Skript zum

JAVA-Kurs

Alfons Brandl, Matthias Göbel, Clemens Harlfinger, David von Oheimb

18. Mai 1999

Inhaltsverzeichnis

1 Einleitung

Generell kann man sich unter einem *Programm* eine Folge von Anweisungen vorstellen, die der Erledigung bzw. Lösung einer Aufgabe dienen. Eine derartige Anweisungsfolge entspricht damit einem Lösungsverfahren. Man spricht auch von einem *Algorithmus*, sofern das Verfahren garantiert in endlicher Zeit ein Ergebnis liefert bzw. *terminiert*.
Programmieren entspricht somit der Umsetzung eines Lösungsverfahrens für ein bestimmtes Problem in eine Anweisungsfolge. Die eigentliche Lösung der Aufgabe erfolgt dann mit der *Programmausführung*.

1.1 Programmiersprachen

Normalerweise geht man bei der Erstellung von Programmen von einem natürlichsprachlich beschriebenen Lösungsverfahren aus und versucht, eine Formulierung des Verfahrens (also ein Programm) zu erstellen, die eine Ausführung des Verfahrens durch einen Rechner ermöglicht.
Eine *Programmiersprache* stellt eine einheitliche Notation zur Formulierung von Programmen dar. Der *Programmtext* ist dann die (schriftliche) Darstellung eines Programms in der jeweils gewählten Programmiersprache.

Programmiersprachen unterscheiden sich vor allem bzgl. ihres *Abstraktionsgrads*. Der Abstraktionsgrad ist ein Maß dafür, inwieweit bei der Programmierung Eigenschaften des ausführenden Rechners berücksichtigt werden müssen. Gering abstrahierende Sprachen werden daher als *maschinennah* bezeichnet. Sprachen mit höherem Abstraktionsgrad heißen dagegen *Hochsprachen*.

1.1.1 Maschinennahe Programmiersprachen

Moderne Rechenprozessoren (CPUs) stellen dem Programmierer eine Reihe sehr schnell ausführbarer, dafür aber wenig ausdrucksstarker Basisoperationen zur Verfügung (z.B. Sprunganweisungen, Speicherzugriffe, einfache Rechenoperationen). Die Menge aller auf einem Prozessortyp verfügbaren Operationen wird auch als *Prozessorbefehlssatz* bezeichnet.
Die unmittelbare Programmierung mit Prozessorbefehlen erfolgt mit Hilfe maschinennaher Sprachen, den sog. *Assembler-Sprachen*, deren Programmanweisungen direkt auf entsprechende Prozessorbefehle abgebildet werden können. Ein Beispiel für eine solche Assembler-Sprache ist der an der TU München entwickelte MI-Assembler.

Da eine Assembler-Sprache auf einen speziellen Prozessorbefehlssatz zugeschnitten ist, sind Assembler-Programme in der Regel nicht *portierbar*, d.h. eine Programmausführung auf anderen Prozessortypen ist nicht möglich. Zudem ist die Erstellung größerer Programmsysteme auf Assembler-Ebene mühsam und fehleranfällig: Jede abstrakte Programmanweisung muß manuell in eine konkrete Folge von Prozessorbefehlen umgesetzt werden.

1.1.2 Hochsprachen

Mit Hilfe von Hochsprachen, zu denen auch *Java* zählt, versucht man, das Abstraktionsgefälle zwischen Lösungsansatz und dem sich daraus ergebenden Programm zu verringern, so daß die Umsetzung des Lösungsansatzes im Programmtext möglichst gut nachvollziehbar ist. Hochsprachlich formulierte Programme entsprechen normalerweise eher natürlichsprachlichen Anweisungen als entsprechende maschinennahe Programme.

Beispiel 1:

Umsetzung einer umgangssprachlich formulierten Programmanweisung

```
Wenn x kleiner ist als y, dann ändere x so,
daß x danach um 5 größer ist als y.
```

Formulierung in einer Hochsprache (Java):

```
if (x < y) { x = y + 5; }
```

Formulierung in Assembler (MI-Assembler):

```
        MOVE W x, R0
        MOVE W y, R1
        SUB  W R0, R1, R2
        JLE  ende
        ADD  W I 5, R1
        MOVE W R1, x
ende: ...
```

Hochsprachen basieren auf abstrakten, vom Prozessorbefehlssatz unabhängigen *Programmierkonzepten* (z.B. strukturierte Datentypen, Funktionsaufrufe, Schleifen; siehe dazu die folgenden Kapitel). Diese Konzepte werden in konkreten *Sprachkonstrukten* umgesetzt. Ein Programmierer kann dann ein Programm ausschließlich auf Basis abstrakter Konzepte entwickeln und mit den entsprechenden Konstrukten hochsprachlich formulieren.

1.2 Übersetzung von Programmen

1.2.1 Übersetzungsvorgang

Vor der Ausführung eines Programms auf einem Prozessor muß bei Hochsprachen ebenso wie bei Assembler-Sprachen eine Abbildung der Programmanweisungen auf den Prozessorbefehlssatz stattfinden. Dazu wird eine Abbildungsvorschrift angegeben, mit Regeln, die beschreiben, welche Prozessorbefehle nötig sind, um eine Anweisung der Sprache auszuführen. Die Komplexität der Abbildungsvorschrift wächst dabei mit dem Abstraktionsgrad der Sprache. Eine einzelne Programmanweisung kann u.U. in eine Vielzahl von Prozessorbefehlen umgesetzt werden.
Die Abbildung anhand einer solchen Abbildungsvorschrift wird als *Übersetzung* bezeichnet. Eine Übersetzung kann auch in mehreren Stufen stattfinden. In diesem Fall wird der Programmtext bzw. *Quelltext* zunächst in eine Zwischensprache übersetzt. Der daraus resultierende Zwischencode kann dann in einem oder mehreren zusätzlichen Übersetzungsschritten weiterverarbeitet werden, bis ausführbarer *Programmcode* erzeugt wurde.

1.2.2 Compilierung vs. Interpretierung

Prinzipiell kann die Übersetzung eines Programms auf zwei Arten geschehen, die sich durch den Zeitpunkt unterscheiden, zu dem die Übersetzung erfolgt. Wird das gesamte Programm in Prozessorbefehle übersetzt, bevor die Programmausführung beginnt, spricht man von *Compilierung*, wobei die Begriffe *Compilierung* und *Übersetzung* häufig synonym verwendet werden. Programme, die diese Form der Übersetzung durchführen, werden als *Compiler* bezeichnet. Die Compilierung ist ein statisches Verfahren, da der Programmcode zur Laufzeit bereits festgelegt ist.
Wenn Programmanweisungen erst während der Programmausführung dynamisch übersetzt werden, so spricht man von einer *Interpretierung* des Programms. Die entsprechenden Übersetzungsprogramme heißen dann *Interpreter*. Ein spezieller Fall der Interpretation ist die sog. *Emulation*, die einer Interpretierung von Prozessorbefehlen entspricht. Damit kann ausführbarer Programmcode auch auf Systemen mit anderen Prozessoren ausgeführt werden.
Anhand des Abstraktionsgrads einer Sprache läßt sich eine grobe Zuordnung treffen, welches Verfahren zur Übersetzung der Sprache benutzt werden sollte. Für Sprachen mit einem hohen Abstraktionsgrad eignet sich meist eher die Interpretierung, während sich gering abstrahierende Sprachen für die Compilierung anbieten.

Beide Verfahren besitzen eine Reihe von Vor- und Nachteilen. Die statische Compilierung erfordert z.B. eine explizite Übersetzung, bevor Änderungen am Programmtext wirksam werden können. Bei dynamisch interpretierten Programmen ist dies nicht nötig. Auf der anderen Seite ist die Ausführungsgeschwindigkeit interpretierter Programme in der Regel deutlich geringer als bei der Compilierung, da während der Programmausführung jede Anweisung jedesmal vor ihrer Ausführung übersetzt wird. Dies ist insbesondere bei Emulationen zu beachten, wo die Prozessorunabhängigkeit oft mit erheblichen Geschwindigkeitsverlusten erkauft werden muß.
Die Einsatzgebiete für Interpreter-Sprachen liegen vor allem in Bereichen, wo hohe Portabilität und einfaches Ändern von Programmen wichtig sind. Compiler-Sprachen dagegen werden bevorzugt dort eingesetzt, wo hohe Ausführungsgeschwindigkeiten benötigt werden. Bekannte Beispiele für compilierbare Sprachen sind Pascal, Modula und C/C++. Zu den Interpreter-Sprachen zählen u.a. Lisp und Gofer, aber auch Skript-Sprachen wie z.B. Perl.

Java kann sowohl als compilierbare als auch als interpretierte Sprache betrachtet werden: Die Sprachkonzeption sieht die Erzeugung eines speziellen Zwischencodes durch Compilierung (siehe Abschnitt 1.6.3) vor, der dann mit Hilfe eines Interpreters ausgeführt werden kann (siehe Abschnitt 1.6.4).

1.3 Ausführung von Programmen

1.3.1 Laufzeitsysteme

Eine entscheidende Voraussetzung für die korrekte Ausführung eines Programms ist der korrekte Umgang mit äußeren Einflüssen, die oft erst zur Laufzeit bekannt sind. Dazu zählen alle Arten von Eingabedaten, aber auch Systeminformationen wie z.B. die Größe des freien Arbeitsspeichers oder die Adresse von Ein-/Ausgabe-Kanälen. Die Kontrolle solcher (dynamischer) Einflußfaktoren erfolgt über das sog. *Laufzeitsystem*, daß zwischen Betriebssystem und (statischem) Programm angesiedelt ist.
Programmanweisungen, zu deren korrekter Ausführung Laufzeitinformationen erforderlich sind, werden so übersetzt, daß die benötigten Informationen aus dem Laufzeitsystem abgerufen werden, sobald die entsprechende Anweisung abgearbeitet wird. Das Laufzeitsystem seinerseits stellt die erforderlichen Informationen z.B. durch Zugriffe auf das Betriebssystem bereit. Aus Sicht des Programmierers ist damit auch das Betriebssystem Bestandteil des Laufzeitsystems, da er durch das Laufzeitsystem auch Systeminformationen erhält.

Laufzeitsysteme können, abhängig vom Abstraktionsgrad der Programmiersprache, für die sie konzipiert sind, sehr komplex aufgebaut sein und eine Vielzahl von Aufgaben übernehmen (z.B. Speicherverwaltung, Abarbeitung von Kommunikationsprotokollen usw.). Die Realisierungsvarianten für Laufzeitsysteme reichen dementsprechend von der Erweiterung des Programms um einige Anweisungen bis hin zu kompletten Ablaufumgebungen, in denen die Programmausführung erfolgt. Auch Interpreter können als Laufzeitsystem (oder als Teil davon) betrachtet werden.

1.3.2 Bibliotheken

Um den Umfang einer Programmiersprache überschaubar zu halten, werden üblicherweise nur wenige zentrale Konzepte (z.B. Grundoperationen zur Verarbeitung von Daten, Steuerung des Programmablaufs) über Sprachkonstrukte realisiert. Alle weiteren Operationen müssen dann als Teilprogramme mit diesen Konstrukten realisiert werden.
Viele Compiler-Sprachen erlauben die Wiederverwendung häufig benötigter Teilprogramme, der sog. *Routinen*[1], in ausführbarer Form, d.h. Routinen müssen bei der Übersetzung des restlichen Programms nicht erneut übersetzt werden. Routinen werden üblicherweise anhand ihrer Aufgaben bzw. Einsatzgebiete in Gruppen eingeteilt. Der Programmcode der einzelnen Routinen wird dann in einer sog. *Bibliothek* (engl. *library*) zusammengefaßt.

1.3.3 Binden von Programmen

Verwendet ein Programm separat erstellte Teilprogramme wie z.B. Bibliotheksroutinen, so muß bei der Programmausführung sichergestellt sein, daß vor dem Aufruf der Routine eine Verbindung zwischen dem Programm und dem entsprechenden Teilprogramm hergestellt wurde. Diese Verknüpfung eines Programms mit den von ihm benötigten Teilprogrammen Programm bezeichnet man auch als das *Binden* des Programms.
Vor der Ausführung des Programms (oft schon während der Übersetzung) werden mit einem *Binder* (engl. *linker*) alle Aufrufe von Routinen innerhalb des übersetzten Programms ermittelt und die nötigen Verbindungen aufgebaut. Dazu kann der Programmcode mit einer Kopie des Codes der Routine ergänzt werden. Die Routinenaufrufe können aber auch durch Zugriffe auf das Laufzeitsystem ersetzt werden, so daß die Routinen während der Programmausführung vom Laufzeitsystem aufgerufen werden.
Falls das Laufzeitsystem programmübergreifend arbeiten kann, ist der gleichzeitige Zugriff mehrerer Programme auf eine gemeinsame Bibliothek möglich. Eine dafür geeignete Bibliothek wird dann als *Shared Library* oder *Dynamic Link Library (DLL)* bezeichnet.

1.4 Hilfsprogramme und Werkzeuge

Um die Programmierarbeit zu erleichtern, existieren viele verschiedene Typen von Hilfsprogrammen. Solche Hilfsprogramme werden oft auch als *Programmierwerkzeuge* bzw. *Tools* bezeichnet. Wichtig sind hier

[1] Typische Routineaufgaben sind z.B. die Ein-/Ausgabe von Texten oder das Bearbeiten von Dateien.

vor allem die Werkzeuge zur Fehlerlokalisierung, zur Leistungsoptimierung und zur Programmdokumentation.
Es gibt noch eine Reihe von Werkzeugen für verschiedenste andere Bereiche, z.B. zur automatischen Korrektheitsüberprüfung oder zur Visualisierung bestimmter Programmeigenschaften. Diese werden im Rahmen dieses Kurses jedoch nicht näher behandelt.

1.4.1 Fehlerlokalisierung

Mit steigender Komplexität eines Programms wächst auch die Wahrscheinlichkeit, daß das Programm fehlerhaft ist. *Syntaktische Fehler*, also reine Schreibfehler im Programmtext, lassen sich normalerweise anhand von Fehlermeldungen bereits während der Übersetzung lokalisieren.
Schwieriger ist dagegen die Lokalisierung *semantischer Fehler*, bei deren Auftreten das Programm zwar problemlos übersetzt werden kann, aber bei der Ausführung kein korrektes Ergebnis liefert. Für diese Zwecke werden zu vielen Programmiersprachen Hilfsprogramme zur genauen Überwachung des Programmablaufs mitgeliefert, die als *Debugger* bezeichnet werden. Mit einem Debugger läßt sich z.B. der Programmzustand zu bestimmten Zeitpunkten während der Programmausführung „einfrieren“ und dann genauer analysieren.

1.4.2 Leistungsoptimierung

Neben der Korrektheit ist oft auch eine möglichst hohe Ausführungsgeschwindigkeit ein wichtiges Ziel bei der Programmentwicklung. Falls ein Programm zu langsam abläuft, kann mit Hilfe eines sog. *Profilers* nach möglichen Leistungsengpässen im Programmablauf gesucht werden. Beispielsweise läßt sich während der Programmausführung überprüfen, welche Teile des Programms besonders großen Anteil an der Gesamtlaufzeit haben. Diese Teile können dann genauer untersucht und eventuell effizienter gestaltet werden.

1.5 Programmdokumentation

Bei der Dokumentation wird unterschieden zwischen der Bedienungsanleitung für den Programmanwender und der Dokumentation des Programmtextes. Für die Weiterentwicklung bestehender Programmsysteme ist vor allem letztere interessant.
Ein gut lesbarer, ausreichend kommentierter Programmtext macht es erheblich einfacher, ein Programm bzw. die Vorgänge während des Programmablaufs zu verstehen. Davon profitieren nicht nur andere Personen, die den Programmtext lesen wollen, sondern auch der Programmierer selbst, wenn er ein Programm nach längerer Zeit erneut bearbeiten muß. Auch die Fehlersuche ist weniger mühsam, wenn für alle wesentlichen Programmstellen dokumentiert ist, wozu der jeweilige Programmteil dienen soll. Daher sollte eine ordentliche Programmdokumentation für jedes Programm erstellt werden.

1.6 Die Programmiersprache Java

Die Programmiersprache *Java* wurde im Jahr 1995 veröffentlicht, mit der Zielsetzung, eine einfache, aber leistungsfähige und möglichst universell einsetzbare objektorientierte Programmiersprache anzubieten.

1.6.1 Überblick über den Sprachstandard

Ein zentraler Punkt bei der Entwicklung von Java war und ist die Systemunabhängigkeit, d.h. ein Java-Programm soll auf allen Rechnersystemen ohne Änderung ausführbar sein. Neben der eigentlichen Programmiersprache enthält der Java-Standard daher auch Festlegungen zur Gestaltung geeigneter Java-Laufzeitsysteme bzw. Ausführungsumgebungen. Diese Kombination aus Programmiersprache und passender Programmablaufumgebung stellt die sog. *Java-Plattform* dar.

Die Basis von Java ist die Beschreibung der zulässigen Sprachkonstrukte in der *Sprachspezifikation*. Die Java-Spezifikation liefert eine ausführliche Beschreibung der einzelnen Sprachkonstrukte. Außerdem existieren verbindliche Beschreibungen aller Bestandteile des *Application Programming Interface (API)* von Java. Das API ist eine Sammlung von Standardbibliotheken, die alle zur Programmierung der Java-Plattform benötigten Routinen beinhalten.
Die eigentliche Programmiersprache Java wurde seit ihrer Einführung nur geringfügig verändert bzw. erweitert, d.h. damals erstellte, korrekte Java-Programme können auch heute noch ohne Änderungen ausgeführt werden. Aufgrund vieler Erweiterungen der Java-Plattform ist die API-Beschreibung jedoch wiederholt aktualisiert worden. Die aktuelle Fassung von Java bzw. der Java-Plattform ist die Version

1.2.

Für viele Rechner bzw. Betriebssysteme existieren inzwischen Realisierungen der Java-Plattform. Im Rahmen dieses Kurses wird das kostenlos verfügbare *Java Development Kit (JDK)* verwendet. Das JDK ist als Referenz-Implementation der jeweils aktuellen Java-Version und der zugrundeliegenden Java-Plattform konzipiert und existiert für eine Reihe unterschiedlicher Betriebssysteme. Im Rahmen dieses Kurses wird Java 1.1 verwendet, da das JDK für Java 1.2 noch nicht ausgereift genug ist.

1.6.2 Bestandteile des JDK

Das JDK ist eine komplette Implementation der Java-Plattform. Daher beinhaltet es neben allen für das API benötigten Bibliotheken auch einen Compiler (`javac`, siehe 1.6.3) und eine Laufzeitumgebung zur Ausführung der übersetzten Programme (`java`, siehe 1.6.4).

Zusätzlich enthält das JDK einen einfachen Debugger namens `jdb`, der für die Analyse von Abläufen in Java-Programmen entwickelt wurde. Er ist jedoch nicht sehr komfortabel zu bedienen und eher für fortgeschrittene Programmierer geeignet.
Auch Profiling ist mit dem JDK prinzipiell möglich. Das Laufzeitsystem kann durch einen entsprechenden Aufrufparameter (`java -prof`) so konfiguriert werden, daß während der Programmausführung die für das Profiling benötigten Laufzeitinformationen protokolliert werden. Das eigentliche Profiling, also die Analyse dieser Informationen, wird aber nicht direkt unterstützt.

1.6.3 Compilierung von Java-Programmen

Die Übersetzung von Java-Programmen erfolgt in zwei Stufen: Zunächst wird der Programmtext mit dem Compiler `javac` in einen Zwischencode (den sog. *Bytecode*) übersetzt. Dieser Bytecode ist dann mit Hilfe des Java-Interpreters `java` ausführbar (siehe 1.6.4).

Deklarationen von *Java-Klassen* (siehe 2.1) werden in Form von Textdateien (durch die Dateiendung `.java` als Java-Quelltexte identifizierbar) erstellt. Eine Quelltextdatei kann dabei auch mehrere Klassendeklarationen enthalten. Die Quelltextdatei wird bei der Compilierung mit `javac` komplett übersetzt.
Ein Aufruf des Java-Compilers `javac` hat folgende Form[2]:

`javac` *Textdateiname*`.java`

Wichtig: Die Datei-Endung `.java` darf beim Aufrufen des Compilers nicht weggelassen werden! Auch die Verwendung einer anderen Datei-Endung als `.java` wird vom Compiler nicht akzeptiert.

`javac` kann auch mehrere Java-Quelltextdateien innerhalb eines Compileraufrufs übersetzen. Dies ist insbesondere bedeutsam, wenn Quelldateien gegenseitig voneinander abhängen, die Programmtexte in den einzelnen Dateien also wechselseitige Bezüge enthalten.

Aufrufe der Art

```
javac Datei1.java Datei2.java Datei3.java
```

sind ebenso zulässig wie die Benutzung von sog. *Wildcards* bzw. *Jokerzeichen*:

```
javac Datei*.java
```

Die folgende Deklaration der Klasse `HelloWorld` ist bereits ein vollständiges, korrektes Java-Programm: Bei Ausführung des Programms wird mit Hilfe der Methode `System.out.println()` aus einer der Java-Klassenbibliotheken der Text `Hello World!` auf dem Bildschirm ausgegeben. Genauere Erklärungen zur Bedeutung der einzelnen Programmteile folgen in den nächsten Kapiteln.

Beispiel 2:

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

Wenn man diesen Java-Programmtext in eine Datei `HelloWorld.java` geschrieben hat, kann diese Datei mit folgendem Aufruf compiliert werden:

[2] Es wird hier eine textorientierte Kommandoeingabe angenommen, z.B. über eine Text-Shell.

```
javac HelloWorld.java
```

Bei der Compilierung einer Quelltextdatei wird für jede in der Datei enthaltene Klassendeklaration eine eigene *Klassendatei* erzeugt, die den für die jeweilige Klasse generierten Bytecode enthält. Der Name einer Klassendatei setzt sich aus dem Klassennamen und der Endung `.class` zusammen.

Klassenname`.class`

Die Compilierung von `HelloWorld.java` liefert demnach die Bytecode-Datei

```
HelloWorld.class
```

Durch die Zusammenfassung mehrerer Klassendeklarationen in einer Quelltextdatei läßt sich die Anzahl der Quelltextdateien gering halten, und eng verwandte Klassendeklarationen können während der Programmerstellung gemeinsam bearbeitet werden. Auf der anderen Seite entsteht dadurch kein direkter Bezug von Quelltext- zu Codedateien, so daß die Suche nach der zu einer bestimmten Klasse gehörenden Quelltextdatei mühsam werden kann. Normalerweise ist es zweckmäßig, jede Klasse jeweils in einer eigenen Datei mit demselben Namen abzulegen: Eine Klasse mit Namen `A` sollte also in der Datei `A.java` zu finden sein.

Hinweis: Es ist nicht möglich, nur Teile einer Textdatei zu übersetzen. Damit sind Programmtextdateien die kleinsten übersetzbaren Einheiten (die sog. *Übersetzungseinheiten*) in Java.

1.6.4 Ausführung von Java-Programmen

Wie zuvor gezeigt entstehen bei der Compilierung eines Programms eine oder mehrere Bytecode-Dateien. Der erzeugte Bytecode ist allerdings auf den heutzutage eingesetzten Betriebssystemen (noch) nicht direkt ausführbar. Stattdessen wird der Bytecode über einen Interpreter abgearbeitet. Für den im JDK enthaltenen Interpreter `java` entspricht ein Aufruf zur Ausführung eines Java-Programms folgenden Schema:

`java` *Programmname*

Ein Aufruf des in Beispiel 2 gezeigten Programms `HelloWorld` sähe also so aus:

```
java HelloWorld
```

Wichtig: Die Datei-Endung `.class` wird *nicht* angegeben!

Werden zur Programmausführung noch weitere Klassen benötigt, so werden diese vom Interpreter automatisch dazugeladen und müssen daher nicht explizit angegeben werden.

Hinweis: Java bietet mit den sog. *Applets* auch die Möglichkeit, Programme als nicht-eigenständige Anwendungen zu gestalten. Applets sind spezielle Java-Programme mit graphischer Bedienoberfläche, die als aktive Komponenten z.B. in eine HTML-Seite eingebunden werden können. Applets lassen sich deshalb nur mit einem geeigneten Präsentationsprogramm (z.B. WWW-Browser) ausführen.
Da Applets einer Reihe von Einschränkungen unterliegen, steht in diesem Kurs die Erstellung eigenständiger Anwendungsprogramme im Mittelpunkt. Applets werden aber später noch genauer behandelt werden.

1.6.5 Bytecode-Portabilität

Java-Interpreter wie `java` können als Bytecode-Prozessoren angesehen werden, d.h. die Menge aller gültigen Bytecode-Befehle entspricht dem Befehlssatz eines Java-Interpreters. Das Laufzeitsystem von Java emuliert also einen Rechner mit Bytecode-Prozessor. Die Java-Spezifikation bezeichnet diesen Rechner bzw. seine Emulation als *Java Virtual Machine (JVM)*.
Mit der Spezifikation der JVM ist auch der Bytecode eindeutig festgelegt. Damit läßt sich der Bytecode unverändert auf allen Rechnern ausführen, auf denen eine JVM (ein Java-Interpreter) existiert. Eine erneute Compilierung des Programmtexts ist nicht erforderlich.

Die JVM ist nicht für einen speziellen Prozessortyp entworfen, läßt sich aber einfach auf verschiedenen Prozessortypen emulieren. Das erleichtert die Implementierung von Java-Interpretern. Inzwischen werden auch Prozessoren entwickelt, deren Befehlssatz den Bytecode-Befehlen entspricht, so daß die zeitraubende Emulation entfallen kann.

1.6.6 Dokumentation in Java

Zur Dokumentation innerhalb von Programmtexten bietet Java zwei Alternativen. Damit sind ein- und mehrzeilige Kommentare im Programmtext möglich, die bei der Übersetzung des Programms vollständig ignoriert werden.
Einzeilige Kommentare werden mit der Zeichenfolge `//` eingeleitet. Alles, was in der Textzeile rechts von diesen beiden Zeichen steht, wird dann als Kommentar betrachtet. Der Kommentar ist mit dem Zeilenende abgeschlossen, so daß keine weitere Abschlußmarkierung mehr nötig ist. Mehrzeilige Kommentare werden mit `/*` eingeleitet und mit `*/` explizit abgeschlossen.
Kommentare können prinzipiell überall im Programmtext plaziert werden, also auch nach Programmanweisungen. Eine Kommentierung innerhalb einer Programmanweisung ist mit abgeschlossenen Kommentaren (`/*...*/`) zwar auch möglich, wirkt aber normalerweise nicht sehr übersichtlich und sollte daher vermieden werden.

Eine kommentierte Fassung des „Hello World"-Programms aus Beispiel 2 könnte z.B. so aussehen:

Beispiel 3:

```
// HelloWorld
// Ein einfaches Java-Programm, diesmal mit Kommentaren

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

  /* mehrzeiliger
     Kommentar
   */
}
```

2 Prozedurale Konzepte

Dieses Kapitel ist eine kurze Einführung in die Konzepte bzw. Konstrukte, die Java zur *prozeduralen Programmierung* (siehe 2.6.1) bereitstellt. Dazu müssen, nachdem Java zu den *objektorientierten* (Abk.: *OO*) Sprachen gehört, bereits einige Begriffe aus dem OO-Bereich eingeführt werden. Eine ausführliche Beschreibung der OO-Programmierung mit Java folgt aber erst in Kapitel 4.

2.1 Klassen

Der Begriff *Klasse* stammt aus der OO-Programmierung, bei der die Bildung von Klassen ein zentrales Konzept ist. Ein vollständiges Java-Programm besteht aus mindestens einer *Klasse*, die sich wiederum aus einem Klassenrahmen und dem darin enthaltenen eigentlichen Programm zusammensetzt. Die Gesamtheit aus Rahmenbeschreibung und Programmtext ist dann eine *Klassendeklaration*.
Bei einigen Programmiersprachen wird der Begriff der *Deklaration* anders verwendet. In diesen Sprachen setzt sich ein Programm bzw. ein Programmteil aus einem *Kopf* und einem *Rumpf* zusammen, wobei die Deklaration im wesentlichen nur der Programmkopf ist, und die Verbindung von Kopf und Rumpf als *Definition* bezeichnet wird. Der Kopf liefert dabei nur eine Beschreibung, wie das zu definierende (Teil-)Programm verwendet werden kann. Dies kann z.B. der Name sein, über den das (Teil-)Programm aufrufbar ist. Der Rumpf enthält dann die eigentlichen Anweisungen des (Teil-)Programms.
Diese Unterscheidung zwischen Deklaration und Definition ist in Java unnötig. Daher wird im folgenden nur von Deklarationen gesprochen.

Eine Klassendeklaration kann *Methoden* und *Variablen* als *Komponenten* enthalten[1]. Daraus ergibt sich das folgende (vereinfachte) Syntaxschema für Klassendeklarationen in Java:

`class` *Klassenname* {
Variablen
Methoden
}

[1]Seit Java 1.1 sind zusätzlich auch sog. *inner classes* zulässig, d.h. eine Klasse wird innerhalb einer anderen Klasse deklariert. Diese Variante der Klassendeklaration wird an dieser Stelle jedoch nicht weiter vertieft.

Auch die Bezeichnung *Methode* hat ihren Ursprung im OO-Bereich. Als Methode bezeichnet man eine (evtl. auch von außerhalb der Klasse) aufrufbare *Funktion* bzw. *Prozedur*, die innerhalb einer Klasse deklariert ist. Methoden werden in Abschnitt 2.6 genauer erläutert.

Variablen sind aus Sicht des Programmierers im Grunde nichts anderes als Namen für gespeicherte bzw. zu speichernde Daten. Über einen Variablennamen kann die unter diesem Namen gespeicherte Information (der Wert der Variablen) - abhängig von der geplanten Aktion - gelesen oder überschrieben werden. Der Umgang mit Variablen wird in Abschnitt 2.4 ausführlich erklärt.

Eine Klasse kann somit als eine Ansammlung von Methoden und Daten betrachtet werden.

Der Klassenbegriff in der objektorientierten Programmierung basiert darauf, daß in einer Beschreibung eine ganze Klasse von *Objekten* zusammengefaßt werden kann, die die gleichen Eigenschaften bzw. *Attribute* (Methoden und Variablen) besitzen. Jedes dieser Objekte ist damit ein Repräsentant bzw. eine sog. *Instanz* seiner Klasse. Eine Klasse kann prinzipiell beliebig viele Instanzen besitzen.
In diesem Kapitel wird mit sog. *Klassenmethoden* und *Klassenvariablen* gearbeitet. Alle derartigen Variablen und Methoden sind fester Bestandteil einer Klasse und können damit ohne Instanziierung der Klasse verwendet werden. Der Zugriff auf diese Werte erfolgt direkt über die Klasse.
Die Alternative wären die sog. *Instanzvariablen* und *Instanzmethoden*, die für jede Instanz einer Klasse separat angelegt werden und nur über ihre jeweilige Instanz verfügbar sind. Das Instanzenkonzept wird aber im Moment noch nicht benötigt und daher an dieser Stelle nicht weiter vertieft. Es wird im nächsten Kapitel ausführlich besprochen.
Klassenkomponenten (Variablen und Methoden) werden durch Voranstellen des *Schlüsselworts* `static` gekennzeichnet. `static` zählt zu den sog. *Modifikatoren* von Java, mit denen die Zugriffsrechte für einzelne Komponenten bei deren Deklaration genau festgelegt werden können.

Bemerkung: Als *Schlüsselwörter* werden alle Wörter bezeichnet, die fester Bestandteil bzw. eindeutiges Erkennungszeichen eines Sprachkonstrukts sind. Die Übersetzung von Programmen basiert auf der Analyse von Schlüsselworten. Deshalb dürfen Schlüsselworte nicht für andere Aufgaben (z.B. als Namen für Variablen) verwendet werden.

2.2 Namen und Bezeichner

Für Klassen, Methoden, Variablen und andere Java-Sprachelemente muß vom Programmierer ein eindeutiger *Bezeichner* bzw. *Name* festgelegt werden. Über diesen Bezeichner kann dann im weiteren Programm auf das entsprechende Element zugegriffen werden.

2.2.1 Konventionen zur Bezeichnerwahl

Die Namensgebung ist in Java nicht in allen Einzelheiten verbindlich festgelegt. Prinzipiell ist jede Folge von Buchstaben und Ziffern ein gültiger *Java-Bezeichner* bzw. *Name*. Die einzigen Einschränkungen bestehen darin, daß ein Java-Bezeichner mit einem Buchstaben anfangen muß und sich von allen Java-Schlüsselwörtern unterscheiden muß.
Es existieren aber Konventionen für eine einheitliche Bezeichnerwahl, die die Lesbarkeit von Programmtexten erhöhen sollen. Eine sinnvolle Namensgebung beginnt danach schon mit der Auswahl aussagekräftiger Bezeichner, die beim Lesen des Programmtextes die Bedeutung der bezeichneten Sprachelemente so gut wie möglich verdeutlichen. Eine Methode z.B. sollte mit einem Verb bezeichnet werden, das der durch die Methode ausgeführten Aktion entspricht.

Dementsprechend existieren auch Vorschläge zur Schreibweise von Bezeichnern: Klassennamen sollten als Substantive gesehen werden und demnach stets mit einem Großbuchstaben beginnen (z.B. `HelloWorld` in Beispiel 2). Methodennamen (Verben) und Variablen (Attribute bzw. Adjektive) beginnen dagegen mit einem Kleinbuchstaben.
Sofern der Bezeichner aus mehreren Worten zusammengesetzt ist, werden die nachfolgenden Teilworte ohne weiteres Trennzeichen in Großschreibung angehängt (wie auch bei `HelloWorld`). Konstanten, die in Java als unveränderliche Variablen definiert werden (siehe 2.4.5), werden laut Konvention komplett mit Großbuchstaben geschrieben, wobei einzelne Teilworte mit „_“ zusammengesetzt werden (z.B. `EINE_KONSTANTE`).

Wichtig: Für alle Java-Schlüsselwörter ist Kleinschreibung zwingend vorgeschrieben !

2.2.2 Zusammengesetzte Bezeichner

Komponenten einer Klasse sind in Java nur innerhalb ihrer Klasse verwendbar. Dementsprechend sind auch die Bezeichner von Komponenten nur innerhalb der jeweiligen Klasse gültig. Komplexe Programme werden in Java aber normalerweise in eine Reihe von Klassen zerlegt. Um aus einer Klasse auf die Komponenten anderer Klassen (Klassenvariablen, Klassenmethoden usw.) zugreifen zu können, werden *zusammengesetzte Bezeichner* benötigt.
Ein zusammengesetzter Bezeichner kann als eine Art *Pfadangabe* verstanden werden, die den Weg zur gewünschten Komponente in einer *Namenshierarchie* eindeutig festlegt. Die einzelnen Namensteile werden dazu jeweils durch einen Punkt verbunden. Beim direkten Zugriff auf eine Komponente einer Klasse z.B. ergibt sich somit die folgende Bezeichnerstruktur:

Klassenname.*Komponentenname*

In zusammengesetzten Bezeichnern können auch Objektinstanzen und deren Komponenten auftreten. Statt dem Klassennamen wird für eine Objektinstanz ein sog. *Ausdruck* (siehe 2.7) angegeben, der die Objektinstanz bezeichnet. Dies kann z.B. der Name einer entsprechenden Objektvariablen sein.
Da Komponenten von Objektinstanzen wiederum Objektinstanzen sein können, ist theoretisch eine beliebig tiefe Verschachtelung von Instanzen und damit auch von Objektbezeichnern möglich. Die einzelnen Bezeichner werden jeweils wieder mit einem Punkt verbunden. Ein Beispiel für einen solchen Bezeichner ist der Methodenname `System.out.println` in Beispiel 2. Dieser Bezeichner beschreibt die Methode `println` der Objektvariablen `out`, die wiederum eine Komponente der Klasse `System` ist.
An dieser Stelle sei nur erwähnt. daß die Klasse `System` eine der im Java-Standard festgelegten Klassen ist. Die Verwendung von Java-Standardklassen wird in den folgenden Kapiteln ausführlich behandelt werden.

2.3 Einfache Datentypen

Die zentrale Aufgabe von Programmen ist die Bearbeitung von Daten, sei es die Verarbeitung von eingegebenen Datensätzen oder die Datenausgabe für ein Berechnungsergebnis. Um mit den Daten sinnvoll umgehen zu können, müssen zum Zeitpunkt des Datenzugriffs die Größe und die Struktur der Daten bekannt sein. Diese Informationen werden in vielen Programmiersprachen durch Angabe der *Sorte* bzw. des *Datentyps* der zu bearbeitenden Daten bereitgestellt.
Auch in Java werden Datentypen meist explizit festgelegt. Java arbeitet mit *strenger Typisierung*, d.h. alle Daten müssen einem Datentyp zugeordnet sein. Dies unterscheidet Java von Sprachen ohne strenge Typisierung, z.B. Skriptsprachen wie Perl. Erfolgt für einen Wert keine explizite Typangabe, wird der Wert implizit einem bestimmten Datentyp zugeordnet. Die Regeln für diese Zuordnung sind in Java genau festgelegt.
Durch die Typangabe kann exakt bestimmt werden, wieviel Speicherplatz ein Datum belegt und wie das Datum in diesem Speicherplatz abgelegt werden muß. Sowohl Variablen als auch Methoden besitzen einen Datentyp. Für Methoden entspricht der Typ der Methode dem Datentyp des Ergebnisses, das die Methode nach ihrer Ausführung zurückliefert.

Das Typkonzept von Java ist zweigeteilt. In diesem Abschnitt werden zunächst nur die sog. *Grundtypen* vorgestellt. Die andere Klasse von Datentypen, die *Referenztypen*, unterscheidet sich von den Grundtypen vor allem in der Art des Datenzugriffs. Grundtypen erlauben den direkten Zugriff auf Daten, während bei Referenztypen nur mit Verweisen (Referenzen) auf die eigentlichen Daten gearbeitet wird.
In Java sind alle Datentypen, die nicht zu den Grundtypen zählen, Referenztypen. Darunter sind alle Typen für *Felder* und *Objekte*. Auch Felder von Grundtypen oder Zeichenkettenobjekte (Strings) gehören damit einem Referenztyp an.

In diesem Kapitel werden nur die für Grundtypen und Felder charakteristischen Merkmale behandelt. Die ausführliche Vorstellung von Objekten folgt im nächsten Kapitel.

2.3.1 Grundtypen

Java bietet für Zahlen, Textzeichen und Boolesche Werte einige einfache *Grundtypen* (engl.: *primitive types*) an, die jeweils durch eigene Schlüsselwörter bezeichnet werden.

Schlüsselwort	Speicherplatz	Wertebereich
Datentypen für ganzzahlige Werte (mit Vorzeichen)		
`byte`	8 bit	$-2^7 \ldots 2^7-1$ ($-128 \ldots 127$)
`short`	16 bit	$-2^{15} \ldots 2^{15}-1$ ($-32768 \ldots 32767$)
`int`	32 bit	$-2^{31} \ldots 2^{31}-1$ ($-2147483648 \ldots 2147483647$)
`long`	64 bit	$-2^{63} \ldots 2^{63}-1$ ($-9223372036854775808 \ldots 9223372036854775807$)
Datentypen für Gleitkommazahlen		
`float`	32 bit	$\pm 3.40282347E+38 \ldots$ $\pm 1.40239846E-45$
`double`	64 bit	$\pm 1.79769313486231570E+308 \ldots$ $\pm 4.94065645841246544E-324$
Datentyp für Boolesche Werte		
`boolean`	1 bit	$false, true$
Datentyp für Unicode[2]-Textzeichen		
`char`	16 bit	$\backslash u0000 \ldots \backslash uFFFF$

Hinweis: Der Datentyp `char` ist auch für ganze Zahlen ohne Vorzeichen (Wertebereich: $0 \ldots 2^{16}-1$) verwendbar, allerdings nur innerhalb von Berechnungen. In allen anderen Situationen, z.B. bei der Ausgabe eines `char`-Wertes auf dem Bildschirm, wird der Wert als Textzeichen interpretiert.

