rf = new RandomAccessFile(f, "rw");
if(exists && rf.length() > 0) {
   entries = rf.readLong();
   // ...
}
```

Nach der Anzahl der enthaltenen Einträge folgen diese hintereinander in der Datei. Sie sind als `UTF-8`-Strings abgelegt und können deshalb mit Hilfe der Methode `getUTF()` eingelesen werden. Dies geschieht innerhalb einer Schleife, die so lange durchlaufen wird, bis der Positions-Zeiger am Ende der Datei angekommen ist:

```
while(rf.getFilePointer() < rf.length()) {
   String line = rf.readUTF();
   System.out.println(line);
}
```

Wenn die Klasse mit einem Kommandozeilen-Parameter aufgerufen worden ist, wird der Inhalt dieses Parameters am Ende eingefügt. Zuvor muss jedoch der Positions-Zeiger auf die Startposition zurückgesetzt werden, da hier die Anzahl der enthaltenen Einträge abgelegt ist. Dies geschieht mit Hilfe der Methode `seek()`. Danach kann die um 1 erhöhte Anzahl der Einträge durch die Methode `writeLong()` gespeichert werden:

```
if(args.length > 0 && null != args[0]) {
   rf.seek(0);
   rf.writeLong(++entries);
   // ...
}
```

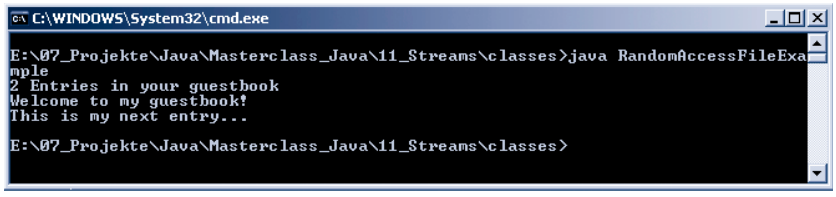
Jetzt kann der Positions-Zeiger ans Ende der Datei gesetzt werden, damit wir den neuen Eintrag anfügen können. Auch dies erledigt die Methode `seek()`, der

wir diesmal die Größe der Datei als Parameter übergeben. Durch ein `writeUTF()` wird die übergebene Zeichenkette an die Datei angefügt:

```
rf.seek(rf.length());
rf.writeUTF(args[0]);
```

Nach dem Schließen der Datei ist der Vorgang abgeschlossen. Nutzer, die die Klasse ohne Parameter aufrufen, erhalten eine Auflistung aller gespeicherten Nachrichten:

Abbildung 11.4
Auflistung aller Einträge unseres Gästebuchs

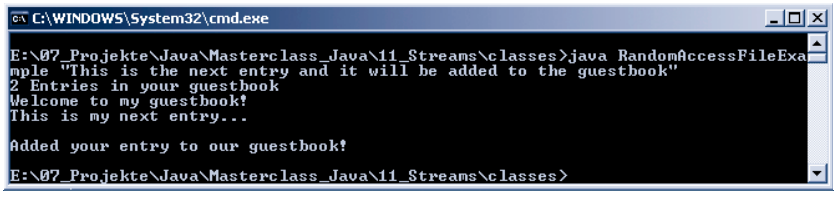


Wird die Klasse mit einem Parameter aufgerufen, wird dieser der Datei hinzugefügt. Rufen wir die Datei per

```
java RandomAccessFileExample "This is the next entry and it will
  be added to the guestbook"
```

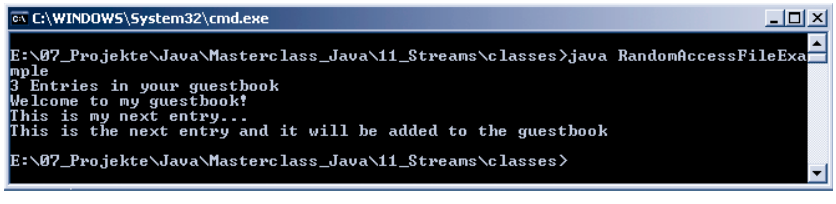
auf, wird dieser gespeichert und eine entsprechende Meldung ausgegeben:

Abbildung 11.5
Ein Eintrag wurde angefügt ...



Ein erneuter Aufruf der Klasse ohne weitere Parameter zeigt uns, dass sich sowohl die Anzahl der Einträge im Gästebuch als auch die Liste der Einträge geändert haben:

Abbildung 11.6
... und kann wieder ausgelesen werden



11.5 File

Die Klasse `File` repräsentiert eine Datei oder ein Verzeichnis im Datei-System. Die Klasse stellt Methoden zur Verfügung, mit deren Hilfe Informationen über eine Datei erhalten werden können. Ebenfalls verfügt sie über Methoden, die Dateien anlegen oder löschen können.

Der Konstruktor der `File`-Klasse ist mehrfach überladen:

```
public File(String pathname)

public File(String parent, String child)

public File(File parent, String child)

public File(URI uri)
```

Die einfachste Variante des `File`-Konstruktors nimmt eine relative oder absolute Angabe des Datei-Namens entgegen. Die beiden Konstruktoren mit zwei Parametern erlauben eine getrennte Angabe von Pfad und Datei-Name, wobei ersterer auch durch eine `File`-Instanz repräsentiert werden kann. Die vierte Überladung erlaubt die Angabe einer `URL`-Instanz, die aber nicht etwa auf eine Ressource im Internet, sondern auf eine lokale Ressource verweisen kann.

Beispiele für gültige Konstruktoren wären:

```
new File("testFile.txt"); // relative to current position
new File("c:/temp", "file.txt"); // absolute path
new File("\\\\host\\share\\file.txt"); // Windows-Fileshare
new File("/usr/data/someFile.txt"); // Unix, absolute path
new File(new URL("file:///c:/temp/file.txt")); // URL
```

Beachten Sie, dass Sie unter Microsoft Windows Backslashes verdoppeln müssen, um sie nutzen zu können. Alternativ können Sie auch den Schrägstrich statt des Backslashes verwenden. Sie müssen diesen dann auch nicht verdoppeln.

Statt der Angabe

```
c:\\temp\\file.txt
```

können Sie unter Windows auch

```
c:/temp/file.txt
```

schreiben. Dies ist nicht nur kürzer, sondern auch deutlich übersichtlicher und weniger fehlerträchtig.

11.5.1 Pfad-Bestandteile ermitteln

Nachdem eine `File`-Instanz erzeugt worden ist, können wir mit deren Hilfe Informationen zu den einzelnen Bestandteilen des Pfades und des Datei-Namens ermitteln. Folgende Methoden stehen uns dafür zur Verfügung:

```
public String getName()

public String getParent()

public File getParentFile()

public String getPath()

public boolean isAbsolute()

public String getAbsolutePath()
```

```
public File getAbsoluteFile()

public String getCanonicalPath()
   throws IOException

public File getCanonicalFile()
   throws IOException
```

Die Methode `getName()` gibt uns den Namen der Datei oder des Verzeichnisses zurück, das durch die aktuelle `File`-Instanz repräsentiert wird. Eventuelle andere Bestandteile des Pfades werden entfernt.

Mit Hilfe von `getAbsolutePath()` können wir den absoluten Verzeichnispfad zum durch das `File`-Objekt repräsentierten Element ermitteln. Sollte das `File`-Objekt mit Hilfe einer relativen Pfad-Angabe erstellt worden sein, wird der absolute Pfad einfach vorangestellt. Wenn die `File`-Instanz beispielsweise mit einem Parameter `.\\file.txt` erzeugt worden ist, so liefert `getAbsolutePath()` eine Rückgabe wie `C:\temp\.\file.txt` – der Punkt im Pfad bleibt also erhalten.

Derartige Angaben sind nicht immer erwünscht, weshalb es ebenfalls die Methode `getCanonicalPath()` gibt. Diese gibt einen eindeutigen Pfad, bereinigt um alle überflüssigen Angaben, zurück – statt `C:\temp\.\file.txt` wäre dies `C:\temp\file.txt`.

Um den Namen des übergeordneten Verzeichnisses zu ermitteln, können Sie die Methode `getParent()` verwenden.

Wenn Sie statt der Angaben als `Strings` lieber `File`-Instanzen mit diesen Informationen füttern wollen, können Sie die Methoden `getParentFile()`, `getAbsoluteFile()` und `getCanonicalFile()` verwenden.

Die Methode `isAbsolute()` gibt uns Informationen darüber, ob der im Konstruktor übergebene Pfad absolut oder relativ angegeben worden ist.

11.5.2 Detaillierte Informationen über die File-Instanz

Das `File`-Objekt liefert uns wichtige Informationen über das repräsentierte Element. Folgende Methoden stehen dafür zur Verfügung:

```
public boolean canRead()

public boolean canWrite()

public boolean exists()

public boolean isDirectory()

public boolean isFile()

public boolean isHidden()

public long lastModified()

public long length()
```

Um festzustellen, ob es sich bei dem durch eine `File`-Instanz repräsentierten Element um eine Datei oder ein Verzeichnis handelt, können wir die Methoden `isDirectory()` und `isFile()` verwenden. Zuvor sollten wir mit Hilfe von `exists()` feststellen, ob das repräsentierte Element überhaupt existiert.

Wollen wir herausfinden, ob aus dem angegebenen Element gelesen werden kann, können wir die Methode `canRead()` verwenden. Ihr Gegenstück `canWrite()` gibt uns Auskunft darüber, ob in die Datei geschrieben werden kann.

Ist die Datei versteckt, finden wir dies mit Hilfe der Methode `isHidden()` heraus. Die Datei-Länge liefert uns die Methode `length()`. Der Zeitpunkt des letzten Änderns der Datei wird mit Hilfe von `lastModified()` ermittelt. Diese Methode liefert uns die Anzahl der Millisekunden seit dem 01.01.1970. Diese Information kann mit dem Konstruktor des `Date`-Objekts verwendet werden, wodurch wir eine `Date`-Instanz erhalten würden, mit deren Hilfe wir das genaue Datum der letzten Änderung der Datei erfahren können:

```
File f = new File("c:/temp/testfile.txt");
Date d = new Date(f.lastModified());
```

Diese `Date`-Instanz könnte nun weiterverarbeitet werden.

11.5.3 Datei-Aktionen

Sollte es sich beim durch die `File`-Instanz repräsentierten Element um eine Datei handeln, können wir diese Datei löschen, neu anlegen oder umbenennen:

```
public boolean createNewFile()
   throws IOException

public boolean delete()

public void deleteOnExit()

public boolean renameTo(File dest)

public boolean setLastModified(long time)

public boolean setReadOnly()
```

Die Methode `createNewFile()` legt eine leere Datei neu an. Dies geschieht allerdings nur, wenn die Datei zuvor nicht existierte. Sie können die Methode also ohne vorherige Prüfung, ob die Datei existiert, aufrufen.

Um die durch die File-Instanz repräsentierte Datei zu löschen, können wir gleich zwei Methoden verwenden: `delete()` löscht die Datei sofort, dagegen sorgt die Methode `deleteOnExit()` dafür, dass die Datei erst beim Terminieren der JVM gelöscht wird. Dies geschieht dann aber auch garantiert, selbst wenn dabei `Exceptions` auftreten sollten.

Eine Datei umbenennen oder verschieben? Kein Problem für die Methode `renameTo()`: Diese nimmt als Parameter eine `File`-Instanz entgegen, die das

Ziel repräsentiert. Um eine Datei zu verschieben, benötigen wir also lediglich folgenden Code:

```
File f = new File("c:/temp/file.txt");
File target = new File("e:/temp/test.txt");
f.renameTo(target);
```

Da die Implementierung der Methode systemabhängig ist, kann nicht garantiert werden, wie die Methode sich verhält, wenn das Ziel auf einem anderen Laufwerk liegt oder die Datei bereits existiert. Es ist aber davon auszugehen, dass es in einem derartigen Fall zu einer `IOException` kommen kann.

Um zu verhindern, dass Dritte eine Datei überschreiben, können wir deren Schreibschutz-Attribut setzen. Dies erledigt die Methode `setReadOnly()`. Schlecht nur, dass keine Methode existiert, die das Schreibschutz-Attribut wieder entfernt – wir sollten also mit Hilfe von `canWrite()` überprüfen, ob ein schreibender Zugriff möglich ist:

```
File f = new File("c:/temp/file.txt");
f.setReadOnly();
if(!(f.canWrite()))  {
   System.out.println("Write-access not possible!");
}
```

11.5.4 Verzeichnis-Aktionen

Wenn das durch die `File`-Instanz repräsentierte Element ein Verzeichnis ist, können wir naturgemäß andere Aktionen vornehmen, als dies bei einer Datei der Fall wäre. Folgende Methoden stehen uns zur Verfügung:

```
public boolean delete()

public void deleteOnExit()

public String[] list()

public String[] list(FilenameFilter filter)

public File[] listFiles()

public File[] listFiles(FilenameFilter filter)

public File[] listFiles(FileFilter filter)

public boolean mkdir()

public boolean mkdirs()

public boolean setLastModified(long time)

public boolean setReadOnly()

public static File[] listRoots()
```

Die Methoden `delete()` und `deleteOnExit()` funktionieren bei Verzeichnissen analog zu ihren Pendants bei Dateien. Allerdings setzt ein erfolgreiches Löschen voraus, dass das Verzeichnis leer ist.

Es empfiehlt sich also im Vorfeld, alle untergeordneten Elemente des Verzeichnisses mit Hilfe der Methoden `list()` oder `listFiles()` und deren Überladungen abzurufen. Mit Hilfe von `FilenameFilter`- und `FileFilter`-Instanzen können einzelne Verzeichnis-Elemente akzeptiert oder zurückgewiesen werden. Zu diesem Zweck erfordern beide Interfaces die Implementierung einer Methode `accept()`.

Für das Interface `FilenameFilter` hat diese Methode folgende Signatur:

```
boolean accept(File dir, String name)
```

Das Interface `FileFilter` erfordert die Implementierung einer Methode `accept()` mit dieser Signatur

```
boolean accept(File pathname)
```

Die Rückgabe beider Methoden soll dann `true` sein, wenn das übergebene Element der Rückgabe-Liste hinzugefügt (also akzeptiert) werden soll. Anderenfalls sollte die Rückgabe der Methode `false` sein – die Datei oder das Unterverzeichnis werden in diesem Fall ignoriert. Ein Beispiel für eine `FileFilter`-Implementierung könnte etwa so aussehen:

```
import java.io.FileFilter;
import java.io.File;

/**
 * A FileFilter accepting files with the extension .txt
 */
public class TxtFileFilter implements FileFilter {

   public boolean accept(File pathname) {
      return pathname.isFile() &&
         pathname.getName().endsWith(".txt");
   }
}
```

Eine Liste aller Wurzelverzeichnisse des aktuellen Systems erhalten Sie mit Hilfe der statischen Methode `listRoots()`. Auf Unix-Systemen beinhaltet die Rückgabe der Methode nur ein Element – das Wurzelverzeichnis "/". Unter Windows wird jedes einzelne Laufwerk zurückgegeben.

```
String() roots = File.listRoots();
for(String root : roots) {
   System.out.println(root);
}
```

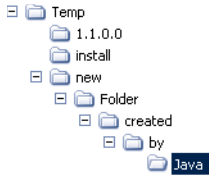
Wollen Sie ein Verzeichnis neu anlegen, erzeugen Sie eine neue `File`-Instanz, die das Verzeichnis repräsentieren soll. Anschließend rufen Sie deren Methode `mkdir()` auf, die das Unterverzeichnis erzeugt. Sollten Sie einen kompletten Verzeichnis-Pfad erzeugen wollen oder sich nicht sicher sein, ob eines der über-

geordneten Verzeichnisse wirklich existiert, rufen Sie die Methode `mkdirs()` auf, die jedes der fehlenden übergeordneten Verzeichnisse erstellt:

```
File newDir = new File("c:/temp/new/Folder/created/by/Java/");
newDir.mkdirs();
```

Sollte nur der Ordner `c:\temp` existieren, werden die Ordner *new*, *Folder*, *created*, *by* und *Java* erzeugt:

Abbildung 11.7
Per mkdirs() erzeugte Ordner-Struktur



Die Methode `setReadOnly()` erlaubt es, ein Verzeichnis mit einem Schreibschutz zu versehen. Mit Hilfe von `setLastModified()` können wir das letzte Änderungsdatum eines Verzeichnisses setzen.

11.6 Fazit

Mit Hilfe von Byte- und Character-Streams können wir auf Datenquellen zugreifen. Die verschiedenen Streams bieten uns eine übergreifende API, so dass wir in der Lage sind, diese Zugriffe ohne große Änderungen an unserem Code durchzuführen. Während sich Byte-Streams für den Zugriff auf binäre Daten eignen, bieten sich Character-Streams für die Arbeit mit textuellen Informationen an.

`FileStreams` und ihre `Reader`-/`Writer`-Gegenstücke erlauben einen ausschließlich lesenden oder schreibenden Zugriff auf die Datenquelle. Das Manko, dass Lesen und Schreiben nicht zugleich möglich sind, behebt die Klasse `RandomAccessFile`. Diese ähnelt zwar Streams, ist aber eine eigenständige Implementierung, die einen Zugriff auf Dateien mit bekannten Strukturen erlaubt.

Richtig interessant wird der Zugriff auf Dateien erst mit der Klasse `File`. Diese kann sowohl einzelne Dateien als auch Verzeichnisse repräsentieren und erlaubt es uns, mit dem Datei-System zu arbeiten. Wir können Informationen zu den repräsentierten Elementen auslesen oder diese Elemente neu anlegen, manipulieren oder löschen.

Wir sind nach der Lektüre dieses Kapitels nunmehr in der Lage, Informationen persistent zu speichern. Leider sind Dateien nicht das Nonplusultra, wenn größere Datenmengen zu verarbeiten sind. Hier kommt die `JDBC`-API ins Spiel, die eine Anbindung von fast beliebigen Datenbanken erlaubt. Dies wird Thema des folgenden Kapitels sein, in dem wir uns unter anderem auch mit der Anbindung einer MySQL-Datenbank befassen werden.

12

JDBC

JDBC ist Javas Schnittstelle zur Welt der Datenbanken. Mit JDBC können Sie fast jede relationale Datenbank an Java anbinden und aus Java-Applikationen heraus verwenden – egal, ob es sich um eine Oracle-Datenbank, IBMs DB2, den Microsoft SQL-Server oder die populäre Open-Source-Datenbank MySQL handelt. JDBC selbst stellt die API bereit, mit deren Hilfe Sie über Treiber auf die Datenbank zugreifen können. Die Abkürzung JDBC steht für Java DataBase Connectivity und der Ansatz von JDBC orientiert sich am ODBC-Ansatz.

Aufgrund des schieren Umfangs und der Komplexität des Themas werden wir in diesem Kapitel nur auf die grundlegenden Mechanismen hinter JDBC eingehen können. Wir werden lernen, wie wir eine Verbindung zu einer Datenbank herstellen oder wie wir Daten in diese Datenbank einfügen und auch wieder auslesen können.

12.1 Das JDBC-Prinzip

JDBC stellt Methoden zur Verfügung, mit deren Hilfe wir eine Verbindung zu einer Datenbank herstellen sowie Daten einfügen, auslesen oder ändern können. JDBC greift nicht direkt auf die Datenbank zu, sondern verwendet datenbankspezifische Treiber (gerne auch *Provider* genannt).

JDBC unterscheidet zwischen vier Typen von Treibern, die jeweils spezifische Vor- und Nachteile haben:

- Typ-1-Treiber: Dies sind keine spezifischen JDBC-Treiber, sondern ODBC-Treiber. Diese können nicht direkt verwendet werden, sondern Java kommuniziert mit ihnen über die im Lieferumfang enthaltene JDBC-ODBC-Bridge. Typ-1-Treiber stellen die langsamste und fehlerträchtigste Variante des Zugriffs auf eine Datenbank dar – sind aber gleichzeitig oftmals die einzige Möglichkeit, um auf eine Datenbank zugreifen zu können, für die kein JDBC-Treiber existiert.

- Typ-2-Treiber: Typ-2-Treiber sind zwar selbst in Java programmiert, greifen aber intern auf datenbankspezifische Nicht-ODBC-Treiber zurück. Dies kann sehr problematisch werden, wenn für das System, auf dem die Java-Applikation läuft, gar kein datenbankspezifischer Treiber existiert.
- Typ-3-Treiber: Diese Art von Treibern ist komplett in Java geschrieben und erfordert keine Installation systemspezifischer Software. Typ-3-Treiber greifen allerdings nicht direkt auf die Datenbank zu, sondern nutzen eine interne Zwischenschicht. Diese Treiber-Variante ist deutlich performanter als die Typ-1- oder Typ-2-Treiber.
- Typ-4-Treiber: Wenn ein JDBC-Treiber direkt in Java programmiert ist und darüber hinaus direkt mit einer Datenbank über deren spezifisches Kommunikationsprotokoll Verbindung aufnehmen kann, so ist er ein Typ-4-Treiber. Diese Art von Treibern ist die schnellste Variante des Zugriffs auf eine Datenbank und andererseits allerdings auch besonders komplex zu programmieren.

In der Praxis werden Sie recht selten auf Typ-1- und Typ-3-Treiber stoßen. Wesentlich häufiger anzutreffen sind Typ-2- und Typ-4-Treiber, wobei letztere mittlerweile für jedes größere Datenbank-System angeboten werden.

Leider sind die Treiber nicht immer kostenlos, aber beim Datenbank-Anbieter erhalten Sie in der Regel einen Typ-2- oder Typ-4-Treiber, der eventuell nicht so performant ist wie sein kostenpflichtiges Gegenstück, dafür aber eben auch nichts kostet.

Nachdem wir für unsere Datenbank einen Treiber heruntergeladen und bei Bedarf auch installiert haben, können wir auf eine Datenbank zugreifen. Diese Datenbank-Zugriffe finden in der Regel nach folgendem Schema statt:

1. Treiber laden und instanzieren
2. Verbindungsobjekt erzeugen und befüllen
3. Verbindung herstellen
4. Aktionen durchführen (Daten anlegen, auslesen, ändern oder löschen)
5. Verbindung zur Datenbank wieder schließen

Der Zugriff auf Datenbanken per `JDBC` findet unter Verwendung der Packages `java.sql.*` und `javax.sql.*` statt.

12.2 MySQL, PostGre-SQL, Cloudscape, Oracle ...?

Eigentlich ist es völlig egal, mit welcher Datenbank Sie anfangen zu arbeiten. Der Autor empfiehlt den Einsatz von MySQL, der beliebtesten und bekanntesten Open-Source-Datenbank der Gegenwart. In verschiedenen Tutorials werden Sie gerne auf Cloudscape, HSQL oder ähnliche in Java implementierte Datenbanken verwiesen – nur ist deren Funktionsumfang in der Regel beschränkt und über deren Performance braucht man kein Wort zu verlieren.

Etwas anders liegt der Fall beim Einsatz von Oracle oder PostGre-SQL. Beides sind voll ausgestattete relationale Datenbank-Systeme. Allerdings ist Oracle sehr teuer, und PostGre-SQL ist zwar ein viel beachteter MySQL-Konkurrent (der tatsächlich sogar mehr kann), hat aber bei weitem nicht die Verbreitung, wie es bei MySQL der Fall ist.

Auch im Internet-Bereich ist MySQL stark verbreitet. Wenn Sie also bereits mit MySQL gearbeitet haben, werden Sie erfreut feststellen, dass dieses Kapitel ebenfalls MySQL einsetzt. Wenn dies Ihre ersten Geh-Versuche mit Datenbanken sind, dann bietet Ihnen MySQL einen recht einfachen und komfortablen Einstieg.

Nebenbei: Der Einsatz von MySQL ist für private oder nicht-kommerzielle Projekte kostenlos. Und die Lizenzen für kommerzielle Projekte sind ebenfalls mehr als erschwinglich.

12.2.1 MySQL, der JDBC-Treiber und die Admin-Werkzeuge

Die Installation von MySQL ist nicht schwierig. Zum Zeitpunkt der Drucklegung dieses Buches ist die Version 4.0 aktuell. Sie können diese Version unter *http://dev.mysql.com/downloads/* in der für Ihr System geeigneten Version herunterladen und installieren.

Anschließend können Sie von der Kommandozeile aus das Datenbank-System ein wenig testen und mit Hilfe der Kommandozeilen-Tools beispielsweise neue Nutzer oder Datenbanken anlegen. Unter Windows NT / 2000 / XP / 2003 und natürlich den verschiedenen Unix-Derivaten sowie Linux kann MySQL als Dienst oder Daemon laufen. Einzelheiten, wie Sie die Datenbank als Dienst oder Daemon installieren können, entnehmen Sie bitte der Dokumentation.

Administrations-Tools

Mit der Installation der Datenbank-Software alleine ist es leider nicht getan. Wir haben nun zwar die Engine, die die Daten verwaltet und ablegt, müssten das Anlegen von Daten und Benutzern aber von Hand erledigen. Wer dies gerne tun möchte, kann sich mit Hilfe der MySQL-Kommandozeilen-Tools gerne austoben – nur ist dies für uns als Entwickler in der Regel nicht sinnvoll.

Also benötigen wir ein Administrations-Werkzeug, das uns die Verwaltung unserer Datenbank erleichtert. Glücklicherweise gibt es das auch kostenlos auf der MySQL-Homepage zum Herunterladen – und selbstverständlich auch in Versionen für Windows und Linux. Sie finden *MySQL-Administrator* unter der Adresse *http://dev.mysql.com/downloads/administrator/index.html* kostenlos zum Download bereitgestellt.

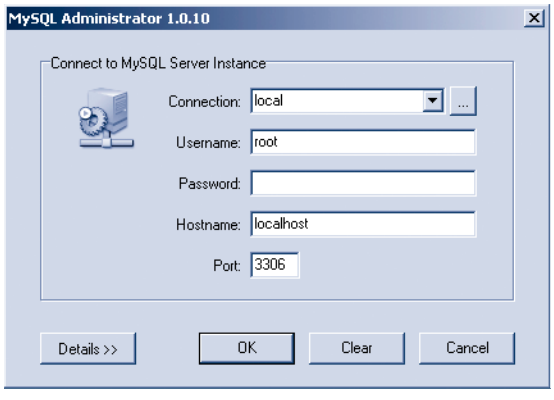
Ebenfalls interessant dürfte der *Query-Browser* sein, der es Ihnen erlaubt, SQL-Statements auf die Datenbank abzusetzen, zu analysieren und zu optimieren. Sie können die Software (die sich derzeit noch im Alpha-Stadium befindet) unter der Adresse *http://dev.mysql.com/downloads/query-browser/index.html* herunterladen. Es gibt Versionen für Windows und Linux.

Zuletzt soll noch das nicht mehr weiterentwickelte *MySQL Control Center* erwähnt werden. Dieses ist eine Art Vorläufer der aktuellen Administrator-Applikation und erlaubt ein einfacheres Anlegen von Datenbanken und Tabellen, als dies bei den derzeitigen Werkzeugen der Fall ist. Das MySQL Control Center können Sie unter der Adresse *http://dev.mysql.com/downloads/other/mysqlcc.html* herunterladen. Es ist in Versionen für Windows und Linux erhältlich.

Nutzer und Datenbank-Tabelle anlegen

Nachdem Sie die Datenbank und die Administrations-Tools installiert haben, sollten Sie einen neuen Datenbank-Benutzer anlegen. Sie können dies mit Hilfe des MySQL-Administrator-Tools erledigen. Zu diesem Zweck sollten Sie die MySQL-Datenbank-Instanz und das MySQL-Administrator-Tool starten. So Sie noch keine Änderungen an den Benutzer-Einstellungen vorgenommen haben, sollten Sie sich mit dem Benutzernamen `root` und einem leeren Password an der Datenbank-Instanz auf `localhost` anmelden können:

Abbildung 12.1
Anmeldebildschirm für den MySQL-Administrator



Nach erfolgreicher Anmeldung befinden Sie sich im Hauptmenü des Administrator-Programms. Hier sollten Sie im Untermenü USER ADMINISTRATION das Password des Benutzers `root` setzen. Dessen leeres Password ist zwar kein großes Sicherheitsrisiko, dennoch kann zumindest vom lokalen System aus jeder Nutzer eine Anmeldung an der Datenbank durchführen.

Vergeben Sie also für den Benutzer `root` ein Kennwort und legen Sie bei der Gelegenheit auch gleich einen weiteren Nutzer für die Beispiele dieses Kapitels an. Der Autor verwendet im Folgenden einen Benutzer `guestbook` mit dem Kennwort `bookguest`, der sich nur von der lokalen Maschine aus anmelden darf.

Anschließend sollten Sie eine Datenbank `test` anlegen, so diese noch nicht existiert. Sie erledigen dies über den Menüpunkt CATALOG. Innerhalb der Datenbank `test` sollten Sie nun noch eine Tabelle `guestbook` anlegen, die folgende Felder definiert:

- `id` vom Typ `INTEGER`, mit den gesetzten Flags `AUTO_INCREMENT` und `NOT NULL`.
- `message` vom Typ `VARCHAR` mit einer maximalen Länge von 255 Zeichen und dem Flag `NOT NULL`.

- `sender` vom Typ `VARCHAR` mit einer maximalen Länge von 50 Zeichen. Das Flag `NOT NULL` sollte nicht aktiviert sein.
- `messageDate` vom Typ `DATETIME` mit dem aktivierten Flag `NOT NULL`

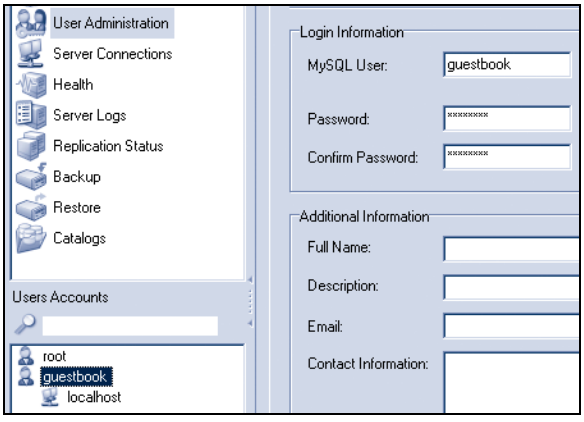


Abbildung 12.2
Der Benutzer guestbook darf sich nur von localhost anmelden

Falls Sie die Tabelle lieber per SQL-Statement anlegen wollen, können Sie dies mit diesem Statement erreichen:

```
CREATE TABLE `guestbook` (
  `id` int(10) unsigned NOT NULL auto_increment,
  `message` varchar(255) NOT NULL default '',
  `sender` varchar(50) default NULL,
  `messageDate` datetime NOT NULL default '0000-00-00 00:00:00',
  PRIMARY KEY  (`id`)
) TYPE=MyISAM;
```

Wenn Sie die Tabelle im MySQL-Administrator betrachten, sollte sie etwa so aussehen:

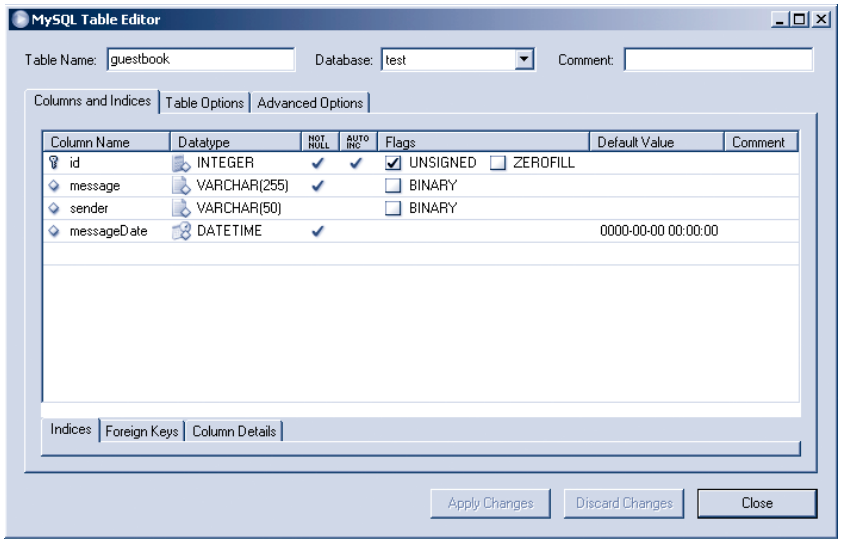


Abbildung 12.3
Die Tabelle guestbook im Editor des MySQL Administrator-Tools

Nach dem Anlegen der Datenbank und der Tabelle sollten Sie wieder in das User-Management zurückwechseln und dem Nutzer über das Tab „Schema Privileges“ SELECT-, INSERT-, UPDATE- und DELETE-Rechte auf der Datenbank test einräumen.

Übrigens: Wenn Sie auch die grafischen Tools für die Administration der Datenbank als nicht bequem und intuitiv genug empfinden, sollten Sie als Windows-Nutzer einen Blick auf *MySQL-Front* werfen. Sie finden die Software unter der Adresse *http://www.mysqlfront.de* und können sich dort eine 30-Tage-Testversion herunterladen.

Sollten Sie über einen Apache- oder IIS-Webserver auf Ihrem lokalen System samt installiertem PHP verfügen, können Sie Ihre Datenbank auch über das Internet administrieren. Installieren Sie sich dann *phpMyAdmin*, das Sie unter der Adresse *http://www.phpmyadmin.net/* finden und kostenlos herunterladen können.

JDBC-Treiber

MySQL stellt für seine Datenbank JDBC-Typ-4-Treiber zum kostenlosen Download zur Verfügung. Sie finden das Treiber-Paket unter der Adresse *http://dev.mysql.com/downloads/connector/j/3.0.html*. Laden Sie es herunter und entpacken Sie das im Archiv enthaltene *JAR*-File mysql-connector-java-3.0.14-production-bin.jar in ein Verzeichnis, das zum Klassenpfad Ihrer Java-Installation gehört, oder kopieren Sie es ins aktuelle Applikations-Verzeichnis.

Damit sind wir mit den Vorbereitungen fertig und können MySQL als Datenbank verwenden.

12.3 Verbindung herstellen

Bevor wir Daten aus einer Datenbank abrufen können, müssen wir eine Verbindung zu dieser Datenquelle öffnen. Zu diesem Zweck laden wir zuerst den JDBC-Treiber, der sich beim Treiber-Manager (DriverManager) registriert, um später Verbindungen aufbauen und handhaben zu können. Anschließend können wir eine Connection anfordern, die die Verbindungsinformationen aufnehmen wird.

12.3.1 Treiber laden

Das Laden eines Treibers geschieht, indem wir seinen kompletten Klassen-Namen inklusive Paket-Angabe an die statische Methode forName() der Class-Klasse übergeben. Für MySQL mit dem MySQL Connector-J-Treiber lautet der komplette Klassenname com.mysql.jdbc.Driver, so dass der Aufruf für die Registrierung des Treibers so aussieht:

```
Class.forName("com.mysql.jdbc.Driver");
```

Der Aufruf von Class.forName() wird aber nur einwandfrei funktionieren, wenn das JAR-File *mysql-connector-java-3.0.14-production-bin.jar* im Klassenpfad definiert ist oder sich im aktuellen Arbeitsverzeichnis befindet.

Da beim Laden des Treibers eine `ClassNotFoundException` geworfen werden kann, sollte das Statement in einem `try-catch`-Block stehen.

12.3.2 Verbindungs-Parameter übergeben

Nun können wir ein `Connection`-Objekt anfordern, das die Verbindungsinformationen halten und uns später weitere Aktionen erlauben wird. Dieses `Connection`-Objekt erhalten wir durch die statische Methode `getConnection()` der `DriverManager`-Klasse aus dem `java.sql`-Namensraum, die mehrfach überladen ist:

```
public static Connection getConnection(String url)
   throws SQLException

public static Connection getConnection(
   String url, String user, String password)
   throws SQLException

public static Connection getConnection(
   String url, Properties info)
   throws SQLException
```

Alle Überladungen haben gemeinsam, dass sie einen `String` entgegennehmen, der die Zugriffsinformationen zur Datenbank beinhaltet. Derartige Strings werden übrigens auch `ConnectionStrings` genannt.

Die weiteren Überladungen nehmen weitere Zugriffsinformationen entgegen. So können Sie den für den Zugriff zu verwendenden Usernamen und das Password explizit angeben oder eine `Properties`-Instanz mit weiteren Parametern übergeben. Welche Parameter dabei zum Einsatz kommen können, ist von der verwendeten Datenbank abhängig.

Der `ConnectionString` für den Zugriff auf eine MySQL-Datenbank hat folgenden Aufbau:

```
jdbc:mysql://[Host][:Port]/[data-
base][?param][=wert][&param2][=wert2]...
```

Dabei ist Host durch den Namen oder die IP-Adresse des Servers zu ersetzen. Der Port einer MySQL-Instanz ist in der Regel 3306. Unter den anzugebenden Parametern sollten sich immer auch Benutzername und Kennwort befinden. Alternativ können Sie diese Informationen auch in Form einer `Properties`-Instanz oder zweier Strings übergeben.

Für den Zugriff auf die Datenbank *test* der lokalen MySQL-Instanz unter Angabe des Benutzernamens und des Kennworts des Users *guestbook* sollte der `ConnectionString` so aussehen:

```
jdbc:mysql://localhost:3306/test?user=guestbook&password=bookguest
```

Wenn wir eine `JDBC`-Connection mit diesem `ConnectionString` aufbauen wollen, sollten sowohl das Registrieren der Treiber-Klasse als auch das Erzeugen der Connection innerhalb eines `try-catch`-Blocks stattfinden:

```
import java.sql.*;

String connStr =
   "jdbc:mysql://localhost:3306/" +
   "test?username=guestbook&pwd=bookguest";
Connection conn = null;

try {
   Class.forName("com.mysql.jdbc.Driver");
   conn = DriverManager.getConnection(connStr);
} catch (ClassNotFoundException e) {
   e.printStackTrace();
} catch (SQLException e) {
   e.printStackTrace();
}
```

Sollte die Verbindungsherstellung nicht möglich sein – etwa weil die Zugriffsinformationen nicht korrekt waren, dann wird eine `SQLException` geworfen, die wir natürlich abfangen sollten. Wenn alles einwandfrei läuft, dann verfügen wir über eine `Connection`-Instanz, mit deren Hilfe wir nun Abfragen an die Datenbank senden können.

12.4 Datenbank-Abfragen

Nach dem Erzeugen einer `Connection`-Instanz können wir mit Hilfe von so genannten `Statements` Daten in die Datenbank einfügen und abrufen. JDBC unterscheidet drei verschiedene `Statement`-Arten, die jeweils in entsprechenden Interfaces definiert sind und von der `Connection`-Instanz erzeugt werden können:

```
Statement createStatement()
   throws SQLException

PreparedStatement prepareStatement(String sql)
   throws SQLException

CallableStatement prepareCall(String sql)
   throws SQLException
```

Die drei Methoden sind mehrfach überladen. In der Praxis arbeiten Sie aber meistens mit den hier gezeigten Varianten, die Ihnen entweder `Statement`-, `PreparedStatement`- oder `CallableStatement`-Instanzen zurückgeben.

Jede dieser Statement-Implementierungen hat dabei eine eigene Einsatzberechtigung:

- `Statement`: Wird für einfache SQL-Statements ohne dynamische Bestandteile eingesetzt.
- `PreparedStatement`: Wird für SQL-Statements mit Parametern eingesetzt.
- `CallableStatement`: Wird für SQL-Statements, die Stored Procedures in einer Datenbank verwenden, eingesetzt.

In Tutorials und Fachartikeln wird gerne das Interface `Statement` für den Zugriff auf Datenbanken verwenden. Der Autor rät davon ab, und zwar aus einem ganz bestimmten Grund: Nur `PreparedStatements` schützen vor der *SQL-Injection*, bei der bösartige Angreifer SQL-Code in die Abfragen einschleusen können.

Achtung

Aus Sicherheitsgründen sollten Sie bei jedem SQL-Statement, das auch nur einen Parameter verwendet, der auf externen Daten beruht, auf `PreparedStatement`-Instanzen zurückgreifen.

12.4.1 Daten einfügen oder ändern

Die Verwendung des Interfaces `Statement` ist der einfachste Weg, um SQL-Statements abzusetzen. Gleichzeitig ist die Verwendung dieses Interfaces aber auch der unsicherste Weg, den Sie wählen können, da `Statement`-Implementierungen keinen Schutz vor *SQL-Injection* bieten. Da die Interfaces `PreparedStatement` und `CallableStatement` aber von `Statement` erben, soll dieses Interface dennoch im Folgenden exemplarisch verwendet werden.

Um Daten in eine Tabelle einzufügen, können Sie die Methoden `execute()` oder `executeUpdate()` der `Statement`-Instanz verwenden. Diese nehmen als Parameter ein SQL-Statement entgegen, das Daten in die Tabelle einfügt oder Datensätze ändert:

```
boolean execute(String sql)
   throws SQLException

int executeUpdate(String sql)
   throws SQLException
```

Zwar existieren auch zu diesen Methoden mehrere Überladungen, aber in der Praxis arbeiten Sie fast ausschließlich mit diesen beiden Varianten. Der Unterschied zwischen `execute()` und `executeUpdate()` besteht darin, dass `executeUpdate()` keine Ergebnisse erzeugen kann. Die Rückgabe der Methode `executeUpdate()` ist ein `int`-Wert, der uns Auskunft über die Anzahl der betroffenen Datensätze gibt. Sollte ein Fehler auftreten, wird eine `SQLException` geworfen.

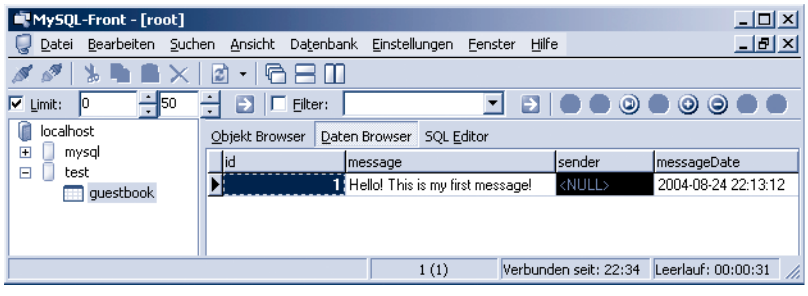
Wenn wir in die Tabelle `guestbook` einen Datensatz einfügen wollen, können wir dies mit diesem Code erreichen:

```
import java.sql.*;

String connStr =
   "jdbc:mysql://localhost:3306/" +
   "test?user=guestbook&password=bookguest";
String statement =
   "INSERT INTO guestbook (message, messageDate) " +
   "VALUES ('Hello! This is my first message!', " +
      "'2004-08-24 22:13:12')";
```

```
Connection conn = null;
try {
   Class.forName("com.mysql.jdbc.Driver");
   conn = DriverManager.getConnection(connStr);

   Statement stmt = conn.createStatement();
   stmt.executeUpdate(statement);

   conn.close();
} catch (ClassNotFoundException e) {
   e.printStackTrace();
} catch (SQLException e) {
   e.printStackTrace();
}
```

Nach dem Ausführen des Statements können wir in der Datenbank-Tabelle unseren neuen Eintrag bestaunen:

Abbildung 12.4
Der neue Datensatz ist in die Tabelle eingefügt worden



Das Ändern von Datensätzen erfolgt mit Hilfe eines `UPDATE`-Statements – der Rest des Codes bleibt unverändert:

```
String statement =
   "UPDATE guestbook SET sender = 'Karsten Samaschke' WHERE id = 1";

stmt.executeUpdate(statement);
```

Wenn Sie einen Datensatz löschen wollen, können Sie ebenfalls die `executeUpdate()`-Methode verwenden:

```
String statement = "DELETE FROM guestbook WHERE id = 1";

stmt.executeUpdate(statement);
```

12.4.2 Abrufen von Daten

Mit Hilfe der Methode `executeQuery()` des `Statement`-Interfaces können wir einen oder mehrere Datensätze aus der Datenbank-Tabelle abrufen. Die Verwendung der Methode ist angenehm einfach – sie ist nicht überladen und gibt immer eine `ResultSet`-Instanz zurück:

```
ResultSet executeQuery(String sql)
   throws SQLException
```

Das Interface `ResultSet` repräsentiert eine oder mehrere Zeilen aus der Datenbank. Nach der Ausführung der Methode `executeQuery()` befindet sich der interne Satzzeiger vor dem ersten Element des `ResultSets`. Mit Hilfe seiner Methode `next()` kann der Satzzeiger auf das nächste Element bewegt werden – die Methode gibt uns darüber hinaus einen booleschen Wert zurück, der angibt, ob sich weitere Elemente im `ResultSet` befinden.

Mit Hilfe verschiedener Methoden können wir nun die im `ResultSet` enthaltenen Daten abrufen:

```
Array getArray(int i)
Array getArray(String colName)

Blob getBlob(int i)
Blob getBlob(String colName)

boolean getBoolean(int columnIndex)
boolean getBoolean(String columnName)

byte getByte(int columnIndex)
byte getByte(String columnName)

byte[] getBytes(int columnIndex)
byte[] getBytes(String columnName)

Date getDate(int columnIndex)
Date getDate(String columnName)

double getDouble(int columnIndex)
double getDouble(String columnName)

float getFloat(int columnIndex)
float getFloat(String columnName)

int getInt(int columnIndex)
int getInt(String columnName)

long getLong(int columnIndex)
long getLong(String columnName)

Object getObject(int columnIndex)
Object getObject(String columnName)

short getShort(int columnIndex)
short getShort(String columnName)

String getString(int columnIndex)
String getString(String columnName)

Time getTime(int columnIndex)
Time getTime(String columnName)

Timestamp getTimestamp(int columnIndex)
Timestamp getTimestamp(String columnName)
```

Denken Sie nicht, dass dies alle Methoden zum Abrufen von Daten sind – aus Platzgründen sind hier nur die wichtigsten Vertreter aufgeführt.

Von jeder Methode existieren mindestens zwei Überladungen: Einmal wird ein `String` entgegengenommen, der den Spalten-Namen in der Datenbank repräsentiert, und in der zweiten Überladung können wir einen `int`-Wert übergeben, der dem Index des Feldes im `ResultSet` (nicht in der Datenbank-Tabelle!) entspricht.

Die Verwendung des `ResultSets` ist recht einfach. Wenn wir beispielsweise alle Datensätze der `guestbook`-Tabelle auslesen wollten, könnten wir folgenden Code verwenden:

```
import java.sql.*;
import java.util.Calendar;
import java.text.SimpleDateFormat;

// ...

String connStr =
   "jdbc:mysql://localhost:3306/" +
   "test?user=guestbook&password=bookguest";
String statement =
   "SELECT message, sender, messageDate " +
   "FROM guestbook";


Connection conn = null;

try {
   Class.forName("com.mysql.jdbc.Driver");
   conn = DriverManager.getConnection(connStr);

   Statement stmt = conn.createStatement();
   ResultSet rs = stmt.executeQuery(statement);

   while(rs.next()) {
      String message = rs.getString(1);
      String sender = rs.getString(2);
      java.util.Date d = rs.getDate(3);
      java.util.Date t = rs.getTime(3);

      if(sender == null) {
         sender = "Unknown sender";
      }

      Calendar c = Calendar.getInstance();
      c.setTime(d);
      SimpleDateFormat df = new SimpleDateFormat("dd.MM.yyyy");
      String date = df.format(d);

      c = Calendar.getInstance();
      c.setTime(t);
      df = new SimpleDateFormat("HH:mm:ss");
      String time = df.format(t);

      System.out.println(
         String.format("Message from %s (%s, %s):\n%s",
            sender, date, time, message));
   }
```

```
   conn.close();
} catch (ClassNotFoundException e) {
   e.printStackTrace();
} catch (SQLException e) {
   e.printStackTrace();
}
```

Die Methode `executeQuery()` nimmt als Parameter ein SQL-Statement entgegen und gibt eine `ResultSet`-Implementierung zurück, die danach durchlaufen werden kann.

Dieses Durchlaufen erfolgt innerhalb einer `while`-Schleife. Deren Abbruchbedingung tritt ein, wenn die Methode `next()` des `ResultSets` den Wert `false` zurückgibt und somit signalisiert, dass keine weiteren Zeilen mehr im `ResultSet` enthalten sind.

Aus Gründen der Lesbarkeit sollten Sie im SQL-Statement auf den Platzhalter * verzichten und stattdessen die einzelnen Spalten explizit angeben. Anmerkung: Die Performance wird dadurch unmerklich beeinflusst, denn die Meta-Informationen einer Relation sind sofort verfügbar bei der Auswertung eines `SELECT`-Statements mit *-Operator.

In der Schleife werden die einzelnen Werte ausgelesen. Der Zugriff kann sowohl über den Spaltennamen als auch über den Index der Spalte erfolgen. Dieser Index bezieht sich nicht auf die Reihenfolge der Spalten in der Datenbank, sondern auf die durch das `SELECT`-Statement definierte Reihenfolge. Die erste im `SELECT`-Statement definierte Spalte hat den Index 1, die zweite Spalte hat den Index 2. Dies ist etwas unüblich – Indizierungen fangen bei Java in der Regel beim Index-Wert 0 an.

Damit Entwickler sich nicht unnötig mit Typ-Konvertierungen herumschlagen müssen, führen die meisten `get`-Methoden implizite Konvertierungen durch. Die generischsten dieser `get`-Methoden sind `getString()` und `getObject()`, die fast alle Arten von Daten auslesen können.

12.5 PreparedStatement

Die Verwendung des Interfaces `PreparedStatement` wird vom Autor dieses Buches explizit empfohlen. Dies geschieht aus vier Gründen:

- Sicherheit: `PreparedStatements` sind sicherer als `Statement`-Implementierungen. Sie schützen in der Regel effektiv vor den gefürchteten *SQL-Injection*-Attacken.
- Performance: `PreparedStatements` sind in der Regel schneller als `Statement`-Implementierungen, da SQL-Statements und Parameter (je nach Datenbank-System) getrennt übergeben werden können. Für das Datenbank-System entfällt die Analyse des SQL-Statements und das Parsen der bei einem Statement immer nur in Text-Form übergebenen Werte.
- Lesbarkeit: Klassen, die `PreparedStatements` verwenden, sind in der Regel besser lesbar, denn die Zuweisung der Werte erfolgt in Form einzelner Methoden-Aufrufe.
- Typ-Sicherheit: Da die Daten über Java-Methoden zugewiesen werden, können selbstverständlich nur die Arten von Daten zugewiesen werden, die auch von den Methoden akzeptiert werden.

Diese vier Vorteile erkaufen wir uns glücklicherweise nicht mit einer komplexen Bedienung des Interfaces: Der Methode `prepareStatement()` des `Connection`-Interfaces übergeben wir das SQL-Statement, das Platzhalter für die Werte enthält. Der Platzhalter ist dabei das Fragezeichen:

```
PreparedStatement prepareStatement(String sql)
   throws SQLException

PreparedStatement prepareStatement(String sql,
   int autoGeneratedKeys)
   throws SQLException

PreparedStatement prepareStatement(String sql,
   int[] columnIndexes)
   throws SQLException

PreparedStatement prepareStatement(String sql,
   int resultSetType, int resultSetConcurrency)
   throws SQLException

PreparedStatement prepareStatement(String sql,
   int resultSetType, int resultSetConcurrency,
   int resultSetHoldability)
   throws SQLException

PreparedStatement prepareStatement(String sql,
   String[] columnNames)
   throws SQLException
```

Die gebräuchlichste Überladung ist die einfachste Variante, wie ein `String` als Parameter zugewiesen worden ist.

Diese Werte werden anschließend mit Hilfe diverser `set`-Methoden zugewiesen:

```
void setArray(int i, Array x)
void setBlob(int i, Blob x)
void setBoolean(int parameterIndex, boolean x)
void setByte(int parameterIndex, byte x)
void setBytes(int parameterIndex, byte[] x)
void setDate(int parameterIndex, Date x)
void setDouble(int parameterIndex, double x)
void setFloat(int parameterIndex, float x)
void setInt(int parameterIndex, int x)
void setLong(int parameterIndex, long x)
void setNull(int parameterIndex, int sqlType)
void setObject(int parameterIndex, Object x)
void setShort(int parameterIndex, short x)
void setString(int parameterIndex, String x)
void setTime(int parameterIndex, Time x)
void setTimestamp(int parameterIndex, Timestamp x)
void setURL(int parameterIndex, URL x)
```

Der erste Parameter bezeichnet den Index des Parameters. Die Zählung startet – anders als etwa bei Arrays – beim Index-Wert 1 für das erste Element.

12.5.1 Datensätze einfügen, ändern oder löschen

Wie bereits mehrfach erwähnt, ist die Verwendung des Interfaces `PreparedStatement` nicht komplexer als die Verwendung des Interfaces `Statement`. Statt der Methode `createStatement()` der `Connection`-Instanz greifen wir beim Einsatz eines `PreparedStatements` auf die Methode `preparedStatement()` zurück, der wir das zu verwendende SQL-Statement samt der Platzhalter als Parameter übergeben:

```
PreparedStatement stmt = conn.prepareStatement(
   "INSERT INTO guestbook (message, messageDate) VALUES (?, ?)");
```

Anschließend können wir die Werte mit Hilfe der entsprechenden `set`-Statements setzen:

```
String message = "Hello! This is my second message!";
java.util.Date date = new java.util.Date();

stmt.setString(1, message);
stmt.setObject(2, date);
```

Die Methode `executeUpdate()` der `PreparedStatement`-Instanz führt dann das Kommando gegen die Datenbank aus. Ihr werden beim `PreparedStatement` keine Parameter übergeben:

```
stmt.executeUpdate();
```

Zwar existieren auch die im Interface `Statement` definierten Versionen, die als Parameter ein SQL-Statement entgegen nehmen, jedoch wäre die Verwendung dieser Varianten relativ sinnfrei, da wir so auf jeglichen Vorteil, den uns das `PreparedStatement` verschafft, verzichten würden. Dann könnten wir gleich wieder auf das Standard-`Statement` zurückgreifen.

Nach Ausführung der Methode `executeUpdate()` befindet sich der neue Datensatz in der Datenbank:

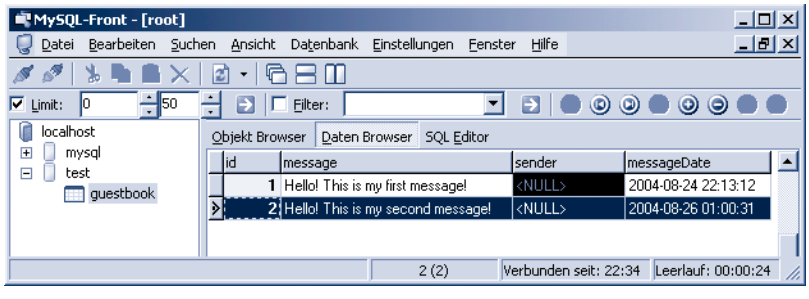


Abbildung 12.5
Der zweite Datensatz ist per PreparedStatement eingefügt worden

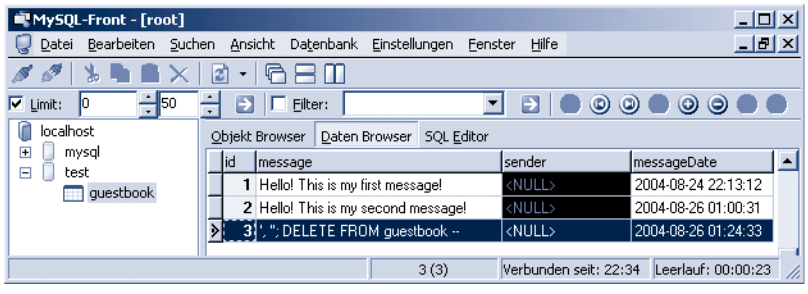
Lassen Sie uns nun einmal einen bösartigen *SQL-Injection*-Versuch starten. Bei Verwendung eines `PreparedStatements` müsste dieser Versuch fehlschlagen und das SQL-Statement, das in die Datenbank eingeschleust und ausgeführt werden sollte, müsste wie ein normaler Text in der Datenbank erscheinen:

```
String message = "', ''; DELETE FROM guestbook --";
Java.util.Date date = new java.util.Date();

stmt.setString(1, message);
stmt.setObject(2, date);

stmt.executeUpdate();
```

Versuchen Sie das nicht mit einer gewöhnlichen `Statement`-Instanz! Das `PreparedStatement` sorgt dafür, dass der bösartige Code unschädlich gemacht und als Text in die Datenbank eingefügt wird. Bei Verwendung von `Statement`-Implementierungen geschieht dies nicht und die Datenbank würde gelöscht werden.

Wenn wir das Statement ausführen, werden wir feststellen, dass wir zwar einen seltsam aussehenden Datenbank-Eintrag vorfinden, die anderen Daten jedoch weiterhin in der Datenbank vorhanden sind:

Abbildung 12.6
Da hat wohl jemand versucht, unsere Datenbank zu löschen ...?



Ändern eines Datensatzes

Wenn wir einen Datensatz ändern wollen, können wir dies ebenfalls mit Hilfe eines `PreparedStatements` und der `executeUpdate()`-Methode erledigen:

```
PreparedStatement stmt = conn.prepareStatement(
   "UPDATE guestbook SET sender = ? WHERE id = ?");

stmt.setString(1, sender);
stmt.setInt(2, id);

stmt.executeUpdate();
```

Löschen eines Datensatzes

Auch das Löschen von Datensätzen funktioniert mit Hilfe eines `PreparedStatements`:

```
PreparedStatement stmt = conn.prepareStatement(
   "DELETE FROM guestbook WHERE id = ?");

stmt.setInt(1, id);

stmt.executeUpdate();
```

Wie auch bei einer Statement-Implementierung erfordert der Einsatz des `PreparedStatements` spätestens beim Aufruf der `executeUpdate()`-Methode die Kapselung innerhalb eines `try-catch`-Blocks, da dabei eine `SQLException` geworfen werden kann.

12.5.2 Abrufen von Daten

Das Abrufen von Daten geschieht genauso, wie es auch beim Interface `Statement` der Fall wäre – nur die Vorbereitung der Ausführung ist anders: Statt einer `Statement`-Implementierung verwenden wir auch hier eine `PreparedStatement`-Instanz.

```
PreparedStatement stmt = conn.prepareStatement(
   "SELECT message, sender, messageDate FROM guestbook " +
   "WHERE id > ?");

stmt.setInt(1, 1);
```

Die Ausführung der durch das `PreparedStatement` repräsentierten Datenbank-Abfrage geschieht durch deren Methode `executeQuery()`, der wir keinen Parameter übergeben:

```
ResultSet rs = stmt.executeQuery();
```

Dieses `ResultSet` können wir nun so lange durchlaufen, bis seine Methode `next()` den Wert `false` zurückgibt:

```
while(rs.next()) {
   // ...
}
```

Die einzelnen Werte rufen wir aus dem `ResultSet` mit Hilfe seiner `get`-Methoden ab. Schwierigkeiten macht dabei das bei MySQL mögliche `DATETIME`-Format – dieses kann durch `JDBC` nicht abgebildet werden. Stattdessen müssen wir den Wert der entsprechenden Spalte zweimal abrufen und erhalten dann zwei `Date`-Instanzen, von denen eine ein Datum und die andere eine Uhrzeit repräsentiert:

```
String message = rs.getString(1);
String sender = rs.getString(2);
java.util.Date d = rs.getDate(3);
java.util.Date t = rs.getTime(3);
```

Mit Hilfe einer `Calendar`-Instanz und der `SimpleDateFormatter`-Klasse können die Werte für Datum und Uhrzeit dann verarbeitet und später ausgegeben werden:

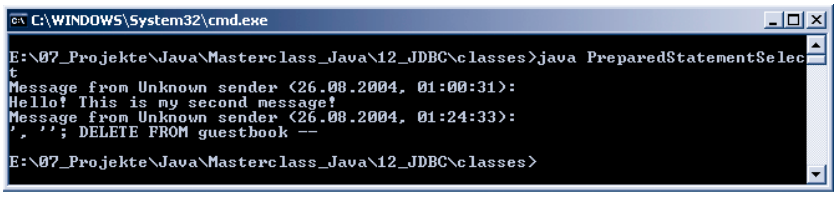


Abbildung 12.7
Ausgabe der Datenbank-Einträge auf der Kommandozeile

12.6 Tipps und Tricks

Der Datenbank-Zugriff per JDBC ist auf der einen Seite recht einfach umzusetzen, andererseits lauern eine Menge Fallstricke in der Arbeit mit Datenbanken. Lassen Sie uns hier also einige wichtige Regel, Tipps und Tricks besprechen, die Ihnen die Arbeit mit Datenbanken und JDBC als Datenbank-Schnittstelle erleichtern werden.

12.6.1 Sichern Sie Ihre Datenbank!

Dies ist schon der wichtigste Tipp, den man Ihnen geben kann. Im Gegensatz zu Dateien, auf die man in der Regel nur vom lokalen System aus zugreifen kann, sind Datenbanken oftmals über das Netzwerk ansprechbar. MySQL beispielsweise lauscht auf den Netzwerk-Port 3306 und der Microsoft SQL-Server ist über den Port 1433 erreichbar. Auch wenn Ihre Datenbank-Software selber „unbreakable" sein sollte: Unsichere Passwörter oder freizügig verteilte Zugriffsrechte schaffen genügend Lücken für Angreifer.

Aus diesem Grund sollten Sie folgende Maßregeln beachten:

- Vergeben Sie an jeden Datenbank-Nutzer ein eindeutiges (und möglichst komplexes) Passwort. Da Datenbank-Nutzer oftmals Applikationen sein werden, müssen Sie auf die Merkbarkeit der Kennwörter keinen großen Wert legen.
- Geben Sie den Datenbank-Nutzern so wenig Rechte wie möglich. Weder müssen gewöhnliche Datenbank-User neue Tabellen anlegen, noch benötigen sie ständig Administrator-Privilegien. Die meisten Applikationen beschränken sich auf INSERT-, UPDATE-, SELECT- und DELETE-Aktionen – dementsprechend müssen Sie auch nur diese Aktionen zulassen.
- Ändern Sie nach Möglichkeit den Port, auf dem die Datenbank lauscht. Denn wenn die Datenbank nicht mehr über den Standard-Port ansprechbar ist, dann wird es für Angreifer auch deutlich schwieriger, Verbindungen zur Datenbank herzustellen.
- Setzen Sie Firewalls ein. Dabei muss es sich nicht um teure High-End-Lösungen handeln, sondern oftmals reichen Desktop- oder Router-Firewalls völlig aus. Diese blocken unerwünschte Zugriffe auf die Datenbank zuverlässig ab – und Ihre Datenbank ist von außen nicht mehr sichtbar.
- Halten Sie Ihre Datenbank auf einem aktuellen Stand. Egal, welche Art von Datenbank Sie einsetzen: Alle großen Datenbank-Hersteller bietet regelmäßig Updates für ihre Produkte an. Diese Sicherheits-Updates sind in der Regel kostenlos zu beziehen – Sie sollten diese Möglichkeit wahrnehmen und neue Updates zeitnah einspielen.

12.6.2 Verwenden Sie PreparedStatements!

Statt des unsicheren `Statement`-Interfaces sollten Sie `PreparedStatement`-Instanzen einsetzen. Diese sind nicht nur sicherer, sondern vielfach auch schneller beim Zugriff auf die Datenbank. Die Handhabung von `PreparedStatements` ist darüber hinaus potentiell intuitiver und eingängiger. Scheuen Sie nicht den zusätzlichen Schreibaufwand, den Ihnen der Einsatz von `PreparedStatements` bescheren könnte – die Vorteile überwiegen die Bequemlichkeit ganz sicher.

12.6.3 Geben Sie bei SELECT-Statements immer alle Spalten an!

Sie können bei `SELECT`-Statements einen Stern als Platzhalter verwenden, wenn Sie nicht alle Spalten angeben wollen, die abgerufen werden sollen:

```
SELECT * FROM guestbook
```

Offensichtlich ist dies sehr effektiv und auch schnell zu schreiben. Besser jedoch – da besser lesbar – ist die explizite Angabe der abzurufenden Spalten:

```
SELECT id, message, sender, messageDate from guestbook
```

Für den Datenbank-Server hat dies zudem den Vorteil, dass ein wenig Aufwand bei Parsen des SQL-Statements entfällt, da er nicht herausfinden muss, welche Spalten in der Tabelle tatsächlich in Frage kommen. Für den Entwickler ist dies vorteilhaft, weil so offensichtlicher wird, welche Spalten tatsächlich abgerufen werden sollen, was die Wartbarkeit der Applikation deutlich erhöht.

12.6.4 Kapseln Sie Datenbankzugriffe in Datenklassen

Folgender Ansatz hat sich in der Praxis bewährt:

- Definieren Sie Klassen, die Ihre Daten repräsentieren sollen.
- Erstellen Sie ein Interface, das Methoden für den Zugriff auf bestimmte Daten bereitstellt.
- Implementieren Sie das Interface in konkreten Klassen.
- Erstellen Sie eine Factory, mit deren Hilfe Sie eine Instanz der konkreten Datenabruf-Klasse erzeugen können.

Klingt nach viel Aufwand? Ist es eigentlich nicht, denn es zwingt Sie, sich konkret zu überlegen, welche Daten wie repräsentiert werden können und wie Sie diese abrufen können. Dies bedeutet zwar, dass Sie im Vorfeld etwas größeren Aufwand betreiben müssen, um ein Konzept und die Klassen zu erstellen, dafür steigern sich aber Performance und Wartbarkeit Ihrer Applikation deutlich.

Dieser Ansatz nennt sich übrigens `DAO`-Ansatz.

Repräsentation eines Datensatzes

Lassen Sie uns diesen Ansatz für den Zugriff auf unsere Gästebuch-Datenbank nachvollziehen. Wir benötigen zunächst eine Klasse, die einen Datensatz im Gästebuch repräsentiert:

Listing 12.1 Repräsentation eines Gästebuch-Eintrags

```
import java.util.Date;

/**
 * Represents an entry in the guestbook
 */
public class GuestbookEntry {

   private int id = 0;
   private String message = "";
   private String sender = "";
   private Date created = new Date();

   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public String getMessage() {
      return message;
   }

   public void setMessage(String message) {
      this.message = message;
   }

   public String getSender() {
      return sender;
   }

   public void setSender(String sender) {
      this.sender = sender;
   }

   public Date getCreated() {
      return created;
   }

   public void setCreated(Date created) {
      this.created = created;
   }
}
```

Die Klasse `GuestbookEntry` selbst verfügt über keine eigene Verarbeitungslogik – sie hält lediglich einen Datensatz und erlaubt einen strukturierten Zugriff darauf. Nebenbei definiert sie Standard-Werte, die zum Einsatz kommen, wenn keine anderen Werte zugewiesen worden sind.

Datenzugriffs-Interface (DAO-Interface)

Das Datenzugriffs-Interface definiert die Methoden, die verwendet werden können, um auf die Daten zuzugreifen. Gleichzeitig erlaubt es implementierenden Klassen, eine komplett eigene Logik zu verwenden, um die Daten abzurufen, zu manipulieren oder zu verwalten. Was zunächst nach mehr Aufwand klingt, bietet zwei große Vorteile:

- In der Applikation wird nur mit dem `DAO`-Interface gearbeitet. Die konkrete Implementierung bleibt der Applikation verborgen. Dies sorgt für eine bessere Wartbarkeit der Applikation, denn die konkrete Datenzugriffs-Klasse kann ausgetauscht oder geändert werden, ohne dass der Code in der Applikation geändert werden müsste.
- Eventuelle Datenbankzugriffe finden über eine zentrale Klasse statt. Ändert sich das Datenmodell in der Datenbank, reicht die Anpassung dieser zentralen Klasse. Die Methoden-Aufrufe in den einzelnen Applikations-Elementen müssen nicht geändert werden.

Es sprechen also einige gewichtige Gründe für den Einsatz von Datenzugriffs-Interfaces. Ein derartiges `DAO`-Interface muss auch nicht komplex sein, wie das folgende Beispiel belegt:

Listing 12.2
Das Interface GuestbookDAO definiert die Methoden, die für den Datenzugriff verwendet werden können

```
/**
 * Defines methods for the access of our
 * guestbook database
 */
public interface GuestbookDAO {

   /**
    * Retrieves an array of entries
    */
   public GuestbookEntry[] retrieve();

   /**
    * Adds an entry to the guestbook
    */
   public void add(GuestbookEntry entry);

   /**
    * Updates an entry in the guestbook
    */
   public void update(GuestbookEntry entry);
}
```

Unser Interface `GuestbookDAO` definiert drei Methoden, die den Zugriff auf die Daten erlauben. Die Methode `retrieve()` gibt ein Array von `GuestbookEntry`-Instanzen zurück. Mit Hilfe von `add()` können wir einen neuen Gästebuch-Eintrag anfügen und per `update()` kann der übergebene Eintrag geändert werden.

Die konkrete Implementierung der Methoden bleibt den einzelnen `DAO`-Objekten überlassen.

DAO-Implementierung

Die DAO-Implementierung ist für den eigentlichen Zugriff auf die zugrunde liegenden Daten zuständig. Sie implementiert zu diesem Zweck das zuvor definierte DAO-Interface und erlaubt uns somit das Abrufen und Manipulieren von Daten-Objekten. Die Benennung der Klassen bleibt dabei grundsätzlich dem Entwickler überlassen.

Eine DAO-Implementierung des GuestbookDAO-Interfaces für eine MySQL-Datenbank könnte so aussehen:

Listing 12.3
Die DAO-Implementierung erlaubt den Zugriff auf die Datenbank

```
import java.sql.*;
import java.util.ArrayList;

/**
 * Concrete implementierung of the GuestbookDAO
 */
public class MySQLGuestbookDAO implements GuestbookDAO {

   private Connection conn;

   MySQLGuestbookDAO() {
      try {
         Class.forName("com.mysql.jdbc.Driver");
         conn = DriverManager.getConnection("...");
      } catch (ClassNotFoundException e) {
         e.printStackTrace();
      } catch (SQLException e) {
         e.printStackTrace();
      }
   }

   /**
    * Retrieves a list of guestbook entries
    */
   public GuestbookEntry[] retrieve() {
      ArrayList<GuestbookEntry> list =
         new ArrayList<GuestbookEntry>();

      if(conn != null) {
         try {

            PreparedStatement stmt = conn.prepareStatement(
               "SELECT id, message, sender, messageDate " +
               "FROM guestbook");

            ResultSet rs = stmt.executeQuery();
            while(rs.next()) {
               int id = rs.getInt(1);
               String message = rs.getString(2);
               String sender = rs.getString(3);
               java.util.Date d = (java.util.Date)rs.getObject(4);

               GuestbookEntry entry = new GuestbookEntry();
               entry.setCreated(d);
               entry.setId(id);
               entry.setMessage(message);
```

Listing 12.3 *(Forts.)*
Die DAO-Implementierung erlaubt den Zugriff auf die Datenbank

```
                entry.setSender(sender);

                list.add(entry);
            }

            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    return list.toArray(new GuestbookEntry[0]);
}

public void add(GuestbookEntry entry) {
    if(conn != null) {
        try {
            PreparedStatement stmt = conn.prepareStatement(
                "INSERT INTO guestbook " +
                "(message, sender, messageDate) " +
                "VALUES (?, ?, ?)");

            stmt.setString(1, entry.getMessage());
            stmt.setString(2, entry.getSender());
            stmt.setObject(3, entry.getCreated());

            stmt.executeUpdate();
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

public void update(GuestbookEntry entry) {
    if(conn != null) {
        try {
            PreparedStatement stmt = conn.prepareStatement(
                "UPDATE guestbook SET " +
                "message = ?, sender = ?, messageDate = ? " +
                "WHERE id = ?");

            stmt.setString(1, entry.getMessage());
            stmt.setString(2, entry.getSender());
            stmt.setObject(3, entry.getCreated());
            stmt.setInt(4, entry.getId());

            stmt.executeUpdate();
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }

    }
}
}
```

Die Klasse `MySQLGuestbookDAO` implementiert das Interface `GuestbookDAO` und greift dabei intern auf eine MySQL-Datenbank zu. Innerhalb ihrer Methode `retrieve()` ruft sie alle Einträge der Tabelle ab und gibt deren Inhalte in einem Array aus `GuestbookEntry`-Instanzen zurück.

Die Methoden `insert()` und `update()` nutzen `PreparedStatements`, um Daten in der Datenbank einzufügen oder zu ändern. Das Vorgehen dabei entspricht den weiter vorne in diesem Kapitel besprochenen Ansätzen – wir erzeugen die Statements und weisen ihnen die Inhalte mit Hilfe ihrer `set`-Methoden zu.

Factory

Für Applikationen, die das `GuestbookDAO` einsetzen wollen, sind diese Implementierungsdetails unwichtig. Sie nutzen eine `Factory`, um eine Instanz der Datenzugriffsklasse zu erzeugen, und müssen sich nicht darum kümmern, welche konkrete Implementierung Sie verwenden. Die `Factory` für dieses Beispiel ist sehr simpel gehalten – schließlich haben wir nur eine potentielle Datenquelle, die wir anbinden müssen:

```
/**
 * Factory for the creation of GuestbookDAO-instances
 */
public class GuestbookDAOFactory {

   public static GuestbookDAO getInstance() {
      return new MySQLGuestbookDAO();
   }
}
```

Die Methode `getInstance()` der `GuestbookDAOFactory`-Klasse gibt eine neue Instanz der `MySQLGuestbookDAO`-Klasse zurück. Da die Signatur der Methode deklariert, dass eine `GuestbookDAO`-Instanz zurückgegeben werden soll, könnten wir hier auch andere Instanzen anderer Klassen zurückgeben – solange diese das Interface implementieren.

Einsatz: Daten abrufen

Das Abrufen von Daten gestaltet sich mit dem `GuestbookDAO`-Interface und dessen Implementierungen sehr einfach:

- Zunächst holen wir uns eine Referenz auf eine `DAO`-Implementierung.
- Anschließend nutzen wir ausschließlich deren Methoden, um auf Daten aus der Datenquelle zuzugreifen.

In unserem Code arbeiten wir nun also nicht mehr mit SQL-Statements, sondern Methoden-Aufrufen:

Listing 12.4
Verwenden einer GuestbookDAO-Instanz

```
/**
 * Selects and displays all entries
 * in the guestbook
 */
public class GuestbookSelect {

   public static void main(String[] args) {
      GuestbookDAO gb = GuestbookDAOFactory.getInstance();
      GuestbookEntry[] entries = gb.retrieve();

      for(GuestbookEntry entry : entries) {
         System.out.println("====================");
         System.out.println(
            String.format(
               "ENTRY #%s", entry.getId()));
         System.out.println(
            String.format(
               "%s wrote: %s (%s)",
                  entry.getSender(),
                  entry.getMessage(),
                  entry.getCreated()));
      }

      System.out.println("====================");
   }
}
```

Neben dem Vermeiden unnötiger Redundanzen aufgrund mehrfach im Code vorhandener SQL-Statements ist der Zugriff auf die Daten nun für unsere Applikation bemerkenswert simpel geworden, denn wir müssen uns nicht mehr darum kümmern, welche Art von Datenquelle (Datenbank, Textdatei, XML-Datei) wir abfragen müssen oder wie diese Abfragen im Detail aussehen. Alles, was nun an Logik für den Datenzugriff nötig ist, passt in zwei Anweisungen:

```
GuestbookDAO gb = GuestbookDAOFactory.getInstance();
GuestbookEntry[] entries = gb.retrieve();
```

Dass sich durch einen derartigen Ansatz die Wartbarkeit der Applikation erhöht, ist offensichtlich. Selbiges gilt für die Sicherheit und potentiell auch die Performance, denn das zentrale Abrufen der Daten innerhalb einer `DAO`-Implementierung erleichtert den Einsatz von Caching-Technologien.

Einsatz: Daten speichern

Analog zum Abrufen von Daten verhält sich das Vorgehen bei deren Speicherung oder Aktualisierung, denn auch hier erzeugen wir mit Hilfe der `DAO`-Factory eine Referenz auf eine spezifische `DAO`-Implementierung, ohne uns um die Details dieser Implementierung kümmern zu müssen:

Listing 12.5
Speichern von Daten mit Hilfe einer DAO-Instanz

```
/**
 * Adds an entry to the guestbook
 */
public class GuestbookInsert {

   public static void main(String[] args) {

         // ...

         GuestbookEntry entry = new GuestbookEntry();
         entry.setMessage(message);
         entry.setSender(sender);

         GuestbookDAO gb = GuestbookDAOFactory.getInstance();
         gb.add(entry);
      }
   }
}
```

Einfacher kann das Speichern von Daten doch fast schon nicht mehr sein: Zunächst erzeugen wir eine neue Instanz der `GuestbookEntry`-Klasse, die die zu speichernden Daten enthält. Anschließend lassen wir uns von der `GuestbookDAOFactory`-Klasse eine Referenz auf die `DAO`-Implementierung zurückgeben, die unsere Daten speichern soll – was sie dann auch mit Hilfe ihrer Methode `add()` erledigt.

DAO oder nicht DAO – ab wann machen derartige Datenklassen Sinn?

Eigentlich machen `DAOs` auch bei kleinsten Applikationen Sinn, denn die Vorteile ihres Einsatzes sind überwältigend:

- Bessere Trennung von Daten und Applikation
- Bessere Wartbarkeit der Applikation
- Geringere Implementierungsaufwände für einbindende Methoden
- Größere Datensicherheit aufgrund zentraler Behandlung
- Deutlich größere Flexibilität der Applikation hinsichtlich ihrer Speichermedien und der verwendbaren Datenbanken
- Bessere Update-Fähigkeit der Daten- und Daten-Zugriffsklassen

Wenn man es streng nimmt, sollte die Verwendung von `DAOs` eigentlich in jedem Projekt stattfinden. Dem entgegen stehen eigentlich nur zwei Punkte: der größere Aufwand bei der Konzeptionierung der Applikation und die Disziplin der Entwickler, die `DAOs` zunächst schreiben und testen müssen, bevor sie sie in ihren Applikationen einsetzen können.

Der Autor empfiehlt den Einsatz von `DAOs` zur Kapselung von Daten(bank)-Zugriffen dringend.

12.6.5 Auslagerung der ConnectionStrings und der Klassen-Namen

Der letzte Tipp dieses Kapitels betrifft das Ablegen der `ConnectionStrings` und der Namen der `JDBC`-Treiber. Letzteres mag – insbesondere beim Einsatz von `DAOs` – nicht zwingend nötig sein, aber `ConnectionStrings` und eventuell Benutzernamen-/Kennwort-Kombinationen haben im Java-Code nichts verloren. Oder möchten Sie Ihre Applikation jedes Mal neu kompilieren müssen, wenn sich die Parameter für den Zugriff auf die Datenbank geändert haben?

Die einfachste Möglichkeit der Auslagerung der `ConnectionStrings` stellt die Verwendung einer `Properties`-Datei dar, die sich im Applikations-Verzeichnis befindet. Vor dem Zugriff auf eine Datenbank laden Sie die `Properties`-Datei in eine `Properties`-Instanz und lesen daraus den zu verwendenden `ConnectionString` aus.

Um diesen Ansatz umzusetzen, reichen wenige Zeilen Code aus. Sehen wir uns dies am Beispiel der gerade vorgestellten `MySQLGuestbookDAO`-Klasse an. In deren Konstruktor wird eine Datenbank-Verbindung definiert, die dann von den Klassen-Methoden verwendet wird. Der Ansatz mit einem festverdrahteten `ConnectionString` sieht derzeit so aus:

```
Class.forName("com.mysql.jdbc.Driver");
conn = DriverManager.getConnection("...");
```

Wenn wir statt der festverdrahteten Parameter die Möglichkeit haben wollen, die Verbindungs-Parameter extern abzulegen, können wir dies mit folgendem Code umsetzen:

```
Properties settings = new Properties();
String fileName = System.getProperty("user.home") +
   "/connection.properties";

settings.load(new FileInputStream(new File(fileName)));
String className = settings.getProperty("className");
String connStr = settings.getProperty("connectionString");

Class.forName(className );
conn = DriverManager.getConnection(connStr);
```

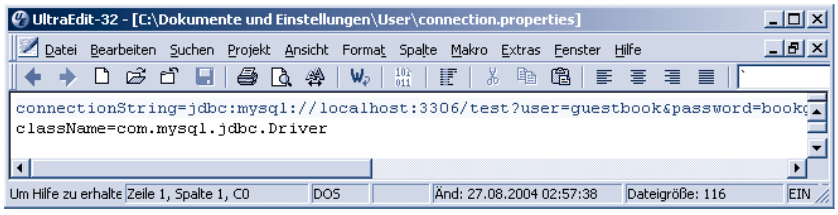
Wir erzeugen hier eine neue `Properties`-Instanz, die die im Home-Verzeichnis des Benutzers unter dem Namen `connection.properties` abgelegten Einstellungen laden wird.

Achtung

Die Property `user.home` zeigt unter Windows nicht auf den – gerne als *Home*-Verzeichnis angesehenen – Ordner *Eigene Dateien*, sondern auf das eigentliche Home-Verzeichnis. Dieses befindet sich bei Windows XP beispielsweise im Order *c:\Dokumente und Einstellungen\<Benutzername>*. Der Ordner *Eigene Dateien* befindet sich dagegen unterhalb des *Home*-Verzeichnisses.

Innerhalb der Datei `connection.properties` sind zwei Schlüssel definiert: `className` mit dem Namen des JDBC-Treibers und `connectionString` mit den Verbindungs-Informationen zur Datenbank:

Abbildung 12.8
In der Datei connection.properties sind die Verbindungs-Informationen abgelegt



Nach dem Laden der Daten in die `Properties`-Instanz können diese Daten nun abgerufen und verwendet werden.

Für uns bietet dieses Vorgehen mehrere Vorteile:

- Keine festverdrahteten Verbindungs-Informationen mehr in Applikationen
- Bessere Wartbarkeit aufgrund zentraler Ablage der Verbindungs-Informationen
- Änderung der Informationen möglich, ohne dass die Applikation neu kompiliert werden müsste

Statt festverdrahteter Verbindungs-Informationen sollten Sie also diese Daten in jedem Fall extern halten.

12.7 Fazit

`JDBC` ist eine Technologie, die es uns erleichtert, auf Datenbanken zuzugreifen. Das Herstellen von Datenbank-Verbindungen wird damit ebenso einfach wie das Auslesen von Daten. Da für fast jedes Datenbank-System spezielle `JDBC`-Treiber existieren, können wir meist mit optimaler Performance auf die Datenbank zugreifen.

Gleichzeitig ist `JDBC` eine Technologie, bei deren Einsatz viele Fehler begangen werden können. Dies betrifft nicht nur den Applikations-Aufbau, sondern ganz besonders auch die Datenbank-Systeme selbst, von deren Unsicherheit und schlechter Konfiguration böswillige Zeitgenossen profitieren können. Als Entwickler sollten wir also unsere Systeme immer auf einem aktuellen Stand halten und auch auf Seiten von Java die Sicherheit unserer Applikation nicht aus den Augen verlieren.

Der ebenfalls in diesem Kapitel vorgestellte `DAO`-Ansatz für die Trennung von Applikation und Datenzugriffsschicht erleichtert den Umgang mit Datenbanken und anderen Datenquellen enorm und erfordert andererseits keinen ungebührlich hohen Aufwand beim Einsatz. Wenn Sie sich mit dem Ansatz vertraut gemacht haben, werden Sie dessen Vorteile nicht mehr missen wollen.

13

Netzwerk

Der Zugriff und die Arbeit mit Netzwerken ist im Java-Umfeld bemerkenswert einfach! In den Paketen `java.net` und `java.nio` werden uns dafür einige Klassen zur Verfügung gestellt, die es uns wirklich leicht machen, mit dem Netzwerk und dem Internet zu arbeiten.

13.1 Auflösung eines Hostnamens

Die Auflösung eines Hostnamens (etwa einer Internet-Adresse) zu einer IP-Adresse können wir mit Hilfe der Klasse `InetAdress` aus dem `java.net`-Paket vornehmen. Wir nutzen zu diesem Zweck deren statische Methode `getByName()`, die einen Hostnamen oder eine IP-Adresse entgegennimmt und eine `InetAdress`-Instanz zurückgibt, mit deren Hilfe wir mehr über den Host erfahren können:

```
static InetAddress getByName(String host)
   throws UnknownHostException
```

Wenn wir die `InetAddress`-Instanz als Rückgabe der Methode erhalten haben, können wir mit deren Hilfe sehr einfach den Hostnamen und die IP-Adresse ermitteln. Außerdem sind wir in der Lage, feststellen zu können, ob der Host erreichbar ist.

Die Ermittlung des Host-Namens geschieht mit Hilfe der Methode `getHostname()`. Die IP-Adresse wird durch die Methode `getHostAddress()` aufgelöst werden. Und die Methode `isReachable()` gibt uns zurück, ob der betreffende Host per `PING` erreicht werden kann:

```
public String getHostName()

public String getHostAddress()

public boolean isReachable(int timeout)
   throws IOException

public boolean isReachable(NetworkInterface netif,
   int ttl, int timeout)
   throws IOException
```

Eine Umsetzung in einer kleinen Applikation, die diese Informationen für einen übergebenen Host-Namen- oder eine IP-Adresse ausgibt, könnte so aussehen:

Listing 13.1 Informationen über einen Netzwerk-Host abrufen

```
import java.net.InetAddress;
import java.io.IOException;

/**
 * Displays information about a network host
 */
public class NetworkHost {

   public static void main(String[] args) throws IOException {

      if(args == null || args.length == 0) {
         System.out.println("Usage: NetworkHost <Host>");
         System.exit(0);
      }

      String host = args[0];
      InetAddress address = InetAddress.getByName(host);

      System.out.println(String.format("Hostname: %s",
         address.getHostName()));
      System.out.println(String.format("Host-Address: %s",
         address.getHostAddress()));
      System.out.println(String.format("Is reachable: %s",
         address.isReachable(2000)));
   }
}
```

Die Applikation arbeitet recht unspektakulär: Sie nimmt über die Kommandozeile eine IP-Adresse oder einen Hostnamen entgegen und übergibt diesen Wert an die statische Methode `getByName()` der `InetAddress`-Klasse, die uns eine Klassen-Instanz zurückgibt, die den angegebenen Wert repräsentiert:

```
InetAddress address = InetAddress.getByName(host);
```

Nun können wir den Hostnamen und die IP-Adresse des Hosts ermitteln lassen. Wurde ein Hostname bereits als Parameter übergeben, wird dieser von `getHostName()` einfach wieder zurückgegeben. Wird eine IP-Adresse angegeben, wird diese als Hostname wieder zurückgegeben.

Die Methode `isReachable()`, die intern einen `PING` auf den angegegeben Host ausführt und dabei das als Parameter übergebene `PING`-Timeout verwendet, stellt fest, ob der durch den Hostnamen oder die IP-Adresse repräsentierte Server erreichbar ist. Die Rückgabe ist entweder `true` oder `false`, wobei auch `false` zurückgegeben wird, wenn der Server zwar erreichbar ist, aber nicht auf `PING`s reagiert.

Wenn wir die Applikation ausführen und als Host beispielsweise `www.karsten-samaschke.de` angeben, können wir diese Ausgabe erhalten:

Abbildung 13.1
Ausgabe der per InetAddress ermittelten Informationen zu einem Host

13.2 Netzwerk-Adapter auflisten

Die Klasse `java.net.NetworkInterface` verfügt über eine statische Methode `getNetworkInterfaces()`, die eine `Enumeration` vom Typ `NetworkInterface` zurückgibt:

```
static Enumeration<NetworkInterface> getNetworkInterfaces()
   throws SocketException
```

Beim Durchlaufen der `Enumeration` erhalten wir jeweils eine Instanz der `NetworkInterface`-Klasse, die einen Netzwerk-Adapter repräsentiert. Diese verfügt über eine Methode `getDisplayName()`, die den Anzeige-Namen des Adapters zurückgibt. Eine Liste aller an den Adapter gebundenen IP-Adressen erhalten wir durch die Methode `getInetAddresses()`:

```
public String getDisplayName()
public Enumeration<InetAddress> getInetAddresses()
```

Mit Hilfe dieser Informationen können wir nun eine Liste aller Netzwerk-Adapter samt den zugeordneten IP-Adressen ausgeben:

```
try {
   Enumeration<NetworkInterface> nis =
      NetworkInterface.getNetworkInterfaces();
   while(nis.hasMoreElements()) {
      NetworkInterface ni = nis.nextElement();

      System.out.println(ni.getDisplayName());
      System.out.println("IP-Addresses:");

      Enumeration<InetAddress> addresses =
         ni.getInetAddresses();
      while(addresses.hasMoreElements()) {
         InetAddress address = addresses.nextElement();
         System.out.println(address.getHostAddress());
      }
   }
} catch (SocketException e) {
   e.printStackTrace();
}
```

Die Liste der Netzwerk-Adapter liefert uns die Methode `getNetworkInterfaces()`. Dabei kann eine `SocketException` auftreten, weshalb wir das Abrufen der Informationen in einem `try-catch`-Block kapseln.