2.3.2 Literale

Die explizite Angabe von Werten eines Grundtyps erfolgt im Quelltext über die sog. *Literale*. Literale können nur bei den Grundtypen für Wertangaben verwendet werden, d.h. Literale für Referenztypen existieren nicht. Die einzige Ausnahme sind *Zeichenketten* (mit der Typbezeichnung `String`), für die aufgrund ihrer häufigen Verwendung eine einfache Literalschreibweise festgelegt ist.
Die folgende Tabelle beschreibt den korrekten Aufbau von Literalen für die verschiedenen Grundtypen und für Zeichenketten:

Art der Darstellung	Schreibweise	Beispiel
ganzzahlige Werte		
dezimal	[−]{1...9}{0...9}*	1234
hexadezimal	[−]0x{{0...9}|{a...f}}+	0x1A2B
oktal	[−]0{0...7}+	01234
Gleitkomma-Werte		
gebrochen rational	[−][{0...9}*].{0...9}+[e[−]{0...9}+]	1.23e4[3]
exponential	{0...9}+[e[−]{0...9}+]	123e4
Boolesche Werte		
Text	true|false	true
einzelne Textzeichen		
Zeichen	'{*Zeichen*}'	'a'
Unicode	{*ganzeZahl*}	97
Zeichenketten (Strings)		
Text	" {*Zeichen*}* "	"Hello World!"

Bei der Verwendung von Buchstaben in numerischen Literalen wird nicht zwischen Groß- und Kleinschreibung unterschieden. Der Beispielwert `0x1A2B` kann genauso als `0X1a2b` geschrieben werden.

Ganzzahlige Literale werden zunächst dem Typ `int` zugeordnet. Wird an ein Literal ein `l` bzw. `L` angehängt, stellt das Literal einen Wert des Typs `long` dar (z.B. `1234L`). Eine Kennzeichnung von Literalen als `byte`- bzw. `short`-Wert ist in Java nicht möglich.

[2] *Unicode* ist ein Standard zur Integration verschiedenster internationaler Zeichensätze. Einer dieser Zeichensätze ist u.a. der ASCII-Zeichensatz.

[3] Negative Exponenten werden durch ein dem Exponenten vorangestelltes Minuszeichen gekennzeichnet (z.B. `-1.23e-4`).

Bei Gleitkomma-Werten kann durch ein nachgestelltes `f` bzw. `d` eine eindeutige Festlegung von `float`- bzw. `double`-Werten erreicht werden. Werte ohne explizite Typangabe sind dabei vom Typ `double`. Wird bei exponentialer Darstellung kein Exponent angegeben, muß eine explizite Typfestlegung mittels `f` oder `d` erfolgen, um eine Unterscheidungsmöglichkeit gegenüber der Dezimaldarstellung ganzer Zahlen zu haben (z.B. `1234d`).

Neben darstellbaren Zeichen wie Buchstaben und Ziffern erlaubt Java auch Steuerzeichen und andere nicht direkt darstellbare Textzeichen in Literalen. Zur Eingabe solcher Zeichen dienen die sog. *Escape-Sequenzen*. Die wichtigsten Escape-Sequenzen sind in der folgenden Tabelle kurz aufgelistet.

\n	Neue Zeile
\t	Tabulator
\'	Einfaches Anführungszeichen
\"	Doppeltes Anführungszeichen
\\	Backslash
\uXXXX	Unicode-Zeichencode (X = Hexadezimal-Ziffer 0...f)
\OO...	Unicode-Zeichencode (O = Oktal-Ziffer 0...7)

Hinweis: Literale können im Programmtext nicht (z.B. zur übersichtlicheren Darstellung) getrennt werden. Ein Zeichenkettenliteral z.B. muß fortlaufend und ohne Zeilenumbrüche eingegeben werden. Eine saubere Formatierung von Programmausgaben ist dagegen durch die Verwendung der oben genannten Steuerzeichen möglich und sollte auch angestrebt werden.

2.4 Variablen

Variablen sind Speicherbereiche, auf die über einen Namen bzw. Bezeichner zugegriffen werden kann. Sie werden zur Speicherung von *Zuständen* (dazu mehr in den folgenden Kapiteln) und von *Zwischenergebnissen* verwendet.
Wie in Abschnitt 2.1 bereits erwähnt wurde, können Variablen in Java in verschiedenen Formen auftreten, nämlich als Klassen- oder Instanzvariablen (siehe Kapitel4), oder als *lokale Variablen* (siehe Abschnitt 2.4.3). Die Unterschiede dieser einzelnen Variablenarten werden im weiteren Verlauf des Kurses noch behandelt werden. Es gilt aber für jede Variable in Java, daß vor ihrer Verwendung zwei Schritte ausgeführt werden müssen, nämlich die *Deklaration* und die *Initialisierung* der Variablen.

2.4.1 Deklaration von Variablen

Bei der Deklaration einer Variable wird Speicherplatz für diese Variable reserviert. Dazu ist die Angabe eines Datentyps für den gewünschten Variablennamen erforderlich, um den Platzbedarf der Variable bestimmen zu können. Will man mehrere Variablen gleichen Typs anlegen, so ist es möglich, die Variablennamen mit Kommata hintereinanderzureihen.

Beispiel 4:

```
int ganzeZahl;
double gleitpunktzahl;
String zeichenkette, nochEineZeichenkette;
```

Variablen werden in Java nach ihrem Namen unterschieden. Daher dürfen zu keinem Zeitpunkt zwei Variablen mit dem selben Namen deklariert sein, selbst wenn sie zu unterschiedlichen Datentypen gehören.

2.4.2 Initialisierung von Variablen

Bei der Initialisierung wird in den für die Variable reservierten Speicherplatz ein Anfangswert (Initialbelegung) eingetragen, d.h. es findet ein schreibender Zugriff auf die Variable statt. Dazu wird der Variablen mit dem *Operator* „=“ (siehe 2.7) ein Anfangswert zugewiesen. Dieser Operator wird deshalb auch *Zuweisungsoperator* genannt. Die Initialisierung ist also eine Zuweisungsanweisung (siehe 2.8).

Beispiel 5:

```
ganzeZahl = 5;
gleitpunktzahl = 3.14159;
zeichenkette = "Irgendein Text";
```

Es ist auch möglich, Deklaration und Initialisierung zusammenzufassen. Dabei wird die Initialisierung als Teil der Deklaration aufgefaßt, d.h. wie bei der normalen Deklaration können mehrere Variablen mit einer Deklarationsanweisung angelegt und einzeln initialisiert werden.

Beispiel 6:

```
int ganzeZahl = 5;
double gleitpunktzahl = 3.14159;
String zeichenkette         = "Irgendein Text",
       nochEineZeichenkette = "Ein anderer Text";
```

Variablendeklarationen und -initialisierungen müssen stets mit einem „;“ abgeschlossen sein. Dadurch können die einzelnen Anweisungen einer Anweisungsfolge sauber voneinander getrennt werden. Andere Text- oder Steuerzeichen, z.B. ein Zeilenumbruch, sind als Abschlußzeichen nicht geeignet.

Wichtig: In Java wird für jede Variable bei ihrer Deklaration implizit eine Initialisierung vorgenommen, d.h. der für die Variable reservierte Speicher wird in einen definierten Anfangszustand gebracht. Auf diesen Wert sollte der Programmierer sich jedoch nicht verlassen, sondern statt dessen die Variable stets explizit initialisieren, bevor sie anderweitig verwendet wird. Viele Java-Compiler, darunter auch `javac`, brechen daher beim Zugriff auf nicht explizit initialisierte Variablen die Übersetzung mit einer Fehlermeldung ab oder geben zumindest entsprechende Warnhinweise aus.

2.4.3 Lokale Variablen

Das zentrale Konzept zur Strukturierung von Programmen in Java sind Blockstrukturen, die einen modularen Programmaufbau ermöglichen. Neben Klassendeklarationen, die auch als Blockstruktur angesehen werden können, sind vor allem *Anweisungsblöcke* (oder kurz *Blöcke*) für die Programmierung interessant. Ein Block besteht aus einem Paar geschweifter Klammern und kann Variablendeklarationen und Programmanweisungen beinhalten. Auch leere Blöcke der Form `{}` sind zulässig.
Blöcke selbst sind Programmanweisungen, d.h. in einem Block können weitere Blöcke auftreten. Durch die Schachtelung von Blöcken ist eine Strukturierung des Programmtextes möglich. Ausgangspunkt der Blockstrukturierung in Java sind die Methoden bzw. deren Rümpfe (siehe 2.6).

Werden innerhalb eines Blocks Variablen deklariert, so bezeichnet man diese als *lokale Variablen*. Lokale Variablen sind im Gegensatz zu Klassenvariablen nur innerhalb des Blocks verwendbar, in dem sie deklariert werden. Ein Zugriff von außen ist nicht zulässig, egal ob von einem umschließenden oder einem separaten Block aus.

```
int variable = 0;
{
  int lokaleVariable = 1;
}
variable = lokaleVariable; // FEHLER: Zugriff auf lokale Variable von aussen!
```

2.4.4 Lebensdauer und Gültigkeitsbereich

Die *Lebensdauer* einer Variablen wird bestimmt durch den Ort ihrer Deklaration im Programmtext. Die Lebensdauer einer lokalen Variable oder eines formalen Parameters endet bei Verlassen des umschließenden Blockes (im Normalfall am Blockende) bzw. der Methode. Nach Verlassen des Blockes steht der Speicherbereich der lokalen Variable wieder für andere Verwendungszwecke zur Verfügung. Klassen- und Instanzvariablen hingegen sind lebendig, solange ihre Klasse bzw. Instanz existiert.
Auf eine Variable kann während ihrer Lebensdauer innerhalb ihres *Gültigkeitsbereiches* zugegriffen werden. Dieser umfasst — je nach Variablenart — einen bestimmten Block, wobei u.U. in einem inneren Block eine Variable gleichen Namens deklariert werden kann, die in diesem Abschnitt die äußere Variable *verschattet*. In Java dürfen lokale Variablen und formale Parameter sich nicht gegenseitig verschatten, aber sie können Klassen- und Instanzvariablen verschatten.

```
int variable = 0;
{
  variable = 1;
}
{
  int variable = 1; // FEHLER: Verschattung in Java nicht erlaubt!
```

```
}
```

Wichtig: Begriffe wie *Lokalität*, *Gültigkeit* und *Lebensdauer* beziehen sich vor allem auf den Programmtext und weniger auf die Programmausführung. Geschachtelte Methodenaufrufe z.B. sind uneingeschränkt möglich, da jeder Methodenaufruf als separater Block abläuft. Man spricht dabei auch von einer *Inkarnation* der Methode.
Alle Inkarnationen einer Methode besitzen für ihre lokalen Variablen jeweils eigene Speicherbereiche. Lokale Variablen werden also zwischen den Inkarnationen nicht geteilt, sondern jede Inkarnation besitzt eigene „Exemplare“ der lokalen Variablen. Dies ist insbesondere von Bedeutung für die *Rekursion* (siehe 2.6.7).

2.4.5 Konstanten

Oft gibt es in einem Programm Größen, die für die Dauer des gesamten Programms einen festen Wert haben sollen. Derartige Größen sollten als *Konstanten* definiert sein, damit ihr Wert nicht während des Programmablaufs versehentlich geändert werden kann.

Eine Konstante kann als Variable betrachtet werden, deren Wert mit ihrer Initialisierung für ihre gesamte Lebensdauer unveränderlich festgelegt wird. Eine Konstante muß bei ihrer Deklaration auch zugleich initialisiert werden.
Um eine Variablendeklaration in eine Konstantendeklaration umzuwandeln, wird der Modifikator `final` vor dem Datentyp der Konstanten angegeben. Damit wird der Speicherbereich der Variable für alle Schreibzugriffe nach der Initialisierung gesperrt. (Der `final`-Modifikator kann auch für Methodendeklarationen benutzt werden. Dieser Gebrauch wird aber erst in Kapitel 5 behandelt werden.)

Beispiel 7:

```
final int EINE_KONSTANTE = 10;      // Konstantendeklaration: EINE_KONSTANTE
int eineVariable = EINE_KONSTANTE;
EINE_KONSTANTE = 11;                // FEHLER: Konstante ist unveraenderlich!

final double PI = 3.141592;
```

2.5 Felder

Felder (auch *Reihungen* genannt, engl.: *arrays*) dienen dazu, mehrere Elemente desselben Datentyps zusammenzufassen. Für ein Feld wird ein fortlaufender Speicherbereich reserviert, der genug Platz bieten muß, um alle einzelnen Elemente aufzunehmen. Der pro Element reservierte Speicherplatz enthält die Daten des jeweiligen Elements (bei Grundtypen) bzw. einen Verweis auf die Elementdaten (bei Referenztypen).

2.5.1 Deklaration von Feldern

Die Deklaration von Feldern erfolgt in Java durch Paare von eckigen Klammern, die direkt hinter den Datentyp der Feldelemente geschrieben werden. Die Anzahl der Klammerpaare bestimmt dabei die *Dimension* des Feldes, d.h. die Anzahl der Indexwerte, die für den Zugriff auf genau ein Element des Feldes benötigt werden.
Ein n-dimensionales Feld kann allgemein als ein i-dimensionales Feld mit $n - i$-dimensionalen Feldern als Elementen betrachtet werden. Mehrdimensionale Felder sind z.B. zur Repräsentierung von Matrizen gut geeignet.

Das folgende Syntaxschema zeigt Variablendeklarationen für ein- bzw. zweidimensionale Felder. Höherdimensionale Felder werden analog mit der entsprechenden Anzahl von Klammerpaaren deklariert.

Elementtyp`[]` *Bezeichner*;
Elementtyp`[][]` *Bezeichner*;

Hinweis: Die Klammerpaare können auch erst hinter den Bezeichner der Feldvariablen gesetzt werden, wobei auch eine Mischung beider Schreibweisen erlaubt ist. Deklarationen der Art

Elementtyp Bezeichner`[]`;
Elementtyp`[]` *Bezeichner*`[]`;

sind also zulässig. Diese Schreibweise ist aber weniger intuitiv und sollte daher vermieden werden.

Nach der Deklaration ist ein Feld zwar bzgl. seines Datentyps festgelegt, aber es wurde noch kein Speicherplatz für die Feldelemente reserviert. Ein derartiges Feld kann auch als Feld ohne Elemente betrachtet werden. Erst durch die Initialisierung des Feldes wird die Anzahl der Feldelemente festgelegt.

2.5.2 Initialisierung von Feldern

Auch eine Feldvariable kann bereits während der Deklaration initialisiert werden, indem man der Variablen eine mit Kommas getrennte Auflistung ihrer Elemente in geschweiften Klammern zuweist. Auch mehrdimensionale Felder können so initialisiert werden, wenn die entsprechenden Teilfelder innerhalb des Feldes angegeben bzw. initialisiert werden.

Beispiel 8:

```
float[] zahlenFeld = { 1.1, 2.2, 3.3, 4.4 };   // eindimensionales Feld

char[][] zeichenFeld = { { 'a', 'b' },          // eindimensionale Felder als
                         { 'c', 'd' }           // Feldelemente
                       };
```

Wichtig: Die explizite Angabe von Feldinhalten zwischen geschweiften Klammern ist nur im Rahmen einer Felddeklaration zulässig. Jede andere Verwendung dieser Schreibweise, z.B. in einer separaten Zuweisungsanweisung, führt zu einer Fehlermeldung.

Einen Sonderfall stellt die explizite Initialisierung eines Feldes mit dem Schlüsselwort `null` dar: Für ein mit `null` initialisiertes Feld wird keinerlei Speicherplatz für Feldelemente reserviert.
Durch `null` wird hierbei die Abwesenheit eines Feldes ausdrückt. Daher kann in diesem Falle über die Feldvariable nicht auf Feldelemente zugegriffen werden. Auch das Abfragen der Anzahl der Elemente führt zu einem Fehler (im Gegensatz zu Feldern mit 0 Elementen).

Beispiel 9:

```
float[] leeresFeld = null;
```

Bemerkung: Da `null` anstelle eines (fehlenden) Verweises auf einen Speicherbereich verwendet werden kann, läßt es sich in Verbindung mit allen Arten von Referenztypen einsetzen, nicht nur mit Feldern. `null` entspricht also bzgl. seines Datentyps einer allgemeinen Referenz.

2.5.3 Zugriff auf Feldelemente

Der Zugriff auf einzelne Elemente eines eindimensionalen Feldes erfolgt über den *Index* des Elements. Der Index entspricht der Position des Elements relativ zum Anfang des Feldes, wobei das erste Element im Feld in Java stets den Index `0` erhält. Der Index numeriert die Elemente lückenlos zum Feldende hin aufsteigend durch. Das letzte Element eines n-elementigen Feldes hat somit den Index $n-1$.
Die Anzahl der Elemente eines Feldes wird auch als *Länge* des Feldes bezeichnet. Die Länge stellt einen Attributwert des Felds dar und kann abgefragt werden, indem man `.length` nach der Angabe des Feldes bzw. seines Namens schreibt. Damit kann zur Laufzeit für jedes beliebige Feld der größte gültige Feldindex ermittelt werden.
Auf Feldelemente wird zugegriffen, indem der Index des gewünschteni Elementes in eckigen Klammern an das Feld bzw. den Feldnamen angehängt wird.

Der Zugriff auf mehrdimensionale Felder verläuft analog zum Zugriff auf eindimensionale Felder, wobei entsprechend der Dimension mehrere Indexwerte zur Adressierung eines Elements benötigt werden. Um auf ein einzelnes Element eines n-dimensionalen Feldes zuzugreifen, ist also ein n-stelliger Index der Form `[`$indexwert_1$`][`$indexwert_2$`]`$\ldots$`[`$indexwert_n$`]` nötig. Allgemein erhält man durch Angabe von m Indexwerten ($m \leq n$) Zugriff auf Teilfelder mit der Dimension $n-m$.

Beispiel 10:

```
float[] zahlenFeld = { 1.1, 2.2, 3.3, 4.4 };
zahlenFeld[3] = 5.0;                     // Wert des 4. Elements zu 5.0 aendern
int letztesElement = zahlenfeld.length -1; // zahlenfeld hat 4 Feldelemente ==>
                                         // Index des letzten Elements ist 3
```

```
zahlenfeld[letztesElement] = 4.0;        // Wert des letzten Element zu 4.0
                                         // aendern

char[][] zeichenFeld = { { 'a', 'b' },
                         { 'c', 'd' }
                       };
zeichenFeld[1][0] = zeichenFeld[0][1];   // 'c' wird mit 'b' ueberschrieben
int AnzahlZeichenFelder = zeichenfeld.length; // zeichenfeld hat 2 Feldelemente
                                              // (mit 2 Elementen pro Teilfeld)
int AnzahlZeichen = zeichenfeld[0].length;    // zeichenfeld[0] ist Teilfeld
```

2.5.4 Erzeugung leerer Felder

Zum Zeitpunkt der Deklaration eines Feldes sind oft dessen Größe bzw. sein Inhalt noch nicht bekannt, so daß eine Initialisierung des Feldes bzw. der Feldelemente an dieser Stelle noch nicht möglich ist. Für solche Situationen bietet Java die Möglichkeit, ein leeres Feld geeigneter Größe zu erzeugen, sobald die Anzahl der Feldelemente bekannt ist. Mit Hilfe des Operators `new` wird explizit ein Speicherbereich reserviert, der groß genug ist, um das gesamte Feld aufzunehmen. Die Feldelemente selbst können dann auch erst zu einem späteren Zeitpunkt initialisiert werden.
Die Syntax zur Erzeugung eines (eindimensionalen) Feldes entspricht folgendem Schema:

`new` *Datentyp* `[` *Feldlänge* `]`

Die Angabe des Datentyps und der Feldlänge ist erforderlich, damit die Größe des benötigten Speicherplatzes genau berechnet werden kann. Bei mehrdimensionalen Feldern findet die Initialisierung nach dem selben Prinzip statt, wobei für die weiteren Dimensionen die jeweiligen Längenangaben in zusätzliche eckige Klammerpaare gesetzt werden[4].

```
char[] feld = new char[80];

int[][] matrix;          // Deklaration (Typfestlegung fuer Variablennamen)
matrix = new int[2][2];  // Initialisierung (Festlegung der Feldlaenge)
```

Wichtig: Die Initialisierung von leeren Feldern (mit `new`) betrifft immer nur das Feld selbst, d.h. Anzahl und Art der Feldelemente werden festgelegt. Eine Initialisierung der einzelnen Feldelemente findet dabei *nicht* statt!

Wird ein Feld bei der Deklaration nicht initialisiert, so entspricht dies einer impliziten Initialisierung mit `null`. Die explizite Initialisierung eines derartigen Feldes könnte dann z.B. so aussehen:

```
int[][] feld;
feld = null;
```

Eine solche Initialisierung erzeugt ein Feld mit 0 Elementen, d.h. es muß kein Speicherplatz für die Feldelemente reserviert werden.

Bei mehrdimensionalen Feldern ist auch eine partielle Initialisierung möglich. Wird z.B. bei einem zweidimensionalen Feld eine Initialisierung nur für die erste Dimension angegeben, so entspricht dies der Initialisierung eines eindimensionalen Feldes mit eindimensionalen Feldern als Elementen. Damit sind auch unregelmäßig geformte Felder möglich, da die einzelnen Teilfelder später mit unterschiedlichen Größen initialisiert werden können.

Beispiel 11:

```
int[][] feld = new int[2][];        // feld hat 2 uninitialisierte Teilfelder
int[] erstesElement = new int[1];
feld[0] = erstesElement;            // 1. Teilfeld hat 1 Element
feld[1] = new int[10];              // 2. Teilfeld hat 10 Elemente

String[]   textFeld1 = { "das", "ist", "ein" };
```

[4] Der `new`-Operator wird auch zur dynamischen Erzeugung von einzelnen Objekten verwendet. Hierbei entfällt logischerweise die Angabe irgendwelcher Feldlängen.

```
String[][] textFeld2 = { textFeld1,
                         { "zweidimensionales", "Textfeld" }
                       };
```

Bei der partiellen Initialisierung ist darauf zu achten, daß immer nur komplette Teilfelder uninitialisiert bleiben dürfen. Das bedeutet, daß nach der ersten nichtinitialisierten Dimension keine weiteren Initialisierungsgrößen mehr folgen dürfen. Initialisierungen wie z.B.

```
int[][]      initFehler1 = new int[][3];          // FEHLER: Anzahl der
                                                  // Teilfelder fehlt!
double[][][] initFehler2 = new double[10][][10]; // FEHLER: Teilfeld nicht
                                                  // komplett uninitialisiert!
```

sind also nicht zulässig.

Ist die Länge eines Feldes durch die Initialisierung erst einmal vorgegeben, kann sie nicht mehr geändert werden. Insbesondere kann ein Feld nicht vergrößert werden. Stattdessen wird normalerweise ein zweites Feld mit entsprechend veränderter Größe erzeugt, in das die ursprünglichen Feldelemente bzw. die noch benötigten Elemente kopiert werden können.

2.6 Methoden

Methoden sind zentrale Bestandteile von OO-Sprachen wie Java. In Java sind ausführbare Anweisungen nur innerhalb von Methoden zulässig. Damit müssen Methoden verwendet werden, um die (in Variablen) gespeicherten Daten in irgendeiner Form bearbeiten zu können. Methoden stellen ausführbare Anweisungsfolgen dar. Der *Aufruf* einer Methode ist selbst eine solche ausführbare Anweisung und startet die Ausführung der Methode bzw. der im Methodenrumpf definierten Anweisungsfolge.
Methoden sind stets Bestandteil einer Klasse, d.h. die Klasse liefert einen Bezugsrahmen (*Kontext*) für die Methode, in dem die Methodenausführung stattfinden kann. Damit ist die Methode eng mit der Klasse verbunden, in der sie deklariert wird, unabhängig davon, ob es sich um eine Klassen- oder eine Instanzmethode handelt (siehe Kapitel 4 zur Unterscheidung). Der Zugriff auf eine Methode (also der *Methodenaufruf*) ist daher bei Klassenmethoden nur über die sie umgebende Klasse (bzw. eine Instanz bei Instanzmethoden) möglich.

2.6.1 Prozeduren und Funktionen

Betrachtet man andere (nicht-objektorientierte) Programmiersprachen, so wird dort oft zwischen *Prozeduren* und *Funktionen* als aufrufbaren Programmteilen unterschieden. Gemeinsames Merkmal beider Konzepte ist die Zusammenfassung einer Anweisungsfolge zu einer Struktur, die einen eindeutigen Namen bzw. Bezeichner erhält. Über diesen Namen kann dann die Anweisungsfolge aufgerufen werden.

Bei Prozeduren im engeren Sinn sind die Daten, die die Prozedur bearbeiten soll, außerhalb der Prozedurstruktur definiert. Das Ergebnis des Prozeduraufrufs sind dann die Änderungen, die während der Prozedurausführung an den Daten vorgenommen wurden. Diese Änderungen werden als *Seiteneffekte* bezeichnet, da sie praktisch nebenbei während der eigentlichen Prozedurausführung stattfinden.
Innerhalb einer Prozedur können Hilfsvariablen definiert werden, die nach dem Ende der Prozedur wieder gelöscht werden. Diese können z.B. zur Speicherung von Zwischenergebnissen verwendet werden. Da die Zuweisung (Speicherung) von Werten in Prozeduren eine zentrale Rolle spielt, spricht man von *zuweisungsorientierter Programmierung*, wenn Prozeduren verwendet werden.

Mit Funktionen wird versucht, Daten und Anweisungen streng voneinander zu trennen. Dazu werden bei einem Funktionsaufruf alle von der Funktion benötigten Daten als *Parameter* zur Verfügung gestellt. Die Funktion wird dann ausgeführt, wobei nur die in den Parametern enthaltenen Daten verwendet werden können. Das Ergebnis des Funktionsaufrufs wird in Form eines *Rückgabewerts* an den Aufrufer übergeben, der damit weiterarbeiten kann. Funktionen verursachen keine Seiteneffekte, da während der Funktionsausführung keine Zugriffe auf Daten außerhalb der Funktion mehr nötig sind.
Programme in funktionalen bzw. *applikativen* Sprachen wie z.B. Gofer bestehen im Prinzip aus einem Funktionsaufruf, dessen Rückgabewert dann das Gesamtergebnis der Programmausführung darstellt. In der Regel verfügen aber auch applikative Sprachen über eine Möglichkeit, Seiteneffekte gezielt zu erzeugen, z.B. zur Ausgabe von Zwischenergebnissen.
In applikativen Sprachen wird häufig die Bezeichnung *Term* anstelle von *Programm* verwendet. Die Auswertung eines Terms entspricht dann der Programmausführung und geschieht, indem alle Teilterme des

Terms ausgewertet und über die Anwendung von Operationen miteinander verknüpft werden. Die so entstehenden Termauswertungen lassen sich in Form eines Baums darstellen, mit dem ursprünglich auszuwertenden Term als Wurzel. Die Darstellung prozeduraler Programmabläufe ist dagegen mit Baumstrukturen allein meist nicht möglich, da Seiteneffekte nicht in eine Baumstruktur umgesetzt werden können.

Bei vielen modernen Programmiersprachen existiert keine klare Trennung mehr zwischen Prozeduren und Funktionen. Oft können Prozeduren mit Parametern und Resultaten versehen werden, und Seiteneffekte können auch aus Funktionen heraus erzeugt werden. Solche Sprachen erlauben sowohl einen eher funktionalen als auch einen prozeduralen Programmierstil.
Auch Java zählt zu diesen Sprachen. Methoden in Java bieten wie Funktionen die Möglichkeit der Parametrisierung und können auch Rückgabewerte liefern. Gleichzeitig können sie aber auch auf Daten außerhalb des Methodenrumpfs zugreifen, solange diese sich in der selben Klasse befinden. Damit vereinen die Java-Methoden die Konzepte von Funktionen und Prozeduren in sich. Java unterstützt somit sowohl zuweisungsorientierte als auch applikative Programmierung.

2.6.2 Anweisungen und Ausdrücke

Analog zur Unterscheidung zwischen Funktionen und Prozeduren können bei zuweisungsorientierten Sprachen wie Java auch einzelne Programmschritte in zwei Kategorien eingeteilt werden.
Dies sind zum einen die *Anweisungen*, die den prozeduralen Ablauf der Programmausführung festlegen. Anweisungen dienen vorrangig der Steuerung des Programmablaufs. Auf der anderen Seite stehen die *Ausdrücke*, die Werte bzw. die Berechnung von Werten repräsentieren, wie z.B. Literale oder Variablenbezeichner.

Ausdrücke besitzen einen Datentyp, der bei streng typisierten Sprachen wie Java eindeutig festgelegt ist, und werden zum Zugriff auf Daten bzw. Berechnungsergebnisse während der Programmausführung benötigt. Ausdrücke können als erweiterte Form von Termen betrachtet werden, die zusätzlich auch zuweisungsorientierte Elemente als Teilausdrücke enthalten dürfen. In Ausdrücken können also, anders als in Termen, auch Seiteneffekte auftreten.
Ausdrücke sind Bestandteil von Anweisungen, d.h. ein korrektes Java-Programm besteht aus einer Folge von Anweisungen, die Ausdrücke enthalten können oder sogar nur aus einem Ausdruck bestehen, wie z.B. bei einer Zuweisungsanweisung. Wird in einer Anweisung ein Wert benötigt, wird dieser Wert durch einen Ausdruck festgelegt.

Weitere Informationen zum Aufbau von Ausdrücken und zu den verschiedenen Anweisungsarten sind in den Abschnitten 2.7 bzw. 2.8 zu finden.

2.6.3 Deklaration von Methoden

Die *Methodendeklaration* wird im folgenden aufgeteilt in die Deklaration des *Methodenkopfs* und des *Methodenrumpfs*. Diese Aufteilung ist in Java sinnvoll, da für die sog. *abstrakten* Methoden (siehe 4.4) nur ein Methodenkopf angegeben wird.
In diesem Kapitel wird davon ausgegangen, daß zu jeder Methodendeklaration auch ein entsprechender Methodenrumpf angegeben wird, d.h. abstrakte Methoden werden zunächst noch nicht behandelt. Die Verwendung abstrakter Methoden wird erst im weiteren Verlauf dieses Kurses erklärt werden. Die Schreibweise zur Deklaration abstrakter Methoden (mit abschließendem „;“ anstelle des Methodenrumpfs) wird jedoch bereits in diesem Kapitel zur Darstellung von Methodenköpfen verwendet.

Der Kopf einer Methode legt neben dem Methodennamen auch die *formalen Parameter* fest. Dies geschieht über eine in runde Klammern gesetzte, durch Kommas getrennte Liste der Parameterdeklarationen, also den Namen und Datentypen aller Parameter. Außerdem wird der Datentyp für den Rückgabewert hinzugefügt, den die Methode nach einem Aufruf als Ergebnis liefert.

Wichtig: Da die formalen Parameter im Methodenrumpf wie Variablen verwendet werden, ist es nicht erlaubt, zwei formale Parameter mit dem gleichen Bezeichner in einem Methodenkopf zu verwenden.

Das Syntaxschema für einen Methodenkopf sieht folgendermaßen aus:

Ergebnistyp Methodenname (*Datentyp Name*, *Datentyp Name*, ...)

Anhand des Methodenkopfs kann bereits bestimmt werden, wie die Eingabedaten für einen korrekten Methodenaufruf beschaffen sein müssen, und was für eine Art von Rückgabewert zu erwarten ist. Der Kopf einer Methode entspricht damit der sog. *Signatur* dieser Methode, die einem Methodenbezeichner

die Datentypen all seiner Parameter zuordnet.

Wenn eine Methode keine Parameter benötigt, dann muß im Methodenkopf eine leere Parameterliste (ein leeres Klammerpaar) angegeben werden. Falls eine Methode keinen Rückgabewert liefern soll, so wird dies durch das Schlüsselwort `void` anstelle einer Datentypangabe gekennzeichnet. Beim Aufruf einer solchen Methode muß dann kein Speicherplatz für irgendwelche Ergebnisse reserviert werden.

Hinweis: `void` ist kein Datentyp im eigentlichen Sinn, sondern steht stellvertretend für das Nichtvorhandensein von typisierbaren Daten. Damit ist `void` auch keine zulässige Typangabe für Daten bzw. Variablen.

Beispiel 12:

```
int  beispielMethode1 (String eingabeText);  // Parameter und Resultat
void beispielMethode2 (char eingabeZeichen, int eingabeWert); // kein Resultat
void beispielMethode3 ();                    // kein Parameter und kein Resultat
```

Der Methodenkopf allein sagt noch nichts darüber aus, was bei einem Aufruf der Methode eigentlich geschehen soll. Der *Methodenrumpf* enthält die Anweisungen, die bei einem Aufruf der Methode ausgeführt werden. Ein Methodenrumpf wird als eine mit geschweiften Klammern eingefaßte Anweisungsfolge geschrieben. Der Rumpf wird unmittelbar nach dem Methodenkopf angegeben:

Ergebnistyp Methodenname (*Datentyp Name*, *Datentyp Name*, ...) {
Methodenrumpf
}

Die in Beispiel 2 deklarierte Methode `main()` kann demnach folgendermaßen in Methodenkopf und Methodenrumpf aufgeteilt werden:

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

Im Methodenrumpf werden die formalen Parameter aus dem Methodenkopf prinzipiell wie lokale Variablen verwendet, d.h. der Gültigkeitsbereich der formalen Parameter ist stets der gesamte Methodenrumpf. Aufgrund der fehlenden Verschattung in Java ist es deshalb nicht erlaubt, eine lokale Variable mit dem selben Bezeichner zu deklarieren wie einen formalen Parameter dieser Methode.

2.6.4 Aufruf von Methoden

Um eine Methode innerhalb eines Programms zu verwenden, ist ein *Aufruf* der Methode erforderlich. Dazu wird der Name der Methode angegeben, gefolgt von der Liste der aktuellen Parameter (in runden Klammern).

Methodenname (*Liste der aktuellen Parameter*)

Jeder aktuelle Parameter ist ein Ausdruck, d.h. er repräsentiert einen Wert bzw. die Berechnung eines Werts. Beim Methodenaufruf werden die Werte der aktuellen Parameter auf die formalen Parameter an der jeweils entsprechenden Position übertragen.
Ein Methodenaufruf kann auch frei von aktuellen Parametern sein; in diesem Fall wird nach dem Methodennamen nur ein leeres Klammerpaar geschrieben.

Liefert eine Methode einen Ergebniswert, dann ist ein Aufruf dieser Methode als Ausdruck zu interpretieren, dessen Typ gleich dem Resultattyp der zugehörigen Methodendeklaration ist. Methoden ohne Rückgabewert (`void`) entsprechen dagegen einer einzelnen Anweisung.

Beispiel 13:

```
int ergebnis1;
ergebnis1 = beispielMethode1(eineStringVariable);
beispielMethode2('a', 1234);
beispielMethode3();
```

Hinweis: Methoden mit Rückgabewert können auch in Anweisungsform aufgerufen werden. Der Ergebniswert wird in diesem Fall zwar zurückgeliefert, steht aber dann nicht mehr zur weiteren Verwendung zur Verfügung. Eine solcher Aufruf ist daher nur für Methoden mit Seiteneffekten sinnvoll.

```
beispielMethode1(eineStringVariable);
```

Wird ein Methodenaufruf ausgeführt, so werden zunächst die Ausdrücke der aktuellen Parameter ausgewertet (von links nach rechts). Ihre Werte werden als entsprechende formale Parameter übergeben, und der Rumpf der Methode wird angesprungen und ausgeführt. Nach der Abarbeitung des Rumpfes wird die Programmausführung unmittelbar „hinter“ dem Methodenaufruf fortgesetzt: Falls der Methodenaufruf einen Ergebniswert geliefert hat, so kann dieser Wert in der Anweisung verwendet werden, in der der Methodenaufruf stattfand. Ansonsten wird die nächste Anweisung ausgeführt.

2.6.5 Rückgabewerte

Die Wertrückgabe aus einer Methode erfolgt über die `return`-Anweisung. Dazu wird nach dem Schlüsselwort `return` ein dem Ergebniswert entsprechender Ausdruck (mit dem in der Methodendeklaration festgelegten Datentyp) angegeben. Liefert eine Methode keinen Rückgabewert (`void`), dann besteht die Anweisung nur aus dem Schlüsselwort `return`.

`return` *Rückgabewert*`;`
`return;`

Die Ausführung einer `return`-Anweisung beendet die Abarbeitung des Methodenrumpfs, d.h. der Methodenrumpf wird verlassen. Die Ergebniswerte werden, sofern vorhanden, an den Aufrufer der Methode zurückgegeben.

Innerhalb eines Methodenrumpfs können auch mehrere `return`-Anweisungen auftreten. Wichtig ist nur, daß die letzte während eines Methodenaufrufs ausgeführte Anweisung immer eine `return`-Anweisung sein muß. So wird sichergestellt, daß jeder Methodenaufruf ein definiertes Ergebnis liefert. Insbesondere darf das Ende des Methodenrumpfs bei der Programmausführung nicht überschritten werden.
Eine Ausnahme stellen dabei die `void`-Methoden dar. Da sie keinen Rückgabewert liefern müssen, ist eine abschließende `return`-Anweisung nicht zwingend notwendig und kann weggelassen werden. Erreicht der Programmablauf das Ende des Methodenrumpfs einer `void`-Methode, so wird implizit eine `return`-Anweisung ohne Rückgabewert ausgeführt.

Beispiel 14:

```
class BeispielMethoden {
  static int anzahlErhoehungen = 0;   // Klassenvariable (fuer Seiteneffekt)

  // eine Methode ohne Resultat und ohne Parameter
  static void tuNichts () {
    return;                   // Rueckkehr zum Aufrufer
  }

  // eine Methode zum Erhoehen des Eingabewerts um 1
  static int erhoehe (int eingabe) {
    anzahlErhoehungen = anzahlErhoehungen + 1;   // Seiteneffekt!
    return (eingabe + 1);                        // Wertrueckgabe
  }

  // die main-Methode
  public static void main (String[] args) {
    int ergebnis;

    ergebnis = erhoehe(1);    // ein Methodenaufruf mit Rueckgabewert
    tuNichts();               // ein Methodenaufruf als Anweisung
    erhoehe(2);               // Verwerfen des Aufrufresultats
                              // (Aufruf nur wegen Seiteneffekt)

    // eine return-Anweisung fuer die main-Methode kann hier entfallen
    // (wie bei allen void-Methoden)
  }
}
```

2.6.6 Überladung von Methoden

Innerhalb einer Klassendeklaration können mehrere gleichnamige Methoden deklariert werden. Die Deklarationen gleichnamiger Methoden müssen sich aber in Anzahl, Reihenfolge und/oder Datentyp ihrer formalen Parameter unterscheiden, damit beim Aufruf einer derartigen Methode eine eindeutige Zuordnung zu dem passenden Methodenrumpf möglich ist. Enthält eine Klasse mehrere Methoden gleichen Namens, so spricht man von einer *Überladung* des Methodennamens.
Die Namen der formalen Parameter spielen bei der Unterscheidung überladener Methoden keine Rolle. Dies liegt daran, daß zum Aufruf einer (überladenen) Methode nur der Methodenname und eine Liste der aktuellen Parameter angegeben werden. Diese Liste enthält nur Werte und Informationen über deren Datentypen, aber keine Parameterbezeichner. Eine Unterscheidung der überladenen Methoden anhand von Parameterbezeichnern wäre zum Zeitpunkt des Aufrufs also nicht möglich. d.h. es könnte nicht eindeutig entschieden werden, welcher Methodenrumpf auszuführen ist.
Auch die Ergebnisdatentypen sind als Unterscheidungskriterium für Methoden nicht geeignet, da das Ergebnis eines Methodenaufrufs nicht zwangsläufig weiterverwendet werden muß.

Beispiel 15:

```
double summe (byte a, byte b);
double summe (double a, double b);
double summe (double a, double b, double c);
double summe (double[] summanden);

byte summe (byte c, byte d);  // FEHLER: Datentypen in Parameterliste
                              //         identisch mit 1. Deklaration!
```

2.6.7 Rekursion

Eine Methode kann sich selbst aufrufen, und zwar direkt oder auch indirekt über andere Methoden. In diesem Fall spricht man von *Rekursion*, der Aufruf selbst wird als *Rekursionsschritt* bezeichnet. Bei der Rekursion ist vor allem darauf zu achten, daß die Ausführung in jedem Falle terminiert, d.h. die Methode darf sich selbst nur endlich oft aufrufen. Dies wird durch die Festlegung einer *Terminierungsbedingung* erreicht, die innerhalb der Methode überprüft werden muß. Ein weiterer Rekursionsschritt wird nur dann ausgeführt, wenn die Terminierungsbedingung noch nicht erfüllt ist.
Rekursion ist ein zentrales Konzept der funktionalen Programmierung, das als Gegenstück zum prozeduralen Schleifen-Konzept (siehe 2.8.5) gesehen werden kann. In Java-Programmen können Rekursion und Schleifen gleichermaßen verwendet werden.

Beispiel 16:

```
// rekursive Implementierung der Fakultaetsfunktion
static int fakultaet (int i) {
  if (i == 0)                  // Terminierungsbedingung
    return 1;                  // Bedingung erfuellt: Terminierung
  else
    return i * fakultaet(i-1); // Bedingung nicht erfuellt: rekursiver Aufruf
}
```

2.6.8 Die Methode `main`

Um ein übersetztes Java-Programm auszuführen, wird die Codedatei der entsprechenden Klasse von der JVM eingelesen und interpretiert. Eine Klasse enthält aber nur Methoden und Variablen; außerhalb von Methodendeklarationen sind keine ausführbaren Programmanweisungen (siehe 2.8) zulässig. Deshalb muß das auszuführende Programm (bzw. die Klasse) eine Methode bereitstellen, die von der virtuellen Maschine als Einstiegspunkt für die Programmausführung verwendet werden kann.
Programme, die als eigenständige Anwendungen mit `java` ausgeführt werden, stellen dafür eine Methode namens `main` zur Verfügung. Die Interpretation des Programmcodes beginnt dann stets mit einem Aufruf der Methode `main`.
Der Methodenkopf der `main`-Methode ist durch die Sprachspezifikation eindeutig festgelegt:

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

`main` muß eine Klassenmethode (mit dem Modifikator `static`, siehe 2.1) sein, da eine Instanz der Klasse erst während des Programmablaufs, also nach dem Aufruf von `main`, gebildet werden könnte.
Der Modifikator `public` (siehe Kapitel 5) erlaubt den Aufruf von `main` auch von außerhalb der Klasse ohne Einschränkungen. Damit ist sichergestellt, daß auch die JVM (`java`) auf die `main`-Methode zugreifen kann.

Der formale Parameter `args` stellt alle beim Programmaufruf evtl. zusätzlich angegebenen Werte in Textform bereit. So können auch Programme parametrisiert werden.
Der Name des formalen Parameters kann, wie bei anderen Methodendeklarationen auch (siehe 2.6.6), prinzipiell beliebig gewählt werden, da für die korrekte Zuordnung von Methodenaufruf und Methodenrumpf nur der Datentyp des Parameters von Bedeutung ist.

Hinweis: Falls ein Programm nur als Applet ausgeführt werden soll, so ist keine `main`-Methode erforderlich. Die Erstellung und Ausführung von Applets wird zu einem späteren Zeitpunkt noch gesondert behandelt.

2.7 Operatoren und Ausdrücke

Ein Ausdruck ist, wie bereits in Abschnitt 2.6.2 erwähnt wurde, eine Repräsentation für einen Wert bzw. die Berechnung eines Werts zur Laufzeit. Ausdrücke können in verschiedenen Ausprägungen auftreten. Einfach aufgebaute Ausdrücke sind Variablenbezeichner, Methodenaufrufe, Literale oder Schlüsselworte wie `null`.
Mit *Operatoren* können aus Teilausdrücken komplexere Ausdrücke zusammengesetzt werden. Dabei werden die Teilausdrücke bzw. deren Werte als *Argumente* des Operators behandelt. Jeder (Teil-)Ausdruck hat einen bestimmten Typ.
In Java gibt es eine Reihe von Operatoren für grundlegende (Rechen-)Operationen. Operatoren existieren z.B. für die Grundrechenarten, für Gleichheitsüberprüfungen und für logische Verknüpfungen.

Es gibt Java-Operatoren mit unterschiedlichen *Stelligkeiten*. Mit der Stelligkeit eines Operators wird die Anzahl der Teilausdrücke bzw. *Argumente* bezeichnet, die bei der dem Operator zugeordneten Operation verwendet bzw. verknüpft werden. Ein *binärer* (zweistelliger) Operator verbindet z.B. zwei Teilausdrücke miteinander, während ein *unärer* (einstelliger) Operator auf einen einzelnen Teilausdruck angewendet wird.
Auch die Position eines Operators relativ zu seinen Argumenten ist normalerweise genau festgelegt. Bei der sog. *Präfix*-Notation wird der Operator vor seinen Argumenten angegeben, bei der *Postfix*-Schreibweise dahinter. Ein *Infix*-Operator dagegen wird zwischen seine Argumente gestellt. In Java wird für alle mehrstelligen Operatoren die Infix-Schreibweise verwendet.

Beispiel 17:

```
6 + (-5)  // hier ist das '-' ein unaerer Operator
6 - 5     // hier ist das '-' ein binaerer Operator
```

Die Auswertungsreihenfolge für Teilausdrücke ergibt sich aus den unterschiedlichen „Bindungsstärken“ bzw. *Präzedenzen* und der *Assoziativität* der einzelnen Operatoren (siehe 2.7.11), falls sie nicht durch Klammerung explizit festgelegt ist. Wie bei Termen in applikativen Sprachen läßt sich auch bei Ausdrücken die Auswertung in Form eines *Operatorbaums* darstellen, wobei die Wurzel des Baums der Auswertung des Gesamtausdrucks entspricht und die einzelnen Teilbäume die entsprechenden Teilauswertungen repräsentieren.

2.7.1 Arithmetische Operatoren

Arithmetische Operatoren sind verfügbar für die vier Grundrechenarten und die Restbildung:

Addition:	`+`
Subtraktion:	`-`
Multiplikation:	`*`
Division:	`/`
Restbildung:	`%`

Die arithmetischen Operatoren sind mit allen numerischen Grundtypen verwendbar. Dabei ergibt sich der Datentyp des arithmetischen Ausdrucks aus den Typen seiner Teilausdrücke. Um den Typ des Gesamtausdrucks eindeutig festlegen zu können, empfiehlt es sich, möglichst für alle Teilausdrücke denselben

Datentyp zu verwenden. Die dafür evtl. erforderliche Typanpassung wird in Abschnitt 2.7.10 genauer erläutert.
Zu beachten ist, daß bei der ganzzahligen Division der Betrag des Ergebnisses stets abgerundet wird, d.h. ein evtl. vorhandener nicht-ganzzahliger Divisionsrest wird nicht berücksichtigt.

Beispiel 18:

```
// Grundrechenarten fuer ganze Zahlen:
int i = 3;

i = i + 1;          // Ergebnis: i erhaelt den Wert 4
i = 3 * 5;          // Ergebnis: i erhaelt den Wert 15
i = -86 - (-5);     // Ergebnis: i erhaelt den Wert -81
i = 11 / 2;         // Ergebnis: i erhaelt den Wert 5 (Abrundung !)

// Restbildung:
i = 7 % 3;          // Ergebnis: i erhaelt den Wert 1
i = -7 % 3;         // Ergebnis: i erhaelt den Wert -1

// Grundrechenarten fuer Gleitkomma-Zahlen:
double x = 3.14;

x = x + 1.00;       // Ergebnis: x erhaelt den Wert 4.14
x = 3.0 * 5.1;      // Ergebnis: x erhaelt den Wert 15.3
x = -86.0 - (-5.1); // Ergebnis: x erhaelt den Wert -80.9
x = 11.0 / 2.0;     // Ergebnis: x erhaelt den Wert 5.5

// Restbildung:
x = 7.14 % 3.00;    // Ergebnis: x erhaelt den Wert 1.14
```

2.7.2 Bitweises Verschieben

Alle Operationen für Bit-Manipulation sind nur in Verbindung mit ganzzahligen Ausdrücken verwendbar. Die Wirkungsweise dieser Operatoren wird am besten deutlich, wenn man ganzahlige Werte in *Binärdarstellung* (Zahldarstellung zur Basis 2) betrachtet. Ein Wert `a` vom Typ `int` z.B. läßt sich dann darstellen als $a_{32}a_{31}\ldots a_1$ ($a_i \in 0,1$), wobei noch ein `B` an die Wertdarstellung angehängt werden kann, um die Binärdarstellung zu kennzeichnen.

Die Binärdarstellung vorzeichenbehafter ganzzahliger Datentypen (mit n Bit) folgt in Java den Regeln der sog. *2-Komplement-Bildung*. Ein negativer Wert wird durch eine 1 im höchstwertigen Bit a_n gekennzeichnet, und die Bits a_{n-1} bis a_1 enthalten die Binärdarstellung des Werts, den man zu -2^{n-1} addieren muß, um auf den Gesamtwert zu kommen.
Damit lassen sich alle negativen Werte von -2^{n-1} bis -1 eindeutig darstellen. Betrachtet man z.B. den Typ `byte` (8 Bit), so ergeben sich für die Zahlen $-2^7 (= -128)$ bis -1 die Binärdarstellungen `10000000B` bis `11111111B`.

Mit den Operatoren `<<`, `>>` und `>>>` kann ein ganzzahliger Wert bitweise nach links oder rechts verschoben werden. Der Wert des Teilausdrucks rechts vom Operator gibt an, um wieviele Bit-Stellen der Wert des linken Teilausdrucks verschoben werden soll.

bitweise Schieben nach links:	`<<`
bitweise Schieben nach rechts (mit Vorzeichen):	`>>`
bitweise Schieben nach rechts (ohne Vorzeichen):	`>>>`

Bei `<<` wird von rechts mit 0-Bits aufgefüllt. Bei `>>>` wird von links mit 0-Bits aufgefüllt, bei `>>` mit dem Vorzeichen-Bit ('+' $\equiv$ '0', '-' $\equiv$ '1'). Der Operator `>>` übernimmt also das Vorzeichen des linken Arguments, d.h. wenn der Wert des linken Arguments von `>>` negativ ist, so ist das Ergebnis ebenfalls negativ. Bei `>>>` dagegen verliert ein negatives linkes Argument sein Vorzeichen (sofern nicht um 0 Bits verschoben wird).

Beispiel 19:

```
// Binaerdarstellung, gekennzeichnet durch nachgestelltes 'B':
```

```
// 9 = 00001001B, -9 = 11110111B (2-Komplement: -9 = -128 + 119)
byte b;

// bitweises Verschieben:
b = 9 << 2;     // Ergebnis: b erhaelt das Vierfache von 9, also 36,
                // denn  00001001B << 2 = 00100100B = 36
b = 9 >> 3;     // Ergebnis: b erhaelt das (abgerundete) Achtel von 9,
                // also 1, denn  00001001B >> 3 = 00000001B = 1
b = -9 >> 3;    // Ergebnis: b erhaelt den Wert -2,
                // denn  11110111B >> 3 = 11111110B = -2
b = 9 >>> 3;    // Ergebnis: b erhaelt das (abgerundete) Achtel von 9,
                // also 1, denn  00001001B >>> 3 = 00000001B = 1
```

Bitweise Operationen auf 8-bit- oder 16-bit-Argumenten werden in Java implizit mit 32-bit durchgeführt, d.h. Argumente eines Datentyps wie z.B. `byte` werden vor und nach der Operationsausführung entsprechend angepaßt. Aufgrund dieser Umrechnungen kann es zu unerwarteten Ergebnissen oder gar zu Fehlern kommen, z.B. durch eine Überschreitung des gültigen Wertebereichs für den Gesamtausdruck. In diesen Fällen bricht `javac` mit einer Fehlermeldung ab.

```
byte b;

b = -9 >>> 3;   // FEHLER: 11110111B >>> 3 = 00011110B = 28 (erwartet)
                //         11111111111111111111111110111B >>> 3 =
                //         00011111111111111111111111110B =
                //         = 11111110B (als byte-Wert) = -2 (nicht 28)!!!
```

Diese Problematik läßt sich durch eine explizite Typanpassung (siehe 2.7.10) zumindest teilweise beseitigen. Generell sollte aber bitweises Schieben nur in solchen Fällen verwendet werden, wo es zum besseren Verständnis des Programms beiträgt. Meistens sind arithmetische Operationen anschaulicher und sollten daher bevorzugt zum Einsatz kommen.

2.7.3 Bitweise logische Verknüpfungen

In Java existieren vier Operatoren zur bitweise logischen Verknüpfung von ganzzahligen Werten:

bitweise AND-Verknüpfung:	`&`
bitweise OR-Verknüpfung:	`|`
bitweise XOR-Verknüpfung:	`^`
bitweise logisches Komplement:	`~`

Der unäre Operator `~` negiert einen Wert bitweise, d.h. ein 0-Bit wird zu einem 1-Bit, und umgekehrt. `~` ist also ein bitweises boolesches `NOT` ($\neg a_i$). Die binären Operatoren `&`, `|` und `^` entsprechen einem bitweisen `UND` ($a_i \wedge b_i$), `ODER` ($a_i \vee b_i$) bzw. `XOR` ($(a_i \wedge \neg b_i) \vee (\neg a_i \wedge b_i)$) zweier Werte (gleichen Typs).

Beispiel 20:

```
byte b = 9;     // 9 = 00001001B

// bitweise AND- / OR- / XOR-Verknuepfung:
b = -123 & 3;   // Ergebnis: b erhaelt den Wert 1,
                // denn  10000101B AND 00000011B = 00000001B = 1
b = 12 | 5;     // Ergebnis: b erhaelt den Wert 13,
                // denn  00001100B OR 00000101B = 00001101B = 13
b = 12 ^ 5;     // Ergebnis: b erhaelt den Wert 9,
                // denn  00001100B XOR 00000101B = 00001001B = 9

// bitweise Negierung:
byte b = ~0;    // Ergebnis; b erhaelt den Wert -1,
                // denn  NOT 00000000B = 11111111B = -1
```

Bitweise logische Verknüpfungen sind bzgl. impliziter Umrechnungen im Gegensatz zu den bitweisen Verschiebeoperationen unkritisch. Allerdings sollte auch bei der Verwendung dieser Operatoren zumindest ein beschreibender Kommentar erfolgen, um das Verständnis des Programms zu erleichtern.

2.7.4 Logische Verknüpfungen

Für den Umgang mit booleschen Ausdrücken bietet Java die folgenden logischen Operationen:

AND-Verknüpfung:	`&` und `&&`
XOR-Verknüpfung:	`^`
OR-Verknüpfung:	`|` und `||`
logisches Komplement:	`!`

Der Unterschied zwischen `&` und `&&` bzw. `|` und `||` liegt darin, daß `&` und `|` *strikte* Operatoren sind. Ein Operator heißt *strikt*, wenn bei seiner Auswertung stets alle seine Teilausdrücke ausgewertet werden. Die Operatoren `&&` und `||` sind die einzigen nicht-strikten binären Operatoren in Java.
Bei `&&` und `||` wird der rechte Teilausdruck nicht ausgewertet, wenn durch den Wert des linken Teilausdrucks das Gesamtergebnis bereits festgelegt ist. Dies tritt ein, falls bei `&&` die Auswertung des linken Teilausdrucks `false` ergibt, oder bei `||` der linke Teilausdruck als Ergebnis `true` liefert.

Beispiel 21:

```
boolean b = true;
int i = 0;

// strikte logische Operatoren:
b = ( 1 > 2 ) & ( -7 < 18 );       // Ergebnis: b erhaelt den Wert "false"
b = ( 2 > 1 ) | b;                 // Ergebnis: b erhaelt den Wert "true"
b = ( 1 <= 2 ) ^ ( 1 > 2 );        // Ergebnis: b erhaelt den Wert "true"
b = !b;                            // Ergebnis: b erhaelt den Wert "false"

// nicht-strikte logische Operatoren:
b = ( 1 > 2 ) && ( 1 != 0 );       // Ergebnis: b erhaelt den Wert "false"
b = ( 2 > 1 ) || ( (1 / i) == 0 ); // Ergebnis: b erhaelt den Wert "true"
                                   // Strikte Auswertung würde ergeben:
                                   //   FEHLER: Division durch 0!
```

2.7.5 Zuweisungsoperatoren

Die Zuweisung ist in Java die einzige Möglichkeit, einen Wert dauerhaft in einer Variablen bzw. in dem für die Variable reservierten Speicherbereich abzulegen. Eine Zuweisung ist also das Schreiben eines Werts in eine Variable. Die Form der Zuweisung ist folgendermaßen festgelegt:

Variable `=` *Ausdruck*

Neben dem einfachen Zuweisungsoperator `=` gibt es in Java noch eine Reihe von zusätzlichen Zuweisungsoperatoren. Es handelt sich dabei um zusammengesetzte Zuweisungsoperatoren, die neben der reinen Zuweisung noch zusätzliche Operationen durchführen.
Zusammengesetzte Zuweisungsoperatoren sind als abkürzende Notation in Verbindung mit allen zuvor vorgestellten binären arithmetischen und bitweisen Operatoren einsetzbar, ausgenommen die nicht-strikten Operatoren `&&` und `||`. Java bietet somit die folgenden zusammengesetzten Zuweisungsoperatoren:

arithmetisch:	`+=`	`-=`	`*=`	`/=`	`%=`
bitweise verschieben:	`<<=`	`>>=`	`>>>=`		
(bitweise) logisch verknüpfen:	`&=`	`|=`	`^=`		

Dabei wird ein Ausdruck der Form

$$Variable_x = Variable_x \circ Ausdruck$$

abgekürzt geschrieben als

$$Variable_x \circ = Ausdruck$$

wobei $\circ$ für einen strikten binären arithmethischen oder bitweisen Operator steht. Soll also eine Operation $\circ$ auf eine Variable angewendet werden, und das Ergebnis anschließend dieser Variablen selbst zugewiesen werden, dann kann die abkürzende Schreibweise verwendet werden. Die zusammengesetzten Zuweisungen können auch als Operatoren zur direkten Veränderung von Variablenwerten gesehen werden.

Beispiel 22:

```
int i;
boolean b;

// Zuweisungsoperatoren bei numerischen Variablen:
i = 3;       // herkoemmliche Zuweisung
i += 4;      // dieselbe Wirkung wie i = i + 4;
i -= 2;      // dieselbe Wirkung wie i = i - 2;
i *= 12;     // dieselbe Wirkung wie i = i * 12;
i /= 3;      // dieselbe Wirkung wie i = i / 3;
i %= 5;      // dieselbe Wirkung wie i = i % 5;

// Zuweisungsoperatoren bei booleschen Variablen:
b = false;   // herkoemmliche Zuweisung
b |= true;   // dieselbe Wirkung wie b = b | true;
b &= false;  // dieselbe Wirkung wie b = b & false;
```

Eine Zuweisung ist selbst ein Ausdruck und kann deshalb Teil eines größeren Ausdrucks sein. Die Auswertung erfolgt in diesem Fall von rechts nach links. Der Wert eines solchen Ausdrucks ist der zugewiesene Wert.

Beispiel 23:

```
int i = 3;
int k = 2;
int r;

int s = 7 + ( r = ( k *= i )); // Ergebnis: k erhaelt den Wert 6,
                               //           r den Wert 6 und s den Wert 13.
                               // Das gleiche Ergebnis ergaebe sich bei:
                               // int s = 7 + r = k *= i;
```

Hinweis: Eine Verschachtelung von mehreren Zuweisungen innerhalb eines Ausdrucks ist normalerweise nicht zu empfehlen, da ein solcher Ausdruck schnell unübersichtlich wird. Statt dessen sollte eher eine Folge von mehreren Zuweisungsanweisungen verwendet werden.

2.7.6 Vergleichsoperatoren

Das Ergebnis eines Ausdrucks mit einem binären Vergleichsoperator ist ein boolescher Wert, d.h. ein derartiger Ausdruck ist vom Typ `boolean`. Ausdrücke des Typs `boolean` werden oft auch als *Bedingungen* bezeichnet und sind wesentlicher Bestandteil einiger wichtiger Anweisungsarten (siehe 2.8).

Gleichheit:	`==`
Ungleichheit:	`!=`
echt kleiner:	`<`
kleiner oder gleich:	`<=`
größer oder gleich:	`>=`
echt größer:	`>`

Die Vergleichsoperatoren sind für verschiedene Argument-Datentypen definiert. Die Operatoren `<`, `<=`, `>` und `>=` sind auf die Datentypen beschränkt, auf denen eine *Ordnung* definiert ist. In Java sind dies nur die numerischen Datentypen.
Die Überprüfung auf Gleichheit (`==`) bzw. Ungleichheit (`!=`) ist dagegen für alle Datentypen möglich, wobei bei Referenztypen nur die Referenzen überprüft werden, ob sie auf dieselbe Speicherposition verweisen.

Beispiel 24:

```
int i = 3;
boolean b;

// Vergleiche:
b = (i == -4);      // Ergebnis: b erhaelt den Wert "false"
b = (i != -4);      // Ergebnis: b erhaelt den Wert "true"
b = (3 > 5);        // Ergebnis: b erhaelt den Wert "false"
b = (5 >= 3);       // Ergebnis: b erhaelt den Wert "true"
b = (i < 0);        // Ergebnis: b erhaelt den Wert "false"
b = (i <= 0);       // Ergebnis: b erhaelt den Wert "false"
```

Wichtig: Der Test auf Gleichheit wird in Java mit einem *doppelten* Gleichheitszeichen (==) geschrieben. Die einfache Zuweisung dagegen wird durch ein einfaches Gleichheitszeichen (=) ausgedrückt. Das versehentliche Weglassen eines = bei der Gleichheitsüberprüfung gehört zu den häufigsten Programmierfehlern, die außerdem u.U. (Gleichheitsprüfung für eine Variable vom Typ `boolean`) nur sehr schwer zu entdecken sind!

2.7.7 Inkrement und Dekrement

Numerische Variablen können in einem Ausdruck mit einem vor- oder nachgestellten `++` oder `--` vorkommen, wobei dabei als Seiteneffekt ihr Wert um 1 erhöht oder erniedrigt wird. Die Position des Operators zur Variablen bestimmt die zeitliche Reihenfolge zwischen Seiteneffekt und der weiteren Verwendung des Wertes:
Ein vorangestelltes `++` oder `--` (Präfix-Schreibweise) erhöht bzw. erniedrigt die Variable vor ihrer Auswertung, so daß innerhalb der weiteren Auswertung des Ausdrucks bereits der neue Wert der Variablen verwendet wird. Man spricht hier auch von einem *Präinkrement* bzw. *Prädekrement*.
Ein nachgestelltes `++` oder `--` (Postfix-Schreibweise) dagegen wird erst nach der Auswertung der Variablen durchgeführt, d.h. es wird der ursprüngliche Wert der Variablen als Auswertungsergebnis geliefert. Dies wird dann als *Postinkrement* bzw. *Postdekrement* bezeichnet.

Beispiel 25:

```
short s = 17;
short t;

t = ++s;    // Praeinkrement: t erhaelt den Wert 18; s hat nun den Wert 18
t = s++;    // Postinkrement: t erhaelt den Wert 18; s hat nun den Wert 19
t = --s;    // Praedekrement: t erhaelt den Wert 18; s hat nun den Wert 18
t = s--;    // Postdekrement: t erhaelt den Wert 18; s hat nun den Wert 17
```

Inkrement und Dekrement können auch als Kurzform für spezielle Zuweisungsanweisungen betrachtet werden. Die folgenden Anweisungen z.B. sind völlig gleichwertig:

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

Hinweis: Anders als bei der Verwendung in Ausdrücken besteht hier kein Unterschied zwischen Prä- und Postinkrement.

2.7.8 Der Konditionaloperator

Der Konditionaloperator „`? :`“ ist der einzige *ternäre* (dreistellige) Operator in Java:

Bedingung `?` *Ausdruck_1* `:` *Ausdruck_2*

Bedingung ist ein Ausdruck vom Typ `boolean`. Ergibt ihre Auswertung `true`, so ist der Wert des Gesamtausdrucks der Wert von *Ausdruck_1* (der Ausdruck nach `?`). Andernfalls ist der Wert von *Ausdruck_2* (der Ausdruck nach `:`) das Gesamtergebnis.

Der Konditionaloperator ist, ebenso wie `&&` oder `||`, nicht-strikt: Wird *Bedingung* zu `true` ausgewertet,

so wird *Ausdruck_2* nicht ausgewertet, ergibt *Bedingung* den Wert `false`, so wird *Ausdruck_1* nicht ausgewertet.

Beispiel 26:

```
boolean istWeiblich = true;
String anrede;

anrede = istWeiblich ? "Frau" : "Herr";
```

Der Konditionaloperator erlaubt es, einen Teilausdruck in Abhängigkeit vom Auswertungsergebnis eines anderen Teilausdrucks (*Bedingung*) auszuwerten. Bei komplexeren Unterscheidungen werden jedoch meist bedingte Anweisungen (siehe 2.8.2) verwendet.

2.7.9 Verknüpfung von Zeichenketten

Zeichenketten nehmen in Java eine Sonderstellung ein: Der Datentyp `String` gehört zwar zu den Referenztypen, aber trotzdem gibt es für ihn sowohl eine Literaldarstellung als auch einen speziellen Verknüpfungsoperator (ähnlich wie bei Grundtypen).
Der Verknüpfungsoperator `+` dient dazu, zwei Zeichenketten (bzw. Kopien der Zeichenketten) miteinander zu verbinden. Er kann auch in Verbindung mit dem Zuweisungsoperator in zusammengesetzten Zuweisungen benutzt werden.

Beispiel 27:

```
String kurzerText = "Das ist ein kurzer Text";
String langerText = kurzer Text + ", der jetzt laenger wird";
langerText += ", und jetzt langt's.";
```

Der Verknüpfungsoperator kann auch für die Textdarstellung von anderen Werten verwendet werden. Ist nur eins der Argumente ein `String`-Wert, so wird implizit der Wert des anderen Arguments in eine Zeichenketten-Repräsentation umgewandelt.

```
final double PI = 3.141592;
String text = "Der Wert von PI ist " + PI;
```

2.7.10 Konvertierung von Datentypen

In streng typisierten Sprachen wie Java ist die Festlegung von Datentypen wesentlicher Bestandteil der Programmierung. Zur flexibleren Behandlung von Daten existiert oft eine Möglichkeit, Daten eines bestimmten Typs an einen anderen Datentyp anzupassen. Diese explizite Umwandlung wird als *Typkonvertierung* bzw. *Cast* bezeichnet.
Der unäre Cast-Operator in Java hat folgende Form:

`(` *Datentyp* `)` *Ausdruck*

Dabei wird der Wert, der sich bei der Auswertung von *Ausdruck* ergibt, in einen Wert des Datentyps *Datentyp* konvertiert.

Casts können nicht zwischen beliebigen Typen durchgeführt werden. Eine Konvertierung von Grundtypen zu Referenztypen oder umgekehrt ist mit dem Cast-Operator nicht möglich. Einzige Ausnahme ist die Konvertierung in Zeichenketten (`String`). Eine Konvertierung in einen `String` ist für alle Grundtypen möglich, ebenso wie für alle Referenztypen.
Auch bei korrekten Konvertierungen ist immer zu berücksichtigen, daß ein Cast oft nur eine näherungsweise Umwandlung bewirkt. So geht z.B. bei der Umwandlung eines `double`-Ausdrucks in einen `int`-Wert zwangsläufig der nicht-ganzzahlige Rest des Ausdrucks verloren.

Beispiel 28:

```
// Genauigkeitsgewinn durch Typkonvertierung
int intVariable = 3;
int intFaktor = 12;
int intWert = (intVariable / 4) * intFaktor; // intWert erhaelt den Wert 0
```

```
intwert = (int) ((((double) intVariable) / 4.0) *
                 ((double) intFaktor));      // intwert erhaelt den Wert 9

// Rundungsfehler durch Typkonvertierung
final double PI = 3.141592;
int pi = (int) PI;                // pi erhaelt den Wert 3
```

2.7.11 Präzedenzen und Assoziativität

Zwischen den Operatoren gibt es eine Rangfolge bzgl. ihrer „Bindungsstärke“ bzw. *Präzedenz*. Die Präzedenz von Operatoren bestimmt dabei die Reihenfolge, in der die Operationen innerhalb eines Ausdrucks ausgewertet werden, sofern die Reihenfolge nicht durch Klammerung explizit festgelegt ist.

Hat ein Operator $\odot$ eine höhere Präzedenz als ein Operator $\oplus$, so bindet $\odot$ stärker als $\oplus$, d.h. ein Ausdruck der Form

$$expr_1 \oplus expr_2 \odot expr_3$$

wird unabhängig von der Assoziativität der Operationen so ausgewertet, als ob er

$$expr_1 \oplus (expr_2 \odot expr_3)$$

lauten würde.

Bei Operatoren mit gleicher Präzedenz wird die Auswertungsreihenfolge anhand der *Assoziativität* der Operatoren ermittelt. Ein Operator $\circ$ wird als *linksassoziativ* bezeichnet, wenn ein Ausdruck der Form

$$expr_1 \circ expr_2 \circ expr_3$$

genauso wie der geklammerte Ausdruck

$$(expr_1 \circ expr_2) \circ expr_3$$

ausgewertet wird, d.h. die Auswertung erfolgt von links nach rechts.
Analog dazu wird bei *rechtsassoziativen* Operatoren von rechts nach links ausgewertet: Ist $\circ$ ein rechtsassoziativer Operator, so wird

$$expr_1 \circ expr_2 \circ expr_3$$

ausgewertet wie der geklammerte Ausdruck

$$expr_1 \circ (expr_2 \circ expr_3)$$

In der obigen Darstellung wird ein binärer Operator in Infixnotation angenommen. Die Assoziativität ist in Java aber für alle Operatoren unabhängig von deren Stelligkeit festgelegt. Alle binären Operatoren mit Ausnahme der Zuweisungen sind linksassoziativ, die restlichen Operatoren rechtsassoziativ.

Tabelle der Operatoren (mit absteigender Präzedenz):

Präzedenz	Operator	Assoziativität	Beschreibung
	unäre Operatoren		
17	`++ --`	←	Postinkrement/-dekrement
16	`+ -`	←	Vorzeichen
15	`++ --`	←	Präinkrement/-dekrement
14	`~ !`	←	(bitweise) Komplementbildung
13	(*Datentyp*)	←	Typkonvertierung (Cast)
	binäre Operatoren		
12	`* / %`	→	Multiplikation, Division, Modulo
11	`+ -`	→	Addition, Subtraktion
10	`<< >> >>>`	→	bitweise Verschiebungen
9	`< > <= >=` `instanceof`	→	Vergleichsoperatoren[5]
8	`== !=`	→	(Un-)Gleichheitsüberprüfung
7	`&`	→	strikter AND-Operator
6	`^`	→	strikter XOR-Operator
5	`|`	→	strikter OR-Operator
4	`&&`	→	nicht-strikter AND-Operator
3	`||`	→	nicht-strikter OR-Operator
	ternäre Operatoren		
2	`? :`	←	Konditionaloperator
	Zuweisungen		
1	`= += -=` `*= /= %=` `<<= >>= >>>=` `&= ^= |=`	←	Zuweisungsoperatoren

→: linksassoziativ ←: rechtsassoziativ

Hinweis: Die Auswertungsreihenfolge kann für einen Ausdruck durch vollständige Klammerung explizit festgelegt werden. Durch die Ausnutzung von Präzedenzen wird oft ein Großteil dieser Klammern eingespart, ohne daß sich dadurch die Auswertungsreihenfolge ändert. Die Darstellung komplexer Ausdrücke wird allerdings durch Klammerung oft übersichtlicher. Daher sollten Klammern zur Gliederung eines Ausdrucks eingesetzt werden, auch wenn sie aufgrund der Operator-Präzedenzen weggelassen werden könnten.

```
a * b + c                       // Vollstaendige Klammerung: (a * b) + c
a * b * c * d + e * f * g * h   // Sinnvolle Klammerung:
                                // (a * b * c * d) + (e * f * g * h)
```

2.8 Anweisungen

Anweisungen dienen zur Steuerung des Ablaufs von Java-Programmen. Anders als Ausdrücke können Anweisungen keinem Datentyp zugeordnet werden, d.h. eine Anweisung liefert keinen Rückgabewert. Anweisungen stellen praktisch den Ablaufrahmen für die Auswertung von Ausdrücken dar.
Neben den schon zuvor verwendeten *Zuweisungsanweisungen* sind die *bedingten Anweisungen* und die *Wiederholungsanweisungen* die wesentlichen Anweisungsarten. In Verbindung mit diesen Konstrukten werden noch einige weitere Sprachelemente benötigt, die in den folgenden Abschnitten ebenfalls vorgestellt werden.
In Java existieren noch andere Anweisungen für spezielle Aufgaben wie z.B. zur Fehlerbehandlung. Diese Anweisungen werden erst in späteren Kapiteln eingeführt und daher hier nicht weiter behandelt.

Einzelne Anweisungen werden durch einen Strichpunkt abgeschlossen. Sind mehrere Anweisungen in einem Block (siehe 2.4.3) zusammengefaßt, ist mit dem Abschluß der letzten Anweisung innerhalb des Blocks der gesamte Block abgeschlossen, d.h. der Block selbst muß nicht mehr mit einem Strichpunkt abgeschlossen werden.

[5] Der Operator `instanceof` ist nur für Referenztypen (Objekte) definiert und wird deshalb erst im Kapitel 4 eingeführt.

Die Abarbeitung von Anweisungen erfolgt sequentiell, die Anweisungen werden also der Reihe nach ausgeführt. Kann eine Anweisung selbst weitere Anweisungen enthalten, so ist eine Schachtelung von Anweisungen möglich. In diesem Fall werden die Unteranweisungen nach den Regeln ausgeführt, die für die jeweilige umfassende Anweisung gelten. Ein Block von Anweisungen wird dabei wie eine einzelne Anweisung behandelt.

Wichtig: Anweisungen lassen oft nur einzelne Unteranweisungen zu. Soll eine Folge von mehreren Anweisungen als Unteranweisung angegeben werden, so muß diese Folge in einem Block geklammert werden.

2.8.1 Zuweisungsanweisungen

Wie in Abschnitt 2.7.5 bereits erwähnt werden Zuweisungen zur Speicherung von Werten benötigt. Neben der Speicherung von Werten innerhalb von Ausdrücken kann eine Zuweisung auch als eigenständige Anweisung betrachtet werden. Alle zuvor erwähnten Zuweisungsausdrücke können als Anweisung verwendet werden, auch zusammengesetzte Zuweisungen und die In- bzw. Dekrementierung von Variablen (siehe 2.7.7).

2.8.2 Einfache Fallunterscheidung

Die *einfache Fallunterscheidung* erlaubt, ausgehend von der Überprüfung einer Bedingung (Ausdruck des Typs `boolean`), die Auswahl einer Alternative. Je nachdem, ob die Bedingung erfüllt ist oder nicht, wird dann genau eine der Alternativen ausgeführt. Die Alternativen sind selbst wieder Anweisungen bzw. Anweisungsblöcke.
Die einfache Fallunterscheidung entspricht zwar in ihrer Struktur dem Konditionaloperator (siehe 2.7.8), ist jedoch das wesentlich ausdrucksstärkere der beiden Sprachkonstrukte. Mit der Fallunterscheidung kann nicht nur ein Wert bedingungsabhängig berechnet werden, sondern es ist möglich, den kompletten Programmablauf über Bedingungen zu steuern.
Die Syntax für eine einfache Fallunterscheidung lautet:

`if (` *Bedingung* `)` $Alternative_1$ `else` $Alternative_2$

Falls die Auswertung der *Bedingung* `true` liefert, gilt die Bedingung als erfüllt, und $Alternative_1$ wird ausgeführt. Ergibt *Bedingung* allerdings den Wert `false`, kommt stattdessen $Alternative_2$ zur Ausführung. Die jeweils nicht gewählte Alternative wird verworfen, d.h. sie wird nicht ausgeführt.

Für den Fall, daß nur bei Erfüllung der Bedingung überhaupt eine Anweisung ausgeführt werden soll, kann die zweite (leere) Alternative weggelassen werden. Dann entfällt auch das Schlüsselwort `else`, das zur Kennzeichnung des Beginns der zweiten Alternative dient:

`if (` *Bedingung* `)` *Anweisung*

Beispiel 29:

```
int wert = -7;
boolean istPositiv = ( wert > 0 );

// Fallunterscheidung für Ausdruecke:
if (istPositiv)
  System.out.println("Wert ist positiv");
else
  System.out.println("Wert ist nicht positiv");

// Entsprechende Ausgabe mit Konditionaloperator:
System.out.println("Wert ist " + (istPositiv ? "" : "nicht ") + "positiv");

// Fallunterscheidung für Anweisungen
// (mit Konditionaloperator nicht möglich):
if (wert < 0)
  System.out.println("Wert ist negativ");
else
  wert = -wert;
```

```
// Bedingte Ausführung (keine else-Alternative):
if (istPositiv) wert = -wert;

// verschachtelte Fallunterscheidung (mit Bloecken):
if (wert < 0)
  System.out.println( "Wert ist negativ" );
else {
  istPositiv = (wert != 0);
  if (wert < 10)
    System.out.println( "Wert ist zwischen 0 und 9" );
  else
    System.out.println( "Wert ist groesser als 9" );
}
```

2.8.3 Marken

Java bietet die Möglichkeit, sog. *Marken* bzw. *Labels* im Programm zu setzen, um einzelne Anweisungen mit einem Bezeichner zu versehen. Anhand einer solchen Marke kann die Ausführung der zugehörigen Anweisung dann entweder gezielt gestartet (siehe 2.8.4) oder abgebrochen werden (siehe 2.8.6).

Marke: *Anweisung*

Als *Marke* kann jeder beliebige, noch nicht anderweitig benutzte Java-Bezeichner eingesetzt werden. Es ist auch möglich, mehrere Marken für eine Anweisung festzulegen.

Beispiel 30:

```
  int wert = 0;

Marke1:                 // Marke1 bezeichnet die gesamte if-Anweisung
  if (wert != 0) {
  Marke2:               // Marke2 bezeichnet die Inkrement-Anweisung
    wert++;
  Marke3:               // Marke3 bezeichnet den folgenden Anweisungsblock
  Marke4:               // Marke4 bezeichnet denselben Anweisungsblock
    {
      wert--;
      wert--;
    }
  }
```

2.8.4 Mehrfache Fallunterscheidung

Die einfache Fallunterscheidung mit der `if`-Anweisung erlaubt maximal zwei Anweisungsalternativen, die über eine Bedingungsauswertung ausgewählt werden. Sollen mehr Alternativen erlaubt sein, so wird eine mehrfache Fallunterscheidung mit der `switch`-Anweisung ausgeführt.
Anstelle einer Bedingung wird bei der mehrfachen Fallunterscheidung zunächst ein nahezu beliebiger ganzzahliger Ausdruck (nur der Typ *long* ist hier nicht zulässig) ausgewertet. Dieser Wert wird dann im folgenden Anweisungsblock zur Fallunterscheidung verwendet.