```
Enumeration<NetworkInterface> nis =
   NetworkInterface.getNetworkInterfaces();
```

Anschließend kann die `Enumeration` durchlaufen werden. Wir geben zunächst den Anzeige-Namen der aktuellen `NetworkInterface`-Instanz aus, auf den wir mit Hilfe ihrer Methode `getDisplayName()` zugreifen:

```
System.out.println(ni.getDisplayName());
```

Die dem Netzwerk-Adapter zugeordneten IP-Adressen rufen wir mit Hilfe der Methode `getInetAddresses()` ab, die uns eine `Enumeration` aus `InetAddress`-Instanzen zurückgibt:

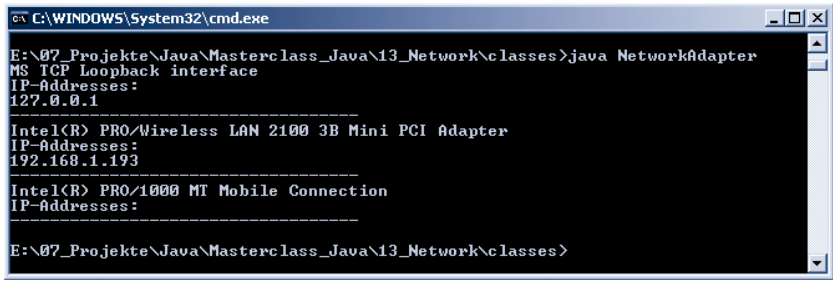
```
Enumeration<InetAddress> addresses = ni.getInetAddresses();
```

Jede der in dieser Enumeration enthaltenen `InetAddress`-Instanzen gibt uns mit Hilfe der Methode `getHostAddress()` die repräsentierte IP-Adresse zurück, die wir ausgeben:

```
System.out.println(address.getHostAddress());
```

Auf dem System des Autors produziert der obige Code folgende Ausgabe:

Abbildung 13.2
Auflistung aller Netzwerk-Adapter



Ob ein Netzwerk-Adapter verwendet werden kann, erkennen Sie übrigens daran, ob ihm IP-Adressen zugeordnet worden sind.

13.3 URL

Die Klasse `URL` aus dem `java.net`-Package repräsentiert eine Ressource, die sich entweder im Netzwerk oder auf dem lokalen System befindet. Die Art einer derartigen Ressource ist nicht vordefiniert – alles, auf das sich verweisen lässt, kann in der Regel auch durch eine `URL`-Instanz repräsentiert werden.

Strukturell gesehen kann ein URL (Unique Resource Location Identifier) in verschiedene Bestandteile zerlegt werden. Analysieren wir zu diesem eine Web-

Adresse, wie etwa *http://www.karsten-samaschke.de/Default.htm*. Diese Adresse besteht aus folgenden Komponenten:

- Protokoll (http)
- Host-Angabe (www.karsten-samaschke.de)
- Port (in diesem Fall nicht vorhanden, wird bei Web-Adressen standardmäßig als Port 80 angenommen)
- Pfad (optional, Verzeichnis- und Datei-Angabe, in diesem Fall /Default.htm).

Neben diesen Standard-Komponenten können URLs auch noch weitere Komponenten enthalten, die aber vom Typ des Elements abhängig sind, auf das verwiesen werden soll.

URL-Angaben müssen nicht absolut erfolgen. Wird ein relativer URL angegeben, dann erfolgt das Setzen der Informationen zu Host und Port implizit.

Das Erzeugen einer `URL`-Instanz, die einen Verweis auf einen URL repräsentiert, erfolgt ausschließlich, indem deren Konstruktor die entsprechenden Informationen übergeben werden:

```
public URL(String spec)
   throws MalformedURLException

public URL(URL context, String spec)
   throws MalformedURLException

public URL(String protocol, String host, String file)
   throws MalformedURLException

public URL(String protocol, String host, int port,
   String file)
   throws MalformedURLException

public URL(String protocol, String host, int port,
   String file, URLStreamHandler handler)
   throws MalformedURLException
```

Die gebräuchlichsten Konstruktor-Überladungen sind die, die nur einen `String` mit der URL-Angabe und die drei Angaben für Protokoll, Host und Pfad-Komponente entgegennehmen. Ebenso ist es möglich, eine `URL`-Instanz für den Kontext und eine relative Angabe für die Pfad-Komponente als Parameter zu übergeben.

Für untypische `URL`-Angaben empfehlen sich die beiden letzten Konstruktoren, die die kompletten Informationen zu Protokoll, Host, Port und Pfad-Komponente entgegennehmen. Der letzte Konstruktor erlaubt sogar die Angabe eines so genannten `URLStreamHandlers`, also einer Klasse, die für das Handling der durch die `URL`-Klasse repräsentierten Ressource und deren Daten zuständig ist. Normalerweise müssen wir einen derartigen Handler nicht angeben, denn diese Handler sind mit dem Protokoll assoziiert. Für eine Angabe der Handler-Instanz sollten also triftige Gründe vorliegen.

13.3.1 Analyse einer URL

Eine erzeugte `URL`-Instanz erlaubt es uns, die Informationen zu Protokoll, Host, Port und Pfad nach Erzeugung wieder abzurufen:

- `public int getDefaultPort()`: Gibt den Standard-Port zurück
- `public String getFile()`: Gibt die Pfad-Komponente und die angehängte Query-Komponente der referenzierten Ressource zurück. Sollte kein Query-String angegeben sein, entspricht die Rückgabe von `getFile()` der Rückgabe von `getPath()`
- `public String getHost()`: Gibt den Host-Namen der Ressource zurück
- `public String getPath()`: Gibt die Pfad-Komponente zurück
- `public int getPort()`: Gibt den Port zurück oder -1, falls keine derartige Angabe erfolgt ist
- `public String getQuery()`: Gibt eine Query-String-Komponente zurück (oder null, falls nicht angegeben). Derartige Query-String-Angaben werden durch das Fragezeichen (?) von der Adresse getrennt und beinhalten Name-/Werte-Parameter, die der Server zur Verarbeitung der Anfrage benötigt. Gibt `null` zurück, falls kein derartiger Wert angegeben ist.
- `public String getProtocol()`: Gibt das angegebene Protokoll zurück
- `public String getRef()`: Gibt die Referenz zu einer Web-Adresse zurück. Derartige Referenzen werden durch die Raute (#) von der eigentlichen Adresse getrennt und dienen der Navigation innerhalb einer Webseite. Gibt `null` zurück, wenn keine Referenz angegeben worden ist.
- `public String getUserInfo()`: Gibt die in der Adresse enthaltene Benutzernamen-Angabe zurück. Diese wird in der Regel durch das `@`-Zeichen von der eigentlichen Adress-Angabe getrennt. Aufgrund einer Sicherheits-Lücke im Microsoft Internet Explorer sollten derartige Angaben nicht mehr verwendet werden. Gibt `null` zurück, wenn keine User-Information angegeben worden ist.

Um die Verwendung dieser Methoden zu demonstrieren, können wir folgenden Code verwenden:

Listing 13.2 Analyse einer URL

```
import java.net.URL;
import java.net.MalformedURLException;

/**
 * Analyses an URL
 */
public class UrlInfo {

   public static void main(String[] args) {

      if(null == args || args.length == 0) {
         System.out.println("Usage: UrlInfo <Address>");
         System.exit(0);
      }

      String address = args[0];
```

Listing 13.2 *(Forts.)*
Analyse einer URL

```
      try {
         URL url = new URL(address);

         System.out.println("Authority   : " +
            url.getAuthority());
         System.out.println("Default-Port: " +
            url.getDefaultPort());
         System.out.println("File        : " + url.getFile());
         System.out.println("Host        : " + url.getHost());
         System.out.println("Path        : " + url.getPath());
         System.out.println("Port        : " + url.getPort());
         System.out.println("Protocol    : " + url.getProtocol());
         System.out.println("Query       : " +
            (null != url.getQuery() ? url.getQuery() : ""));
         System.out.println("Reference   : " +
            (null != url.getRef() ? url.getRef() : ""));
         System.out.println("User-Info   : " +
            (null != url.getUserInfo() ? url.getUserInfo() : ""));
      } catch (MalformedURLException e) {
         e.printStackTrace();
      }
   }
}
```

Die Funktionsweise des Beispiels ist recht selbsterklärend. Lediglich die Verwendung der Methoden `getQuery()`, `getRef()` und `getUserInfo()` erfordert eine besondere Vorsicht, da diese Methoden bei Nichtvorhandensein der entsprechenden Komponenten null zurückgeben.

Wenn ein Nutzer mit der Applikation arbeitet will, kann er eine gültige Adresse als Parameter angeben und bekommt den daraus gebildeten URL analysiert:

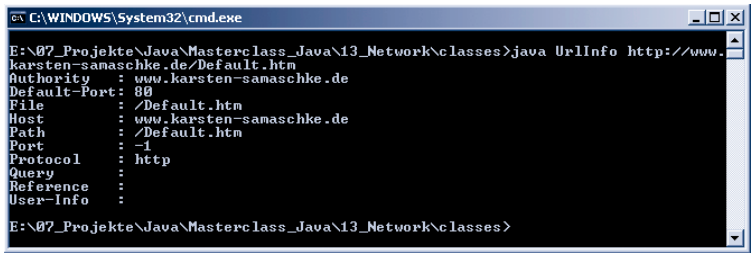


Abbildung 13.3
Zerlegung einer URL in ihre Bestandteile

13.3.2 Verarbeitung des referenzierten Inhalts

Die Verarbeitung einer URL-Ressource erfolgt entweder durch `URLConnection`-Instanzen oder durch `InputStreams`. Dies bedeutet, dass wir die referenzierten Daten beispielsweise auch lokal speichern können. Um auf die `URLConnection`- und `InputStream`-Instanzen zugreifen zu können, verwenden wir die Methoden `openConnection()` und `openStream()`:

```
public URLConnection openConnection()
   throws IOException
public URLConnection openConnection(Proxy proxy)
   throws IOException
public final InputStream openStream()
   throws IOException
```

Um eine HTML-Datei mit Hilfe einer `URL`-Instanz herunterzuladen und lokal abzuspeichern, können wir den `InputStream` der `URL`-Instanz mit Hilfe eines `BufferedReaders` (und der Brückenklasse `InputStreamReader`) einlesen und parallel entweder ausgeben oder in einer Daten speichern.

Dies könnte durch folgenden Code erledigt werden:

Listing 13.3
Einlesen des Inhalts einer per URL repräsentierten Ressource

```
import java.net.URL;
import java.net.MalformedURLException;
import java.io.*;

/**
 * Fetches the contents of a resource and
 * saves it to a local file or displays it
 */
public class UrlFileReader {

   public static void main(String[] args) {

      // ...

      String address = args[0];
      String fileName = null;
      if(args.length > 1) {
         fileName = args[1];
      }

      try {
         URL url = new URL(address);
         BufferedWriter bw;

         if(fileName != null) {
            bw = new BufferedWriter(
               new FileWriter(fileName));
         } else {
            bw = new BufferedWriter(
               new PrintWriter(System.out));
         }

         BufferedReader rdr = new BufferedReader(
            new InputStreamReader(url.openStream()));

         String line;
         while((line = rdr.readLine()) != null) {
            bw.write(line + "\n");
         }

         bw.close();
         rdr.close();
      } catch (MalformedURLException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Nach dem Erzeugen der `URL`-Instanz unter Angabe der vom Nutzer angegebenen Internet-Adresse können wir eine Instanz der `BufferedReader`-Klasse erzeugen. Deren Konstruktor weisen wir einen `InputStreamReader` zu, der die `InputStream`-Instanz, die von der Methode `openStream()` der `URL`-Instanz zurückgegeben wird, kapselt:

```
BufferedReader rdr = new BufferedReader(
   new InputStreamReader(url.openStream()));
```

Da der `BufferedReader` das zeilenweise Verarbeiten des Inhalts erlaubt, gestaltet sich das Einlesen der Webseite sehr einfach: Wir durchlaufen den per `readLine()` zurückgegebenen Inhalt so lange, bis `null` zurückgegeben wird. Der von uns verwendete `BufferedWriter` gibt die Zeile dann auf seinem Ausgabe-Stream (entweder die Standard-Ausgabe oder eine Datei) aus:

```
while((line = rdr.readLine()) != null) {
   bw.write(line + "\n");
}
```

Wenn wir die Klasse `UrlFileReader` ausführen, müssen wir zwingend eine Internet-Adresse angeben. Optional können wir auch einen Datei-Namen definieren, der die Datei angibt, in die der Inhalt der Ressource gespeichert wird. Sollte kein zweiter Parameter definiert sein, wird der abgerufene Inhalt direkt ausgegeben:

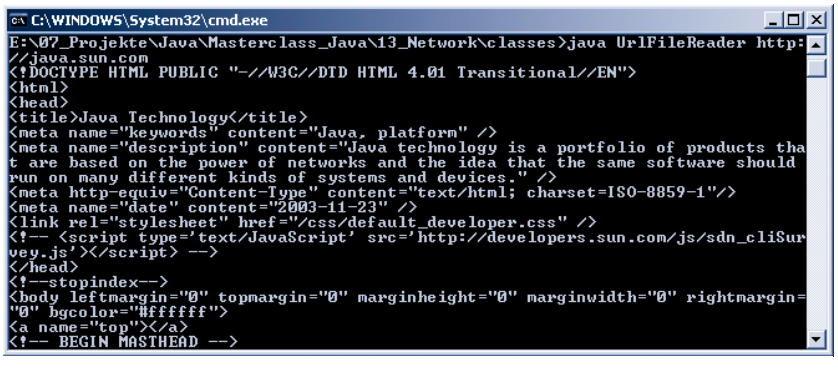


Abbildung 13.4
Abruf und Ausgabe einer HTML-Datei mit Hilfe der URL-Klasse

13.3.3 URLConnection

Die abstrakte Klasse `URLConnection` stellt uns nicht nur eine `InputStream`-Instanz (übrigens die gleiche `InputStream`-Instanz, die über die `URL`-Klasse erreichbar ist) zum Abrufen von Daten zur Verfügung, sondern gibt uns eine Menge mehr an Informationen zurück:

- `int getConnectTimeout()`: Gibt das Timeout der Verbindungsanforderung in Millisekunden zurück
- `String getContentEncoding()`: Gibt das Inhalts-Encoding zurück
- `int getContentLength()`: Gibt die Inhaltslänge in Bytes zurück
- `String getContentType()`: Gibt den Inhalts-Typ zurück

- `long getDate()`: Gibt das auf dem Server gesetzte Datum als Millisekunden seit dem 01.01.1970 zurück
- `long getExpiration()`: Gibt den Ablaufzeitpunkt des abgerufenen Inhalts als Millisekunden seit dem 01.01.1970 zurück
- `String getHeaderField(int n)`: Gibt den Inhalt des n-ten Header-Felds zurück
- `String getHeaderField(String name)`: Gibt den Inhalt des bezeichneten Header-Feldes zurück
- `InputStream getInputStream()`: Gibt den InputStream für das Abrufen des repräsentierten Inhalts zurück
- `long getLastModified()`: Gibt das Datum der letzten Änderung am repräsentierten Inhalt als Millisekunden seit dem 01.01.1970 zurück.
- `int getReadTimeout()`: Gibt das Timeout für das Empfangen von Daten in Millisekunden zurück

Dies ist nur eine kleine Auswahl der verfügbaren Informationen. Die `URLConnection`-Klasse bietet uns noch deutlich mehr Informationen, die aber den Rahmen dieses Buchs sprengen würden. Mit Hilfe der hier angeführten Methoden sind Sie jedoch in der Lage, den Inhalt und die verfügbaren Informationen zu einer Ressource zu analysieren:

```
import java.net.URL;
import java.net.URLConnection;
import java.io.IOException;
import java.util.Date;

// ...

URLConnection conn = url.openConnection();

System.out.println("Connect Timeout: " +
   conn.getConnectTimeout());
System.out.println("Content Encoding: " +
   conn.getContentEncoding());
System.out.println("Content Length: " +
   conn.getContentLength());
System.out.println("Content Type: " +
   conn.getContentType());
System.out.println("Date: " +
   new Date(conn.getDate()));
System.out.println("Exiration: " +
   new Date(conn.getExpiration()));
System.out.println("Last Modified: " +
   new Date(conn.getLastModified()));
System.out.println("Read Timout: " +
   conn.getReadTimeout());
```

Wenn wir uns so die Informationen zur Seite *http://www.karsten-samaschke.de/Default.htm* ausgeben lassen, erfahren wir einiges über die referenzierte Ressource:

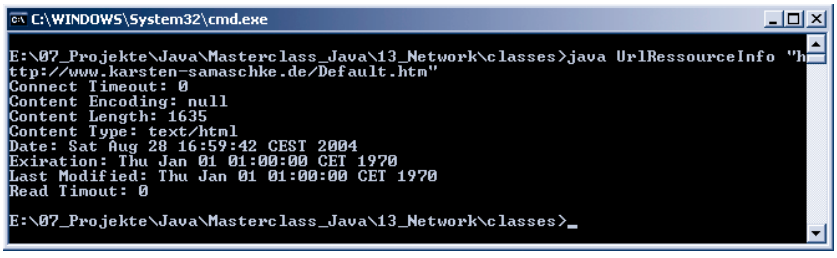


Abbildung 13.5
URLConnection-Informationen zu einer referenzierten Ressource

13.4 Client/Server-Kommunikation

Client/Server-Kommunikation ist die hohe Kunst der Arbeit mit dem Netzwerk. Einerseits ist es sehr einfach, eine derartige Kommunikation mit Hilfe von Java zu implementieren, andererseits sollte man die Fallstricke bei dieser Art der Kommunikation nicht unterschätzen, schließlich müssen Sie die Protokolle selbst implementieren.

13.4.1 Sockets

Die Socket-basierende Kommunikation basiert darauf, dass es einen Server und mindestens einen Client gibt. Der Server lauscht auf Verbindungs-Anfragen von Clients, die auf einem bestimmten Port erfolgen müssen. Die Nummer des Ports liegt für eigene Protokolle in der Regel höher als 1023, denn die Ports von 1 bis 1023 sind standardisiert – wenn Ihr Server also auf einen dieser Ports lauscht, sollte er auch die von anderen Clients erwarteten Kommunikations-Protokolle implementieren. Wenn dies nicht der Fall ist, sollte Ihr Server auf einen der vielen möglichen Ports zwischen 1024 und 65535 ausweichen, denn die stehen für alle Applikationen offen.

Übrigens haben Sie – möglicherweise, ohne es zu wissen – bisher auch schon ganz selbstverständlich mit einer Client-Server-Architektur gearbeitet. Sogar im Rahmen dieses Kapitels haben Sie sie eingesetzt. Sie wissen nicht, wann dies der Fall war? Jedes Mal, wenn Sie auf das Internet zugreifen und eine Webseite abrufen, nutzen Sie diese Architektur, denn Ihr Client (der Web-Browser) greift auf einen Web-Server zu. Mit Hilfe des HTTP-Protokolls kommunizieren die beiden miteinander und der Web-Server liefert den angeforderten Inhalt aus.

Generell findet diese Art von Kommunikation über Sockets statt – und zwar auf beiden Seiten. Der Server lauscht auf Anfragen auf einem definierten Port, an den wiederum ein Socket gebunden ist. Der Client kommuniziert über einen Socket, der an einen Port gebunden ist, mit dem Server.

Clientseitige Kommunikation

Java verfügt über die Klasse `Socket`, die die Kommunikation über Sockets auf Client-Seite erlaubt. Diese Klasse verfügt über einen mehrfach überladenen Konstruktor, der es uns erlaubt, einen Server-Hostnamen und den Server-Port anzugeben, zu dem eine Verbindung aufgebaut werden soll:

```
public Socket()
public Socket(String host, int port)
   throws UnknownHostException, IOException
public Socket(InetAddress address, int port)
   throws IOException
public Socket(String host, int port,
   InetAddress localAddr, int localPort)
   throws IOException
public Socket(InetAddress address, int port,
   InetAddress localAddr, int localPort)
   throws IOException
```

In der Regel wird man den Konstruktor einsetzen, der Zielhost und Zielport als Parameter entgegennimmt. Es ist allerdings auch möglich, wirklich alle Details der Verbindungsanforderung festzulegen – und zwar bis zum lokalen Port, der geöffnet werden soll.

Nach dem Öffnen des `Sockets` findet die weitere Kommunikation über `Streams` statt – mit Hilfe der Methoden `getOutputStream()` und `getInputStream()` erhalten wir Referenzen auf die `Output-` und `InputStreams`, mit deren Hilfe die Kommunikation abgewickelt werden soll:

```
public InputStream getInputStream()
   throws IOException

public OutputStream getOutputStream()
   throws IOException
```

Aufgrund des Wesens von Netzwerk-Verbindungen und weil die Kommunikation in der Regel textuell stattfindet, bietet es sich an, die Byte-Streams in `Reader-` und `Writer` zu überführen und dann mit `BufferedReader-` und `BufferedWriter`-Instanzen zu arbeiten.

Bevor wir allerdings wirklich anfangen, mit einem Server zu kommunizieren, sollten wir festlegen, dass nach einem bestimmten Zeitraum automatisch ein Verbindungsabbruch stattfindet, wenn keine weitere Kommunikation stattgefunden hat. Wir können diesen Zeitraum mit Hilfe der Methode `setSoTimeout()` festlegen, der wir den Zeitraum in Millisekunden übergeben:

```
public void setSoTimeout(int timeout)
   throws SocketException
```

Wenn kein Timeout gesetzt wird, geht die `Socket`-Instanz davon aus, dass einen unendlichen Zeitraum auf das Eintreffen neuer Daten gewartet werden darf. Jede Zeitangabe größer als 0 bricht das Warten auf neue Daten nach dem angegebenen Zeitraum ab. Es empfiehlt sich, immer ein Timeout zu setzen.

Das eigentliche Abrufen von Daten sollte bei Client-Sockets immer mit Hilfe von `BufferedReader`-Instanzen erfolgen, denn diese bieten die sehr bequem zu bedienenden Methoden `readLine()`. Ein weiterer Grund für diese Empfehlung: Die meisten Protokolle arbeiten zeilenbasiert.

Achten Sie beim Einsatz von Sockets immer ganz besonders darauf, dass Sie am Ende der Kommunikation sowohl die `Reader` als auch die `Writer` schließen. Danach sollten Sie den `Socket` ebenfalls explizit schließen:

```
public void close()
   throws IOException
```

Generell gilt folgendes Schema bei der Kommunikation mit einem Server:

1. Öffnen eines Sockets zum Server
2. Erzeugen von `BufferedReader`- und `BufferedWriter`-Instanzen unter Verwendung der `Input`- und `OutputStreams` des Sockets
3. Kommunikation mit dem Server
4. Schließen der `Writer`- und `Reader`-Instanzen
5. Schließen des Sockets

Serverseitige Kommunikation

Die Kommunikation auf Seiten des Servers findet ebenfalls über Sockets statt. Allerdings wird dabei nicht die Klasse `Socket`, sondern deren Gegenstück `ServerSocket` erzeugt. Der `ServerSocket`-Klasse können wir beim Erzeugen angeben, auf welchem Port sie auf Verbindungsaufnahmen von Clients lauschen soll:

```
public ServerSocket()
   throws IOException
public ServerSocket(int port)
   throws IOException
public ServerSocket(int port, int backlog)
   throws IOException
public ServerSocket(
   int port, int backlog, InetAddress bindAddr)
   throws IOException
```

Der zweite mögliche Parameter beim Erzeugen einer `ServerSocket`-Instanz – `backlog` – gibt an, wie viele gleichzeitige Verbindungsanforderungen (nicht offene Verbindungen – dies ist ein häufiger Denkfehler) erlaubt sind.

In der Regel lauscht ein Server auf allen IP-Adressen auf Verbindungsanfragen – mit Hilfe des dritten Parameters können Sie angeben, an welche IP-Adresse der Server gebunden wird, was dazu führt, dass Anforderungen, die über andere IP-Adressen gestellt werden, nicht mehr behandelt werden.

Nach dem Erzeugen muss der Server auf Verbindungsanforderungen warten. Dies geschieht mit Hilfe der Methode `accept()` der `ServerSocket`-Instanz. Diese Methode arbeitet blockierend – sie wartet so lange, bis eine Verbindungs-Anforderung kommt oder das per `setSoTimeout()` eingestellte Timeout eintritt:

```
public Socket accept()
   throws IOException

public void setSoTimeout(int timeout)
   throws SocketException
```

Die Rückgabe von `accept()` ist eine `Socket`-Implementierung, über die die weitere Kommunikation mit dem Client abgewickelt werden kann. Diese `Socket`-Implementierung verhält sich wie ein Client-Socket – wir müssen also `BufferedReader`- und `BufferedWriter`-Instanzen erzeugen, über die wir mit dem Client kommunizieren können.

Am Ende der Kommunikation sollten Sie die verwendeten Streams und Sockets explizit schließen, da sonst unnötig Ressourcen verbraucht und System-Sockets verbraucht werden.

Ein Beispiel für die Kommunikation zwischen Server und Client

Lassen Sie uns hier ein kleines Beispiel für die Kommunikation zwischen Server und Client ansehen. Wir wollen einen Server schreiben, der jedem Client die Anzahl der bisherigen Verbindungsaufnahmen, den Zeitpunkt des Starts des Servers und die aktuelle Uhrzeit übergibt. Der Server soll auf Port 2222 lauschen – einfach, weil der Autor das als eine schöne Zahl empfindet.

Mit folgendem Code können wir einen derartigen Server umsetzen:

Listing 13.4
Die Klasse NetworkTimeServer fungiert als ein Server, zu dem wir connecten können

```
import java.util.Date;
import java.net.ServerSocket;
import java.net.Socket;
import java.io.IOException;
import java.io.BufferedWriter;
import java.io.OutputStreamWriter;

/**
 * A server implemented in Java
 */
public class NetworkTimeServer extends Thread {

   private Date started;
   private int connects = 0;
   private ServerSocket server;

   /**
    * Constructor
    */
   NetworkTimeServer() {
      super();
      try {
         server = new ServerSocket(2222);
         System.out.println("NetworkTimeServer startet...");

         started = new Date();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
```

Listing 13.4 *(Forts.)*
Die Klasse NetworkTimeServer fungiert als ein Server, zu dem wir connecten können

```
    /**
     * Main method of the server
     */
    public void run() {
       if(server == null) {
          return;
       }

       while(server != null) {
          try {
             Socket client = server.accept();

             connects++;
             BufferedWriter out = new BufferedWriter(
                new OutputStreamWriter(
                   client.getOutputStream()));

             out.write("Hello!\n");
             out.write("Server started at " + started + "\n");
             out.write("You are client #" + connects + "\n");
             out.write("Time now is " + new Date());

             out.flush();

             out.close();
             client.close();
          } catch (IOException e) {
             e.printStackTrace();
          }
       }
    }

    public static void main(String[] args) {
       NetworkTimeServer nts = new NetworkTimeServer();
       nts.start();
    }
}
```

Die Klasse `NetworkTimeServer` ist als `Thread`-Ableitung implementiert. In ihrem Konstruktor erzeugt sie einen neuen `ServerSocket`, der auf Port 2222 lauscht:

```
server = new ServerSocket(2222);
```

Anschließend erfasst sie die aktuelle Zeit in der lokalen Variablen `started`. Die Methode `run()` führt den Server aus. Sie wird eingebunden, sobald die Methode `start()` der `Thread`-Klasse aufgerufen worden ist.

Innerhalb der Methode `run()` wird eine `while`-Schleife ausgeführt, die so lange läuft, wie die `ServerSocket`-Instanz `server` ungleich `null` ist. Im Körper der `while`-Schleife warten wir auf die Verbindungs-Aufnahme durch Clients. Dieses Warten geschieht mit Hilfe der Methode `accept()` der `ServerSocket`-Instanz, die uns eine `Socket`-Instanz zurückgibt, wenn eine Kontakt-Aufnahme erfolgt ist:

```
Socket client = server.accept();
```

Sobald ein Client eine Verbindung hergestellt hat, wird die Methode `accept()` verlassen und der weitere Inhalt der `while`-Schleife kann ausgeführt werden. Hier erhöhen wir zunächst einen internen Zähler, der die Anzahl der Verbindungsaufnahmen repräsentiert. Anschließend können wir mit Hilfe eines `OutputStreamWriters` eine `BufferedWriter`-Instanz erzeugen, die den `OutputStream` der Client-`Socket`-Instanz kapselt:

```
BufferedWriter out = new BufferedWriter(
   new OutputStreamWriter(
      client.getOutputStream()));
```

Jetzt sind wir in der Lage, dem Client eine Nachricht zu übermitteln. Dies geschieht mit Hilfe der `write()`-Methode der `BufferedWriter`-Instanz, deren Puffer anschließend per `flush()` geleert wird. Zuletzt schließen wir sowohl den Writer als auch den Socket zum Client und können auf eine weitere Verbindungs-Aufnahme warten:

```
out.close();
client.close();
```

In der statischen Methode `main()` wird der `NetworkTimeServer` gestartet und der oben geschilderte Prozess kann ablaufen.

Nun müssen wir nur noch einen Client schreiben, der mit dem Server Verbindung aufnimmt und dessen Ausgabe abruft:

Listing 13.5
Der NetworkTimeClient verbindet sich mit dem Server auf Port 2222

```
import java.net.Socket;
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;

/**
 * Connects to a server on the network
 */
public class NetworkTimeClient {

   private final static int SERVER_PORT = 2222;

   public static void main(String[] args) {
      if(args == null || args.length == 0) {
         System.out.println(
            "Usage: NetworkTimeClient <Server-name>");
         return;
      }

      String host = args[0];
      try {
         Socket client = new Socket(host, SERVER_PORT);

         BufferedReader reader = new BufferedReader(
            new InputStreamReader(client.getInputStream()));

         String line;
         while((line = reader.readLine()) != null) {
            System.out.println(line);
         }
```

```
            reader.close();
            client.close();
         } catch (IOException e) {
            e.printStackTrace();
         }
      }
}
```

Listing 13.5 *(Forts.)*
Der NetworkTimeClient verbindet sich mit dem Server auf Port 2222

Nach dem Aufruf der Klasse `NetworkTimeClient`, der als Parameter der Hostname oder die IP-Adresse des Servers übergeben werden muss, erzeugt diese eine neue `Socket`-Instanz, die auf den Server verweist:

```
Socket client = new Socket(host, SERVER_PORT);
```

Ein `BufferedReader`, der mit Hilfe eines `InputStreamReaders` den Eingabe-Strom des Sockets kapselt, wird verwendet, um die vom Server zurückgegebenen Daten zu empfangen:

```
BufferedReader reader = new BufferedReader(
   new InputStreamReader(client.getInputStream()));
```

Solange der Server Daten zurückgibt, werden diese eingelesen und über `System.out` wieder ausgegeben. Dies geschieht so lange, bis die Rückgabe des Servers `null` ist und dieser also offensichtlich die Verbindung getrennt hat:

```
String line;
while((line = reader.readLine()) != null) {
   System.out.println(line);
}
```

Zuletzt schließen wir den Reader und auch die Verbindung zum Server, indem wir die Methode `close()` der beiden Instanzen aufrufen:

```
reader.close();
client.close();
```

Damit ist die Kommunikation zwischen Client und Server beendet. Wenn wir den Server gestartet haben, können wir Client und Server miteinander kommunizieren lassen:

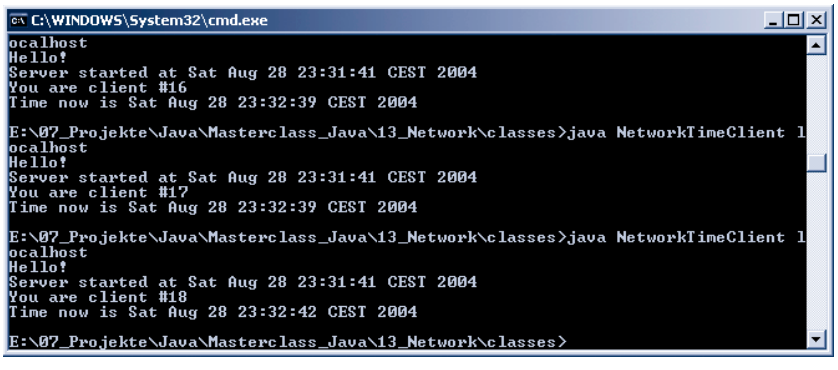


Abbildung 13.6
Kommunikation des Clients mit dem Server

13.5 E-Mails senden

Etwas, das Sie sicherlich schmerzlich bei Java vermissen, ist die Fähigkeit, E-Mails zu versenden. Sie haben – um Mails verschicken zu können – genau zwei Möglichkeiten:

- Schreiben Sie eine Kommunikations-Klasse, die die Mails über das SMTP-Protokoll verschickt – die Basics dafür haben Sie in diesem Kapitel gelernt
- Nutzen Sie `Java Mail`, eine Lösung für das Versenden und Empfangen von E-Mails

Der Autor hat selbst schon einmal eine Lösung implementiert, die E-Mails mit Hilfe von Sockets versendet. Dies ist mit Hilfe des `SMTP`-Protokolls (und ein wenig Geduld) recht einfach umzusetzen, sprengt aber gewiss den Rahmen dieses Buches. Konzentrieren wir uns deshalb auf den zweiten Ansatz und setzen `Java Mail` ein.

13.5.1 Was ist Java Mail?

`Java Mail` ist ein komplettes Framework für das Senden und Empfangen von E-Mails. Es unterstützt verschiedene Protokolle (`SMTP` für den Versand sowie `POP 3` und `IMAP` für den Empfang und das Verarbeiten von Nachrichten) und kann bei Bedarf um eigene Protokolle erweitert werden. `Java Mail` unterstützt verschiedene Nachrichten-Typen und kann sehr flexibel konfiguriert werden. Eigentlich wäre es Verschwendung, `Java Mail` nur zum Versand von einfachen E-Mails einzusetzen – aber auch das erledigt das Framework hervorragend.

13.5.2 Installation von Java Mail

Unter der Adresse *http://java.sun.com/products/javamail/* können Sie die derzeit aktuellste Version von `Java Mail` kostenlos herunterladen. Zum Zeitpunkt der Drucklegung dieses Buches ist dies die Version 1.3.1 – die Version 1.3.2 ist aber bereits in Arbeit.

Ebenfalls benötigen Sie das `JavaBeans Activation Framework`. Die derzeit aktuelle Version 1.0.2 können Sie ebenfalls kostenlos herunterladen. Die Adresse für den Download lautet *http://java.sun.com/products/javabeans/glasgow/jaf.html*.

Nach dem Herunterladen sollten Sie aus beiden Archiven die `JAR`-Dateien entpacken und in ein Verzeichnis kopieren, das Sie problemlos aus Ihren Applikationen heraus erreichen können. Alternativ entpacken Sie die `JAR`-Archive in Ihr Applikations-Verzeichnis. Damit ist die Installation abgeschlossen und wir können E-Mails mit Hilfe von `Java Mail` verschicken.

13.5.3 Versenden von E-Mails

Das Versenden von E-Mails via `Java Mail` gestaltet sich recht simpel – allerdings muss man tatsächlich erst ein Gefühl für den Umgang mit `Java Mail` gewinnen.

Im ersten Schritt sollten wir immer eine `Properties`-Instanz erzeugen, der wir den Namen des zu verwendenden Mail-Servers in der Property `mail.smtp.host` übergeben. Diese `Properties`-Instanz kann dann der Methode `getDefaultSession()` der `javax.mail.Session`-Klasse als Parameter übergeben werden:

```
public static Session getDefaultInstance(
   java.util.Properties props)
public static Session getDefaultInstance(
   java.util.Properties props, Authenticator authenticator)
```

Die so erhaltene `Session`-Instanz kann danach verwendet werden, um eine Instanz der `MimeMessage`-Klasse zu erzeugen, die die zu versendende Nachricht repräsentiert:

```
public MimeMessage(Session session)
```

Die `MimeMessage`-Klasse bietet eine Menge an Methoden an, um die einzelnen Felder einer Nachricht zu setzen. Für uns interessant sollten folgende Felder sein:

- Absender
- Empfänger
- Betreff
- Nachricht

Diese Informationen können wir mit Hilfe der Methoden `setFrom()`, `setRecipient()`, `setSubject()` und `setText()` setzen:

```
public void setSender(Address address)
   throws MessagingException
public void setRecipient(
   Message.RecipientType type, Address address)
   throws MessagingException
public void setSubject(String subject)
   throws MessagingException
public void setText(String text)
   throws MessagingException
```

Neben diesen Methoden bietet die `Message`-Klasse eine Vielzahl weiterer Methoden, um Inhalt und Header-Felder zu setzen. So können Sie unter anderem auch recht einfach Datei-Anhänge definieren oder HTML-E-Mails versenden. Dies alles sprengt allerdings den Rahmen dieses Kapitels.

Die Methode `setRecipient()` erwartet als ersten Parameter die Angabe, was für einen Typ von Empfänger wir zuweisen wollen. Zur Auswahl stehen die Werte dieser in der Klasse `Message.RecipientType` definierten Konstanten:

- `RecipientType.TO`: Direkte Empfänger, werden im „An“-Feld der empfangenen E-Mail angezeigt.
- `RecipientType.CC`: Kopie-Empfänger. Werden im Feld „CC“ der empfangenen E-Mail angezeigt.
- `RecipientType.BCC`: Blindkopie-Empfänger. Tauchen in der empfangenen Mail nicht mehr auf.