`switch (` *Ausdruck* `) {` *Alternativen* `}`

Jede einzelne Alternative (eine oder mehrere Anweisungen) innerhalb der Anweisungsfolge *Alternativen* ist dabei mit einer speziellen Art von Marke gekennzeichnet, die einen ganzzahligen Wert enthält, der dann mit dem Auswertungsergebnis verglichen wird. Bei Übereinstimmung zwischen diesem Wert und dem Auswertungsergebnis wird die Programmausführung ab der entsprechenden Marke fortgesetzt.

`switch (` *Ausdruck* `) {`
`case` $Konstante_1$`:` $Alternative_1$
`case` $Konstante_2$`:` $Alternative_2$
...

`case` $Konstante_n$`:` $Alternative_n$
`default:` $Alternative_{n+1}$
}

Marken für die einzelnen Alternativen werden mit dem Schlüsselwort `case` und einem festen ganzzahligen Wert $Konstante_i$ festgelegt. Die Werte dürfen dabei nur als Konstanten (siehe 2.4.5), konstante Ausdrücke oder ganzzahlige Literale (siehe 2.3.2) angegeben werden. Außerdem ist darauf zu achten, daß die Werte paarweise verschieden sein müssen. Das Ergebnis der Auswertung von *Ausdruck* wird dann mit den Werten $Konstante_1$ bis $Konstante_n$ aller Marken verglichen.
Mit dem Schlüsselwort `default` kann eine zusätzliche Alternative angegeben werden, die ausgeführt wird, wenn keiner der zuvor angegebenen Fälle eingetreten ist. Dies entspricht der `else`-Alternative bei der `if`-Anweisung.

Ein wesentlicher Unterschied zur `if`-Anweisung liegt darin, daß die einzelnen Alternativen bei der `switch`-Anweisung sich nicht gegenseitig ausschließen, sondern in einer einzigen Anweisungsfolge zusammengefaßt sind. Die Marken geben lediglich die „Startposition“ für die einzelnen Alternativen an, wobei auch mehrere Marken zur selben Startposition gehören können.
Prinzipiell werden alle Anweisungen, die nach der gewählten Marke plaziert sind, ebenfalls abgearbeitet, auch wenn sie zu anderen Alternative gehören. Um einzelne Alternativen getrennt voneinander ausführen zu können, ist ein expliziter Abbruch der Ausführung der `switch`-Anweisung erforderlich. Dazu dient die *Abbruchanweisung* `break` (siehe 2.8.6). Die Anweisungsfolge wird dann von der gewählten Marke an bis zur ersten auftretenden `break`-Anweisung ausgeführt.

Beispiel 31:

```
byte wert = 5;

switch (wert+2) {
  case 2:
    System.out.println("Zwei");
    break;
  case 0:
    System.out.println("Null");
    break;
  case 1:
    System.out.println("Eins");
    break;
  case 3:
  case 3+1:
  case 6-1:
  case 6:
    System.out.println("zwischen 3 und 6");
    break;
  case 7:                           // (wert+2) = 7, d.h. diese Alternative
                                    // wird ausgewaehlt
    System.out.println("Sieben");   // Keine "break"-Anweisung zum Abschluss,
                                    // d.h. auch die naechste Anweisung wird
                                    // ausgeführt.
  default:
    System.out.println( "kleiner als 0 oder groesser als 6" );
}
```

Hinweis: Durch die Möglichkeit, Alternativen überlappend (ohne Trennung durch `break`) auszuführen, lassen sich mit der `switch`-Anweisung auch aufwendige Fallunterscheidungen oft sehr kompakt darstellen. Allerdings birgt diese kompakte Darstellung auch ein erhöhtes Fehlerrisiko in sich. Daher sollten überlappende Alternativen stets besonders sorgfältig auf ihre Korrektheit überprüft und ausreichend dokumentiert werden.

2.8.5 Schleifen

Wiederholungsanweisungen bzw. *Schleifen* erlauben die wiederholte Ausführung einer Anweisung bzw. Anweisungsfolge. Wie auch schon bei der `if`-Anweisung gilt auch hier wieder, daß Anweisungsfolgen, die

wiederholt ausgeführt werden sollen, in einem Block zusammengefaßt werden müssen. Die zu wiederholende Anweisung wird auch als *Schleifenkörper* bezeichnet.
In Java gibt es drei verschiedene Wiederholungsanweisungen, die sich in erster Linie dadurch unterscheiden, wie und wann die Anzahl der Wiederholungsschritte bzw. *Iterationen* festgelegt wird. Generell bieten alle Schleifen die Möglichkeit, eine *Wiederholungsbedingung* zu formulieren, die ähnlich wie die Terminierungsbedingung bei der Rekursion (siehe 2.6.7) vor bzw. nach jeder Iteration überprüft wird.

Eine einfache Schleife, bei der die Überprüfung der Wiederholungsbedingung vor der Ausführung des Schleifenkörpers stattfindet, hat folgende Form:

`while (` *Wiederholungsbedingung* `)` *Schleifenkörper*

Vor jeder Iteration innerhalb einer `while`-Anweisung wird zunächst die *Wiederholungsbedingung* ausgewertet. Ergibt sich als Resultat `false`, dann ist die Wiederholungsbedingung nicht erfüllt. Die Ausführung der Schleife ist damit beendet, d.h. auch der Schleifenkörper wird nicht mehr ausgeführt. Ist die *Wiederholungsbedingung* aber erfüllt, so findet die Iteration statt und der Schleifenkörper wird ausgeführt.

Beispiel 32:

```
int zahl = 1000000;

int tmp = 1;
int stellen = 0;

// Stellenanzahl fuer positive ganze Zahlen
while(tmp <= zahl) {
  tmp *= 10;
  stellen += 1;
}
System.out.println("Die Zahl " + zahl + " hat " + stellen + " Stellen.");

// Der Schleifenkoerper wird nie ausgefuehrt
while(false)
  System.out.println("niemals");
```

Wichtig: Ist die Wiederholungsbedingung immer erfüllt, so ergibt sich eine sog. *Endlos-Schleife*, bei der unendlich viele Iterationen ausgeführt werden. Damit terminiert auch das Programm nie. Endlos-Schleifen müssen deshalb vermieden werden, z.B. durch eine Korrektur der Wiederholungsbedingung, oder durch den Gebrauch der Abbruchanweisung `break` (siehe 2.8.6) im Schleifenkörper.

Beispiel 33:

```
int iterationen = 1000000;

// FEHLER: Endlosschleife
while(true)
  System.out.println("immer weiter so...");

// Abbruch der Schleife im Schleifenkoerper
while(true) {
  System.out.println("wie lange denn noch?");
  if (iterationen == 0) {                    // Abbruchbedingung
    System.out.println("na endlich");
    break;                                   // Abbruch der Schleife
  }
  iterationen--;
}
```

Eine Variante der `while`-Anweisung ist die `do-while`-Anweisung oder kurz `do`-Anweisung:

`do` *Schleifenkörper* `while (` *Wiederholungsbedingung* `);`

Im Unterschied zur `while`-Anweisung wird bei der `do`-Anweisung der *Schleifenkörper* ausgeführt, bevor die *Wiederholungsbedingung* ausgewertet wird. Der Schleifenkörper wird also immer mindestens einmal ausgeführt, auch wenn die *Wiederholungsbedingung* nie erfüllt ist. Ansonsten erfolgt die Ausführung der `do`-Anweisung analog zur `while`-Anweisung.

Beispiel 34:

```
int zahl = 1000000;
int tmp = 1;
int stellen = 0;

// Stellenanzahl fuer nichtnegative ganze Zahlen
// Dank garantierter Ausfuehrung des Schleifenkoerpers geht es jetzt auch mit 0
do {
  tmp *= 10;
  stellen += 1;
}
while(tmp <= zahl);
System.out.println("Die Zahl " + zahl + " hat " + stellen + " Stellen.");

// Der Schleifenkoerper wird einmal ausgefuehrt
do
  System.out.println("einmal");
while(false);
```

Die dritte Schleifenvariante ist die `for`-Anweisung, die speziell für Felddurchläufe und andere zählergesteuerte Wiederholungen geeignet ist:

`for (` *Initialisierung*; *Wiederholungsbedingung*; *Fortschaltung* `)` *Schleifenkörper*

Initialisierung und *Fortschaltung* sind jeweils einzelne Anweisungen, die auch weggelassen werden können (die Strichpunkte als Begrenzer der *Wiederholungsbedingung* müssen aber auf alle Fälle gesetzt werden). Die *Wiederholungsbedingung* und der *Schleifenkörper* sind analog zur `while`-Anweisung definiert.
Die Ausführung einer `for`-Anweisung beginnt mit der Ausführung der *Initialisierung*: Hier können Variablen mit geeigneten Werten belegt werden, es können aber auch neue Variablen deklariert werden. Eine solche Variable ist dann nur innerhalb der gesamten `for`-Anweisung gültig.
Die anschließenden Iterationsschritte der `for`-Anweisung laufen ähnlich ab wie bei der `while`-Anweisung: Zunächst wird die *Wiederholungsbedingung* ausgewertet: Liefert die Auswertung das Resultat `false`, so ist die `for`-Anweisung beendet, und der *Schleifenkörper* wird nicht mehr ausgeführt. Ist die *Wiederholungsbedingung* dagegen erfüllt, so werden der *Schleifenkörper* und anschließend die *Fortschaltung* ausgeführt. Damit ist der Iterationsschritt abgeschlossen.

Die `for`-Schleife erleichtert vor allem zählergesteuerte Wiederholungen: Bei der *Initialisierung* kann eine lokale Zählervariable deklariert werden, die bei der *Fortschaltung* erhöht oder erniedrigt werden kann. In der *Wiederholungsbedingung* kann überprüft werden, ob der Zähler bereits weit genug gezählt hat.
Besondere Bedeutung hat ein solcher Zähler als Index (normalerweise vom Typ `int`) für Zugriffe auf ein Feld. Mit einem einfachen Zähler können Durchläufe durch komplette (eindimensionale) Felder erreicht werden, wobei der Schleifenkörper für jedes einzelne Feldelement komplett ausgeführt werden kann.

Beispiel 35:

```
// Zaehle von 1 bis 10:
for(int i = 1 ; i <= 10; i += 1)
  System.out.println(i);

// Zaehle von 10 bis 1:
int i = 10;
for(; i > 0; i--)          // keine Initialisierung noetig
  System.out.println(i);

// Felddurchlauf (Ausgabe von Primzahlen):
short primes[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23 };
```

```
for(int k = 0; k < primes.length; k++) {
  System.out.println(primes[k]);
}

// Bilde Summe und Produkt dieser Primzahlen:
int summe = 0;
long produkt = 1;
for(int k = 0; k < primes.length; k++) {
  summe += primes[k];
  produkt *= primes[k];
}
System.out.println("Summe: " + summe);
System.out.println("Produkt: " + produkt);
```

Hinweis: Die `for`-Anweisung in Java erlaubt die Veränderung der lokal deklarierten Variablen auch innerhalb des Schleifenkörpers. Allerdings ist eine Veränderung dieser lokalen Variablen außerhalb der Fortschaltung im Prinzip ein Seiteneffekt und kann daher bei ungenauer Programmierung zu Fehlern während der Programmausführung führen, deren Ursache dann oft nur noch schwer lokalisierbar ist. Speziell Zählervariablen sollten möglichst nur in der Fortschaltung verändert werden, um den Ort eines evtl. vorhandenen Programmierfehlers einfach eingrenzen zu können.

2.8.6 Abbruchanweisungen

Um die Ausführung einer Anweisung abzubrechen, existiert in Java die *Abbruchanweisung* `break`, evtl. gefolgt von einem Markenbezeichner. Damit lassen sich alle Anweisungen, die andere Anweisungen *umschließen*, also selbst weitere Anweisungen beinhalten können, vorzeitig beenden. Die abzubrechende Anweisung muß in jedem Fall die Abbruchanweisung umschließen, da sonst die abzubrechende Anweisung zum Zeitpunkt des Abbruchs gar nicht ausgeführt würde, d.h. ihre Ausführung könnte auch nicht abgebrochen werden.

```
break;
break Marke;
```

Wird kein Markenbezeichner angegeben, wird die nächste die Abbruchanweisung umschließende Schleife bzw. `switch`-Anweisung abgebrochen. Bei Angabe eines Markenbezeichners *Marke* wird die mit *Marke* bezeichnete Anweisung abgebrochen, unabhängig von der Art der Anweisung.

Beispiel 36:

```
int wert = -1;

Fallunterscheidung:  // Markendeklaration
if (wert != 0) {
  while (true) {     // Ohne Abbruchanweisungen waere das eine Endlos-Schleife
    wert++;
    if (wert >= 10)
      break Fallunterscheidung; // Abbruch der gesamten markierten Anweisung
    else
      if (wert < 0)
        break;       // Abbruch der while-Schleife
  }
  wert *= 100;
}
```

Hinweis: Die Verwendung von Abbruchanweisungen kann rasch zu unübersichtlichen, schwer verständlichen Programmen führen (siehe obiges Beispiel), vor allem, wenn der Abbruch nicht gezielt mit einer Markenangabe erfolgt. Deshalb sollten `break`-Anweisungen, speziell bei verschachtelten Schleifen, stets nur mit großer Sorgfalt eingesetzt werden.

Speziell für Schleifen bietet Java auch die Möglichkeit, nicht die Ausführung der gesamten Schleife, sondern nur den gerade ausgeführten Iterationsschritt zu beenden. Dafür wird `break` durch das Schlüsselwort `continue` ersetzt.

Bei do-Anweisungen ist zu beachten, daß mit einer continue-Anweisung nur die Ausführung des Schleifenkörpers beendet wird; die nachfolgende Auswertung der Wiederholungsbedingung findet auf jeden Fall statt.

Beispiel 37:

```
int wert = 0;

Schleife:                  // Markendeklaration
while (wert < 10) {
  wert++;
  if ((wert % 2) != 0)     // Ueberpruefung, ob 'wert' ungerade ist
    continue Schleife;     // Starte naechsten Iterationsschritt der Schleife
  System.out.println(wert + " ist gerade");
}

do {
  wert++;
  continue;                // Diese Zeile koennte auch weggelassen werden
} while (wert < 20);
```

3 Klassen und Objekte

Dieses Kapitel stellt die grundlegenden Mechanismen zum Umgang mit Klassen und Objekten vor. Auf spezielle objektorientierte Prinzipien wie Abstraktion und Vererbung wird erst in Kapitel 4 eingegangen.

3.1 Objekte als Tupel, Klassen als Tupeltypen

Objekte können als Tupel (z.B. als Datensätze in einer Datenbank) betrachtet werden, deren Typ durch die zugehörige Klasse gegeben ist.
Der folgende Java-Quellcode deklariert eine *Klasse* (engl.: *class*) mit dem Namen Mitarbeiter als Typ eines vierstelligen Tupels. Der Körper in den geschweiften Klammern dieser Klassendeklaration enthält die Beschreibung der vier Tupelkomponenten, in Form von Variablen mit Typ und Namen.

```
class Mitarbeiter {
  String nachname;
  String vorname;
  int personalnummer;
  boolean istWeiblich;
}
```

Jedes Tupel dieses Typs ist eine *Klasseninstanz* der Klasse Mitarbeiter und soll einen Mitarbeiter einer Firma beschreiben. Statt *Klasseninstanz* sagt man oft auch kurz *Instanz* (engl.: *instance*). Instanzen werden auch als *Objekte* bezeichnet.
Jedes Objekt der Klasse Mitarbeiter besitzt jeweils eigene Exemplare der vier Variablen. Die Variablen in Objekten werden *Instanzvariablen* genannt. Ihre Namen beginnen üblicherweise mit einem Kleinbuchstaben.

3.2 Instanziierung einer Klasse, Zugriff auf Instanzvariablen

Wird zur Laufzeit eines Programms ein Objekt von einer Klasse erzeugt („kreiert“), so sagt man auch: „Die Klasse wird instanziiert.“ Die Schaffung neuer Instanzen einer Klasse wird syntaktisch durch das Schlüsselwort new ausgedrückt, gefolgt vom Klassennamen.
Der Zugriff auf eine Instanzvariable eines Objektes erfolgt, indem man das Objekt angibt, gefolgt von einem Punkt und dem Namen der Variable.

```
class NeueInstanz {

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

```
    // Instanziierung: Kreierung eines neuen Objektes:
    m = new Mitarbeiter();

    // schreibender Zugriff auf Instanzvariablen:
    m.nachname = "Mayer";
    m.vorname = "Hans";
    m.personalnummer = 11073;
    m.istWeiblich = false;

    // lesender Zugriff auf Instanzvariablen:
    System.out.println("Nachname: " + m.nachname);
    System.out.println("Vorname: " + m.vorname);
    System.out.println("Personalnummer: " + m.personalnummer);
    System.out.println("Anrede: " + (m.istWeiblich ? "Frau" : "Herr"));

    // Anlegen eines zweiten Objektes:
    Mitarbeiter n = new Mitarbeiter();
    n.nachname = "Bader";
    n.vorname = "Petra";
    n.personalnummer = 865;
    n.istWeiblich = true;
  }

}
```

3.3 Konstruktoren

Konstruktoren sind spezielle Prozeduren, die direkt nach der Erzeugung einer neuen Instanz aufgerufen werden. Ihre Aufgabe ist im allgemeinen, diese neue Instanz in einen geeigneten Startzustand zu versetzen, d.h. ihre Instanzvariablen sinnvoll zu initialisieren.
Jede Klasse besitzt, wenn nichts weiter angegeben ist, einen parameterlosen Standardkonstruktor (engl.: *default constructor*) , der alle Instanzvariablen mit einem Standardwert initialisiert (z. B. für `int`-Variablen mit 0).
Ein Konstruktor hat stets denselben Namen wie die dazugehörige Klasse. Weil ein Konstruktor kein Resultat liefert, kann bei seiner Deklaration kein Resultatstyp angegeben werden. Daher sind im Rumpf von Konstruktoren (wie auch in Methoden mit Resultatstyp `void`) `return`-Anweisungen ausschließlich in der Form „`return;`“ erlaubt, die also keine Ausdrücke beinhalten.
Ein Konstruktor wird im Zusammenhang mit `new` aufgerufen, wenn ein neues Objekt kreiert wird. Die aktuellen Parameter für einen Konstruktor-Aufruf werden hierbei in einem Instanziierungsausdruck nach dem Klassennamen in runden Klammern durch Kommata getrennt angegeben. Der Wert dieses Ausdrucks ist das neu erzeugte und initialisierte Objekt.

Syntax eines Instanziierungsausdrucks:

`new` *Klassenname* `(` *Liste der aktuellen Parameter* `)`

Beispiel:

```
class Mitarbeiter {
  String nachname;
  String vorname;
  int personalnummer;
  boolean istWeiblich;

  // Deklaration eines Konstruktors:
  Mitarbeiter(String nn, String vn, int pn, boolean w) {
    nachname = nn;
    vorname = vn;
    personalnummer = pn;
```

```
    istWeiblich = w;
  }
}

class KonstruktorBeispiel {

  public static void main(String args[]){

    Mitarbeiter m,n;

    // Konstruktor-Aufrufe:
    m = new Mitarbeiter("Mayer", "Hans", 11073, false);
    n = new Mitarbeiter("Bader", "Petra", 865, true);
  }
}
```

3.4 Klassenvariablen

Im Gegensatz zu Instanzvariablen existiert von einer *Klassenvariable* für alle Objekte einer Klasse nur ein Exemplar. Dieses Exemplar ist auch dann vorhanden, wenn zu einem Zeitpunkt keine Instanz der Klasse existiert.
Klassenvariablen werden auch *statische Variablen* genannt, da ihre Deklarationen mit dem Schlüsselwort `static` geschrieben werden. Bei der Deklaration von Instanzvariablen dagegen fehlt dieser Modifikator.
Der Zugriff auf eine Klassenvariable erfolgt durch die Angabe des zugehörigen Klassennamens, gefolgt von einem Punkt und dem Namen der Variablen.
Ein typisches Beispiel für eine Klassenvariable ist ein Zähler, dessen Wert die Anzahl der bereits erzeugten Instanzen einer Klasse angibt.

```
class Mitarbeiter {
  String nachname;
  String vorname;
  int personalnummer;
  boolean istWeiblich;
  static int anzahl = 0;   // eine Klassenvariable, initialisiert mit 0

  Mitarbeiter(String nn, String vn, int pn, boolean w) {
    nachname = nn;
    vorname = vn;
    personalnummer = pn;
    istWeiblich = w;
    anzahl++;
  }
}

class AnzahlBeispiel {

  public static void main(String args[]){

    System.out.println("Anzahl: " + Mitarbeiter.anzahl);
    Mitarbeiter m1 = new Mitarbeiter("Mayer", "Hans", 11073, false);
    System.out.println("Anzahl: " + Mitarbeiter.anzahl);
    Mitarbeiter m2 = new Mitarbeiter("Bader", "Petra", 865, true);
    System.out.println("Anzahl: " + Mitarbeiter.anzahl);
  }
}
```

3.5 Klassen als Typen

Klassen sind zulässige Typen für Variablen und Parameter. Variablen, deren Typ eine Klasse ist, können Instanzen dieser Klasse[6] zugewiesen werden.
Um auszudrücken, daß in einer Variable gerade *kein* Objekt gespeichert ist, gibt es den speziellen Wert `null`. Dieser Wert wird jeder Instanz- und jeder Klassenvariable eines Referenztyps bei ihrer Deklaration implizit zugewiesen.

Beispiel: Die Klasse `Mitarbeiter` wird variiert, so daß sie nun auch eine Instanzvariable mit Namen `vorgesetzter` enthält. Diese Variable ihrerseits ist wiederum vom Typ `Mitarbeiter`.

```
class Mitarbeiter {
  String nachname;
  String vorname;
  int personalnummer;
  boolean istWeiblich;
  Mitarbeiter vorgesetzter;   // eine Klasse als Typ einer Instanzvariable

  // Konstruktor:
  Mitarbeiter(String nn, String vn, int pn, boolean w, Mitarbeiter v) {
    nachname = nn;
    vorname = vn;
    personalnummer = pn;
    istWeiblich = w;
    vorgesetzter = v;
  }
}

class NullBeispiel {

  public static void main(String args[]){

    Mitarbeiter m, chef;     // eine Klasse als Typ von lokalen Variablen

    chef = new Mitarbeiter("Schmidt", "Martin", 341, false, null);
    m = new Mitarbeiter("Mayer", "Hans", 11073, false, chef);
  }
}
```

In diesem Beispiel wird `null` verwendet, um zu verdeutlichen, daß das in `chef` gespeicherte Objekt keinen Vorgesetzten besitzt.

Instanziierungsausdrücke:

Die Instanziierung einer Klasse mit `new` ist ein Ausdruck, dessen Typ die Klasse ist. Ein solcher Ausdruck kann beispielsweise nach „=“ stehen, um ein Objekt anzugeben, mit dem eine Variable initialisiert werden soll:

```
Mitarbeiter chef = new Mitarbeiter("Schmidt", "Martin", 341, false, null);
Mitarbeiter m = new Mitarbeiter("Mayer", "Hans", 11073, false, chef);
```

Verschachtelte Instanziierungsausdrücke:

Wird als Argument eines Konstruktors ein Objekt erwartet, so kann bei seinem Aufruf ebenfalls ein `new`-Ausdruck stehen. Hierbei entstehen Verschachtelungen von `new`-Ausdrücken, auch *Konstruktor-Terme* genannt:

```
Mitarbeiter m = new Mitarbeiter("Mayer", "Hans", 11073, false,
                      new Mitarbeiter("Schmidt", "Martin", 341, false, null));
```

[6] Die Instanzen dürfen auch einer Unterklasse der angegebenen Klasse angehören. Genauere Erläuterungen hierzu folgen im nächsten Kapitel.

3.6 Werte und Referenzen

3.6.1 Grundtypen und Referenztypen

Zwischen den Grundtypen auf der einen Seite und den Referenztypen, also Klassen, Schnittstellen (siehe Abschnitt 4) und Feldtypen auf der anderen Seite gibt es einen wesentlichen Unterschied: Eine Variable eines Grundtyps enthält einen Wert dieses Grundtyps. Eine Variable eines Referenztyps enthält eine *Referenz* (oder auch *Verweis* oder *Zeiger*; engl.: *reference* oder *pointer*) auf ein Objekt (und nicht das Objekt selbst), oder aber die spezielle Referenz `null`.
Der vordefinierte Typ `Object` stellt den allgemeinsten Referenztyp dar. Eine Variable vom Typ `Object` kann daher Werte eines beliebigen Referenztyps annehmen.

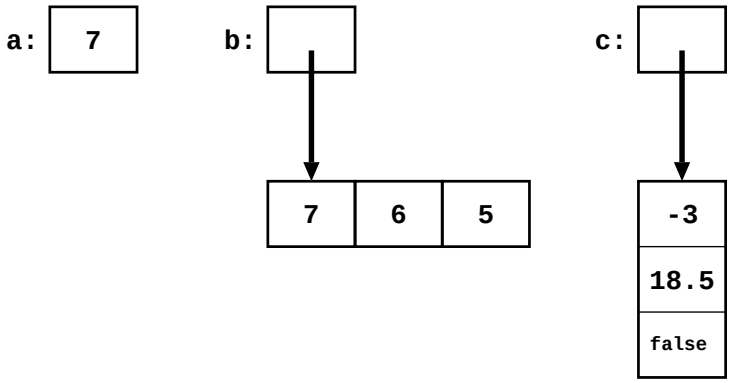
Beispiel: Es werden drei Variablen initialisiert und verglichen; der Typ der ersten ist ein Grundtyp, die Typen der anderen beiden sind Referenztypen, nämlich ein Feldtyp bzw. eine Klasse:

```
int a = 7;
int b[] = { 7, 6, 5 };
Triple c = new Triple(-3, 18.5, false);
```

Die Klasse `Triple` habe dabei folgende Gestalt:

```
class Triple {
  short s;
  double d;
  boolean b;

  Triple(short ss, double dd, boolean bb){
    s = ss;
    d = dd;
    b = bb;
  }
}
```

Nach Initialisierung dieser drei Variablen kann man sich den Inhalt des Arbeitsspeichers wie folgt vorstellen:



Die Variablen `b` und `c` enthalten also jeweils eine Referenz auf ein Feld bzw. ein Objekt.

3.6.2 Felder als Objekte

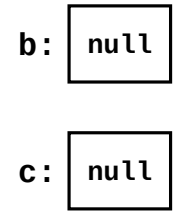
Felder werden in Java wie Objekte von imaginären Klassen behandelt. Die „Klasse“ eines Felds entspricht dabei dem Datentyp des Felds. Felder können daher wie Objekte nicht direkt, sondern nur über Referenzen angesprochen werden.
Neue Felder werden wie neue Objekte mit `new` erzeugt. Auch `null` als „leere“ Referenz ist für Felder und Objekte einheitlich verwendbar. Das Attribut `length`, das für alle Felder definiert ist, kann in diesem Zusammenhang auch als (nicht veränderbare) Instanzvariable betrachtet werden.

Felder können zwar als Objekte benutzt werden, Feldtypen und Klassen sind aber nur bis zu einem gewissen Grad vergleichbar. Feldtypen entsprechen am ehesten festgelegten, nicht mehr erweiterbaren Klassen. Sie können daher im Gegensatz zu Klassen nicht direkt für die im nächsten Kapitel vorgestellten Verfahren zur objektorientierten Modellierung bzw. Programmierung eingesetzt werden.

3.6.3 Die Referenz `null`

Wie bereits zuvor erwähnt, verweist eine Variable mit dem Wert `null` nicht auf ein Objekt, d.h. über die `null`-Referenz ist kein Zugriff auf irgendein Objekt möglich. Dies gilt unabhängig vom Typ der Variablen. Über die Referenz `null` kann daher auch nicht auf Instanzvariablen und -methoden zugegriffen werden.

```
Triple c = null;
short x = c.s;     // Fehler: Zugriff ueber "null"

int b[] = null;
int i = b[1];      // Fehler: Zugriff ueber "null"
```



3.6.4 Gleichheit bei Referenztypen

Mit den Vergleichsoperatoren `==` und `!=` können auch Ausdrücke von Referenztypen verglichen werden. Allerdings wird in diesem Fall nur geprüft, ob die zwei Referenzen auf dasselbe Objekt verweisen (Identitätsprüfung). Zwei verschiedene Instanzen derselben Klasse, deren Instanzvariablen jeweils dieselben Werte besitzen, gelten daher nicht als gleich (bzgl. `==` oder `!=`); dasselbe gilt für zwei verschiedene Felder, in denen dieselben Werte gespeichert sind.
Eine Überprüfung auf inhaltliche Gleichheit von Objekten muß in der Regel gesondert definiert werden.

```
class ObjektVergleich {

  public static void main(String args[]){

    Mitarbeiter m1 = new Mitarbeiter("Schmidt", "Martin", 341, false, null);
    Mitarbeiter m2 = new Mitarbeiter("Schmidt", "Martin", 341, false, null);

    boolean b = (m1 == m2);     // b wird der Wert "false" zugewiesen.
    System.out.println(b);

    b = (null != null);         // b wird der Wert "false" zugewiesen.
  }
}
```

Hinweis: Für viele Klassen sind Methoden zur Überprüfung auf inhaltliche Gleichheit vorgegeben. Die textuelle Gleichheit von Zeichenketten (d.h. Objekten der Klasse `String`) kann z.B. mit der Instanzmethode `compareTo()` der Klasse `String` festgestellt werden. Die Operatoren `==` und `!=` sind hierfür aus dem zuvor genannten Grund *nicht* geeignet.

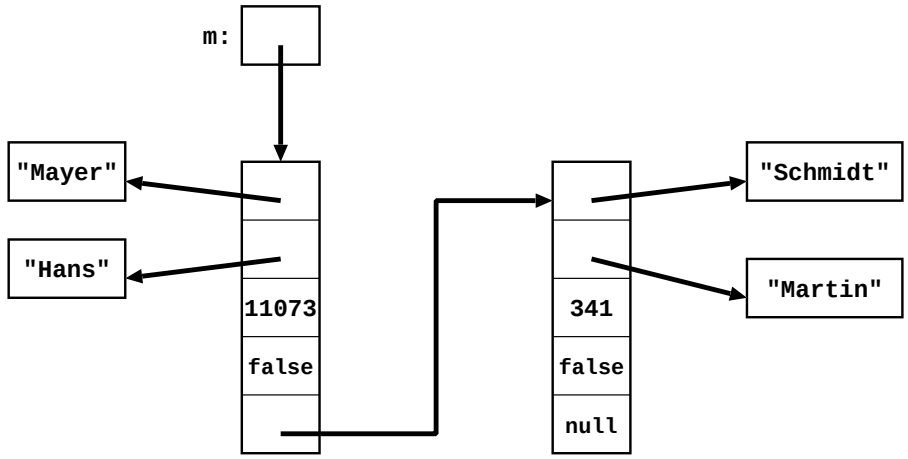
3.6.5 Referenzgeflechte und dynamische Speicherbereinigung

Objekte können über Referenzen miteinander verbunden werden. Auf diese Weise entstehen *Geflechte* von Objekten.

Beispiel: Es wird nochmals der verschachtelte Initialisierungsausdruck aus dem vorausgehenden Abschnitt aufgegriffen:

```
Mitarbeiter m = new Mitarbeiter("Mayer", "Hans", 11073, false,
                      new Mitarbeiter("Schmidt", "Martin", 341, false, null));
```

Nach der Initialisierung von `m` ergibt sich folgender Inhalt des Arbeitsspeichers:



Man beachte hierbei, daß Zeichenketten Objekte sind und deren Klasse `String` ein Referenztyp ist.

Das oben dargestellte Geflecht ist zyklenfrei. Im allgemeinen können aber über Referenzen zwischen Objekten auch Zyklen entstehen. Solche Zyklen stellen an sich keine Fehler dar. Zu Fehlern im Programmen kommt es dagegen oft, wenn man nicht berücksichtigt, daß Zyklen auftreten können.
Auf ein Objekt dürfen auch mehrere Referenzen zugleich zeigen. Falls aber auf ein Objekt keine Referenz mehr verweist, so ist es „verloren“, und sein Speicherplatz wird zu einem späteren Zeitpunkt automatisch wieder freigegeben. Dieser Vorgang wird *dynamische Speicherbereinigung* (engl.: *garbage collection*) genannt. Der Programmierer braucht sich also um die Freigabe nicht mehr benötigter Speicherzellen nicht selbst zu kümmern.

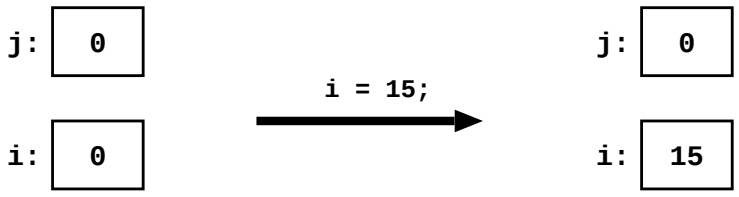
3.6.6 Wertübergabe und Referenzübergabe

Grund- und Referenztypen werden auch als Typen für formale Parameter von Methoden eingesetzt. Da formale Parameter im Methodenrumpf wie lokale Variablen behandelt werden, darf eine Inkarnation einer Methode während ihrer Ausführung auch schreibend auf ihre formalen Parameter zugreifen.

Zunächst ein Beispiel mit einem formalen Parameter `i`, dessen Typ der Grundtyp `int` ist:

```
class Wertuebergabe {

  static void m(int i){
    i = 15;
  }

  public static void main(String args[]){
    int j = 0;
    m(j);
    System.out.println(j);    // Ergebnis: Es wird der Wert 0 ausgegeben.
  }
}
```

In diesem Fall wird der Wert von `j` durch den Aufruf der Methode `m` nicht geändert. Allgemein findet bei Grundtypen *Wertübergabe* (engl.: *call by value*) statt.
Man kann sich den Arbeitsspeicher während des Programmlaufs vor bzw. nach der Zuweisung `i = 15;` wie folgt vorstellen:

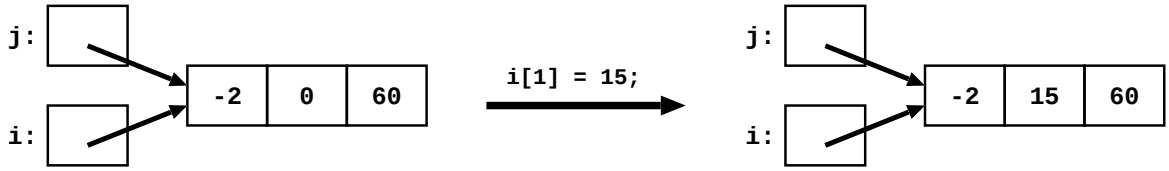


Nun folgt der andere Fall, bei dem der Typ eines formalen Parameters ein Referenztyp ist, was man *Referenzübergabe* (engl.: *call by reference*) nennt. Im Beispiel wird ein Feld übergeben, d.h. der Parametertyp

ist ein Feldtyp:

```
class Referenzuebergabe {

  static void m(int i[]){
    i[1] = 15;
  }

  public static void main(String args[]){
    int j[] = new int[3];
    j[0] = -2;
    j[1] = 0;
    j[2] = 60;
    m(j);
    System.out.println(j[1]);    // Ergebnis: Es wird der Wert 15 ausgegeben.
  }
}
```

Hier wird das übergebene Feld, auf das eine Referenz in `j` gespeichert ist, durch den Aufruf der Methode `m` so manipuliert, daß diese Änderung auch nach Rückkehr von der Methode `m` noch gilt.
Die entsprechenden Skizzen zum Arbeitsspeicher haben nun folgende Gestalt:



3.7 Instanzmethoden

Innerhalb einer Klasse beschreibt jede Methodendeklaration ohne das Schlüsselwort `static` eine *Instanzmethode*. Eine Inkarnation einer Instanzmethode nimmt automatisch Bezug auf eine Instanz der Klasse, dem *impliziten Argument*.
Eine Instanzmethode einer Klasse wird aufgerufen, indem man ein Objekt der Klasse angibt, gefolgt vom Methodennamen und den aktuellen Parametern in runden Klammern. Dieses Objekt ist dann das implizite Argument des Aufrufs.

```
class Mitarbeiter {
  String nachname;
  String vorname;
  int personalnummer;
  boolean istWeiblich;

  // Konstruktor:
  Mitarbeiter(String nn, String vn, int pn, boolean w){
    nachname = nn;
    vorname = vn;
    personalnummer = pn;
    istWeiblich = w;
  }

  // eine Instanzmethode:
  void ausgabe(){
    System.out.print(istWeiblich ? "Frau " : "Herr ");
    System.out.print(vorname + " ");
    System.out.print(nachname);
    System.out.println(" (Personalnummer: " + personalnummer + ")");
  }
}
```

```
class InstanzmethodenBeispiel {

  public static void main(String args[]){

    Mitarbeiter m1 = new Mitarbeiter("Schmidt", "Martin", 341, false);
    Mitarbeiter m2 = new Mitarbeiter("Bader", "Petra", 865, true);

    // zwei Aufrufe der obigen Instanzmethode:
    m1.ausgabe();
    m2.ausgabe();

  }
}
```

3.8 Das Schlüsselwort `this`

Im Rumpf einer Instanzmethode kann mit `this` das Objekt referenziert werden, für das die Methode aufgerufen wurde. Analog dazu wird im Rumpf eines Konstruktors mit `this` das Objekt angesprochen, das gerade neu kreiert wurde. In beiden Situationen ist der Gebrauch des Schlüsselwortes `this` in folgenden zwei Fällen notwendig:

1. wenn in einem Ausdruck eine Referenz auf das Objekt auftreten soll,
2. wenn eine Instanzvariable von einem formalen Parameter oder einer lokalen Variable mit demselben Namen verschattet wird und diese Verschattung umgangen werden soll.

Ansonsten kann in einer Instanzmethode oder in einem Konstruktor eine Formulierung der Art

`this.`*Bezeichner*

Auch kürzer als

Bezeichner

geschrieben werden, wobei *Bezeichner* für eine beliebige Instanzvariable bzw. für einen Aufruf einer Instanzmethode steht.

Beispiele für den Einsatz von `this`:

1. Referenz auf das implizite Argument einer Instanzmethode:

```
class Kettenglied {

  Kettenglied vorgaenger;
  Kettenglied nachfolger;

  void anhaengen(Kettenglied neu){
    // haenge "neu" nach "this" an die Kette an:
    if(neu != null){
      this.nachfolger = neu;    // oder auch: "nachfolger = neu;"
      neu.vorgaenger = this;    // Zuweisen der Referenz "this"
    }
  }
}
```

2. Umgehen von Verschattung:

```
class Mitarbeiter {
  String nachname;
  String vorname;
  int personalnummer;
  boolean istWeiblich;
```

```
    // Konstruktor:
    Mitarbeiter(String nachname, String vorname, int pn, boolean w){
      // Umgehen der Verschattung durch formale Parameter:
      this.nachname = nachname;        // "this" ist notwendig
      this.vorname = vorname;          // "this" ist notwendig
      // Umgehen der Verschattung durch lokale Variablen:
      int personalnummer;
      boolean istWeiblich;
      this.personalnummer = pn;        // "this" ist notwendig
      this.istWeiblich = w;            // "this" ist notwendig
    }
  }
```

3.9 Klassenmethoden

Instanzmethoden sind Methoden, bei deren Ausführung auf ein Objekt Bezug genommen wird. Im Gegensatz dazu ist eine *Klassenmethode* eine globale Methode für die ganze Klasse, nicht für eine bestimmte Instanz. Sie besitzt kein implizites Argument. Demnach ist in ihrem Rumpf auch das Schlüsselwort `this` unzulässig.
Klassenmethoden sind wie Klassenvariablen am Schlüsselwort `static` zu erkennen; daher werden sie auch *statische Methoden* genannt. Sie können aufgerufen werden, indem man vor dem Methodennamen den Namen der Klasse angibt, gefolgt von einem Punkt. Befinden sich Aufruf und Deklaration innerhalb derselben Klasse, so genügt beim Aufruf der Name der Methode ohne vorangestellten Klassennamen.

```
class Mitarbeiter {
  String nachname;
  String vorname;
  int personalnummer;
  boolean istWeiblich;
  static int anzahl = 0;

  // Konstruktor:

  Mitarbeiter(String nn, String vn, int pn, boolean w){
    nachname = nn;
    vorname = vn;
    personalnummer = pn;
    istWeiblich = w;
    anzahl++;
  }

  // zwei Klassenmethoden:

  static int liesAnzahl(){
    return anzahl;
  }

  static void druckeAnzahl(){
    System.out.print("Anzahl der Mitarbeiter: ");
    // ein Aufruf einer Klassenmethode (ohne Klassennamen):
    System.out.println(liesAnzahl());
  }
}

class KlassenmethodenBeispiel {

  public static void main(String args[]){

    Mitarbeiter chef = new Mitarbeiter("Schmidt", "Martin", 341, false);
```

```
    // Aufrufe der obigen Klassenmethoden (mit Klassennamen):
    int a = Mitarbeiter.liesAnzahl();
    Mitarbeiter.druckeAnzahl();
  }
}
```

3.10 Implementierung von Datenstrukturen

Ein wichtiger Bereich der praktischen Programmierung ist die Gliederung und Strukturierung von Daten. Ein Beispiel für eine einfache Form der Strukturierung sind z.B. die bereits bekannten Felder, die somit als eine Art von *Datenstruktur* gesehen werden können. Die Erstellung von beliebig komplexen, evtl. auch rekursiven Datenstrukturen erfolgt in Java durch die Deklaration von Klassen. Die Daten werden dann in Instanzen dieser Klasse(n) gespeichert (Datenobjekte). Durch den Aufbau eines Geflechts von Datenobjekten entsteht schließlich die eigentliche Datenstruktur.
Rekursive Datenstrukturen und die zur Bearbeitung der Daten erforderlichen (rekursiven) Algorithmen sind ein zentraler Bereich der Informatik. In der Praxis häufig verwendete rekursive Datenstrukturen sind Listen und Bäume. Für diese Strukturen wird im folgenden je eine mögliche Implementierung präsentiert.

3.10.1 Listen

Listen sind linear geordnete Ansammlungen von Elementen. Im Gegensatz zu Feldern können Listen (theoretisch) beliebig viele Elemente aufnehmen, und bestimmte Operationen sind für Listen erheblich eleganter und effizienter formulierbar als für Felder.

Die Klasse `MitarbeiterListe` des folgenden Java-Quelltextes stellt Listen von Mitarbeitern dar. Es handelt sich hierbei um „einfach verkettete“ Listen, da jedes Element der Liste *eine* Referenz (auf ihr Nachfolger-Element) besitzt.[7]

```
class Mitarbeiter {
  String nachname;
  String vorname;
  int personalnummer;
  boolean istWeiblich;

  Mitarbeiter(String nn, String vn, int pn, boolean w){
    nachname = nn;
    vorname = vn;
    personalnummer = pn;
    istWeiblich = w;
  }
}

class MitarbeiterListe {

  Mitarbeiter element;
  MitarbeiterListe nachfolger;

  // Konstruktor fuer Listen mit einem Element:
  MitarbeiterListe(Mitarbeiter m){
    element = m;
    nachfolger = null;
  }

  // Konstruktor fuer Listen mit mehr als einem Element:
  MitarbeiterListe(Mitarbeiter m, MitarbeiterListe l){
    element = m;
    nachfolger = l;
  }
}
```

[7] Bei *doppelt verketteten Listen* besitzt jedes Element jeweils eine Referenz auf seinen Vorgänger und seinen Nachfolger.

```
class ListenBeispiel {

  public static void main(String args[]){
    Mitarbeiter m1 = new Mitarbeiter("Schmidt", "Martin", 341, false);
    Mitarbeiter m2 = new Mitarbeiter("Bader", "Petra", 865, true);
    Mitarbeiter m3 = new Mitarbeiter("Mayer", "Hans", 11073, false);

    // Aufbau einer Liste mit diesen drei Mitarbeitern:
    MitarbeiterListe l = new MitarbeiterListe(m1,
                           new MitarbeiterListe(m2,
                             new MitarbeiterListe(m3)));
  }
}
```

3.10.2 Bäume

Bäume sind spezielle Graphen, die hierarchische Anordnungen von Knoten darstellen. Ein Beispiel hierfür sind Stammbäume: Objekte der unten aufgelisteten Klasse `Person` sind Personen (Knoten) in einem Stammbaum. Die Knoten der Eltern (bzw. die Referenzen dieser Knoten) sind für jede `Person` individuell festlegbar.[8]
Solche Stammbäume sind Beispiele für *Binärbäume*: Jeder Baumknoten enthält höchstens zwei Referenzen auf weitere Knoten.

```
class Person {
  String vorname, nachname;
  Person vater, mutter;

  Person(String vorname, String nachname, Person vater, Person mutter){
    this.vorname  = vorname;
    this.nachname = nachname;
    this.vater  = vater;
    this.mutter = mutter;
  }

  void ausgabe(int einrueckung){
    for(int s = einrueckung ; s > 0 ; s--) System.out.print(' ');
    System.out.println(vorname + " " + nachname);
    if( vater != null)  vater.ausgabe(einrueckung+4);
    if(mutter != null) mutter.ausgabe(einrueckung+4);
  }
}

class Stammbaum {

  public static void main(String args[]){
    Person p =
      new Person("Martin", "Schmidt",
         new Person("Hans", "Schmidt",                      // Vater
            new Person("Ulrich", "Schmidt", null, null),    // Vater des Vaters
            new Person("Susanne","Schmidt", null, null)),   // Mutter des Vaters
         new Person("Petra","Schmidt",                      // Mutter
            new Person("Josef",  "Mayer"  , null, null ),   // Vater der Mutter
            new Person("Claudia","Mayer"  , null, null)));  // Mutter der Mutter
    p.ausgabe(0);
  }
}
```

[8] Zur Vereinfachung sei angenommen, daß eine Person keine Ahnen mit gemeinsamen Vorfahren besitzt.

4 Grundlagen der Objektorientierung

Im vorigen Kapitel wurden Objekte vor allem als Behälter für Daten betrachtet, die mit Hilfe von Methoden bearbeitet werden können. Dieses Konzept erlaubt die Erstellung komplexer Datenstrukturen und ist für eine in erster Linie an Datenstrukturen orientierte Programmentwicklung sinnvoll.
Speziell bei der Entwicklung größerer Programmsysteme spielen aber auch Begriffe wie Wiederverwendbarkeit und Flexibilität eine große Rolle. Es werden Techniken benötigt, um z.B. die mehrfache Erstellung ähnlicher Klassen zu vermeiden und bereits erstellte Klassen einfach an neue Anforderungen anpassen zu können. Die Objektorientierung liefert hierfür einige wichtige Konzepte. Die Umsetzung dieser Konzepte mit Java wird in den folgenden Abschnitten genauer behandelt.

Objektorientierte Programmierung setzt eine sinnvolle objektorientierte Modellierung des zu erstellenden Programmsystems voraus. Deshalb sollte vor der Erstellung eines Programms zumindest ein einfaches Grobmodell des zu realisierenden Systems entwickelt werden. Dadurch können prinzipielle Fehler oft schon frühzeitig erkannt und beseitigt werden, so daß die Programmentwicklung effizienter wird.

Bemerkung: Die Modellierung großer Systeme erfolgt häufig mit Hilfe von Modellierungswerkzeugen. Besonders beliebt sind Werkzeuge, die die einzelnen Systemkomponenten und ihre Beziehungen zueinander in graphischer Form darstellen können. In den folgenden Abschnitten wird eine graphische Notation für Klassen, Objekte und ihre Beziehungen verwendet, die eng an die *UML*[9] (*Unified Modeling Language*) angelehnt ist.

4.1 Objekte und Objektklassen

Natürliche Systeme sind in der Regel sehr komplex, d.h. sie bestehen aus einer Vielzahl von *Objekten* bzw. *Komponenten* mit verschiedenen *Eigenschaften*, die in unterschiedlichen Beziehungen zueinander stehen können. Objekte lassen sich anhand ihrer jeweiligen Eigenschaften charakterisieren. Die Eigenschaften eines Objekts werden dabei vor allem durch seine *Attribute* und die *Operationen* bestimmt, die auf dem Objekt durchführbar sind. Die Operationen werden auch als *Methoden* bezeichnet.
Unter den Attributen eines Objekts versteht man alle Bestandteile des Objekts, die Informationen über das Objekt beinhalten. Attribute selbst können wiederum Objekte sein. Diese Schachtelung erlaubt den Aufbau beliebig komplexer Objekte. Die Gesamtheit der in den Attributen enthaltenen Informationen (die Attributwerte) beschreibt den *Zustand*, in dem das Objekt sich gerade befindet.
Zur Modellierung dynamischer Vorgänge in einem System dienen die Methoden, die festlegen, ob und wie ein Objekt auf äußere Einflüsse reagiert.

4.1.1 Instanziierung

Ein realitätsnahes Modell für ein natürliches System muß die Eigenschaften der einzelnen Systemkomponenten möglichst genau nachbilden. Um die Vielzahl existierender Objekte in kompakter Form beschreiben zu können, werden *Klassen* gebildet, die alle Objekte mit den gleichen Eigenschaften umfassen. Werden Objekte in das Modell eingefügt, die dieser Klasse angehören, so geschieht dies über eine sog. *Instanziierung* der Klasse, wobei die Klassenbeschreibung einer Bauanleitung für das neu erstellte Objekt entspricht. Durch Instanziierung läßt sich das Ausmaß der explizit aufzuschreibenden Objektinformationen bereits erheblich verringern.

Bemerkung: Die hier eingeführten bzw. wiederholten Begriffe aus dem OO-Bereich entsprechen weitestgehend der Java-Terminologie. Unterschiedliche Bezeichnungen treten auf für Attribute, die in Java mit Instanzvariablen realisiert werden, und für Methoden, die den Instanzmethoden von Java entsprechen. Die Ursache dafür ist die Java-spezifische Unterscheidung von Klassen- und Instanzvariablen bzw. -methoden, die nicht auf den gesamten OO-Bereich verallgemeinerbar ist.

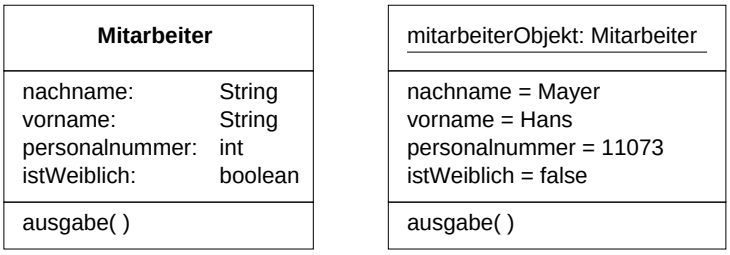
Betrachtet man das Personalverwaltungs-Beispiel aus dem vorigen Kapitel, so ergibt sich z.B. für einen `Mitarbeiter` folgende Darstellung der Objektbeschreibung:

Beispiel 38:

```
class Mitarbeiter {
  String nachname;
  String vorname;
```

[9] UML entspricht im Prinzip einer graphischen Programmiersprache zur Beschreibung der verschiedenen Aspekte eines objektorientieren Systemmodells. Aufgrund der Vielzahl möglicher Darstellungsarten und der damit verbundenen Ausdrucksstärke entwickelt sich UML zunehmend zur Standardsprache für die objektorientierte Modellierung.

```
  int personalnummer;
  boolean istWeiblich;

  // leerer Standard-Konstruktor (Bedeutung spaeter)
  Mitarbeiter () {
  }

  // Konstruktor
  Mitarbeiter (String nachname, String vorname,
               int personalnummer, boolean istWeiblich) {
    this.nachname = nachname;
    this.vorname = vorname;
    this.personalnummer = personalnummer;
    this.istWeiblich = istWeiblich;
  }

  void ausgabe () {
    System.out.print(istWeiblich ? "Frau " : "Herr ");
    System.out.print(vorname + " " + nachname);
    System.out.println(" (Personalnummer: " + personalnummer + ")");
  }
}
```



Klassenbeschreibung Objektbeschreibung

Hinweis: Zu beachten ist bei der Referenzierung durch `this`, daß damit ein Objekt referenziert wird und keine Klasse. Klassenvariablen und -methoden sollten also nicht über `this` angesprochen werden.

4.1.2 Kapselung

Ein wesentliches Prinzip der Objektorientierung ist die sog. *Kapselung* (engl.: *encapsulation*), also die Zusammenfassung von Attributen und Methoden zu einem Objekt. Die Kapselung dient dazu, überwachte Zugriffe auf Attribute zu gewährleisten. Dadurch sollen Seiteneffekte (siehe 2.6.1) zwischen verschiedenen Objekten vermieden werden.
Bei strenger Auslegung des Kapselungsprinzips dürfen nur die in einem Objekt definierten Methoden auf die Attribute dieses Objekts zugreifen, d.h. ein direkter Zugriff von außerhalb des Objekts auf die Attribute ist nicht möglich. Ein Objekt kann ein anderes Objekt also nur dadurch beeinflussen, daß es eine der Methoden des zu beeinflussenden Objekts aufruft. Im obigen Beispiel könnte also nicht einfach das Attribut `nachname` direkt verändert werden. Die Änderung des Attributwerts müßte durch einen Methodenaufruf (z.B. `aendereNachname()`) erfolgen. Wird (wie in diesem Fall) keine Methode für den Attributzugriff deklariert, ist kein Zugriff auf das Attribut möglich.

Hinweis: Wie einige andere OO-Sprachen erlaubt auch Java entgegen dem Kapselungsprinzip direkte Zugriffe auf Objektattribute von außerhalb des Objekts (mit Hilfe von zusammengesetzten Bezeichnern, siehe Abschnitte 2.2). Um dennoch eine Zugriffskontrolle zu ermöglichen, können die Attribute explizit mit verschiedenen Zugriffsrechten versehen werden. Die Vergabe solcher Zugriffsrechte in Java wird in Kapitel 5 detailliert behandelt.

4.2 Klassenhierarchien

4.2.1 Generalisierung und Spezialisierung

Für eine genaue Modellierung komplexer Objekte sind oft viele Attribute und Fähigkeiten zu modellieren, d.h. die entsprechenden Klassenbeschreibungen können sehr umfangreich werden. Existieren Gemeinsam-

keiten zwischen verschiedenen Klassen, bietet sich jedoch oft eine weitere Möglichkeit zur Vereinfachung. In diesem Fall werden gemeinsame Eigenschaften nicht für jede einzelne Klasse erneut festgelegt, sondern in einer eigenen Klasse, der sog. *Oberklasse*, zusammengefaßt. Die Oberklasse hat einen höheren *Abstraktionsgrad* als ihre Unterklassen, da sie nur eine Schnittmenge der Eigenschaften ihrer Unterklassen enthält und daher weniger spezialisiert ist. Die Bildung einer solchen Oberklasse nennt man auch *Generalisierung*.
Die Klassen, die die Eigenschaften der Oberklasse besitzen, werden als *Unterklassen* oder *Subklassen* der Oberklasse bezeichnet. Da Unterklassen und Oberklasse in direkter Beziehung zueinander stehen, spricht man in diesem Fall auch von *direkten Unterklassen* und einer *direkten Oberklasse*.

```
class Student {
  String nachname;
  String vorname;
  boolean istWeiblich;
  String studienfach;
  byte semester;

  // Konstruktor
  Student (String nachname, String vorname, boolean istWeiblich,
           String studienfach, byte semester) {
    this.nachname = nachname;
    this.vorname = vorname;
    this.istWeiblich = istWeiblich;
    this.studienfach = studienfach;
    this.semester = semester;
  }
}
```

Existiert z.B. im Programm für die Personalverwaltung auch noch eine Klasse `Student`, die zur Beschreibung studentischer Hilfskräfte dient, so ergeben sich beim Vergleich beider Klassen einige Gemeinsamkeiten: Die Instanzvariablen `nachname`, `vorname` und `istWeiblich` sind für beide Klassen deklariert. Eine sich daraus ergebende Oberklasse `Person` könnte folgendermaßen aussehen:

```
class Person {
  String nachname;
  String vorname;
  boolean istWeiblich;

  // leerer Standard-Konstruktor (Bedeutung spaeter)
  Person () {
  }

  // Konstruktor
  Person (String nachname, String vorname, boolean istWeiblich) {
    this.nachname = nachname;
    this.vorname = vorname;
    this.istWeiblich = istWeiblich;
  }
}
```

Die *Spezialisierung* funktioniert genau entgegengesetzt zur Generalisierung. Eine bestehende Klasse wird spezialisiert, indem den in der Klasse bereits enthaltenen Merkmalen neue, nur für eine spezielle Teilklasse charakteristische Merkmale hinzugefügt werden und so eine neue Unterklasse gebildet wird. Man spricht auch von *Konkretisierung*, um den Gegensatz zu der bei einer Generalisierung stattfindenden Abstraktion zu verdeutlichen.
Eine Klasse kann beliebig oft spezialisiert werden, ebenso wie eine Oberklasse durch die Generalisierung beliebig vieler Klassen entstehen kann.

Ein Beispiel für die Spezialisierung wäre z.B. eine Klasse `Angestellter`, die zusätzlich zu den von der Klasse `Mitarbeiter` übernommenen Eigenschaften noch eine Variable `tarifgruppe` enthält:

```
class Angestellter {
  String nachname;     // aus der Klasse Mitarbeiter kopiert
  String vorname;      // aus der Klasse Mitarbeiter kopiert
  int personalnummer;  // aus der Klasse Mitarbeiter kopiert
  boolean istWeiblich; // aus der Klasse Mitarbeiter kopiert
  String tarifgruppe;

  // Konstruktor
  Angestellter (String nachname, String vorname,
                int personalnummer, boolean istWeiblich,
                String tarifgruppe) {
    this.nachname = nachname;
    this.vorname = vorname;
    this.personalnummer = personalnummer;
    this.istWeiblich = istWeiblich;
    this.tarifgruppe = tarifgruppe;
  }
}
```

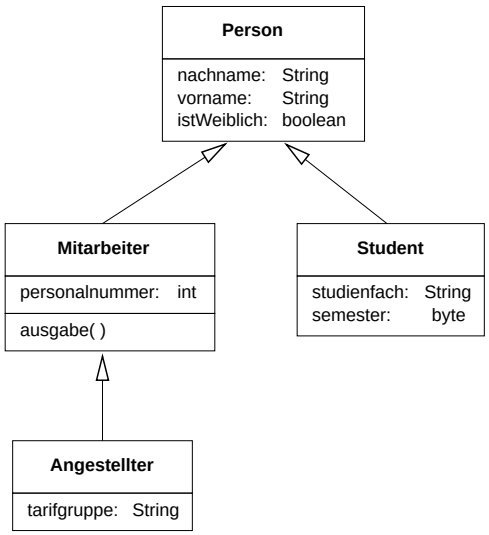
Generalisierung kann auch für einzelne Klassen sinnvoll sein, wenn ein Teil der Eigenschaften der Klasse sich zu einer sinnvollen Oberklasse zusammenfassen läßt, die voraussichtlich für eine spätere Spezialisierung genutzt werden kann.

Wichtig: Die Klasse `Person` ist auch eine Oberklasse der Klasse `Angestellter`. Sie ist aber keine direkte Oberklasse. Umgekehrt ist auch die Klasse `Angestellter` eine (indirekte) Unterklasse der Klasse `Person`. Allgemein gilt, daß zwischen einer Ober- und einer Unterklasse beliebig viele (Unter- bzw. Ober-)Klassen vorhanden sein dürfen. Die Ober- bzw. Unterklassenbeziehungen sind also *transitiv*.

4.2.2 Vererbung

Anstatt die in einer Oberklasse zusammengefaßten gemeinsamen Eigenschaften in jeder Unterklasse erneut zu beschreiben, ist es günstiger, die Unterklassen nur noch mit einem Verweis auf die Oberklasse zu versehen. Dadurch wird für die Unterklasse festgelegt, daß für sie auch die Eigenschaften der Oberklasse gelten. In den einzelnen Klassenbeschreibungen müssen dann lediglich noch die für die jeweilige Klasse typischen Eigenschaften angegeben werden, die sie von den anderen Klassen unterscheidet. Auf diese Weise läßt sich der Umfang der Klassenbeschreibungen oft erheblich reduzieren.
Einer solcher Vermerk definiert praktisch eine Art Verwandtschaftsverhältnis zwischen Ober- und Unterklasse. Die Weitergabe von Eigenschaften der Ober- an die Unterklasse wird daher auch als *Vererbung* (engl.: *inheritance*) bezeichnet. Die Vererbung ist ein weiteres zentrales Prinzip der Objektorientierung.
Wichtig: Analog zu den Ober- und Unterklassenbeziehungen funktioniert auch die Vererbung transitiv. Eine Unterklasse erbt also nicht nur die speziellen Eigenschaften ihrer direkten Oberklasse, sondern auch die Eigenschaften all ihrer indirekten Oberklassen.

Vererbung wird in UML mit Hilfe eines Pfeils von der Unter- zur Oberklasse dargestellt. Für die zuvor verwendeten Klassen ergibt sich somit folgende Darstellung:



Die Abbildung zeigt, wie durch Spezialisierung und Generalisierung eine *Klassenhierarchie* (in Form eines Baums) entsteht. Die Höhe dieser Hierarchie ist prinzipiell nicht beschränkt, d.h. es können (in einem Pfad des Klassenbaums) beliebig viele Generalisierungen und Spezialisierungen stattfinden.

In Java wird Vererbung durch das Schlüsselwort `extends` gekennzeichnet:

`class` *Unterklasse* `extends` *Oberklasse* {...}

Die Klasse *Unterklasse* stellt also eine Erweiterung der Klasse *Oberklasse* dar, d.h. prinzipiell sind alle in *Oberklasse* deklarierten Variablen und Methoden implizit auch in *Unterklasse* vorhanden und können entsprechend genutzt werden.
Wenn also eine Klasse `Person` wie zuvor beschrieben deklariert wurde, können die Klassen `Mitarbeiter` und `Student` als deren direkte Unterklassen deklariert werden. Die Klasse `Angestellter` wird dann als direkte Unterklasse von `Mitarbeiter` deklariert:

Beispiel 39:

```
// Mitarbeiter als Unterklasse von Person
class Mitarbeiter extends Person {
  // ererbte Komponenten:
  // direkt von Person:     nachname, vorname, istweiblich

  // neu deklarierte Objektkomponenten:
  int personalnummer;

  // Konstruktor
  Mitarbeiter (String nachname, String vorname,
               int personalnummer, boolean istWeiblich) {
    this.nachname = nachname;
    this.vorname = vorname;
    this.personalnummer = personalnummer;
    this.istWeiblich = istWeiblich;
  }

  void ausgabe() {
    System.out.print(istWeiblich ? "Frau " : "Herr ");
    System.out.print(vorname + " " + nachname);
    System.out.println(" (Personalnummer: " + personalnummer + ")");
  }
}

// Student als Unterklasse von Person
```

```
class Student extends Person {
  // ererbte Komponenten:
  // direkt von Person:     nachname, vorname, istweiblich

  // neu deklarierte Objektkomponenten:
  String studienfach;
  byte semester;

  // Konstruktor
  Student (String nachname, String vorname, boolean istWeiblich,
           String studienfach, byte semester) {
    this.nachname = nachname;
    this.vorname = vorname;
    this.istWeiblich = istWeiblich;
    this.studienfach = studienfach;
    this.semester = semester;
  }
}

class Angestellter extends Mitarbeiter {
  // Vererbte Komponenten:
  // indirekt von Person:     nachname, vorname, istweiblich
  // direkt von Mitarbeiter:  personalnummer, ausgabe()

  // neu deklarierte Objektkomponenten:
  String tarifgruppe;

  // Konstruktor
  Angestellter (String nachname, String vorname,
                int personalnummer, boolean istWeiblich,
                String tarifgruppe) {
    this.nachname = nachname;
    this.vorname = vorname;
    this.personalnummer = personalnummer;
    this.istWeiblich = istWeiblich;
    this.tarifgruppe = tarifgruppe;
  }
}
```

Die Instanzvariablen `nachname`, `vorname` und `istWeiblich` werden in den Unterklassen durch die Vererbung implizit deklariert. Sie können daher in der Instanzmethode `ausgabe()` genauso verwendet werden wie die explizit deklarierte Instanzvariable `personalnummer`. Dies gilt auch für die Klasse `Angestellter`, die zusätzlich noch die Variable `personalnummer` und die Methode `ausgabe()` erbt.

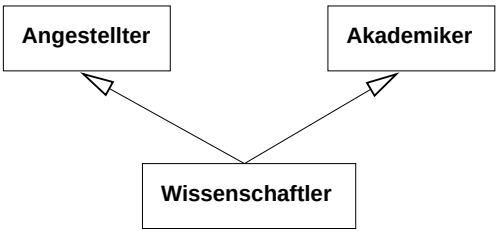
Bemerkung: Vererbung findet stets nur von Ober- zu Unterklasse statt. Da die Spezialisierung prinzipiell in derselben Richtung abläuft, werden die beiden Begriffe oft synonym verwendet. Die Spezialisierung ist aber eine Vorgehensweise zur Modellierung, während die Vererbung letzten Endes nur ein Hilfsmittel zur einfacheren Programmierung von Klassenhierarchien ist. Für die Programmierung ist es ohne Bedeutung, ob die einzelnen Ober- und Unterklassen der Klassenhierarchie durch Spezialisierung oder Generalisierung modelliert wurden.

4.2.3 Mehrfachvererbung

Die einfache Vererbung erlaubt nur die Erzeugung von Klassenhierarchien mit Baumstruktur. Innerhalb des Klassenbaums existiert somit jeweils genau ein Weg von der allgemeinsten Oberklasse (der Wurzel) zu einer beliebigen Unterklasse. Damit kann jede Unterklasse nur einer direkten Oberklasse angehören. Für die Modellierung von Klassenhierarchien ist es aber oft wünschenswert, die Eigenschaften mehrerer Oberklassen in einer Unterklasse zu vereinen. Dadurch lassen sich verschiedene Aspekte dieser Unterklasse separat auf einer allgemeinen Ebene modellieren.

Angenommen, es existiert noch eine weitere Klasse `Akademiker` zur Beschreibung der akademischen Lauf-

bahn eines Menschen (Titel, Abschlußnote usw.). Eine Klasse für wissenschaftliche Angestellte könnte dann auf elegante Weise mit zwei Oberklassen modelliert werden, nämlich `Angestellter` und `Akademiker`. Auf diese Weise könnte die Beschreibung des Arbeitsverhältnisses einer Person mit Informationen über deren Ausbildung direkt verknüpft werden.



Können mehrere Klassen ihre Eigenschaften an eine gemeinsame direkte Unterklasse vererben, so spricht man von *Mehrfachvererbung* (engl.: *multiple inheritance*). Der wesentliche Vorteil der Mehrfachvererbung besteht darin, daß unterschiedliche Eigenschaften, die in getrennten Oberklassen festgelegt wurden, einfach kombiniert werden können. Mehrfachvererbung kann aber auch zu Problemen führen, wenn die verschiedenen Eigenschaften nicht aufeinander abgestimmt sind.
Java erlaubt deshalb (im Gegensatz zu einigen anderen OO-Sprachen wie z.B. C++) prinzipiell keine Mehrfachvererbung, d.h. nach dem Schlüsselwort `extends` kann nur eine direkte Oberklasse angegeben werden. Eine aufwendigere, aber oft weniger problematische Alternative zur Mehrfachvererbung wird in Abschnitt 4.4 vorgestellt.

4.2.4 Klassenhierarchien in Java

Wie in den vorigen Kapiteln gezeigt, muß eine Klassendeklaration keine Angabe einer direkten Oberklasse (mit dem Schlüsselwort `extends`) enthalten. Wird keine Oberklasse angegeben, so wird implizit die vordefinierte Klasse `java.lang.Object` (kurz: `Object`) als direkte Oberklasse angenommen, um alle Objekte mit einer einheitlichen Grundfunktionalität auszustatten.
Die Klasse `Object` gehört zu den Standardklassen von Java. Der Bezeichner `java.lang.Object` ist ein sog. *vollständig qualifizierter Bezeichner*. Der Aufbau solcher Bezeichner wird im Zusammenhang mit der Einführung von *Paketen* in Abschnitt 5.1 besprochen werden.
Die im Beispiel verwendete Klassendeklaration

```
class Person {
  ...
}
```

ist somit gleichbedeutend zu

```
class Person extends Object {
  ...
}
```

Auch alle in Java definierten Standardklassen sind letzten Endes ebenfalls Unterklassen der Klasse `Object`. Dies hat zur Folge, daß alle Klassen direkt oder indirekt Unterklassen der Oberklasse `Object` sind. Die Hierarchie sämtlicher in einem Java-Programm verwendeten Klassen ist demnach immer ein Baum mit der Klasse `Object` als Wurzel.

4.2.5 Vererbung und Konstruktoren

Wie bereits an den vorigen Beispielen gezeigt wurde, wäre es oft bei der Deklaration von Unterklassen sinnvoll, auch die Konstruktoren der Oberklasse verwenden zu können, um die Konstruktoren der Unterklasse überschaubar zu halten. Konstruktoren werden aber nicht an Unterklassen weitervererbt, d.h. eine Klasse verfügt nicht automatisch über die für ihre Oberklasse definierten Konstruktoren. Statt dessen können innerhalb eines Konstruktors die Konstruktoren der direkten Oberklasse explizit oder implizit aufgerufen werden.
Es ist zu beachten, daß der Aufruf eines anderen Konstruktors nur zu Beginn eines Konstruktors stattfinden darf, d.h. ein derartiger Aufruf muß die erste Anweisung im Konstruktorrumpf sein. Es kann also nur *ein* anderer Konstruktor direkt aufgerufen werden. Weiterhin dürfen nur Konstruktoren aufgerufen

werden, die entweder in der Klasse selbst oder in ihrer direkten Oberklasse deklariert sind. Der Aufruf von Konstruktoren aus derselben Klasse ist vor allem dann sinnvoll, wenn man Konstruktoren mit bestimmten Vorgabewerten festlegen will, um die Anzahl der für die Objektinitialisierung benötigen Parameter zu reduzieren.

Konstruktoren können nicht wie Methoden über ihren Bezeichner (in diesem Fall also der Klassenname) aufgerufen werden. Statt dessen erfolgt der Aufruf über den Bezeichner `this` (gefolgt von der Liste der aktuellen Parameter) beim Aufruf eines anderen, in der selben Klasse deklarierten Konstruktors. Für den Aufruf eines Oberklassen-Konstruktors ist statt `this` das Schlüsselwort `super` zu verwenden. `super` erlaubt den Zugriff auf die in der Oberklasse deklarierten Objektkomponenten. Dies ist vor allem für sog. *überschriebene* Methoden hilfreich, die in Abschnitt 4.3.1 vorgestellt werden.
Wird in einer Klassendeklaration kein Konstruktor explizit deklariert, so wird implizit der sog. *Standard-Konstruktor* ergänzt, dessen Rumpf lediglich die Anweisung „`super();`“ enthält. Der Standard-Konstruktor ruft also den parameterlosen Konstruktor der direkten Oberklasse auf. Wird allerdings ein parametrisierter Konstruktor explizit deklariert, so erfolgt keine automatische Deklaration des Standard-Konstruktors.

Die Klasse `Student` kann z.B. den in der Klasse `Person` angegebenen Konstruktor aufrufen, so daß nur noch für die in der Klasse `Student` selbst deklarierten Variablen eine Initialisierung erforderlich ist.

Beispiel 40:

```
class Student extends Person {
  String studienfach;
  byte semester;

  // allgemeiner Konstruktor:
  Student (String nachname, String vorname, boolean istWeiblich,
           String studienfach, byte semester) {
    // Aufruf des Konstruktors von "Person" mit "super":
    super(nachname, vorname, istWeiblich);

    this.studienfach = studienfach;
    this.semester = semester;
  }

  // Konstruktor fuer Informatik-Studenten im ersten Semester:
  Student (String nachname, String vorname, boolean istWeiblich) {
    // Aufruf des Konstruktors von "Student" mit "this":
    this(nachname, vorname, istWeiblich, "Informatik", 1);
  }
}
```

Fehlt im Rumpf eines Konstruktors ein Aufruf eines anderen Konstruktors, so wird implizit der Aufruf `super();` als erste Anweisung ergänzt. Es wird also vor der Ausführung des Konstruktorrumpfs der parameterlose Konstruktor der direkten Oberklasse aufgerufen. Dadurch wird sichergestellt, daß in jedem Fall irgendwann ein Konstruktor der Oberklasse aufgerufen wird.

Ein Beispiel für die Nutzung dieser Eigenschaft wäre die Zählung aller Mitarbeiter. Dazu könnte die Klasse `Mitarbeiter` folgendermaßen ergänzt werden.

```
class Mitarbeiter extends Person {
  static int anzahlMitarbeiter = 0;
  int personalnummer;

  // Geaenderter Standard-Konstruktor
  // (Mitarbeiterzaehler wird bei jeder Instanziierung erhoeht)
  Mitarbeiter () {
    anzahlMitarbeiter++;
  }

  // Konstruktor
```

```
  Mitarbeiter (String nachname, String vorname,
               int personalnummer, boolean istWeiblich) {
    super(nachname, vorname, istWeiblich);
    anzahlMitarbeiter++;
    this.personalnummer = personalnummer;
  }

  void ausgabe() {
    System.out.print(istWeiblich ? "Frau " : "Herr ");
    System.out.print(vorname + " " + nachname);
    System.out.println(" (Personalnummer: " + personalnummer + ")");
  }
}
```

Wird eine Instanz einer Unterklasse von `Mitarbeiter` erzeugt, so wird zwangsläufig auch einer der Konstruktoren aufgerufen, die in der Klasse `Mitarbeiter` deklariert sind. Daher ist garantiert, daß der Wert von `anzahlMitarbeiter` auch wirklich alle kreierten `Mitarbeiter`-Objekte angibt. Der parameterlose Konstruktor erlaubt die Zählung von Unterklasseninstanzen auch bei impliziten Aufrufen des Konstruktors.

Hinweis: Um Probleme bei der Vererbung zu vermeiden, empfiehlt es sich stets, neben der Deklaration parametrisierter Konstruktoren auch einen parameterlosen Konstruktor zu definieren. Dadurch können implizite Aufrufe dieses Konstruktors fehlerfrei durchgeführt werden.

4.3 Polymorphismus

Ein weiteres Grundprinzip der Objektorientierung ist der sog. *Polymorphismus* (engl.: *polymorphism*), was sich am ehesten mit dem Begriff „Vielgestaltigkeit“ übersetzen läßt. Wird bei der Deklaration von Objektvariablen oder formalen Parametern als Datentyp eine Oberklasse angegeben, erlaubt der Polymorphismus auch die Verwendung eines Objekts als Wert bzw. aktueller Parameter, das einer beliebigen Unterklasse der in der Deklaration angegebenen Oberklasse angehört. Die Typdeklaration ist also keine strenge Festlegung auf eine einzelne Klasse, sondern erlaubt auch die Verwendung aller Unterklassen, da für diese sichergestellt ist, daß sie sämtliche Eigenschaften ihrer Oberklasse ebenfalls besitzen.
Die Vielgestaltigkeit bezieht sich somit darauf, daß über einen formalen Parameter bzw. eine Objektvariable (mit der Oberklasse als Typ) auch auf ein Objekt zugegriffen werden kann, dessen Typ beliebig aus einer Vielzahl unterschiedlicher Objekttypen (nämlich aller Unterklassen) gewählt sein kann.

Java unterstützt Polymorphismus, d.h. einer Objektvariable mit dem Typ `Oberklasse` kann auch ein Objekt zugewiesen werden, das zu einer Unterklasse von `Oberklasse` gehört. Dies gilt auch für die Verwendung von Objekten als aktuelle Parameter.

Beispiel 41:

```
class PolymorphismusBeispiel {
  public static void bezahlung (Mitarbeiter mitarbeiter) {
    System.out.println("Mitarbeiter " + mitarbeiter.nachname +
                       " bekommt Geld.");
  }

  public static void main (String[] args) {
    Mitarbeiter mitarbeiter;

    // Polymorphismus bei Zuweisung
    mitarbeiter = new Angestellter("Mayer", "Hans", false, 1, "IIa");

    // Polymorphismus bei Parameteruebergabe
    bezahlung(mitarbeiter);
  }
}
```

4.3.1 Überschreiben von Methoden

Methoden in Oberklassen sind meist sehr allgemein gehalten, da sie den Anforderungen all ihrer Unterklassen genügen müssen. Oft müssen solche Methoden für eine Unterklasse explizit verändert werden, um die erweiterten Eigenschaften korrekt wiederzugeben, oder es bietet sich aus Effizienzgründen an, die spezifischen Eigenschaften der Unterklasse in der Methode gezielt auszunutzen. In solchen Fällen können aus der Oberklasse geerbte Methoden innerhalb der Unterklasse verändert werden. Man spricht dabei vom *Überschreiben* der geerbten Methode.
Eine Methode der Oberklasse wird überschrieben, wenn in der Unterklasse eine Methode mit derselben Methodensignatur (gleicher Name, gleiche Liste formaler Parameterdeklarationen, gleicher Datentyp des Rückgabewerts) deklariert ist.

Beispiel 42:

```
class Angestellter extends Mitarbeiter {
  String tarifgruppe;

  // ueberschriebene Methode
  void ausgabe () {
    System.out.print(istWeiblich ? "Frau " : "Herr ");
    System.out.print(vorname + " " + nachname);
    System.out.println("  Personalnummer: " + personalnummer);
    System.out.println("  Tarifgruppe:    " + tarifgruppe);
  }
}

class UeberschreibungsBeispiel {
  public static void main (String[] args) {
    Mitarbeiter m;

    m = new Mitarbeiter("Mayer", "Hans", false, 1);
    m.ausgabe();    // ausgabe() der Klasse "Mitarbeiter" aufrufen

    m = new Angestellter("Schmidt", "Martin", false, 2, "IIa");
    m.ausgabe();    // ausgabe() der Klasse "Angestellter" aufrufen
  }
}
```

Wichtig: Beim Überschreiben von Methoden ist auch der Ergebnisdatentyp der Methode von großer Bedeutung. Aufrufe der überschriebenen Methode müssen aufgrund des Polymorphismus sowohl für die Ober- als auch für die Unterklasse korrekt sein, d.h. auch der Rückgabewert muß vom gleichen Typ sein.

Das Überschreiben steht in engem Zusammenhang mit dem sog. *dynamischen Binden* von Methodenaufrufen, das zur Umsetzung des Polymorphismus benutzt wird. Wird eine Instanzmethode eines Objekts aufgerufen, so wird zur Laufzeit die Klassendeklaration des Objekts durchsucht, ob eine Instanzmethode mit der entsprechenden Signatur in dieser Klasse deklariert ist. Bei erfolgreicher Suche wird der zugehörige Methodenrumpf ausgeführt. Andernfalls wird die direkte Oberklasse nach dem gleichen Verfahren überprüft, bis eine geeignete Methodendeklaration gefunden wurde.
Klassenmethoden dagegen sind fest an eine Klasse gebunden. Beim Aufruf einer Klassenmethode findet keine dynamische Bindung statt, d.h. auch Polymorphismus ist mit Klassenmethoden nicht möglich (und nicht sinnvoll).

4.3.2 Zugriff auf überschriebene Methoden

Wenn man die beiden Rümpfe der Methode `ausgabe()` der Klassen `Mitarbeiter` und `Angestellter` näher betrachtet, stellt man Gemeinsamkeiten fest. Im Rumpf der Unterklassenmethode kann man dies ausnutzen, da dort über das Schlüsselwort `super` die für die Oberklasse deklarierte Methode aufgerufen werden kann. Die Verwendung von `super` erfolgt dabei analog zu der Verwendung von `this`, d.h. `super` wird wie eine Objektreferenz eingesetzt.
Die Unterklasse `Angestellter` hat dann folgende Gestalt:

```
class Angestellter extends Mitarbeiter {
```

```
  String tarifgruppe;

  // ueberschriebene Methode
  void ausgabe () {
    super.ausgabe();
    System.out.println("  Tarifgruppe:    " + tarifgruppe);
  }
}
```

4.4 Abstrakte Klassen und Schnittstellen

Wie in Abschnitt 4.2.1 bereits erwähnt wurde, sind Oberklassen allgemeiner gehalten als ihre Unterklassen. *Abstrakte Klassen* erlauben die Festlegung von Klassen auf einem höheren Abstraktionsniveau, da sie den Programmierer davon entbinden, konkrete Methoden für bestimmte Problemstellungen zu deklarieren. Die Konkretisierung bleibt den Unterklassen überlassen.

4.4.1 Abstrakte Methoden

Abstrakte Methoden sind Instanzmethodendeklarationen ohne Methodenrumpf, d.h. es wird nur die Signatur der Methode festgelegt. Die Deklaration einer abstrakten Methode erfolgt in Java durch Voranstellen des Modifikators `abstract` vor den Methodenkopf. Die Angabe eines Methodenrumpfs entfällt. Stattdessen wird die Deklaration durch einen Strichpunkt abgeschlossen:

`abstract` *Methodenkopf*;

Enthält eine Klasse eine oder mehrere abstrakte Methoden, so wird sie dadurch zu einer *abstrakten Klasse*. Abstrakte Klassen dürfen in Java verwendet werden wie alle anderen Klassen auch, unterliegen aber einer wesentlichen Einschränkung: Eine Instanziierung abstrakter Klassen ist nicht möglich. Um eine Klasse explizit als abstrakt zu kennzeichnen, muß der Modifikator `abstract` auch vor dem Klassennamen gesetzt werden. Dies ist auch möglich, wenn keine abstrakte Methode innerhalb der Klasse deklariert wird.

`abstract` *Klassenname* { ... }

Bemerkung: Der häufigste Grund dafür, eine Klasse als abstrakt zu deklarieren, ist, daß für eine oder mehrere Methoden kein „vernünftiger“ Rumpf mit Anweisungen gegeben werden kann.

Ein Beispiel für eine abstrakte Methode könnte in die Klasse `Person` integriert werden. Die Methode `haupttaetigkeit()` ist für diese Klasse noch nicht sinnvoll realisierbar und sollte deshalb als abstrakt deklariert werden.