Die beiden Methoden `setSender()` und `setRecipient()` erwarten weiterhin eine `Address`-Instanz, die Sender und Empfänger repräsentiert. Die `Address`-Klasse selbst ist allerdings abstrakt – jedoch trifft das nicht für die Klasse `InternetAddress` zu, die von ihr erbt. Diese deklariert einen Konstruktor, dem wir eine E-Mail-Adresse als Parameter übergeben können:

```
public InternetAddress(String address)
   throws AddressException
public InternetAddress(String address, boolean strict)
   throws AddressException
public InternetAddress(String address, String personal)
   throws java.io.UnsupportedEncodingException
```

Der Parameter `address` repräsentiert stets eine E-Mail-Adresse. In der Standard-Einstellung wird diese auf syntaktische Gültigkeit geprüft, was aber nicht immer sinnvoll ist und deshalb durch Übergabe des Wertes `false` für den booleschen Parameter `strict` abgeschaltet werden kann. Wenn Sie möchten, können Sie durch Übergabe eines Anzeigenamens auch dafür sorgen, dass das E-Mail-Programm des Empfängers einen Namen statt der nackten E-Mail-Adresse anzeigt.

Nachdem man so die Nachricht erstellt hat, wird sie mit Hilfe der statischen Methode `send()` der `Transport`-Klasse verschickt:

```
public static void send(Message msg)
   throws MessagingException
```

Das klingt aufwändig? Dieser Aussage kann man zustimmen – aber andererseits macht die weit reichende Strukturierung das `Java Mail`-Framework sehr flexibel und leistungsfähig.

Sehen wir uns anhand einer kleinen Kommandozeilen-Applikation an, wie wir eine E-Mail mit Hilfe des `Java Mail`-Frameworks versenden können:

Listing 13.6
Versand einer E-Mail mit Hilfe von Java Mail

```
import javax.mail.*;
import javax.mail.internet.MimeMessage;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.AddressException;
import java.util.Properties;

/**
 * Sends a mail using the Java Mail-Framework
 */
public class SendAMail {

   public static void main(String[] args) {

      if(null == args || args.length < 5) {
         System.out.println("Usage: SendAMail <Server> <From> " +
            "<To> <Subject> <Message>");
         return;
      }

      String server = args[0];
      String from = args[1];
      String to = args[2];
      String subject = args[3];
```

Listing 13.6 *(Forts.)*
Versand einer E-Mail mit Hilfe von Java Mail

```
        String body = args[4];

        Properties props = new Properties();
        props.setProperty("mail.smtp.host", server);

        Session session = Session.getDefaultInstance(props);
        Message message = new MimeMessage(session);
        try {
           Address sender = new InternetAddress(from);
           Address recipient = new InternetAddress(to);

           message.setFrom(sender);
           message.setRecipient(
              Message.RecipientType.TO, recipient);
           message.setSubject(subject);
           message.setText(body);

           Transport.send(message);
        } catch (AddressException e) {
           e.printStackTrace();
        } catch (MessagingException e) {
           e.printStackTrace();
        }
     }
}
```

Die Klasse `SendAMail` erwartet, dass wir ihr beim Aufruf fünf Parameter übergeben:

- Zu verwendender Mail-Server
- Absender
- Empfänger
- Betreff
- Nachrichten-Text

Nach dem Einlesen der Parameter erzeugen wir eine neue `Properties`-Instanz, der wir den Parameter für den zu verwendenden Mailserver zuweisen:

```
Properties props = new Properties();
props.setProperty("mail.smtp.host", server);
```

Nun können wir eine Instanz der `Session`-Klasse erzeugen, wobei wir der statischen Methode `getDefaultMessage()` die zuvor definierte `Properties`-Instanz als Parameter übergeben:

```
Session session = Session.getDefaultInstance(props);
```

Die eigentliche E-Mail wird durch eine Instanz der `MimeMessage`-Klasse repräsentiert, deren Konstruktor wir die soeben erzeugte `Session`-Instanz als Parameter übergeben:

```
Message message = new MimeMessage(session);
```

Nach dem Erzeugen der beiden `InternetAddress`-Instanzen, die Absender und Empfänger repräsentieren, können wir diese Informationen zusammen mit dem Betreff und dem Nachrichten-Text an die `Message`-Instanz übergeben:

```
message.setFrom(sender);
message.setRecipient(Message.RecipientType.TO, recipient);
message.setSubject(subject);
message.setText(body);
```

Zuletzt kann die E-Mail versendet werden:

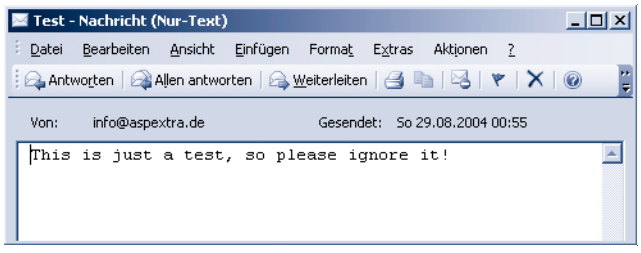


Abbildung 13.7
Diese Nachricht wurde per Java Mail verschickt

13.6 Fazit

Networking mit Java ist nicht schwierig! Egal welche Aufgabenstellung – ob Informationen zu einem Host, Abrufen von Daten per URL oder Kommunikation zwischen Client und Server –, Java bietet uns die Klassen und Methoden, mit denen wir dies lösen können. Bemerkenswert einfach funktioniert vor allem die bei anderen Sprachen sehr komplexe Kommunikation zwischen Client und Server mit Hilfe von `Sockets` und `ServerSockets`.

Das `Java Mail`-Framework, auf das wir ebenfalls im Rahmen dieses Kapitels einen Blick geworfen haben, bietet uns Ansätze, die über das reine Versenden von E-Mails hinausgehen. So können wir mit Hilfe von Java Mail beispielsweise sogar eigene E-Mail-Programme schreiben – das Framework gibt es jedenfalls her.

Dennoch sollten Sie nicht zu euphorisch werden: Auch Java kann nicht verhindern, dass die Arbeit auf verteilten Systemen und im Netzwerk tückisch sein kann – denken wir bei der Gelegenheit etwa an Firewalls, die die Kommunikation behindern (auch wenn dies mit dem wichtigen Hintergedanken der Sicherheit geschieht) oder an Verbindungsabbrüche aufgrund von Netzwerk- oder Server-Überlastungen.

Was Java aber kann: Die Entwicklung von netzwerkfähigen Applikationen extrem vereinfachen – hier spielt die Sprache ihre Stärken voll aus!

A

Lizenzvereinbarung

Sun Microsystems, Inc. Binary Code License Agreement for the JAVA 2 PLATFORM STANDARD EDITION DEVELOPMENT KIT 5.0

SUN MICROSYSTEMS, INC. (”SUN”) IS WILLING TO LICENSE THE SOFTWARE IDENTIFIED BELOW TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS BINARY CODE LICENSE AGREEMENT AND SUPPLEMENTAL LICENSE TERMS (COLLECTIVELY ”AGREEMENT”). PLEASE READ THE AGREEMENT CAREFULLY. BY DOWNLOADING OR INSTALLING THIS SOFTWARE, YOU ACCEPT THE TERMS OF THE AGREEMENT. INDICATE ACCEPTANCE BY SELECTING THE ”ACCEPT” BUTTON AT THE BOTTOM OF THE AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND BY ALL THE TERMS, SELECT THE ”DECLINE” BUTTON AT THE BOTTOM OF THE AGREEMENT AND THE DOWNLOAD OR INSTALL PROCESS WILL NOT CONTINUE.

1. DEFINITIONS. ”Software” means the identified above in binary form, any other machine readable materials (including, but not limited to, libraries, source files, header files, and data files), any updates or error corrections provided by Sun, and any user manuals, programming guides and other documentation provided to you by Sun under this Agreement. ”Programs” mean Java applets and applications intended to run on the Java 2 Platform Standard Edition (J2SE platform) platform on Java-enabled general purpose desktop computers and servers.

2. LICENSE TO USE. Subject to the terms and conditions of this Agreement, including, but not limited to the Java Technology Restrictions of the Supplemental License Terms, Sun grants you a non-exclusive, non-transferable, limited license without license fees to reproduce and use internally Software complete and unmodified for the sole

purpose of running Programs. Additional licenses for developers and/or publishers are granted in the Supplemental License Terms.

3. RESTRICTIONS. Software is confidential and copyrighted. Title to Software and all associated intellectual property rights is retained by Sun and/or its licensors. Unless enforcement is prohibited by applicable law, you may not modify, decompile, or reverse engineer Software. You acknowledge that Licensed Software is not designed or intended for use in the design, construction, operation or maintenance of any nuclear facility. Sun Microsystems, Inc. disclaims any express or implied warranty of fitness for such uses. No right, title or interest in or to any trademark, service mark, logo or trade name of Sun or its licensors is granted under this Agreement. Additional restrictions for developers and/or publishers licenses are set forth in the Supplemental License Terms.

4. LIMITED WARRANTY. Sun warrants to you that for a period of ninety (90) days from the date of purchase, as evidenced by a copy of the receipt, the media on which Software is furnished (if any) will be free of defects in materials and workmanship under normal use. Except for the foregoing, Software is provided ”AS IS”. Your exclusive remedy and Sun’s entire liability under this limited warranty will be at Sun’s option to replace Software media or refund the fee paid for Software. Any implied warranties on the Software are limited to 90 days. Some states do not allow limitations on duration of an implied warranty, so the above may not apply to you. This limited warranty gives you specific legal rights. You may have others, which vary from state to state.

5. DISCLAIMER OF WARRANTY. UNLESS SPECIFIED IN THIS AGREEMENT, ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT THESE DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

6. LIMITATION OF LIABILITY. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH

DAMAGES. In no event will Sun's liability to you, whether in contract, tort (including negligence), or otherwise, exceed the amount paid by you for Software under this Agreement. The foregoing limitations will apply even if the above stated warranty fails of its essential purpose. Some states do not allow the exclusion of incidental or consequential damages, so some of the terms above may not be applicable to you.

7. TERMINATION. This Agreement is effective until terminated. You may terminate this Agreement at any time by destroying all copies of Software. This Agreement will terminate immediately without notice from Sun if you fail to comply with any provision of this Agreement. Either party may terminate this Agreement immediately should any Software become, or in either party's opinion be likely to become, the subject of a claim of infringement of any intellectual property right. Upon Termination, you must destroy all copies of Software.

8. EXPORT REGULATIONS. All Software and technical data delivered under this Agreement are subject to US export control laws and may be subject to export or import regulations in other countries. You agree to comply strictly with all such laws and regulations and acknowledge that you have the responsibility to obtain such licenses to export, re-export, or import as may be required after delivery to you.

9. TRADEMARKS AND LOGOS. You acknowledge and agree as between you and Sun that Sun owns the SUN, SOLARIS, JAVA, JINI, FORTE, and iPLANET trademarks and all SUN, SOLARIS, JAVA, JINI, FORTE, and iPLANET-related trademarks, service marks, logos and other brand designations ("Sun Marks"), and you agree to comply with the Sun Trademark and Logo Usage Requirements currently located at http://www.sun.com/policies/trademarks. Any use you make of the Sun Marks inures to Sun's benefit.

10. U.S. GOVERNMENT RESTRICTED RIGHTS. If Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in Software and accompanying documentation will be only as set forth in this Agreement; this is in accordance with 48 CFR 227.7201 through 227.7202-4 (for Department of Defense (DOD) acquisitions) and with 48 CFR 2.101 and 12.212 (for non-DOD acquisitions).

11. GOVERNING LAW. Any action related to this Agreement will be governed by California law and controlling U.S. federal law. No choice of law rules of any jurisdiction will apply.

12. SEVERABILITY. If any provision of this Agreement is held to be unenforceable, this Agreement will remain in effect with the provision omitted, unless omission would frustrate the intent of the parties, in which case this Agreement will immediately terminate.

13. INTEGRATION. This Agreement is the entire agreement between you and Sun relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification of this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

SUPPLEMENTAL LICENSE TERMS

These Supplemental License Terms add to or modify the terms of the Binary Code License Agreement. Capitalized terms not defined in these Supplemental Terms shall have the same meanings ascribed to them in the Binary Code License Agreement . These Supplemental Terms shall supersede any inconsistent or conflicting terms in the Binary Code License Agreement, or in any license contained within the Software.

A. Software Internal Use and Development License Grant. Subject to the terms and conditions of this Agreement and restrictions and exceptions set forth in the Software "README" file, including, but not limited to the Java Technology Restrictions of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license without fees to reproduce internally and use internally the Software complete and unmodified for the purpose of designing, developing, and testing your Programs.

B. License to Distribute Software. Subject to the terms and conditions of this Agreement and restrictions and exceptions set forth in the Software README file, including, but not limited to the Java Technology Restrictions of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license without fees to reproduce and distribute the Software, provided

that (i) you distribute the Software complete and unmodified and only bundled as part of, and for the sole purpose of running, your Programs, (ii) the Programs add significant and primary functionality to the Software, (iii) you do not distribute additional software intended to replace any component(s) of the Software, (iv) you do not remove or alter any proprietary legends or notices contained in the Software, (v) you only distribute the Software subject to a license agreement that protects Sun's interests consistent with the terms contained in this Agreement, and (vi) you agree to defend and indemnify Sun and its licensors from and against any damages, costs, liabilities, settlement amounts and/or expenses (including attorneys' fees) incurred in connection with any claim, lawsuit or action by any third party that arises or results from the use or distribution of any and all Programs and/or Software.

C. License to Distribute Redistributables. Subject to the terms and conditions of this Agreement and restrictions and exceptions set forth in the Software README file, including but not limited to the Java Technology Restrictions of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license without fees to reproduce and distribute those files specifically identified as redistributable in the Software "README" file ("Redistributables") provided that: (i) you distribute the Redistributables complete and unmodified, and only bundled as part of Programs, (ii) the Programs add significant and primary functionality to the Redistributables, (iii) you do not distribute additional software intended to supersede any component(s) of the Redistributables (unless otherwise specified in the applicable README file), (iv) you do not remove or alter any proprietary legends or notices contained in or on the Redistributables, (v) you only distribute the Redistributables pursuant to a license agreement that protects Sun's interests consistent with the terms contained in the Agreement, (vi) you agree to defend and indemnify Sun and its licensors from and against any damages, costs, liabilities, settlement amounts and/or expenses (including attorneys' fees) incurred in connection with any claim, lawsuit or action by any third party that arises or results from the use or distribution of any and all Programs and/or Software.

D. Java Technology Restrictions. You may not create, modify, or change the behavior of, or authorize your licensees to create, modify, or change the behavior of,

classes, interfaces, or subpackages that are in any way identified as "java", "javax", "sun" or similar convention as specified by Sun in any naming convention designation.

E. Distribution by Publishers. This section pertains to your distribution of the Software with your printed book or magazine (as those terms are commonly used in the industry) relating to Java technology ("Publication"). Subject to and conditioned upon your compliance with the restrictions and obligations contained in the Agreement, in addition to the license granted in Paragraph 1 above, Sun hereby grants to you a non-exclusive, nontransferable limited right to reproduce complete and unmodified copies of the Software on electronic media (the "Media") for the sole purpose of inclusion and distribution with your Publication(s), subject to the following terms: (i) You may not distribute the Software on a stand-alone basis; it must be distributed with your Publication(s); (ii) You are responsible for downloading the Software from the applicable Sun web site; (iii) You must refer to the Software as JavaTM 2 Platform Standard Edition Development Kit 5.0; (iv) The Software must be reproduced in its entirety and without any modification whatsoever (including, without limitation, the Binary Code License and Supplemental License Terms accompanying the Software and proprietary rights notices contained in the Software); (v) The Media label shall include the following information: Copyright 2004, Sun Microsystems, Inc. All rights reserved. Use is subject to license terms. Sun, Sun Microsystems, the Sun logo, Solaris, Java, the Java Coffee Cup logo, J2SE , and all trademarks and logos based on Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. This information must be placed on the Media label in such a manner as to only apply to the Sun Software; (vi) You must clearly identify the Software as Sun's product on the Media holder or Media label, and you may not state or imply that Sun is responsible for any third-party software contained on the Media; (vii) You may not include any third party software on the Media which is intended to be a replacement or substitute for the Software; (viii) You shall indemnify Sun for all damages arising from your failure to comply with the requirements of this Agreement. In addition, you shall defend, at your expense, any and all claims brought against Sun by third parties, and shall pay all damages awarded by a court of competent jurisdiction, or such settlement amount negotiated by you, arising out of or in connection with your use, reproduction or distribution of the Software and/or the Publication.

Your obligation to provide indemnification under this section shall arise provided that Sun: (i) provides you prompt notice of the claim; (ii) gives you sole control of the defense and settlement of the claim; (iii) provides you, at your expense, with all available information, assistance and authority to defend; and (iv) has not compromised or settled such claim without your prior written consent; and (ix) You shall provide Sun with a written notice for each Publication; such notice shall include the following information: (1) title of Publication, (2) author(s), (3) date of Publication, and (4) ISBN or ISSN numbers. Such notice shall be sent to Sun Microsystems, Inc., 4150 Network Circle, M/S USCA12-110, Santa Clara, California 95054, U.S.A , Attention: Contracts Administration.

F. Source Code. Software may contain source code that, unless expressly licensed for other purposes, is provided solely for reference purposes pursuant to the terms of this Agreement. Source code may not be redistributed unless expressly provided for in this Agreement.

G. Third Party Code. Additional copyright notices and license terms applicable to portions of the Software are set forth in the THIRDPARTYLICENSEREADME.txt file. In addition to any terms and conditions of any third party opensource/freeware license identified in the THIRDPARTYLICENSEREADME.txt file, the disclaimer of warranty and limitation of liability provisions in paragraphs 5 and 6 of the Binary Code License Agreement shall apply to all Software in this distribution.

For inquiries please contact: Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. (LFI#141623/Form ID#011801)

Stichwortverzeichnis

Symbole

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Copyright

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf der Website ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.

Hinweis

Dieses und andere eBooks können Sie rund um die Uhr und legal auf unserer Website

(http://www.informit.de)

herunterladen