```
abstract class Person {
  String nachname;
  String vorname;
  boolean istWeiblich;

  Person () {
  }

  // Konstruktor
  Person (String nachname, String vorname, boolean istWeiblich) {
    this.nachname = nachname;
    this.vorname = vorname;
    this.istWeiblich = istWeiblich;
  }

  // Abstrakte Methode
  abstract void haupttaetigkeit();
}
```

Wichtig: Abstrakte Methoden werden, wie andere Methoden auch, vererbt. Erbt eine Klasse eine oder mehrere abstrakte Methoden, ohne diese zu konkretisieren, wird sie dadurch selbst zur abstrakten Klasse,

d.h. auch sie kann nicht instanziiert werden. Um eine abstrakte Methode zu konkretisieren, muß sie mit einer nicht-abstrakten Methodendeklaration überschrieben werden.

Im Beispiel müssen also die Klassen `Student` und `Mitarbeiter` um eine nicht-abstrakte Methodendeklaration für `haupttaetigkeit()` ergänzt werden.

```
class Mitarbeiter extends Person {
  int personalnummer;

  Mitarbeiter (String nachname, String vorname,
               int personalnummer, boolean istWeiblich) {
    super(nachname, vorname, istWeiblich);
    this.personalnummer = personalnummer;
  }

  void ausgabe() {
    System.out.print(istWeiblich ? "Frau " : "Herr ");
    System.out.print(vorname + " " + nachname);
    System.out.println(" (Personalnummer: " + personalnummer + ")");
  }

  // Ueberschreiben der geerbten abstrakten Methode
  void haupttaetigkeit() {
    System.out.println("Arbeiten");
  }
}

class Student extends Person {
  String studienfach;
  byte semester;

  Student (String nachname, String vorname, boolean istWeiblich,
           String studienfach, byte semester) {
    super(nachname, vorname, istWeiblich);
    this.studienfach = studienfach;
    this.semester = semester;
  }

  // Ueberschreiben der geerbten abstrakten Methode
  void haupttaetigkeit() {
    System.out.println("Lernen");
  }
}
```

4.4.2 Schnittstellen

Eine spezielle Form von abstrakten Klassen sind *Schnittstellen* (engl.: *interfaces*). Schnittstellen enthalten nur abstrakte Instanzmethoden und Konstanten, d.h. innerhalb einer Schnittstelle treten keinerlei Anweisungen oder Variablen auf. Damit ist eine Schnittstelle ab ihrer Deklaration unveränderlich festgelegt. Wie auch normale abstrakte Klassen kann eine Schnittstelle nicht instanziiert werden.
Eine Schnittstellen-Deklaration hat große Ähnlichkeit mit der Deklaration einer Klasse. An die Stelle des Schlüsselworts `class` tritt aber das Schlüsselwort `interface`.

`interface` *Schnittstellenname* { ... }

Da Schnittstellen vollständig abstrakten Klassen entsprechen und von daher auch komplett überschrieben werden müssen, ist in ihrem Fall eine normale Vererbung (mit dem Schlüsselwort `extends` gekennzeichnet) nicht sinnvoll. Stattdessen werden alle Schnittstellen, die von einer Klasse konkretisiert werden, nach dem Schlüsselwort `implements` angegeben (jeweils durch ein Komma getrennt). Damit wird ausgedrückt, daß die Klasse alle in den Schnittstellen angegebenen abstrakten Methoden implementieren muß, um die Spezifikation der Schnittstellen zu erfüllen. Die Angabe einer Oberklasse mit `extends` ist parallel dazu weiterhin möglich.

`class` *Klassenname* `implements` *Schnittstellennamen* { ... }

Wichtig: Im Gegensatz zur normalen Vererbung kann eine Klasse mehrere Schnittstellen implementieren. Da Schnittstellen wie Klassen als Datentypen verwendbar sind, kann durch Angabe von mehreren Schnittstellen (auch in Kombination mit einfacher Vererbung) eine leicht abgeschwächte Form der Mehrfachvererbung (siehe Abschnitt 4.2.3) erreicht werden.

Als Beispiel sei eine Schnittstelle `Akademiker` gegeben:

```
interface Akademiker {
  byte KEIN_ABSCHLUSS = 0;
  byte DIPLOM = 1;
  byte DOKTOR = 2;
  byte PROFESSOR = 3;

  void titel();
}
```

Soll nun eine Klasse `Wissenschaftler` zusätzlich noch Informationen über die akademische Laufbahn von Angestellten beinhalten, so kann dies folgendermaßen erreicht werden:

```
class Wissenschaftler extends Angestellter implements Akademiker {
  byte titel;

  Wissenschaftler (String nachname, String vorname,
                   int personalnummer, boolean istWeiblich,
                   int tarifgruppe, byte titel) {
     super(nachname, vorname, personalnummer, istWeiblich, tarifgruppe);
     this.titel = titel;
  }

  void haupttaetigkeit() {
    System.out.println("Forschen");
  }

  // Implementieren der abstrakten Methode
  void titel() {
    switch(titel) {
      case DIPLOM:
      case DOKTOR:
      case PROFESSOR:
        System.out.println("Hochschulabschluss");
        break;
      case KEIN_ABSCHLUSS:
      default:
        System.out.println("Kein Hochschulabschluss");
        break;
    }
  }
}
```

Hinweis: Bei den Konstanten und abstrakten Instanzmethoden einer Schnittstelle sollte auf die Angabe aller Arten von Modifikatoren verzichtet werden, da die Zugriffsrechte (siehe Kapitel 5) der einzelnen Komponenten implizit festgelegt sind. Generell gilt für alle Komponenten einer Schnittstelle, daß sie ohne Einschränkungen lesbar sind (`public`). Konstanten werden automatisch als nicht veränderbare Klassenvariablen betrachtet (`final static`). Methoden sind definitionsgemäß abstrakt (`abstract`).

Schnittstellen können in Java an Unterschnittstellen vererbt werden. Dies geschieht wie bei Klassen mit Hilfe des Schlüsselworts `extends`. Anders als bei Klassen ist allerdings für Schnittstellen Mehrfachvererbung zulässig, d.h. es dürfen nach `extends` auch mehrere Schnittstellen (jeweils durch ein Komma getrennt) angegeben werden.
Die Mehrfach-Vererbung kann zu Mehrdeutigkeiten bei Namen für Konstanten führen: Erbt eine Schnittstelle gleichnamige, aber verschiedene Konstanten von verschiedenen Schnittstellen, so muß der vollständig

qualifizierte Bezeichner (mit vorangestelltem Schnittstellenbezeichner) verwendet werden. Andernfalls kommt es zu einem Übersetzungsfehler.

Zusammenfassung: Unterschiede zwischen Klassen und Schnittstellen

- Schnittstellen sind implizit mit den Modifikatoren `abstract` und `public` versehen.
- Eine Schnittstellen-Deklaration umfaßt keine Anweisungen.
- Eine Konstante in einer Schnittstelle besitzt implizit die Modifikatoren `public`, `static` und `final`. In ihrer Deklaration muß sie stets mit einem Wert initialisiert werden.
- Eine Methode einer Schnittstelle besitzt implizit die Modifikatoren `public` und `abstract`; an Stelle des Methodenrumpfs steht nur ein Strichpunkt.
- Eine Klasse hat höchstens eine direkte Oberklasse, implemetiert aber ggf. mehrere Schnittstellen.
- Bei Schnittstellen gestattet Java Mehrfachvererbung, bei Klassen jedoch nicht.

4.4.3 Überprüfung von Objekteigenschaften

Aufgrund des Polymorphismus können mit der Angabe einer allgemeinen Klasse (im Extremfall `Object`) als Datentyp für Objektvariablen bzw. formale Parameter Objekte einer großen Zahl von Klassen einheitlich genutzt werden. Oft ist es aber wünschenswert, die Zugehörigkeit eines Objekts zu einer Klasse explizit zu überprüfen, um festzustellen, ob das Objekt die für diese Klasse charakteristischen Eigenschaften besitzt. Dies ist in Java mit dem `instanceof`-Operator möglich:

Objektvariable `instanceof` *Klassenname*

Der `instanceof`-Operator liefert einen Wert des Typs `boolean`, der anzeigt, ob die *Objektvariable* der mit *Klassenname* bezeichneten Klasse angehört. Wichtig ist dabei vor allem, daß nicht nur die Zugehörigkeit zu Klassen bzw. Oberklassen überprüft werden kann, sondern auch die Implementierung einer Schnittstelle durch eine Klasse. Anstelle eines Klassennamens kann auch ein Schnittstellenname verwendet werden. Dadurch lassen sich die Eigenschaften eines Objekts sehr genau überprüfen, auch wenn der deklarierte Typ des Objekts evtl. sehr allgemein und wenig konkret ist.

Bemerkung: Der Operatorenname `instanceof` ist etwas irreführend, da nicht überprüft wird, ob ein Objekt als Instanz einer speziellen Klasse erzeugt wurde. Nachdem eigentlich die Zugehörigkeit des Objekts zu der angegebenen Klasse bzw. Schnittstelle überprüft wird, wäre ein Name wie `memberof` hier vielleicht passender.

5 Zugriffskontrolle

Die genaue Kontrolle des Zugriffs auf Klassen und Schnittstellen ist eine wichtige Voraussetzung für die Entwicklung robuster und sicherer Programmsysteme. Die in Java vorhandenen Möglichkeiten zur Zugriffskontrolle werden in diesem Kapitel kurz vorgestellt.

5.1 Pakete

Oft umfaßt ein Java-Programm viele Klassen und Schnittstellen. Um Programme strukturierter zu gestalten, können Klassen und Schnittstellen in sog. *Paketen* (engl.: *packages*) zusammengefaßt werden. Die Pakete in Java entsprechen damit den Bibliotheken anderer Programmiersprachen. Auch die Java-Standardklassen werden in Paketen untergebracht.
Pakete erlauben außerdem eine Bereichsabgrenzung für die Zugriffskontrolle, d.h. es kann genau festgelegt werden, ob der Zugriff auf Klassen und Schnittstellen paketübergreifend möglich ist oder nicht. Damit stellen Pakete ein wichtiges Hilfsmittel bei der Gestaltung sicherer Programme dar.

Ein Paket kann Klassen, Schnittstellen und *Unterpakete* (engl.: *subpackages*) enthalten. Letztere sind selbst Pakete. Pakete sind also hierarchisch angeordnet. Bei dateiorientierten Systemen entspricht ein Paket einem Dateiverzeichnis mit allen in diesem Verzeichnis befindlichen Klassen- und Schnittstellendateien (mit der Endung `.class`). Den Unterpaketen eines Paketes entsprechen dann Unterverzeichnisse dieses Verzeichnisses.
Die Klassen, Schnittstellen und Unterpakete eines Paketes müssen unterschiedliche Namen besitzen (analog zu Dateiverzeichnissen, in denen auch keine zwei Dateien gleichen Namens existieren dürfen).

5.1.1 Vollständig qualifizierte Bezeichner

Ähnlich wie über absolute Pfadangaben in UNIX-Dateisystemen können Klassen, Schnittstellen und Pakete über ihren *vollständig qualifizierten Bezeichner* (engl.: *fully qualified identifier*) direkt angesprochen werden. Ein vollständig qualifizierter Bezeichner besteht dabei aus dem Namen des Pakets bzw. der Unterpakete, in dem die zu bezeichnende Komponente sich befindet, und dem Komponentenbezeichner selbst. Die einzelnen Paket- bzw. Komponentenbezeichner werden dabei wie bei zusammengesetzten Bezeichnern jeweils durch einen Punkt getrennt.
Sind z.B. die in den vorigen Kapiteln eingeführten Klassen in einem Paket `personalverwaltung` zusammengefaßt, so hätte die Klasse `Person` den vollständigen Bezeichner `personalverwaltung.Person`.
Ein Beispiel für ein Standard-Paket ist `java.lang` (`lang` ist ein Unterpaket des Paketes `java`), in dem u.a. auch die Klasse `Object` deklariert ist, die im vorigen Kapitel bereits eingeführt wurde (siehe Abschnitt 4.2).

Generell müssen alle Klassen und Schnittstellen in Java einem Paket zugeordnet sein. Der Zugriff auf sie ist dann nur noch über ihren vollständig qualifizierten Bezeichner möglich. Wird die Paketzugehörigkeit nicht explizit vereinbart (wie in den bisherigen Beispielen), so werden die Klassen und Schnittstellen einem *namenlosen* Paket zugeordnet, d.h. der einfache Bezeichner einer Komponente ist auch ihr vollständig qualifizierter Bezeichner.
In der Regel ist das namenlose Paket auf das Verzeichnis begrenzt, in dem die jeweilige Klassen- bzw. Schnittstellendatei sich befindet. Die Bezeichner von evtl. vorhandenen Unterpaketen entsprechen dann den jeweiligen Unterverzeichnisnamen.

5.1.2 Erstellung von Paketen

Um eigene Pakete zu erstellen, wird zu Beginn des Programmtextes (vor der Deklaration von Klassen und Schnittstellen) der Paketname explizit angegeben:

`package` *Paketname*;

Damit wird festgelegt, daß alle Klassen und Schnittstellen innerhalb der Programmtext-Datei zu dem Paket *Paketname* gehören. *Paketname* muß ein vollständig qualifizierter Bezeichner sein, d.h. bei der Deklaration eines Unterpakets müssen auch die Namen aller Oberpakete (jeweils durch Punkte getrennt) angegeben werden.
Fehlt in einem Java-Quelltext die `package`-Deklaration, so sind die deklarierten Klassen und Schnittstellen Teil eines namenlosen Paketes.

Beispiel 43:

```
// Deklaration des Paketnamens
package personalverwaltung;

class Person {
  String nachname;
  String vorname;
  boolean istWeiblich;

  Person () {
  }

  Person (String nachname, String vorname, boolean istWeiblich) {
    this.nachname = nachname;
    this.vorname = vorname;
    this.istWeiblich = istWeiblich;
  }
}
```

5.1.3 Die `import`-Deklaration

Im allgemeinen muß im Java-Quellcode eine Klasse oder Schnittstelle mit ihrem vollständig qualifizierten Namen angegeben werden. Da solche Bezeichner oft recht lang werden können, kann man `import`-Deklarationen einsetzen. Dadurch können die Bezeichner auch ohne Angabe ihres Paketes verwendet

werden. Im Prinzip entspricht eine `import`-Deklaration dem Setzen eines Suchpfads in einem Dateisystem.

`import` *Paketname*.*Typname*;

`import` *Paketname*.`*`;

Bei der ersten Form wird der vollständig qualifizierte Bezeichner für eine Klasse oder Schnittstelle angegeben. Anstelle von *Paketname*.*Typname* kann dann innerhalb der Textdatei der einfache Bezeichner *Typname* verwendet werden.
Bei der zweiten Form können alle Klassen und Schnittstellen innerhalb des angegebenen Paketes *Paketname* über ihren jeweiligen einfachen Bezeichner angesprochen werden.
In beiden Fällen werden lediglich Klassen und Schnittstellen importiert, die bei der `import`-Deklaration sichtbar sind (s. Unterabschnitt 5.2.1). Normalerweise müssen sie hierfür mit dem Schlüsselwort `public` versehen sein.

`import`-Deklarationen müssen im Programmtext vor der ersten Klassen- oder Schnittstellen-Deklaration stehen, aber hinter einer evtl. vorhandenen Paketnamen-Deklaration (mit dem Schlüsselwort `package`).

Beispiel 44:

```
// Deklaration des Paketnamens
package personalverwaltung;

// Importierung der Klasse java.lang.Object
// (Nicht zwingend erforderlich, siehe naechster Abschnitt)
import java.lang.Object;

class Person extends Object {
  String nachname;
  String vorname;
  boolean istWeiblich;

  Person () {
  }

  Person (String nachname, String vorname, boolean istWeiblich) {
    this.nachname = nachname;
    this.vorname = vorname;
    this.istWeiblich = istWeiblich;
  }
}
```

Wichtig: Bei der Verwendung verkürzter Bezeichner muß auf deren Eindeutigkeit geachtet werden. Sind z.B. zwei Klassen gleichen Namens in verschiedenen Paketen deklariert, die beide importiert werden, müssen für beide Klassen die vollständig qualifizierten Bezeichner verwendet werden, um eine eindeutige Zuordnung der Bezeichner zu gewährleisten.

5.1.4 Java-Standardpakete

Für Java existiert eine ganze Reihe von Standardpaketen, die Klassen und Schnittstellen für die verschiedensten Aufgaben zur Verfügung stellen. Die Standardpakete sind als Unterpakete in das Paket `java` eingegliedert. Die in `java` zusammengefaßten Pakete werden auch als *Java API* (*Application Programming Interface*) bezeichnet.
Eine besondere Stellung nimmt dabei das Paket `java.lang` ein. Dieses Paket enthält die für die Sprache Java zentralen Klassen und Schnittstellen, u.a. auch die Klasse `Object`, die als gemeinsame Oberklasse aller Objekte in Java eine entscheidende Rolle spielt. Da die in diesem Paket enthaltenen Klassen von den meisten Java-Programmen verwendet werden, werden implizit alle in `java.lang` deklarierten Klassen und Schnittstellen importiert (entsprechend der Deklaration `import java.lang.*;`). Dadurch können seine Klassen und Schnittstellen über ihre einfachen Bezeichner angesprochen werden.

Da im Java API sehr viele Klassen und Schnittstellen enthalten sind, soll an dieser Stelle nur ein grober Überblick vermittelt werden:

- `java.lang`: grundlegende Klassen und Schnittstellen (s. oben)
- `java.io`: Ein- und Ausgabe auf Kanäle, Dateien und Filter
- `java.net`: Unterstützung für Rechnernetze (Sockets, URLs, ...)
- `java.util`: Klassen für diverse Hilfsstrukturen, darunter Listen, Stacks, Hash-Tabellen, Zufallsgeneratoren, kalendarisches Datum und Zeitzonen
- `java.util.zip` (ab Java 1.1): Datenkompression
- `java.text` (ab Java 1.1): Unterstützung für Programme, die unterschiedliche Sprachen und Nationen berücksichtigen (z.B. Ausgabeformate für Zahlen, Datum oder Zeitzone)
- `java.math`: Unterstützung von Ganzzahl- und Gleitpunktzahl- Typen beliebiger Breite / Genauigkeit
- `java.lang.reflection` (ab Java 1.1): dynamischer Zugriff auf die geladenen Klassen und Schnittstellen (und deren Variablen, Methoden und Konstruktoren) während der Ausführung eines Programms
- `java.beans` (ab Java 1.1): Unterstützung von sog. *beans* (in Java geschriebene wiederverwendbare, modulare Software-Komponenten)
- `java.applet`: Hilfsklassen für *Applets* (Java-Programme, die unter einem Browser als Teil einer WWW-Seite ablaufen)
- `java.awt`: das sog. *Abstract Windowing Toolkit* (kurz: AWT), eine Sammlung von Klassen und Schnittstellen für graphische Benutzungsoberflächen
- `java.awt.event` (ab Java 1.1): das (weiterentwickelte) Ereignis-Modell des AWT
- `java.awt.image`: Verarbeitung von Bildern
- `java.awt.peer`: Schnittstellen, die für die Realisierung des AWT bedeutsam sind

5.2 Zugriffsbeschränkungen durch Modifikatoren

Bisher wurde davon ausgegangen, daß prinzipiell an jeder beliebigen Stelle eines Java-Programms auf alle Klassen und deren Komponenten zugegriffen werden kann. In der Praxis ist es aber oft nötig, die Zugriffsmöglichkeiten für bestimmte Klassen bzw. Komponenten einzuschränken oder komplett zu unterbinden.
Java bietet mit den Modifikatoren `public`, `protected` und `private` die Möglichkeit, die Sichtbarkeit bzw. den Gültigkeitsbereich von Variablen, Methoden und Klassen bzw. Schnittstellen genau festzulegen. Damit können exakt diejenigen Informationen über ein Programm, die zur Nutzung von Klassen und Schnittstellen erforderlich sind, verfügbar gemacht werden, während die übrigen Informationen vor evtl. unberechtigten Zugriffen geschützt sind.
Im allgemeinen sollten die Modifikatoren so eingesetzt werden, daß Zugriffsrechte so restriktiv wie möglich und nur so großzügig wie nötig gehandhabt werden.

5.2.1 Sichtbarkeit von Klassen und Schnittstellen

Ob auf ein Paket (und damit auf seine Klassen und Schnittstellen) zugegriffen werden darf, hängt vom verwendeten System ab, auf dem das Paket sich befindet. Die Sprachbeschreibung von Java macht hierzu keine näheren Angaben oder Forderungen. Die Erstellung, Erweiterung und Veränderung von Paketen sollte aber so abgesichert sein, daß nur der Paketentwickler selbst die Möglichkeit hat, das Paket bzw. die in diesem enthaltenen Klassen und Schnittstellen zu manipulieren.

Falls eine Klasse oder Schnittstelle mit dem Modifikator `public` (vor dem Schlüsselwort `class` bzw. `interface`) versehen ist, dann kann auf sie von jeder Stelle des Programms zugegriffen werden, von der aus auch auf das Paket zugegriffen werden kann, zu dem die Klasse bzw. Schnittstelle gehört.
Andernfalls können Zugriffe nur innerhalb des Paketes erfolgen, zu dem die Klasse bzw. Schnittstelle gehört.

5.2.2 Sichtbarkeit von Komponenten von Klassen und Schnittstellen

Als Komponenten einer Klasse werden in diesem Zusammenhang alle in der Klasse deklarierten Variablen, Methoden und Konstruktoren bezeichnet. Die Komponenten einer Schnittstelle sind ihre Konstanten und abstrakten Methoden. Der Zugriff auf eine Methode bzw. einen Konstruktor entspricht einem Aufruf dieser Methode bzw. dieses Konstruktors.
Innerhalb eines Programms ist der Zugriff auf Komponenten von Klassen genau dann möglich, wenn die Klasse selbst am Ort des Zugriffs sichtbar ist und die Deklaration der Komponente den Zugriff erlaubt.

Der Modifikator `public`

Der Modifikator `public` wird für uneingeschränkte Zugriffsrechte verwendet. Enthält die Deklaration einer Komponente diesen Modifikator, so ist der Zugriff auf sie gestattet. Auch Zugriffe aus anderen Paketen heraus sind ohne Einschränkung möglich.
Die Konstanten und abstrakten Methoden einer Schnittstelle sind implizit `public`. Daraus folgt, daß auf sie im ganzen Paket zugegriffen werden kann, zu dem die Schnittstelle gehört. Der Modifikator `public` sollte nur für die Komponenten gewählt werden, die für Zugriffe von außen vorgesehen sind. Variablen sollten nur in Ausnahmefällen (z.B. bei häufigem Zugriff) als `public` deklariert werden, um das Kapselungsprinzip nicht zu sehr aufzuweichen.

Der Modifikator `protected`

Der Modifikator `protected` schränkt die Zugriffsrechte dahingehend ein, daß Zugriffe auf die Komponenten nur noch von innerhalb desselben Pakets möglich sind. Die einzige Ausnahme sind Unterklassen der Klasse, die die Komponente beinhaltet. Diese können auf die Komponente zugreifen, auch wenn sie selbst einem anderen Paket angehören.
Der paketübergreifende Zugriff durch Unterklassen unterliegt einer zusätzlichen Beschränkung: Der Zugriff ist hier nur möglich, wenn die Komponente entweder über die Oberklassen-Referenz `super` angesprochen wird, oder wenn sie an die Unterklasse vererbt wurde und von daher in der Unterklasse selbst als Komponente verfügbar ist.
Der Zugriff auf einen Konstruktor, der als `protected` gekennzeichnet ist, ist demnach nur innerhalb eines Unterklassenkonstruktors über `super(...)` möglich. Der Konstruktor kann nicht für die Bildung einer neuen Instanz der Oberklasse verwendet werden.
Komponenten als `protected` zu deklarieren ist sinnvoll, wenn die Komponenten nicht zur allgemeinen Nutzung einer Klasse verwendet werden, aber zur Realisierung von Unterklassen auch in anderen Paketen benötigt werden könnten.

Standard-Zugriff

Wird einer Komponente keiner der Modifikatoren `public`, `protected` oder `private` vorangestellt, so spricht man von *Standard-Zugriff* (engl.: *default access*) oder auch der „Paket-Sichtbarkeit“. In diesem Fall ist der Zugriff auf die Komponente erlaubt, sofern er innerhalb desselben Pakets stattfindet, in dem auch die Komponente bzw. ihre Klasse deklariert sind.
Der Standard-Zugriff empfiehlt sich für alle Komponenten, die nur innerhalb eines Pakets, dort aber uneingeschränkt, nutzbar sein sollen.

Der Modifikator `private`

Durch `private` wird die schärfste Variante der Zugriffskontrolle gekennzeichnet. Komponenten mit diesem Modifikator sind nur innerhalb der Klasse verwendbar, in der sie deklariert werden. Für diese Komponenten findet auch keine Vererbung statt, d.h. auch für Unterklassen bleibt die Komponente unsichtbar.
Die Verwendung von als `private` deklarierten Komponenten bietet sich an für klasseninterne Zwecke. Die Komponente wird vor sämtlichen direkten Zugriffen von außerhalb der Klasse versteckt.

5.2.3 Unterbinden von schreibenden Zugriffen

Wie schon in Unterabschnitt 2.4.5 (s. S. 17) erläutert, dient der Modifikator `final` in Java dazu, nur lesende Zugriffe auf Variablen zuzulassen, schreibende Zugriffe aber zu unterbinden. Dadurch können unerwünschte Manipulationen verhindert werden. Konstanten werden auf diese Weise gegen jegliche Art von Veränderungen abgesichert.

5.2.4 Unterbinden des Überschreibens von Methoden

Wird eine Methode als `final` deklariert, so darf diese Methode nicht in einer Unterklasse überschrieben werden. Das garantiert, daß die Methode in jedem Fall das in der Oberklasse festgelegte Verhalten zeigt. Eine Manipulation durch das Ändern dieses Verhaltens in der Unterklasse ist somit ausgeschlossen. Wird eine Klassendeklaration mit `final` eingeleitet, so ist die gesamte Klasse gegen Änderungen und Erweiterungen geschützt, d.h. eine Verwendung dieser Klasse als Oberklasse (nach `extends`) ist nicht möglich.

5.3 Beispiele für den Einsatz von Zugriffsbeschränkungen

5.3.1 Nur-Lese-Zugriff

Oftmals soll ein Objekt eine Instanzvariable besitzen, die ausschließlich von Methoden seiner Klasse gesetzt wird; ihr Wert soll aber auch von Methoden in anderen Klassen abgefragt werden können. Ein solcher „Nur-Lese-Zugriff von außen“ wird realisiert, indem man die Variable als `private` deklariert und eine (nicht-`private`) Methode ergänzt, die ihren Wert als Ergebnis liefert.
Soll der Wert auch in Anweisungen ermittelt werden können, die sich nicht in demselben Paket befinden wie die Klasse, so müssen die Klasse und die Methode beide als `public` deklariert werden:

Beispiel 45: Implementierung eines Nur-Lese-Zugriffs

```
public class NurLeseZugriff {

  // die Variable:
  private byte variable = 8;

  // die Methode, die ihren Wert liefert:
  public byte liesVariable() {
    return variable;
  }

  ...

}
```

In Anweisungen, die innerhalb der Klasse stehen, kann nach wie vor schreibend auf die Variable zugegriffen werden.

5.3.2 Kontrolliertes Setzen einer Variablen

Das vorausgehende Beispiel soll erweitert werden um die Möglichkeit, den Wert der Variablen zu setzen; dieser schreibende Zugriff soll jedoch von der Klasse kontrolliert werden, z.B. um sicherzustellen, daß gewisse Bedingungen gelten. Hierzu wird die Klasse um eine weitere Methode ergänzt, die die Bedingung berücksichtigt, bevor der Wert der Variablen geändert wird.
Im folgenden Beispiel garantiert die Methode `setzeVariable`, daß der Wert von `variable` eine Dezimalziffer ist. Sie ist hier nicht als `public` deklariert, so daß sie nicht in Anweisungen außerhalb des Paketes der Klasse aufgerufen werden kann.

Beispiel 46: Kontrolliertes Setzen einer Variablen

```
public class KontrolliertesSetzen {

  // die Variable:
  private byte variable = 8;

  // eine Methode, die ihren Wert liefert:
  public byte liesVariable() {
    return variable;
  }

  // eine Methode, die ihren neuen Wert prueft und ggf. an sie zuweist:
```

```
    void setzeVariable(byte neu) {
      if (neu >= 0 && neu <= 9) variable = neu;
    }

    ...

  }
```

Entsprechend gibt es in vielen Klassen Methoden zum Lesen und Setzen von Variablen. Ihre Namen beginnen häufig mit `get`... bzw. `set`.... In der Dokumentation solcher Klassen findet man oft nur noch diese Methoden; die Variablen selbst sind dagegen nicht mehr genannt, da sie `private` und somit von außen nicht erreichbar sind. Von einer vorgegebenen Klasse werden also nur noch die Variablen und Methoden beschrieben, die ein Programmierer „von außen“ verwenden kann.
Es gibt praktisch kaum Variablen, die völlig unkontrolliert geändert werden dürfen. Es ist daher typisch, daß Variablen einer Klasse als `private` oder `protected` deklariert werden. Eine Ausnahme stellen Variablen dar, die `final` sind: Sie besitzen jeweils einen konstanten Wert, der überhaupt nicht verändert werden kann.

5.3.3 Kapselung innerhalb eines Paketes

Innerhalb der Klassen und Schnittstellen eines Paketes gibt es nur zwei Abstufungen für den Zugriff auf eine Variable, eine Methode oder einen Konstruktor: Mit `private` beschränkt sich der Zugriff auf die Klasse, in der die Komponente deklariert wurde. Bei Verwendung von `protected`, `public` oder Standard-Zugriff ist die Komponente von jeder Klasse oder Schnittstelle des Paketes aus erreichbar. Dies ist vor allem dann zu beachten, wenn mehrere Programmierer die Klassen eines Paketes entwickeln.

5.3.4 Zugriff auf Komponenten eines anderen Paketes

Soll auf eine Variable, eine Methode oder einen Konstruktor zugegriffen werden, die/der in einem anderen Paket deklariert ist als die Stelle des Zugriffs, so muß notwendigerweise gelten:

- Die Variable bzw. die Methode bzw. der Konstruktor ist als `public` oder `protected` deklariert.
- Die Klasse bzw. Schnittstelle, die die Deklaration enthält, ist `public` (s. Unterabschnitt 5.2.1).
- Auf das Paket der Klasse bzw. Schnittstelle kann zugegriffen werden (s. Unterabschnitt 5.2.1).

6 Ausnahmen

In diesem Kapitel wird die Ausnahmebehandlung in Java besprochen, d.h. der sichere und saubere Umgang mit Fehlern, die während des Programmlaufs auftreten.

Während der Ausführung von Java-Programmen kann es zu Laufzeitfehlern kommen. Beispiele für solche Fehler sind etwa:

- Beispiel 1: Es wird mit einem unzulässigen Index auf ein Feld zugegriffen, d.h. das angesprochene Feldelement existiert nicht.
- Beispiel 2: Es wird versucht, auf ein Objekt über die `null`-Referenz zuzugreifen.
- Beispiel 3: Es wird versucht, eine Datei zu öffnen, die nicht existiert.
- Beispiel 4: Es wird versucht, Daten aus einer Datei zu lesen, obwohl das Ende der Datei bereits erreicht ist.
- Beispiel 5: Der Arbeitsspeicher ist voll; es können keine weiteren Speicherbereiche reserviert werden.

Wird das Auftreten eines solchen Fehlers in einem Programm nicht gesondert behandelt, so wird die Ausführung abgebrochen, und man erhält eine Fehlermeldung, die u.a. den Fehlertyp beinhaltet.
Java stellt jedoch einem Programmierer Sprachmittel zur Verfügung, mit denen auf das Auftreten derartiger Fehler reagiert werden kann, ohne daß das Programm abgebrochen wird. Ein solcher Fehler wird eine *Ausnahme* (engl.: *exception*) genannt. Tritt während der Programmausführung ein Fehler auf, so

wird eine Ausnahme *ausgelöst* (engl.: *An exception is thrown.*). Wird ein Fehler behandelt, so wird die Ausnahme *abgefangen* (engl.: *The exception is caught.*).

Die Fehlertypen, für die oben fünf Beispiele genannt wurden, können klassifiziert werden. So deuten die ersten beiden Situationen auf unvorsichtige Programmierung hin. Die letzte Situation, ein voller Arbeitsspeicher, kann jederzeit auftreten, ohne daß der Programmierer dies in der Hand hat.
Die Einteilung dieser Fehlerarten geschieht in Java über Klassen: Die Fehlertypen sind Java-Klassen, und eine Ausnahme ist ein Objekt einer solchen Klasse. Tritt ein Fehler auf, so wird eine passende Instanz des Fehlertyps erzeugt, entweder implizit wie in den fünf oben genannten Situationen, oder explizit im Zusammenhang mit einer `throw`-Anweisung (s. unten in Abschnitt 6.2).
Die Klassen für die oben genannten Ausnahmen sind alle vordefiniert:

- Beispiel 1: `java.lang.ArrayIndexOutOfBoundsException`
- Beispiel 2: `java.lang.NullPointerException`
- Beispiel 3: `java.io.FileNotFoundException`
- Beispiel 4: `java.io.EOFException`
- Beispiel 5: `java.lang.OutOfMemoryError`

Für Ausnahmetypen, die Gemeinsamkeiten besitzen, gibt es wiederum gemeinsame Oberklassen. So haben `java.lang.ArrayIndexOutOfBoundsException` und `java.lang.NullPointerException` die gemeinsame Oberklasse `java.lang.RuntimeException`. Die gemeinsame Oberklasse aller Ausnahme-Klassen ist die Klasse `java.lang.Throwable`. Somit sind Ausnahmen genau Instanzen von `Throwable` (oder deren Unterklassen).

Die nun folgenden Unterabschnitte führen in die *Ausnahmebehandlung* (engl.: *exception handling*) in Java ein.

6.1 Abfangen einer Ausnahme: Die `try`-Anweisung

Möchte man auf das Auftreten eines Fehlers reagieren, so sollte dem kritischen Programmteil das Schlüsselwort `try` vorausgehen. Eine dort erzeugte Ausnahme kann dann in den `catch`-*Klauseln* abgefangen werden, die dem Programmteil folgen. Eine solche Klausel umfaßt u.a. die Angabe eines Ausnahmetyps, der angibt, welche Ausnahmen in der Klausel behandelt werden: Steht dort die Klasse `K`, so werden in der Klausel Ausnahmeobjekte abgefangen, die Instanzen von `K` oder von Unterklassen von `K` sind.
Tritt nun im kritischen Programmteil während der Ausführung eine Ausnahme auf, so wird zur textuell ersten passenden Klausel gesprungen, und es wird mit dem Block der Klausel fortgefahren. Es erfolgt also ein Sprung von der Stelle, die die Ausnahme ausgelöst hat, zu der Stelle im Programm, die sie abfängt.

Syntax der beiden Formen der `try`-Anweisung:

> `try` *Block Klauselliste*
>
> `try` *Block Klauselliste* `finally` *Block*

Ein *Block* ist eine Folge von Anweisungen, eingeschlossen in geschweiften Klammern stehen. Das `finally`-Konstrukt ist hierbei optional; es darf also auch fehlen.
Falls vorhanden, so wird der *Block* nach dem Schlüsselwort `finally` *in jedem Falle* ausgeführt, unabhängig davon, ob die Ausführung des *Blocks* nach `try` zu einer Ausnahme geführt hat oder nicht. Über `finally`-Konstrukte werden Anweisungen formuliert, die beim Verlassen der gesamten `try`-Anweisung abgearbeitet werden sollen, um z.B. Ressourcen wieder freizugeben oder einen konsistenten Zustand zu erreichen.

Syntax einer `catch`-Klausel:

> `catch (` *Typ Ausnahmeparameter* `)` *Block*

Der *Typ* muß der Name einer Ausnahmeklasse sein, d.h. `java.lang.Throwable` oder eine Unterklasse davon. Für den *Ausnahmeparameter* kann ein beliebiger Bezeichner gewählt werden. Im *Block* kann dann über diesen Bezeichner das Ausnahmeobjekt angesprochen werden. Der *Ausnahmeparameter* besitzt daher Ähnlichkeiten zu formalen Parametern einer Methode.

Das folgende Beispiel demonstriert eine `try`-Anweisung mit einer `catch`-Klausel für Ausnahmen vom Typ `java.lang.ArithmeticException`, die z.B. bei Ganzzahl-Division durch 0 ausgelöst werden. Beim Abfangen der Ausnahme wird ihr Typ ausgegeben.

Beispiel 47:

```
class TryAnweisung {

  public static void main(String args[]) {
    int i = 0;
    try {
      int k = 1 / i;  // Division durch 0
      System.out.println("Test");
    }
    catch (ArithmeticException e) {
      System.out.println(e);
    }
    System.out.println("Ende");
  }

}
```

Dieses Beispielprogramm gibt niemals das Wort „`Test`“ aus, da die zugehörige Anweisung stets übersprungen wird aufgrund der ausgelösten Ausnahme in der vorausgehenden Zeile.

Das nächste Beispiel demonstriert, daß `try`-Anweisungen selbst von anderen Anweisungen umschlossen sein können, etwa von einer `for`-Schleife. Es werden in dieser Schleife zehn Zufallszahlen zwischen 0.0 und 1.0 erzeugt. Falls bei einem Durchlauf ein Wert kleiner als 0.5 entsteht, so wird eine Ausnahme der Klasse `NullPointerException` erzeugt, indem verucht wird, über die Referenz `null` auf ein Feld zuzugreifen. Man beachte, daß die Ausgabe nach `finally` in jedem Fall erfolgt.

Beispiel 48:

```
class FinallyBeispiel {

  public static void main(String args[]) {
    for (int i = 1; i <= 10 ; i++) {
      char[] feld = null;
      char c;
      double d;
      try {
        d = java.lang.Math.random();
        System.out.print("Die " + i + ". Zufallszahl ist ");
        if (d < 0.5)
          c = feld[0];  // Ausnahme: Feldzugriff ueber die Referenz "null"
        System.out.println(d);
      }
      catch (NullPointerException e) {
        System.out.println("kleiner als 0.5");
      }
      finally {
        System.out.println("Ende des Schleifendurchlaufs");
      }
    }
  }

}
```

Da die textuell *erste* `catch`-Klausel zu einem Ausnahmeobjekt angesprungen wird, sollten in einer Klauselliste Klauseln für Unterklassen *vor* Klauseln für ihre Oberklassen stehen.
Da `NullPointerException` eine Unterklasse von `Throwable` ist, ist der folgende Programmausschnitt *nicht* sinnvoll:

```
catch (Throwable t) {
  ...
}
catch (NullPointerException n) {   // Diese Klausel wird nie angesprungen !
  ...
}
```

6.2 Explizites Auslösen einer Ausnahme: Die `throw`-Anweisung

Ausnahmen können durch die `throw`-Anweisung auch explizit ausgelöst werden. Hierzu muß nach dem Schlüsselwort `throw` lediglich ein Ausdruck angegeben werden, der das Ausnahmeobjekt angibt. Der Typ dieses Ausdrucks muß dabei die Klasse `Throwable` oder eine ihrer Unterklassen sein. Oft wird das Ausnahmeobjekt direkt mittels `new` erzeugt.
Im folgenden wird das letzte Beispiel nochmals aufgegriffen und durch den Einsatz von `throw` vereinfacht:

Beispiel 49:

```
class ThrowBeispiel {

  public static void main(String args[]) {
    for (int i = 1; i <= 10 ; i++) {
      double d;
      try {
        d = java.lang.Math.random();
        System.out.print("Die " + i + ". Zufallszahl ist ");
        if (d < 0.5)
          throw new NullPointerException();
        System.out.println(d);
      }
      catch (NullPointerException e) {
        System.out.println("kleiner als 0.5");
      }
      finally {
        System.out.println("Ende des Schleifendurchlaufs");
      }
    }
  }

}
```

6.3 Aufrufe und Ausnahmen: Die `throws`-Klausel

Es kann vorkommen, daß eine Ausnahme ausgelöst, aber nicht abgefangen wird, da eine geeignete `catch`-Klausel in einer `try`-Anweisung fehlt. In diesem Fall wird die Ausführung der aktuellen Methode beendet und die Ausnahme an die aufrufende Methode weitergereicht. Somit sind nicht-behandelte Ausnahmen auch mögliche Ergebnisse von Methodenaufrufen. Entsprechendes gilt für Konstruktoren.
In diesem Fall kann die aufrufende Methode die Ausnahme abfangen, wie das folgende Beispiel zeigt:

Beispiel 50:

```
class AusnahmeBeiAufruf {

  static void erzeugeAusnahme() {
    throw new NullPointerException();  // eine nicht-behandelte Ausnahme
  }

  public static void main(String args[]) {
    try {
      erzeugeAusnahme();
      System.out.println("Alles in Ordnung.");  // wird niemals ausgegeben
    }
```

```
    catch (Throwable t) {
      System.out.println(t);
    }
    System.out.println("Ende");
  }

}
```

Eine nicht behandelte Ausnahme wird auf diese Weise immer an die aufrufende Methodeninkarnation weitergereicht, bis schließlich eine geeignete `catch`-Klausel erreicht ist. Fehlt eine passende Klausel völlig, so wird das Programm (bzw. der Thread) abgebrochen.
Um Java-Programme sicherer zu gestalten, muß eine Methode meist bekannt geben, welche Typen von Ausnahmen sie auslösen kann, ohne sie abzufangen. Dies geschieht, indem direkt vor den Methodenrumpf das Schlüsselwort `throws` mit einer durch Kommata getrennten Liste der Ausnahmeklassen geschrieben wird:

`throws` *Liste der Ausnahmetypen*

Wenn ein Programmierer einen Methodenaufruf formuliert, so kann er der Methodendeklaration nicht nur den Ergebnistyp entnehmen, sondern auch die Typen der Ausnahmen, die der Methodenaufruf liefern kann. Um den Methodenaufruf richtig einzusetzen, muß bei der Programmierung beides berücksichtigt werden.

Wie aber das vorausgehende Beispiel zeigt, ist diese Angabe nicht immer nötig: Sie darf z.B. bei den Klassen `java.lang.Error` und `java.lang.RuntimeException` sowie deren Unterklassen entfallen.
Die erstere, `Error`, ist die Klasse aller Ausnahmen, die jederzeit „unkontrolliert“ auftreten können und von denen sich ein Programm so gut wie nicht erholen kann. Ein Beispiel hierfür ist die Unterklasse `OutOfMemoryError`, deren Ausnahmen ausgelöst werden, wenn während der Ausführung kein freier Arbeitsspeicher mehr vorhanden ist.
Unter der zweiten oben genannten Ausnahmeklasse, `RuntimeException`, werden Ausnahmen zusammengefasst, die so häufig auftreten können, daß ihre Auflistung nach `throws` umständlich wäre und die Programme schlechter lesbar machen würde. Dies trifft insbesondere für `ArithmeticException` und eben `NullPointerException` zu.
Die Ausnahmen aller verbleibenden Ausnahmeklassen werden *geprüfte Ausnahmen* (engl.: *checked exceptions*) genannt, da der Übersetzer ihre Nennung in `throws`-Klauseln verlangt. Ein Beispiel hierfür ist `java.io.IOException`. Man vergleiche den obigen Programmtext mit dem folgenden:

Beispiel 51: Methodendeklaration mit `throws`

```
class AusnahmeBeiAufruf {

  static void erzeugeAusnahme() throws java.io.IOException {
    throw new java.io.IOException();  // eine nicht-behandelte Ausnahme
  }

  public static void main(String args[]) {
    try {
      erzeugeAusnahme();
      System.out.println("Alles in Ordnung.");
    }
    catch (Throwable t) {
      System.out.println(t);
    }
    System.out.println("Ende");
  }

}
```

Dieser Programmtext wäre ohne die `throws`-Klausel fehlerhaft.

Überschreibung und `throws`-Klauseln

Wenn eine Methode $\tilde{\texttt{m}}$ in einer Unterklasse eine weitere Methode `m` in einer Oberklasse überschreibt, so darf $\tilde{\texttt{m}}$ nicht mehr geprüfte Ausnahmeklassen mittels `throws` deklarieren als `m`: Wenn in der `throws`-Klausel von $\tilde{\texttt{m}}$ eine Ausnahmeklasse `K` vorkommt, dann muß die `throws`-Klausel von `m` die Klasse `K` oder eine Oberklasse von `K` enthalten.

6.4 Benutzerdefinierte Ausnahmeklassen

Statt lediglich die vordefinierten Ausnahmetypen zu verwenden, kann ein Programm auch eigene Klassen für Ausnahmen einführen, indem es neue Unterklassen von `java.lang.Throwable` enthält. Diese Unterklassen werden dann genauso verwendet, wie es schon für die vordefinierten in den letzten Unterabschnitten vorgeführt wurde: Ausnahmeobjekte werden mit `new` kreiert, mit `throw` ausgelöst und durch geeignete `catch`-Klauseln abgefangen. Ist die neue Ausnahmeklasse keine Unterklasse von `Error` oder `RuntimeException`, so handelt es sich um eine geprüfte Ausnahmeklasse, die somit ggf. in `throws`-Klauseln genannt werden muß.
Da ein Programmierer die Gestalt seiner Ausnahmeklassen selbst bestimmt, kann er über geeignete Instanzvariablen festlegen, welche Information über eine Fehlersituation in einem Ausnahmeobjekt enthalten ist. Somit kann eine Ausnahme u.a. eine Beschreibung des Laufzeitfehlers enthalten.
Beispiele für diese Techniken enthält der folgende Unterabschnitt.

6.5 Abbrechen einer Schleife

In Unterabschnitt 2.8.6 wurde bereits erwähnt, daß Schleifen auch über die `break`-Anweisung verlassen werden können. Ein ähnlicher Effekt kann durch eine Ausnahme erzielt werden, die im Schleifenkörper ausgelöst wird: Wird sie erst außerhalb der Schleife durch eine passende `catch`-Klausel abgefangen, so erfolgt ein Sprung aus dem Schleifenkörper heraus.
Ein kleines Beispiel mit einer eigens deklarierten Ausnahmeklasse soll dies verdeutlichen: Hier wird mittels `throw` eine Endlos-Schleife abgebrochen.

Beispiel 52:

```
class NeueAusnahme extends Exception {
}

class ThrowInSchleife {

  public static void main(String args[]) {
    try {
      while (true) {
       System.out.println("Schleife wird abgearbeitet");
       throw new NeueAusnahme();
      }
    }
    catch (NeueAusnahme e) {
      System.out.println("Schleife abgebrochen");
    }
  }

}
```

Das folgende Beispiel enthält eine `for`-Schleife, die ein Feld von Zeichen (vom Typ `char`) abarbeitet, als Feldelemente Dezimalziffern erwartet und den Inhalt des Feldes in eine ganze Zahl umrechnet. Für den Fall, daß das Feld die Länge 0 hat, wird eine Ausnahme angelöst; hierfür ist die Klasse `LeeresFeld` deklariert. Entsprechendes gilt für die Klasse `UngueltigeZiffer`, falls das Feld ein Zeichen enthält, das keine Dezimalziffer ist.

Beispiel 53:

```
class LeeresFeld extends Exception {
}
```

```
class UngueltigeZiffer extends Exception {

  char zeichen;   // das unzulaessige Zeichen, das gefunden wurde
  int position;   // der Feldindex, bei dem es gefunden wurde

  UngueltigeZiffer(char z, int pos) {
    zeichen = z;
    position = pos;
  }

}

class FeldMitDezimalziffern {

  static long wert(char[] feld) throws LeeresFeld, UngueltigeZiffer {
    if( feld.length == 0 )
      throw new LeeresFeld();
    int ergebnis = 0;
    for (int i = 0; i < feld.length; i++) {
      if (feld[i] < '0' || feld[i] > '9')
        throw new UngueltigeZiffer( feld[i], i );
      ergebnis = ergebnis * 10 + feld[i] - '0';
      if (ergebnis < 0)
        throw new ArithmeticException();   // arithmetischer Ueberlauf
    }
    return ergebnis;
  }

  public static void main(String args[]) {
    try {
      char feld1 = { '8' };
      System.out.println( "Wert von feld1: " + wert( feld1 ) );
      char feld2 = { '4', '3', '6', '5', '9', '0' };
      System.out.println( "Wert von feld2: " + wert( feld2 ) );
      char feld3 = { '2', '7' ,'?', '1' };
      System.out.println( "Wert von feld3: " + wert( feld3 ) );
    }
    catch (UngueltigeZiffer e) {
      System.out.println("Ungueltige Dezimalziffer: '"
        + e.zeichen + "' an Position " + e.position);
    }
    catch (Throwable t) {
      System.out.println(t);
    }
  }

}
```

6.6 Weitere Details

Bisher wurde noch nicht angesprochen, daß auch die Blöcke nach `catch` oder `finally` Ausnahmen auslösen können. Tatsächlich kann die Ausführung einer `try`-Anweisung auch mit einer ausgelösten, aber noch nicht abgefangenen Ausnahme enden, und zwar in den folgenden drei Fällen:

1. Im Block nach `try` wird eine Ausnahme ausgelöst, für die es keine passende `catch`-Klausel gibt. In diesem Fall wird zunächst der `finally`-Block angesprungen, falls vorhanden. (Man beachte hierbei Punkt 3 !)

2. In einem `catch`-Block wird eine Ausnahme ausgelöst (und nicht behandelt). In diesem Fall wird zunächst der `finally`-Block angesprungen, falls vorhanden. (Man beachte hierbei Punkt 3 !) Die

Klauselliste wird aber *nicht* nochmals betrachtet, um eine passende Klausel zu finden.

3. Im `finally`-Block wird eine Ausnahme ausgelöst (und nicht behandelt).

Die Ausnahmen unter Punkt 3 bewirken jeweils, daß ggf. eine vorausgegangene unbehandelte Ausnahme „vergessen“ wird.
`try`-Anweisungen dürfen verschachtelt werden. Daher können die Ausnahmen aller drei Fälle durch Klauseln einer weiteren `try`-Anweisung abgefangen werden, die die erstere `try`-Anweisung in ihrem `try`-Block umfaßt.
Doch bevor man durch Verschachtelung ein Programm verunstaltet, empfiehlt es sich, bei der Programmierung der `catch`- und `finally`-Blöcke besonders sorgfältig vorzugehen, um dort das Auslösen von Ausnahmen zu vermeiden.

7 Threads

Dieses Kapitel befaßt sich mit nebenläufig ablaufenden Kontrollflüssen in einem Java-Programm, den Threads.

7.1 Motivation

Meistens betrachtet man sequentielle Abläufe, bei denen die einzelnen Schritte nacheinander ausgeführt werden. Dabei könnten einige Schritte auch nebenläufig oder parallel erledigt werden.
Man denke etwa an Fahrzeuge im Straßenverkehr: Im allgemeinen kann jeder Fahrer sein Fahrzeug lenken, ohne dabei von anderen abhängig zu sein. Wenn ein Fahrzeug fährt, so müssen deswegen nicht alle anderen Verkehrsteilnehmer stehenbleiben und warten, bis es an seinem Ziel angekommen ist.
Eine vergleichbare Situation gibt es bei Programmen: Zum Beispiel können Daten gleichzeitig auf dem Bildschirm ausgegeben und in eine Datei geschrieben werden. Diese beiden Vorgänge dürfen also parallel ablaufen.
Dennoch gibt es Einschränkungen: Im Straßenverkehr dürfen an Kreuzungen oder Engstellen nicht alle gleichzeitig fahren, oder es kommt zu Unfällen. Um sie zu vermeiden, gibt es Regelungen, die die Vorfahrt festlegen. Solche Regelungen sollten andererseits keine zu große Einschränkung bedeuten: Wenn möglich, so sollte kein Fahrzeug unnötig lange warten müssen, bevor es weiterfahren darf. Außerdem sollte jeder irgendwann am Ziel ankommen können, ohne unterwegs für immer blockiert zu sein.
Verkehrsampeln erfüllen diese beiden Forderungen: Sie sorgen einerseits dafür, daß zwei Fahrzeuge nicht an den Kreuzungspunkten ihrer Fahrtstrecken kollidieren, und garantieren andererseits, daß jeder irgendwann grünes Licht bekommt und an einer Kreuzung nicht ewig warten muß.
Überträgt man diese beiden Forderungen auf parallele Abläufe bei Programmen, so nennt man sie „wechselseitiger Ausschluß“ bzw. „schwache Fairness“. In Java gibt es Sprachelemente, die beides realisieren: Die Fahrzeuge entsprechen den Threads und die Verkehrsampeln den Sperren.

7.2 Die Klasse `Thread`

Bei den bisher präsentierten Java-Programmen wurden alle Rechenschritte sequentiell ausgeführt. Parallele Verarbeitung fand nicht statt. Nun werden aber Programme betrachtet, die aus mehreren Kontrollflüssen bestehen, die nebenläufig voranschreiten. Ein solcher Kontrollfluß wird *Thread* genannt. In Java ist ein Thread ein Objekt der vordefinierten Klasse `java.lang.Thread`.
Neue Threads können demnach erzeugt werden, indem mit `new` ein neues `Thread`-Objekt kreiert wird. Allerdings wird dadurch der zusätzliche Kontrollfluß noch nicht gestartet. Hierfür muß erst die Instanzmethode `start()` für dieses Objekt aufgerufen werden. Dieser Aufruf bewirkt unter anderem, daß die Instanzmethode `run()` zu dem Objekt aufgerufen wird, womit die Abarbeitung des neuen Threads beginnt. Diese Methode ist somit vergleichbar mit der Methode `main(...)` einer startbaren Java-Klasse.
Die Methode `run()` der vordefinierten Klasse `Thread` hat jedoch einen Rumpf ohne Anweisungen. Einen neuen Thread von dieser Klasse zu erzeugen und zu starten ist deshalb wenig sinnvoll: Die Ausführung des Threads wäre wirkungslos. Stattdessen wird eine Programmiertechnik eingesetzt, die in Abschnitt 8.1 skizziert wird: Man deklariere eine neue Unterklasse der Klasse `Thread` und überschreibe dort die Methode `run()`. In dieser Methode stehen dann die Anweisungen, die den parallel ablaufenden Code bestimmen.

Beispiel 54: Erzeugen und Starten eines Threads

```
class NeuerThread extends Thread {

  public void run() {
    System.out.println("Hier laeuft ein neuer Thread.");
  }

}

class ThreadBeispiel {

  public static void main(String args[]) {
    Thread t = new NeuerThread();
    t.start();
  }

}
```

Da die `run()`-Methode der Klasse `Thread` den Modifikator `public` besitzt, muß die überschreibende Methode `run()` in der Unterklasse `NeuerThread` ebenfalls als `public` deklariert sein.

Eine `Thread`-Unterklasse kann natürlich auch mehrmals instanziiert werden, wobei dann alle erzeugten Objekte gestartet werden können. Die Klasse `PrintThread` des folgenden Beispiels soll dies demonstrieren; sie ist außerdem zugleich eine startbare Klasse mit einer `main()`-Methode. Dort werden drei Instanzen erzeugt und gestartet, die jeweils in einer Endlos-Schleife erst ein Wort ausgeben und sich dann für höchstens eine Sekunde deaktivieren.

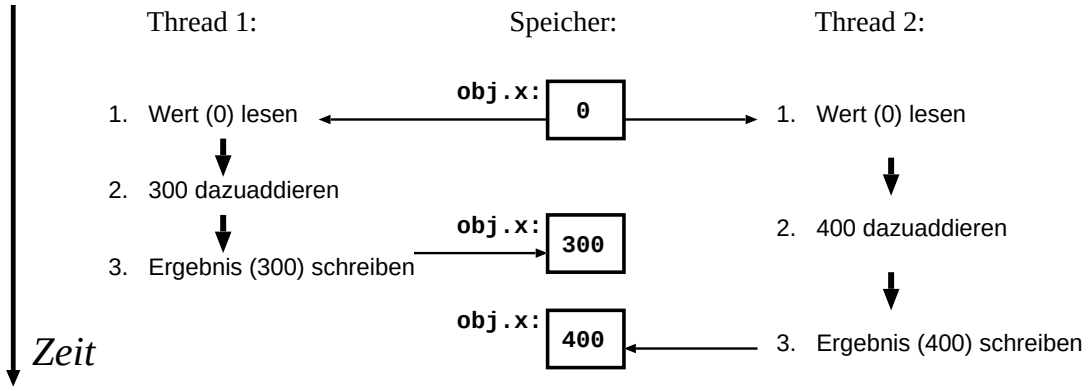
Beispiel 55:

```
class PrintThread extends Thread {

  private String text;

  PrintThread( String text ) {
    this.text = text;
  }

  public void run() {
    while (true) {
      System.out.println( text );
      try {
        // Bilde eine ganzzahlige Zufallszahl zwischen 0 und 1000:
        int millisekunden = (int) ( 1000 * Math.random() );
        sleep( millisekunden );
      }
      catch( Exception e ) break;
    }
  }

  public static void main(String args[]) {
    new PrintThread("morgens").start();
    new PrintThread("mittags").start();
    new PrintThread("abends").start();
  }

}
```

7.3 Synchronisation zwischen Threads

Wenn zwei Fahrzeuge aus verschiedenen Richtungen gleichzeitig durch eine Engstelle fahren, so führt dies in der Regel zu keinem erfreulichen Ergebnis. Auch wenn zwei Threads sich in die Quere kommen, ist das Resultat oft ein anderes als erwartet. Angenommen, zwei Threads möchten gleichzeitig den Wert

einer Variablen erhöhen, etwa durch die Anweisung `obj.x = obj.x + 300` bzw. `obj.x = obj.x + 400`. Man denke hierbei etwa an den Stand eines Bankkontos, auf das zwei Beträge zur selben Zeit eingezahlt werden.
War der Wert der Variable anfangs gleich 0 (d.h. das Konto war leer), so würde man ein Resultat von 700 erwarten. Greifen die beiden Threads jedoch zeitlich verzahnt auf `obj.x` zu, so kann es vorkommen, daß beide den Wert 0 vorfinden, das Teilergebnis 300 bzw. 400 berechnen und schließlich an `x` zuweisen möchten. Da jedoch nur die zeitlich letzte Zuweisung wirkt, wird nur eines der beiden Teilergebnisse in `x` dauerhaft gespeichert, an Stelle des gewünschten Wertes 700. Der Kontoinhaber würde also um eine der beiden Gutschriften betrogen.
Die folgende Skizze deutet eine Möglichkeit für den zeitlichen Verlauf der einzelnen Schritte der beiden Threads an, bei der `obj.x` schließlich den unerwünschten Wert 400 erhält:



Die beiden Anweisungen in den Threads (jeweils mit 2 Variablenzugriffen: erst lesend, dann schreibend), bei denen falsche Resultate aufgrund verzahnter paralleler Ausführung auftreten können, bilden einen *kritischen Bereich*. Um derartige Fehlberechnungen zu verhindern, müssen Threads in bestimmten Situationen synchronisiert werden. Dies erfolgt über *Sperren* (engl.: *locks*): Zu jedem Objekt gibt es eine Sperre, d.h. einen Zähler, der anfangs mit 0 initialisiert ist und von Threads um 1 erhöht und erniedrigt werden kann.
Wenn ein Thread eine Sperre von 0 auf 1 erhöht, so ist er von diesem Moment an der „Eigentümer“ der Sperre. Er bleibt solange der Eigentümer, bis er wieder den Zähler von 1 auf 0 erniedrigt und die Sperre somit freigibt. Während dieser Zeit kann kein anderer Thread den Zähler ändern; bei einem Versuch werden andere Threads blockiert, d.h. ihre Ausführung wird unterbrochen.
Sperren werden jedoch in einem Programm nicht direkt manipuliert. Java bietet stattdessen hierfür das Schlüsselwort `synchronized` an, mit dem der wechselseitige Ausschluß für kritische Bereiche realisiert wird: Mittels `synchronized` wird ein kritischer Bereich festgelegt, in dem zu jedem Zeitpunkt höchstens ein Thread voranschreitet, nämlich der Eigentümer der zugehörigen Sperre.

Syntax der `synchronized`-Anweisung:

`synchronized ( `*Ausdruck*` )` *Block*

Der *Ausdruck* muß ein Objekt angeben. Liefert der *Ausdruck* dagegen `null`, so wird eine Ausnahme vom Typ `NullPointerException` ausgelöst.
Nach der Auswertung des *Ausdrucks* versucht ein Thread T, die zum Objekt gehörige Sperre um 1 zu erhöhen. Dies gelingt im nur, falls der Zähler der Sperre in diesem Moment den Wert 0 hat oder T der Eigentümer der Sperre ist; ansonsten wird T blockiert, d.h. die Ausführung von T schreitet nicht voran. Nach dem Erhöhen beginnt T mit der Abarbeitung des *Blocks*, und während dieser Zeit gilt: Führen weitere Threads ebenfalls eine `synchronized`-Anweisung für dasselbe Objekt aus, so werden sie blockiert, da T gerade Eigentümer der Sperre ist.
Wenn T den *Block* verläßt, wird der Zähler der Sperre erniedrigt. Erhält er dadurch den Wert 0, so darf einer der blockierten Threads fortfahren und wird zum Eigentümer der Sperre; die übrigen Threads bleiben weiterhin blockiert.
Damit ist das oben beschriebene Problem lösbar: Die Erhöhung von `obj.x` muß als Anweisung jeweils nach `synchronized` für das Objekt `obj` geschrieben werden:

```
synchronized(obj) { obj.x = obj.x + 300; }
```

bzw.

```
synchronized(obj) { obj.x = obj.x + 400; }
```

Da die zwei Anweisungen gemeinsam den kritischen Bereich bilden, müssen sie *beide* mit `synchronized` geschrieben werden, da ansonsten nach wie vor falsche Ergebnisse möglich sind.
Man beachte, daß durch `synchronized` und die Sperren nicht der Zugriff auf Objekte verhindert wird. Vielmehr dienen Objekte dazu, um mittels `synchronized` deutlich zu machen, welche Programmteile unter wechselseitigem Ausschluß abgearbeitet werden. Wenn zu einem Zeitpunkt mehrere Threads jeweils `synchronized`-Anweisungen zu demselben Objekt erreicht haben, so ist garantiert, daß höchstens einer unter ihnen die zugehörige *Anweisung* ausführt — die anderen sind blockiert. Für welchen unter ihnen als nächstes die Blockierung aufgehoben wird, kann nicht vorausgesagt werden. Es ist also offen, welcher der wartenden Threads in diesem Fall zuerst voranschreiten darf.
Des weiteren ist `synchronized` auch ein zulässiger Modifikator für Methoden: Wird eine Instanzmethode als `synchronized` bezeichnet, so hat dies dieselbe Wirkung wie das Umschließen des Methodenrumpfes mit `synchronized(this)`. Es ist also beispielsweise

```
synchronized void erhoehe(int delta) {
  obj.x = obj.x + delta;
}
```

gleichbedeutend mit

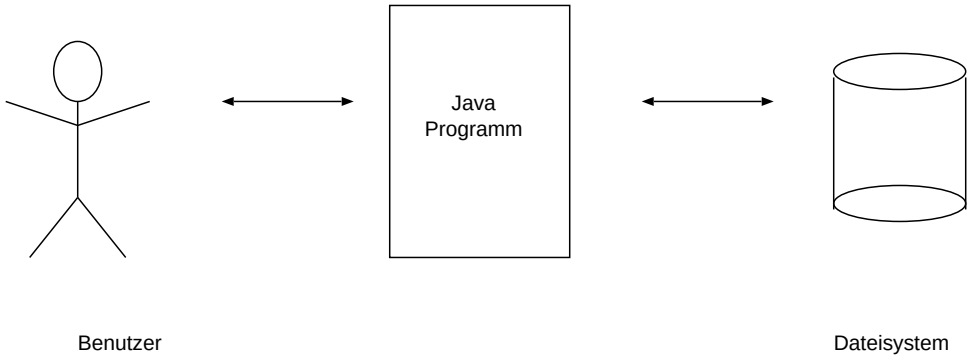
```
void erhoehe(int delta) {
  synchronized(this) {
    obj.x = obj.x + delta;
  }
}
```

Bei Klassenmethoden, die innerhalb der Deklaration der Klasse `K` stehen, wird entsprechend an Stelle von `this` das zu `K` gehörende Objekt der Klasse `java.lang.Class` verwendet.

8 Vordefinierte Klassen

In diesem Kapitel soll exemplarisch gezeigt werden, wie man vordefinierte Java-Klassen verwenden kann, um häufig auftretende Programmieraufgaben zu lösen.
Zunächst wird in Abschnitt 8.1 allgemein beschrieben, wie mit vordefinierten Klassen gearbeitet wird. Anschließend werden in den Abschnitten 8.2 und 8.3 zwei Anwendungsbereiche, bei denen oft mit vordefinierten Klassen gearbeitet wird, etwas näher betrachtet:

- Benutzerinterakton mit graphischen Benutzungsoberflächen.
- Ein- und Ausgabe in Dateien.



Warum wird bei diesen Anwendungsbereichen mit vordefinierten Klassen gearbeitet ?

Bei beiden Anwendungsbereichen kommuniziert das Java-Programm mit seiner Außenwelt. Aus technischer Sicht sind dazu oft recht komplexe Folgen von Betriebssystemaufrufen notwendig, bei denen leicht Programmierfehler entstehen können. Außerdem sind Betriebssystemaufrufe plattformabhängig.

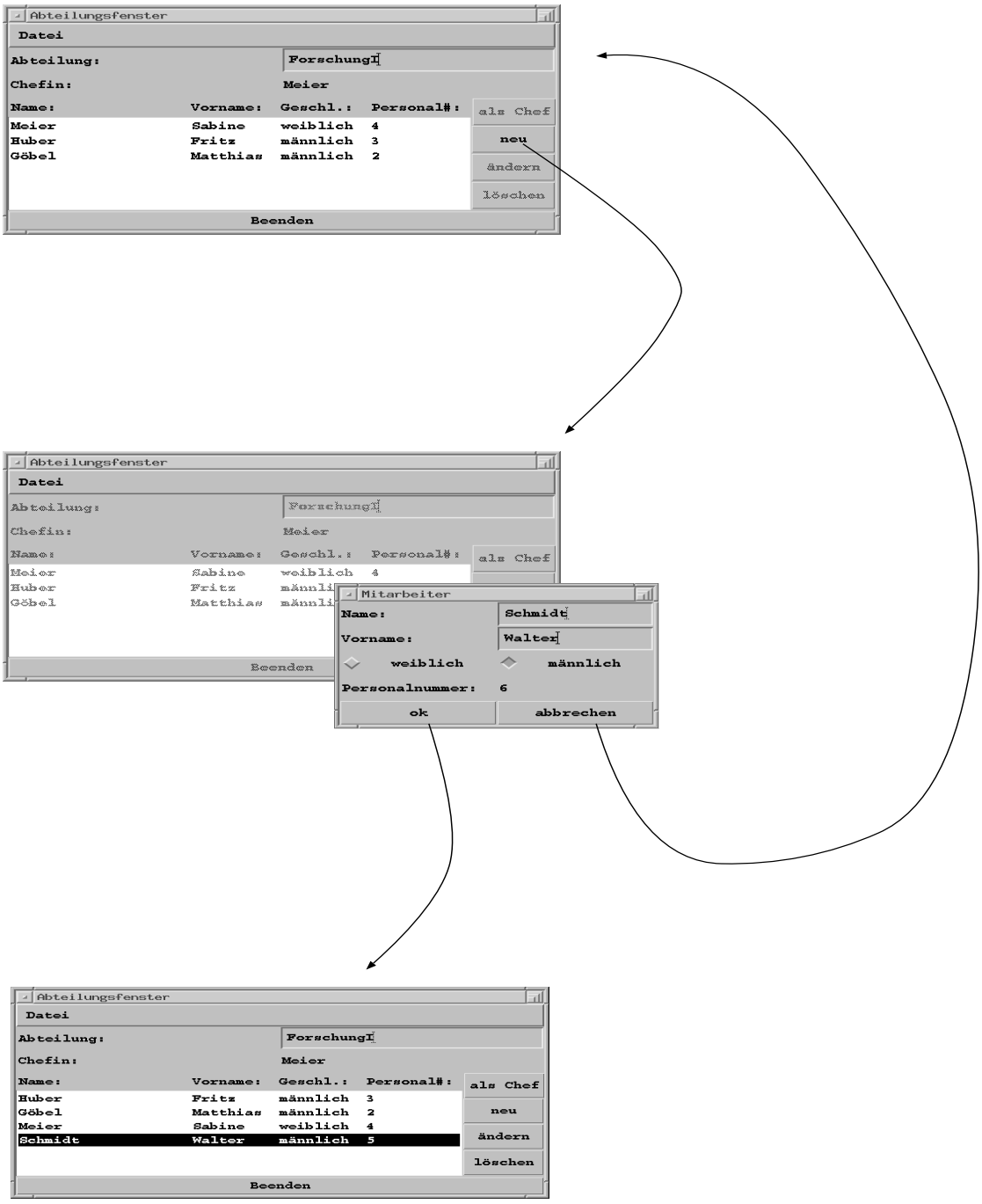
D.h. Programme, die direkte Betriebssystemaufrufe tätigen, können z.B. nicht auf PC's mit einem MS-Windows-Betriebssystem und UNIX-Maschinen ausgeführt werden, ohne daß die Betriebssystemaufrufe für die jeweils andere Plattform geeignet umformuliert werden. Daher werden in Java Betriebssystemaufrufe durch vordefinierte Klassen gekapselt. Dadurch wird dem Programmierer eine plattformunabhängige und in der Regel einfachere Möglichkeit für Kommunikationsaufgaben bereitgestellt.

Vorgehensweise in diesem Kapitel

Die in diesem Kapitel betrachteten Anwendungsbereiche sind in ihrer Gesamtheit sehr komplex. Es kann daher nicht annähernd eine vollständige Betrachtung vorgenommen werden. Anstelle dessen wird exemplarisch an einer komplexeren und einigen kleineren Aufgaben gezeigt, wie man mit vordefinierten Klassen arbeitet. Sie sollen dadurch einen Einblick in die beiden Anwendungsbereiche erhalten und die Methodik, mit vordefinierten Klassen zu arbeiten, erlernen. Sie werden jedoch damit kein vollständiges Wissen, z.B. über graphische Benutzungsoberflächen, erlangen.
Am Ende des Kapitels sollten Sie in der Lage sein, folgende Programmieraufgabe zu lösen.

Programmieraufgabe: Ein System zur Verwaltung von Abteilungen

Es soll ein System zur Verwaltung von Abteilungen erstellt werden. Dieses System erweitert die bereits eingeführte Klasse `Abteilung` um eine graphischen Benutzungsoberfläche und um die Möglichkeit, Objekte der Klasse `Abteilung` persistent abzuspeichern. D.h. auf die Objekte kann auch nach Beendigung des Programms noch zugegriffen werden. Über die Oberfläche sollen die Mitarbeiter der Abteilung angezeigt werden. Außerdem soll es möglich sein, Mitarbeiter neu aufzunehmen, Mitarbeiterdaten zu ändern, sowie Mitarbeiter aus der Mitarbeiterliste der Abteilung zu löschen. Der Chef einer Abteilung soll auch als Mitarbeiter der Abteilung geführt werden. Der Benutzer soll außerdem die Möglichkeit haben, den Chef der Abteilung durch einen anderen Mitarbeiter ersetzen zu können.
Die folgende Abbildung soll einen Eindruck über Teile der Funktionalität des zu entwickelnden Programms vermitteln. Die Pfeile sollen andeuten, was passiert, wenn der Benutzer auf einen der Knöpfe (Buttons) drückt.



Bemerkung: Wenn im folgenden von Benutzer und Programmierer die Rede ist, so ist damit selbstverständlich auch die entsprechende weibliche Form gemeint.

8.1 Allgemeines

In Java werden vordefinierte Klassen für einen Anwendungsbereich in einem oder mehreren Paketen verwaltet.

Beispiel: Vordefinierte Klassen zum Programmieren von Ein- und Ausgabe in Dateien befinden sich im Paket `java.io`.

Dem Programmierer stehen zwei Basistechniken zur Verfügung, um mit vordefinierten Klassen zu arbeiten:

- direktes Instanziieren von vordefinierten Klassen
- Ableiten und Überschreiben geeigneter Methoden

„Ableiten“ bedeutet: Entwickeln von neuen Unterklassen zu vorgegebenen Klassen (bzw. von Klassen, die vorgegebene Schnittstellen implementieren).

Zur Unterscheidung dieser beiden Fälle können folgende Überlegungen gemacht werden:
Vordefinierte Klassen können als Lösungsschablonen für Programmieraufgaben aufgefaßt werden. Diese Schablonen müssen zur Lösung einer konkreten Programmieraufgabe jedoch noch ergänzt bzw. angepaßt werden. Ist der dazu notwendige Programmcode eher umfangreich, so ist es sinnvoll, ihn in einer eigenen Klasse zu kapseln, die von der vordefinierten abgeleitet ist. Dies ist insbesondere dann sinnvoll, wenn die entsprechende Konkretisierung der vordefinierten Klassen öfter verwendet werden soll.
Besteht der Code für die Anpassung der Lösungsschablone jedoch nur aus wenigen Anweisungen, so reicht es, ein Objekt der vordefinierten Klasse zu instanziieren. Die Ergänzungen bzw. Anpassungen werden in diesem Fall dem Objekt entweder im Konstruktoraufruf oder mittels entsprechender Nachrichten mitgeteilt. Voraussetzung für diese Vorgehensweise ist allerdings eine entsprechende Anpassungsfähigkeit der vorgegebenen Klasse.
Beiden Techniken ist gemein, daß schließlich Objekte instanziiert werden, die zur Erledigung bestimmter Aufgaben benötigt werden. In den Beispielaufgaben kommen beide vorgestellten Techniken vor.

Beispiel: Wenn in einem Programm die Interaktion mit dem Benutzer notwendig ist, dann braucht man ein Objekt, das die Interaktion ermöglicht.

Aus Sicht des Programmierers erhält man durch ein solches Objekt die Möglichkeit, als Folge der Interaktion bestimmte Methoden aufzurufen, die zur Lösung der Programmieraufgabe notwendig sind.

Beispiel: Zum Abspeichern des Inhalts einer Variablen in einer Datei wird ein Objekt instanziiert, das entsprechende Methoden bereitstellt.

Es ist auch möglich, daß ein Objekt als ein Empfänger einer externen Nachricht dient. In diesem Fall wird durch das Objekt eine vom Benutzer angegebene Methode ausgeführt.

Beispiel: In einem email-Programm ist ein Objekt damit beschäftigt, die Datei, in der neu ankommende Post gespeichert wird, zu beobachten. Wurde auf die Datei zugegriffen, so wird durch das Objekt eine vom Programmierer angegebene Methode aufgerufen. In dieser Methode können z.B. Anweisungen zur Ausgabe eines Signaltons stehen.

Häufig ist es bei der Lösung von speziellen Programmieraufgaben notwendig, Objekte aus mehreren im Java *application programming interface (API)* vordefinierten Klassen zur Lösung einer Teilaufgabe kombiniert zu verwenden. Dabei ergeben sich für den Programmierer u.a. folgende Fragen:

1. Welche Objekte sind notwendig ?
2. In welcher Reihenfolge müssen Methoden ausgeführt werden ?

Zur Lösung dieser und ähnlicher Fragen ist es unumgänglich, daß der Programmierer eine möglichst vollständige und ausführliche Dokumentation über die entsprechenden vordefinierten Klassen zur Verfügung hat. Diese Dokumentation sollte im Hinblick auf die Verwendung vordefinierter Klassen einen schnellen Überblick und gleichzeitig eine detaillierte Beschreibung der Funktionalität aller Klassen bieten. Durch die Entwicklung von Hypertextsystemen ist es möglich, entsprechende Dokumentationen am Rechner online zur Verfügung zu stellen [10].

Bemerkung: An dieser Stelle soll nicht verschwiegen werden, daß die Verwendung vordefinierter Klassen Probleme aufwerfen kann. Das kann an komplexen, für den Programmierer nicht leicht zu verstehenden Techniken und Konzepten liegen, die die Grundlage für die verwendete vordefinierte Klasse darstellen. Als andere Ursachen kommen aber auch eine zu knappe Dokumentation oder die immer noch relativ häufig auftretenden Fehler (Bugs) in vordefinierten Klassen in Frage.
In solchen Fällen empfiehlt es sich, funktionierende Beispielprogramme, die die entsprechenden vordefinierten Klassen auch verwenden (siehe z.B. in der MeDoc-Java-Dokumentation), zu analysieren, um so die Nutzung dieser Klassen besser zu verstehen.

8.2 Graphische Benutzungsoberflächen mit `java.awt`

Die bisherigen dargestellten allgemeinen Überlegungen zum Arbeiten mit vordefinierten Klassen sollen nun exemplarisch konkretisiert werden.

[10] siehe z.B. `http://medoc.informatik.tu-muenchen.de/Java/java11/api/packages.html`.

Dazu wird zunächst die Kommunikation (Interaktion) zwischen Anwendungsprogrammm und Benutzer betrachtet. Sie erfolgt heutzutage meist über *graphische Benutzungsoberflächen* (engl.: *graphical user interfaces, GUI*).
Java stellt zur Programmierung von GUIs das *Abstract Windowing Toolkit* (*AWT*) zur Verfügung[11]. Das AWT ist eine Sammlung von vordefinierten Klassen, die eine große Zahl von für den Aufbau von GUIs benötigten Grundelementen beschreiben. Diese Klassen befinden sich in den Paketen `java.awt` und `java.awt.event`. Eine vollständige Dokumentation der AWT-Klassen ist wie schon angesprochen über das WWW abrufbar.

8.2.1 Grundlagen

Als Benutzer von Programmen mit GUI (z.B. `xemacs`,`netscape`) hat man bereits eine Vorstellung von einer graphischen Benutzungsoberfläche. Durch eine Benutzungsoberfläche kann der Benutzer:

- Daten ansehen (z.B. die Mitarbeiterliste).
- Daten eingeben (z.B. neue Mitarbeiter).
- Aktionen ausführen (z.B. das Programm beenden).

Für jede dieser Aufgaben stellt das AWT geeignete Klassen zur Verfügung.

Beispiele:

- Zur Darstellung einer Liste von Zeichenreihe gibt es die vordefinierte Klasse

  `java.awt.List`.

- Zur Eingabe und Veränderung einer einzelnen Zeichenreihen gibt es die vordefinierte Klasse

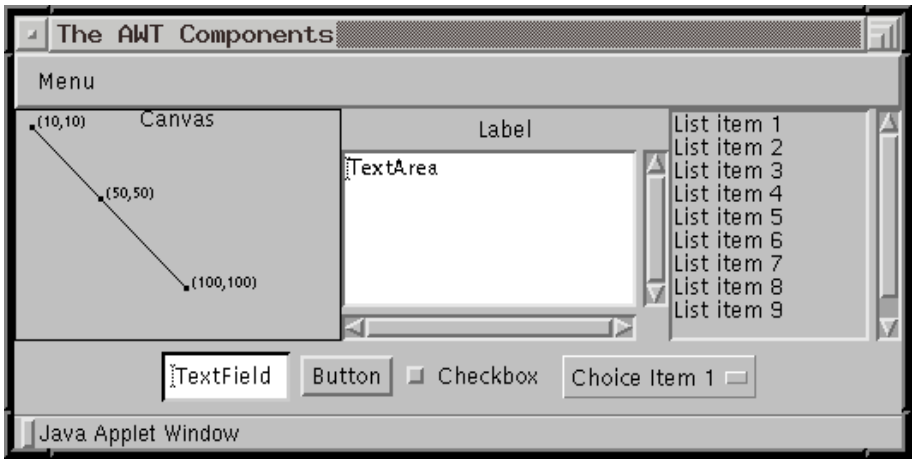
  `java.awt.TextField`.

- Zur Ausführung von Aktionen gibt es unter anderem die Klassen

  `java.awt.Button` und

  `java.awt.MenuItem`.

Objekte dieser Klassen werden allgemein *Interaktionsobjekte* genannt. Folgendes Bild (aus dem Java-Tutorial von Sun[12]) zeigt eine Reihe von Interaktionsobjekten, die das AWT zur Verfügung stellt.



Bibliotheken von vordefinierten Interaktionsobjekten werden auch *User Interface Toolkits* (*UI-Toolkits*) genannt. Ein bekanntes UI-Toolkit ist z.B. auch *OSF Motif*.
Objekt-orientierte UI-Toolkits werden aufgrund einer Vielzahl von Gemeinsamkeiten der verschiedenen Interaktionsobjekte (z.B. jedes Interaktionsobjekt hat eine Farbe) normalerweise in einer Klassenhierarchie organisiert. So sind beim AWT die meisten Interaktionsobjekte (bzw. die entsprechenden Klassen)

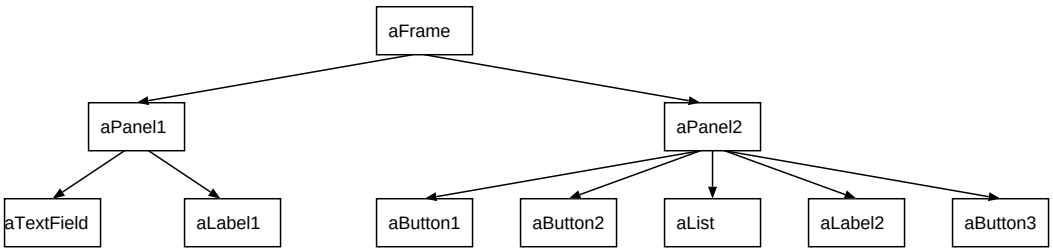
[11] Inzwischen existiert mit *Java Swing* eine Alternative zum AWT. Die Grundkonzepte der AWT-Programmierung lassen sich aber weitgehend auch auf Swing übertragen.

[12] `http://medoc.informatik.tu-muenchen.de/Java/tutorial/ui/overview/components.html`

von der Klasse `java.awt.Component` abgeleitet, in der ein Großteil der Methoden und Variablen definiert sind, die zur Nutzung der Interaktionsobjekte benötigt werden.
Die zu einem Programm gehörenden Interaktionsobjekte werden (deutlich am Bildschirm sichtbar) in einem oder mehreren *Fenstern* gekapselt.
Fenster können in der Regel am Bildschirm vom Benutzer verschoben und evtl. auch in ihrer Größe verändert werden. Für diesen Zweck gibt es in Java die vordefinierte Klasse `java.awt.Frame`. Auch Objekte der Klasse `Frame` werden als Interaktionsobjekte bezeichnet. Interaktionsobjekte, die sich aus Sicht des Benutzers innerhalb eines Frames befinden, werden aus Sicht des Programmierers von einem Objekt vom Typ `Frame` in einer Liste verwaltet und als *Kind-Interaktionsobjekte* bezeichnet. Mittels der Methoden `add(...)` und `remove(...)` können einem Frame weitere Interaktionsobjekte (z.B. Buttons) hinzugefügt bzw. weggenommen werden.

Wie werden Interaktionsobjekte in einem Frame angeordnet ?

Eine komplexe Aufgabe für den Programmierer kann die graphische Anordnung von Interaktionsobjekten darstellen. Auch dabei hilft das AWT mit vordefinierten Klassen. Diese Klassen werden *Layout-Manager* genannt und implementieren die Schnittstelle `java.awt.LayoutManager`. Beispiele für Layout-Manager sind die Klassen `BorderLayout` und `GridLayout`.
Einem Objekt vom Typ `Frame` kann ein Objekt vom Typ `LayoutManager` durch die Methode `setLayout(...)` zugeordnet werden. Nachdem ein Layout-Manager zugeordnet wurde, übernimmt er die Anordnung der Kind-Interaktionsobjekte. Bei einigen Layout-Managern hat der Programmierer die Möglichkeit, die Anordnung der Objekte über Attribute zu steuern. Diese Attribute sind als Konstanten in den verschiedenen Layout-Manager-Klassen definiert.
Beispielsweise bietet ein Layout-Manager vom Typ `BorderLayout` Attribute mit den Namen der Himmelsrichtungen an, so daß z.B. der Aufruf `add(abutton,BorderLayout.WEST);` den Button `abutton` links am Bildschirm anordnet. Demgegenüber ordnet ein Layout-Manager vom Typ `GridLayout` die Kind-Interaktionsobjekte in Form einer Matrix an. Der Programmierer kann die Anzahl der Zeilen und der Spalten beispielsweise beim Konstruktoraufruf von `GridLayout` angegeben. Die Kind-Interaktionsobjekte füllen die Matrix von links nach rechts und von oben nach unten. Anders als bei `BorderLayout` ist es hier nicht notwendig, bei jeder `add(...)`-Anweisung ein Anordnungsattribut zu übergeben.
Bei einem etwas komplexeren Fenster kann es sinnvoll sein, die Anordnung der Interaktionsobjekte auf mehrere Bereiche aufzuteilen. Hierzu wird die vordefinierte Klasse `java.awt.Panel` verwendet. Panel kann genauso wie Frame eine Liste von Kind-Interaktionsobjekten verwalten und wird seinerseits von einem übergeordneten Interaktionsobjekt verwaltet. Dieses stellt üblicherweise ein Objekt der Klasse `java.awt.Frame` dar. Es kann sich aber auch wiederum um ein Panel handeln. Ein Objekt vom Typ `java.awt.Frame` kann damit an der Wurzel einer hierachischen Struktur von Interaktionsobjekten stehen, wie die folgende Abbildung zeigt.



Wie werden Aktionen ausgeführt ?

Wie schon erwähnt, kann der Benutzer über Interaktionsobjekte (z.B. Objekte der vordefinierten Klasse `java.awt.Button`) Aktionen ausführen. Dazu muß der Programmierer dem Interaktionsobjekt mitteilen, welche Aktion z.B. bei der Betätigung eines Buttons auszuführen ist.

Das Drücken eines Buttons, aber auch andere Benutzeraktionen wie z.B. ein Tastendruck, stellen aus Sicht des Programms *Ereignisse* (engl.: *events*) dar, auf die das Programm reagiert.
Viele Interaktionsobjekte sind in der Lage, auf verschiedene Ereignisse auch unterschiedlich zu reagieren. Dazu müssen die eingetretenen Ereignisse voneinander unterscheidbar sein. In Java wird daher nach jeder Benutzeraktion ein Objekt erzeugt, das eine genaue Beschreibung der Benutzeraktion bzw. des sich daraus ergebenden Ereignisses liefert. Bei der Betätigung eines Buttons z.B. wird ein Objekt des Typs `java.awt.event.ActionEvent` erzeugt.

Was ist eine Aktion aus Sicht des Programmierers ?

Eine Aktion ist eine Instanzmethode, die in einer beliebigen Klasse definiert werden kann, wobei Name und Argument für die verschiedenen Ereignisarten vorgegeben sind. Die für die Auswertung von Ereignissen des Typs `ActionEvent` (siehe oben) vorgesehene Methodensignatur ist z.B. folgendermaßen festgelegt:

```
public void actionPerformed(ActionEvent e)
```

Die Informationen über das eingetretene Ereignis, die in dem Argument `e` übergeben werden, können zur Steuerung der Programmaktion genutzt werden. Beispielsweise wird in `e` eine Referenz auf das Interaktionsobjekt abgelegt, über das die Aktionsausführung ausgelöst wurde. Diese Referenz kann z.B. nützlich sein, wenn eine Aktion an mehrere Interaktionsobjekte gekoppelt ist und innerhalb der Aktion eine Fallunterscheidung nach dem auslösenden Interaktionsobjekt notwendig ist.

Damit ein Interaktionsobjekt eine Methode aufrufen kann, muß man ihm zuerst mitteilen, welche Methode verwendet werden soll. Wenn es sich bei dem Interaktionsobjekt beispielsweise um einen Button handelt, so geschieht diese Mitteilung durch folgende Methode der Klasse `java.awt.Button`:

```
public synchronized void addActionListener(ActionListener l)
```

Das übergebene Objekt ist dabei eine Instanz einer Klasse, in der die Methode `actionPerformed` definiert wurde. Um diese Eigenschaft sicherzustellen, muß die Klasse des Argument-Objekts die Schnittstelle `ActionListener` implementieren. Die Schnittstelle `ActionListener` beinhaltet nämlich genau die Methode `actionPerformed`.
Durch mehrfaches Aufrufen der Methode `addActionListener()` mit verschiedenen `ActionListener`-Objekten als Argumenten lassen sich einem Button auch mehrere Objekte zuordnen, die auf das Drücken des Buttons dann individuell reagieren können.

8.2.2 Beispielprogramme

Das folgende Java-Programm ist eines der einfachsten Programme mit GUI. Es besteht lediglich aus einem Button, der das Programm beendet.

Beispiel 56:

```
import java.awt.*;
import java.awt.event.*;

// Minimal ist ein spezieller Frame und wird daher von Frame abgeleitet.
// Minimal implementiert die Schnittstelle ActionListener, um sicherzustellen,
// dass es ActionEvents ueber die Methode actionPerformed(...) empfangen kann.
public class Minimal extends Frame implements ActionListener  {
  public Minimal() {

    // Interaktionsobjekte werden hier anonym (lokale Variablen) verwaltet
    Button abutton = new Button("Beenden");

    // Beim Betaetigen des Buttons soll "diese" Klasse benachrichtigt werden
    // ==> die Methode "actionPerformed()" des Minimal-Objekts wird aufgerufen
    abutton.addActionListener(this);
    add(abutton);
    pack();           // ordnet die Kind-Interaktionsobjekte
                      // entsprechend der Anweisungen an den Layout-Manager an
    setVisible(true); // zeigt das Fenster am Bildschirm
  }

  // wird ueber die Ereignisquelle "abutton" aufgerufen
  public void actionPerformed(ActionEvent e) {
     System.exit(0);
  }

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

```
    Minimal m = new Minimal();
  }
}
```

Das nächste Java-Programm zeigt dem Benutzer eine Liste von Namen von Mitarbeitern an. Durch Betätigung des Buttons wird das Programm beendet.

Beispiel 57:

```
import java.awt.*;
import java.awt.event.*;

public class Mitarbeiterfenster extends Frame implements ActionListener  {
  Button buttonende;
  List listmitarbeiter;

  public Mitarbeiterfenster() {
    // Erzeugen und Einsetzen eines Layout-Managers
    setLayout(new BorderLayout());

    // neue Liste mit 3 gleichzeitig sichtbaren Eintraegen,
    // ohne Mehrfachselektion ==> 2. Konstruktorargument ist "false"
    listmitarbeiter = new List(3,false);

    // Einfuegen der Listeneintraege
    listmitarbeiter.add("Huber");
    listmitarbeiter.add("Schmidt");
    listmitarbeiter.add("Obermeier");
    listmitarbeiter.add("Adam");

    // Einfuegen von listmitarbeiter in der Mitte des Frames
    add(listmitarbeiter,BorderLayout.NORTH);

    // neuer Button
    buttonende = new Button("Beenden");
    buttonende.addActionListener(this);
    // der Button soll im Frame unten angeordnet werden ==> SOUTH
    add(buttonende,BorderLayout.SOUTH);

    pack();
    setVisible(true);
    // "setTitle()" funktioniert u.U. nicht auf HP-Rechnern !
    setTitle("MitarbeiterFenster");
  }

  public void actionPerformed(ActionEvent e) {
    System.exit(0);
  }

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

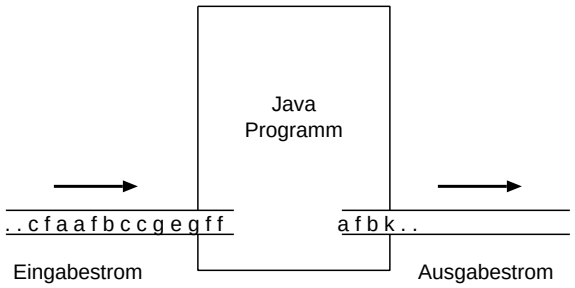
Die Ausgabe für Beispiel 57 am Bildschirm sieht etwa wie folgt aus:

Am Kapitelanfang wurden die beiden grundlegenden Programmiertechniken erwähnt, mit vordefinierten Klassen zu arbeiten. Man kann Objekte vordefinierter Klassen direkt instanziieren oder von ihnen zunächst eine eigene Klasse ableiten und davon ein Objekt instanziieren. Im Beispiel wird von `Frame` eine eigene Klasse `Mitarbeiterfenster` abgeleitet, während von `Button` direkt ein Objekt instanziiert wurde.

Die letzten Anweisungen im Konstruktor von `Mitarbeiterfenster` wurden vorher noch nicht erwähnt. Durch die Anweisung `pack()` wird die Anordnung der Kind-Interaktionsobjekte gemäß den vorher gemachten Anweisungen an den Layout-Manager berechnet. In dieser Berechnung ermittelt der Layout-Manager die Koordinaten der Interaktionsobjekte. Der Programmierer muß sich darüber jedoch keine Gedanken machen. Durch die Anweisung `setVisible(true)` wird das Fenster schließlich am Bildschirm sichtbar geschaltet. Wird diese Anweisung nicht gemacht, so existiert zwar ein Objekt vom Typ `Mitarbeiterfenster`, aber der Benutzer bekommt davon nichts zu sehen.

8.3 Ein- und Ausgabe in Dateien mit `java.io`

Java verwendet für die Ein- und Ausgabe von Daten das Konzept der *Ströme* (engl. *streams*). Ströme werden in Eingabe - und Ausgabeströme unterteilt. Entsprechend gibt es im Paket `java.io` die Klassen `InputStream` und `OutputStream`. Durch einen Eingabestrom kann ein Java-Programm externe Daten lesen. Durch einen Ausgabestrom können Daten nach außen übermittelt werden. Im Java-Programm wird ein Eingabestrom durch ein Objekt der Klasse `InputStream` und ein Ausgabestrom durch ein Objekt der Klasse `OutputStream` implementiert.



Neben der Unterscheidung in Ein- und Ausgabeströme werden Ströme entsprechend der Datentypen, die sie transportieren können, unterschieden. Während durch Objekte der Klassen `InputStream` und `OutputStream` Bytes (8 Bit) transportiert werden, können durch Objekte der Klassen `Reader` und `Writer` Unicode-Zeichen (16 Bit) transportiert werden. Durch `Reader` wird ein Unicode-Zeichen-Eingabestrom und durch `Writer` ein Unicode-Zeichen-Ausgabestrom implementiert. Die Klasse `InputStreamReader` ermöglicht es, Byte-Eingabeströme in Unicode-Eingabeströme umzuwandeln. Entsprechend gibt es die Klasse `OutputStreamWriter` zur Umwandlung von Byte-Ausgabeströmen in Unicode-Ausgabeströme.
Die meisten Klassen von `java.io` sind von einer der bisher betrachteten Klassen abgeleitet. Sie stellen auf bestimmte Anwendungsfälle spezialisierte Klassen von Strömen dar. So gibt es etwa die Klasse `FileReader` (abgeleitet von `Reader`) zum Lesen des Inhalts einer Datei und die Klasse `FileWriter` (abgeleitet von `Writer`) zum Schreiben einer Datei.
Die Verwendung der Stromklassen zur Ausgabe und Eingabe von Daten soll nun exemplarisch erklärt werden.

8.3.1 Ausgabe von Zeichenreihen in Dateien

Wir stehen vor der Aufgabe, einen Text, der in Form eines Objekts vom Typ `String` vorliegt, in einer Datei abzuspeichern. Da durch `String` Unicode-Zeichen verwaltet werden, ist es sinnvoll, einen Unicode-Zeichen-Ausgabestrom dafür zu verwenden. Da außerdem das Ziel der Ausgabe eine Datei ist, ist für diese Aufgabe die Klasse `FileWriter` geeignet.
Zunächst ist ein Objekt der Klasse `FileWriter` zu instanziieren. Der Konstruktor von `FileWriter` erwartet als Argument einen String mit dem Namen der Datei.
Beim Konstruktoraufruf kann es jedoch zu Problemen kommen, beispielsweise, wenn das Verzeichnis, in dem die Datei liegt, nicht existiert, oder wenn keine Schreibberechtigung vorliegt. In diesen und ähnlichen Fällen werden Ausnahmen generiert, die der Programmierer behandeln sollte. Die Sprachmittel und Techniken hierfür wurden bereits in Abschnitt 6 vorgeführt.
Wenn ein Objekt des Typs `FileWriter` vorliegt, so kann dessen Instanzmethode `write(String str)` (geerbt von der indirekten Oberklasse `Writer`) zum Schreiben des Ausgabetexts verwendet werden. Nach diesem Aufruf ist die Zeichenreihe allerdings in der Regel noch nicht vollständig in die Datei geschrieben, sondern wird zumindest teilweise im Datenstrom gepuffert. Das Abschließen des Schreibvorgangs in die Datei geschieht entweder durch explizites „Leeren“ des Stroms durch die Anweisung `flush()` oder durch Schließen des Stroms mittels der Anweisung `close()`.

Das folgende Programm schreibt den String `hallo` in die Datei `test`. Im `catch`-Block hat man die Möglichkeit, über das Objekt `e` der Klasse `Exception` Informationen über die Ausnahme zu erlangen und diese beispielsweise auszugeben.

Beispiel 58:

```
import java.io.*;
public class dateischreiben {
  public static void main(String[] args) {
    try {
      String ausgabe = "hallo";
      FileWriter fw = new FileWriter("test");
      fw.write(ausgabe);
      fw.close();
    } catch (Exception e) {
      System.out.println("Fehler beim Erstellen der Datei: " + e);
    }
  }
}
```

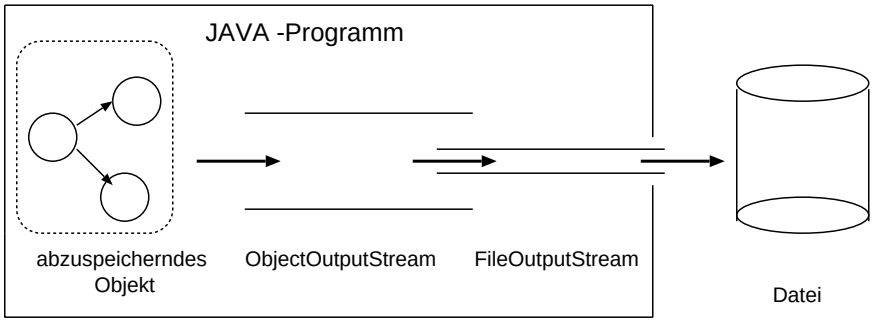
8.3.2 Einlesen von Zeichenreihen von Dateien

Die Umkehrung der voherigen Aufgabe stellt das Einlesen von einer Zeichenreihe aus einer Datei in eine `String`-Variable dar. Dazu kann die Klasse `FileReader` verwendet werden. `FileReader` erbt von `Reader` die Methode `int read()`, durch die einzelne Zeichen als `int`-Werte vom Eingabestrom gelesen werden können. Es liegt also nahe, in einer Schleife die Methode `read()` solange aufzurufen, bis das Ende der Datei erreicht ist. Im Schleifenblock wird das `String`-Objekt, das den Dateiinhalt aufnehmen soll, um jedes einzelne Zeichen erweitert. Da `read()` einen `int`-Wert liefert, ist ein Cast auf `char` notwendig. Beim Dateiende liefert `read()` den Wert `-1`. Das Ausprogrammieren dieser Aufgabe bleibt dem Kursteilnehmer überlassen.

8.3.3 Abspeichern von Objekten in Dateien

Das Paket `java.io` unterstützt nicht nur das Abspeichern von Zeichenreihen, sondern bietet auch die Möglichkeit an, beliebig komplexe Objekte in Dateien abzuspeichern. Dies könnte beispielsweise von Nutzen sein, um Objekte vom Typ `Abteilung` dauerhaft anzulegen. In Zusammenhang mit der zugehörigen graphischen Benutzungsoberfläche kann somit schnell ein durchaus nützliches System zur Verwaltung von Abteilungen entstehen.
Die Klasse, die uns bei dieser Aufgabe hilfreich ist, heißt `ObjectOutputStream`. Wie der Name schon verrät, ist sie von `OutputStream` abgeleitet. Die Möglichkeiten von `ObjectOutputStream` sind nicht nur auf das Abspeichern von Objekten in Dateien beschränkt. So ist es mit dieser Klasse z.B. auch möglich, Objekte in Strömen im Internet zu versenden. Daher hat `ObjectOutputStream` auch keinen

Konstruktor, der als Argument einen Dateinamen erwartet. Stattdessen erwartet der Konstruktor von `ObjectOutputStream` bereits ein Objekt vom Typ `OutputStream`. Dabei kann es sich beispielsweise um ein Objekt vom Typ `FileOutputStream`, einer Unterklasse von `OutputStream`, handeln. `FileOutputStream` wiederum ist auf die Ausgabe in Dateien spezialisiert und erwartet demzufolge im Konstruktor einen String mit dem Dateinamen als Argument.



Um ein Objekt in einen Ausgabestrom zu schreiben, ist es notwendig, es in einen Zeichenstrom umzuwandeln. Man spricht dabei von der *Serialisierung* des Objekts. Der entsprechende Algorithmus ist nicht trivial, da Objekte beliebig komplexe und sogar rekursive Strukturen aufweisen können. Darüberhinaus muß die Serialisierung auch umkehrbar sein, weil man in der Regel das abgespeicherte Objekt später wieder verwenden will.
In `ObjectOutputStream` ist die Objekt-Serialisierung bereits implementiert. Damit ist es möglich, Objekte ohne Kenntnis der durch ihre Klassenzugehörigkeit vorgegebenen Struktur zu serialisieren. Wie das funktioniert, wird hier nicht weiter dargestellt.

Aus bestimmten Gründen (z.B. Sicherheit) ist es sinnvoll, daß nicht alle Arten von Objekten serialisiert werden können. Im Normalfall sind Objekte nicht serialisierbar. Um Objekte serialisieren zu können, müssen Klassen, deren Objekte serialisierbar sein sollen, an ihrer Definitionsstelle explizit markiert werden. Für solche Zwecke benutzt man in Java das Schnittstellenkonzept (siehe Kapitel 4).
Klassen mit serialisierbaren Objekten müssen die Schnittstelle `java.io.Serializable` implementieren. Da `Serializable` nur zur Markierung dient, deklariert diese Schnittstelle keine Methoden. Es ist also nicht notwendig, zur Serialisierung weitere Methoden für die zu serialisierenden Objekte zu implementieren.
Man muß jedoch darauf achten, daß die Klassen der Instanzvariablen, die ebenfalls zu serialisieren sind, ihrerseits `Serializable` implementieren müssen.

Beispiel: Zur Serialisierung der Klasse `Abteilung` ist es notwendig, neben `Abteilung` auch `Mitarbeiter` und `Table` als serialisierbar zu markieren, weil Instanzvariablen entsprechender Typen in `Abteilung` vorhanden sind. Grundtypen wie z.B. `String` (Typ von `Abteilung.name`) sind bereits von vornherein serialisierbar.

Es gibt noch eine Reihe weiterer Aspekte zur Serialisierung, auf die hier jedoch nicht weiter eingegangen wird.
Wenn nun ein Objekt vom Typ `ObjectOutputStream` instanziiert wurde und ein serialisierbares Objekt vorliegt, kann mittels der Instanzmethode `writeObject(Object obj)` das abzuspeichernde Objekt über einen Datenstrom an die Datei weitergegeben werden. Wiederum liegt das Objekt erst vollständig in der Datei vor, nachdem `flush()` oder `close()` aufgerufen wurde.

Der folgende Programmteil schreibt ein Objekt vom Typ `Abteilung` in die Datei `test`.

Beispiel 59:

```
Abteilung abteilung = new Abteilung( /* geeignet konstruiert */ );
try {
  ObjectOutputStream out =
    new ObjectOutputStream(new FileOutputStream("test"));
  out.writeObject(abteilung);
  out.close();
} catch (Exception e) {
  System.out.println(e);
```

```
}
```

8.3.4 Einlesen von Objekten aus Dateien

Zum Einlesen von Objekten gibt es in `java.io` die Klasse `ObjectInputStream`. Sie stellt das Gegenstück zu `ObjectOutputStream` dar. Ähnlich wie bei der Konstruktion von `ObjectOutputStream` wird auch hier ein Strom, der auf das Lesen von Dateien spezialisiert ist (`FileInputStream`) mit einem Strom, der zum Einlesen von Objekten geeignet ist (`ObjectInputStream`) verknüpft.
Anschließend kann mittels der für `ObjectInputStream` deklarierten Methode `readObject()` das Objekt eingelesen werden. Der Rückgabetyp von `readObject()` ist `Object`. Das ist sinnvoll, weil damit beliebige Objekte eingelesen werden können. Wenn an diese Stelle ein Objekt eines konkreten Typs erwartet wird, z.B. ein Objekt vom Typ `Abteilung`, so ist ein entsprechender Cast nötig.

Der folgende Programmteil liest ein Objekt vom Typ `Abteilung` aus der Datei `test`. Beim Einlesen des Objekts ergibt sich eine weitere Möglichkeit einer Ausnahme: Es kann sein, daß in `test` keine Zeichenreihe gespeichert ist, die als ein Objekt vom Typ `Abteilung` eingelesen werden kann.

Beispiel 60:

```
Abteilung abteilung;
try {
  ObjectInputStream in =
    new ObjectInputStream(new FileInputStream("test"));
  abteilung = (Abteilung) in.readObject();
  in.close();
} catch (Exception e) {
  System.out.println(e);
}
```

8.4 GUI-Einbindung des Lesens/Schreibens von Dateien

Im folgenden wird kurz angedeutet, wie die zuvor beschriebenen Methoden zur Speicherung von Daten in ein GUI-basiertes Programm integriert werden können. Die folgenden Hinweise beschreiben die einzelnen Sachverhalte nur in groben Zügen. Für Details wird an dieser Stelle auf die API-Beschreibung verwiesen.

Menüs

Aktionen wie das Lesen und Abspeichern von Daten werden dem Benutzer üblicherweise über Menüs angeboten. Man unterscheidet prinzipiell zwischen Pulldown-Menüs und Popup-Menüs. Wir werden im weiteren nur Pulldown-Menüs betrachten:
Pulldown-Menüs sind in der Regel in einem Balken, der sog. *Menüleiste*, am oberen Fensterrand angeordnet. Die Menüleiste wird in `java.awt` durch die Klasse `MenuBar` implementiert. Einem `Frame`-Objekt kann durch die Methode `setMenuBar()` ein Objekt vom Typ `MenuBar` zugeordnet werden.
Ein Objekt vom Typ `MenuBar` verwaltet eine Liste von Objekten vom Typ `Menu`. Analog zu den Objektlisten der Klasse `Frame` stehen auch für die Klasse `MenuBar` Methoden zum Hinzufügen und Wegnehmen von `Menu`-Objekten (`add(Menu)`, `remove(Menu)`) zur Verfügung. Im Konstruktor von `Menu` wird der Menüname als `String` übergeben. Der Benutzer muß dann den Menünamen anwählen (normalerweise durch einen Mausklick), um das entsprechende Menü zu öffnen.
Aktionen werden vom Benutzer durch die Auswahl eines Menüeintrags ausgelöst. Der Programmierer instanziiert dafür Objekte der Klasse `MenuItem` und ordnet sie einem Objekt der Klasse `Menu` zu. Der Aufbau von Menüs geschieht wiederum durch Methoden `add(MenuItem)`, `remove(MenuItem)`, die in der Klasse `Menu` deklariert sind. Die Zuordnung von Programmaktionen zu Menüeinträgen erfolgt analog zu Buttons.

Dateidialog

Beim Lesen und Schreiben von Dateien wird der Benutzer oftmals in einem separaten Dialog nach dem Namen einer Datei gefragt. Da dieser Anwendungsfall häufig auftritt, bietet `java.awt` dafür eine vorgefertigte Klasse an. Diese Klasse heißt `FileDialog` und stellt eine Spezialisierung der Klasse `Dialog` dar.

Der Konstruktor von `FileDialog` erwartet als Argument unter anderem eine Referenz auf ein Objekt vom Typ `Frame`. Das heißt, ein Dialog kann nur instanziiert werden, wenn ein Frame existiert. Nach der Instanziierung kann der Dialog durch die Methode `show()` sichtbar geschaltet werden.
Ein `FileDialog`-Objekt bietet dem Benutzer die Möglichkeit, eine Datei auszuwählen. Er kann dazu durch die Verzeichnisstruktur navigieren. Der Programmierer kann auf den ausgewählten Dateinamen über die in `FileDialog` deklarierte Methode `getFile()` zugreifen.

Arbeiten mit mehreren Fenstern

Viele Programme bieten dem Benutzer eine Oberfläche, die aus mehreren Fenstern (Frames) besteht. Allerdings sind meist nicht in jeder Situation alle Fenster eines Programms sichtbar, oder zu einem Zeitpunkt reagiert eines dieser Fenster nicht auf Benutzereingaben.
Für solche Zwecke gibt es für den Programmierer die Möglichkeit, mit der Methode `setVisible(boolean)` Frames sichtbar bzw. unsichtbar zu schalten und mit der Methode `setEnabled(boolean)` die Interaktion mit dem Frame zu ermöglichen oder zu verhindern.

8.5 Applets

Mit Java kann man sehr einfach Programme schreiben, die mit Hilfe von HTML-Browsern über das Internet ausführbar sind. Diese Tatsache hat sicherlich viel zum Erfolg von Java beigetragen. Die innerhalb von HTML-Dokumenten ausführbaren Programme heißen *Applets*. Dementsprechend wird die Klasse für ein solches Programm von der vordefinierten Klasse `java.applet.Applet` abgeleitet.
Die Programmierung von Applets unterscheidet sich in einigen Dingen von der Programmierung herkömmlicher Java-Programme (im folgenden auch *Java-Applikationen* genannt), die direkt von einer JVM interpretiert werden. Einer dieser Unterschiede besteht darin, daß aus Sicherheitsgründen nicht auf die gleiche Weise mit Systemressourcen wie etwa Dateien umgegangen werden kann wie bei Java-Applikationen. Wir werden auf diesen Aspekt jedoch hier nicht weiter eingehen. Ausserdem werden in diesem Abschnitt nicht die vielfältigen graphischen und multimedialen Möglichkeiten betrachtet, mit denen Java-Applets HTML-Seiten optisch bereichern können. Die dazu notwendigen Techniken, wie auch weitere Einzelheiten über Applets können z.B. den Online-Büchern des MeDoc-Systems entnommen werden. Wir beschränken uns hier darauf, exemplarisch einige Konzepte zu erklären, die zur Lösung einer konkreten Programmieraufgabe notwendig sind.

Aufgabe: Es soll ein Applet entwickelt werden, mit dem es möglich ist, über das Internet Daten über Abteilungen anzusehen. Dabei soll es nicht möglich sein, diese Daten zu ändern. Technisch soll das Applet auf persistent abgespeicherte Objekte der Klasse `Abteilung` zugreifen. Diese werden mit dem bisherigen System als Dateien in einem bestimmten Verzeichnis abgespeichert. Das Aussehen des Applets soll in etwa dem Aussehen der Benutzungsoberfläche des bisherigen Systems entsprechen. Es entfallen jedoch die Buttons und Menüs, weil keine Bearbeitung der Abteilungsobjekte notwendig ist. Zum Zugriff auf eine konkrete Abteilung soll der Benutzer in einem Textfeld den Abteilungsnamen der gewünschten Abteilung angeben und auf ENTER drücken. Anschließend soll der Inhalt der Abteilung über die gleichen Interaktionsobjekte wie beim bisherigen System angezeigt werden.

Folgende Fragen entstehen unter anderem bei der Lösung dieser Programmieraufgabe:

- Wie werden Applets erstellt ?
- Wie wird ein Applet in eine HTML-Datei eingebunden ?
- Wie kann man auf persistent abgespeicherte Abteilungen über das Internet zugreifen ?

8.5.1 Applets und HTML

Ein Applet wird programmiert, indem eine Unterklasse von `java.applet.Applet` erstellt wird. Die Instanziierung dieser Unterklasse wird jedoch anders als bei Java-Applikationen nicht vom Programmierer durchgeführt. Stattdessen erzeugt der HTML-Browser eine Instanz der Unterklasse und ruft anschließend zu dieser Instanz die Methode `init()` auf, die für alle Applets deklariert ist. Dies ist vergleichbar mit dem Aufruf der Methode `main` bei Java-Applikationen.
Der Aufruf eines Applets wird in einem HTML-Dokument durch eine spezielle Markierung (engl.: *tag*) eingeleitet. Folgendes HTML-Dokument ruft das Applet `Minimal` auf. Der Browser weist dem Applet eine Fläche mit der angegebenen Breite und Höhe zu.

Beispiel 61:

```
<html>
<head>
</head>
<body>
    <applet code=Minimal width=200 height=100></applet>
</body>
</html>
```

Zur Ausführung ist zu beachten, daß der HTML-Browser nach dem Laden der HTML-Seite auf die Datei mit dem Bytecode von `Minimal` zugreifen können muß. Das ist z.B. dann möglich, wenn sich `Minimal.class` im selben Verzeichnis wie das HTML-Dokument befindet.
Die Darstellung des HTML-Dokuments und die damit verbundene Ausführung des Applets erfolgt schließlich durch die Angabe der *URL* [13] (*Uniform Resource Locator*) des Dokuments im HTML-Browser.

Zum Testen von Applets ist der `appletviewer` (eine Komponente des JDK) gut geeignet. Falls der oben angegebene HTML-Text in einer Datei `Minimal.html` abgelegt wurde, könnte das Applet mit dem folgenden Aufruf des `appletviewer` ausgeführt werden:

```
appletviewer Minimal.html
```

Wichtig: Der `appletviewer` erwartet ein HTML-Dokument als Parameter. Es ist nicht möglich, hier wie bei der JVM `java` direkt eine Klassendatei anzugeben !

8.5.2 Zusammenspiel von HTML-Browser und Applet

Wenn eine Applet-Instanz von einem HTML-Browser ausgeführt wird, kann es unter anderem zu folgenden Ereignissen kommen:

- Der Benutzer öffnet das HTML-Dokument mit dem Appletaufruf.
- Der Benutzer scrollt das HTML-Dokument, so daß der sichtbare Bereich des Applets vom Bildschirm verschwindet.
- Der Benutzer scrollt das HTML-Dokument, so daß der sichtbare Bereich des Applets wieder am Bildschirm erscheint.
- Der Benutzer wechselt im Browser zu einer anderen Seite.

Bei der Programmierung von Applets kann es sein, daß man auf diese Ereignisse reagieren muß. Daher ruft der HTML-Browser bei jedem dieser Ereignisse jeweils eine bestimmte Methode von `java.applet.Applet` auf. Als Programmierer hat man damit die Möglichkeit, durch Überschreiben dieser Methoden auf das jeweilige Ereignis zu reagieren.
Im einzelnen handelt es sich dabei um folgende Methoden:

- `init()`: Wird (wie schon erwähnt) aufgerufen, nachdem der Browser die Applet-Instanz erzeugt hat. In dieser Methode sollten Initialisierungen aller Art stattfinden.
- `start()`: Wird vom Browser aufgerufen, wenn die Applet-Instanz sichtbar wird und daher aktiv sein sollte.
- `stop()`: Wird aufgerufen, wenn die Applet-Instanz vorübergehend unsichtbar ist und daher inaktiv sein sollte. Das kann z.B. bei Animationen sinnvoll sein, um unnötige Berechnungen zu vermeiden.
- `paint(...)`: Wird aufgerufen, wenn das Applet neu gezeichnet soll, z.B. weil es vorher von einem anderen Fenster verdeckt wurde.
- `destroy()`: Wird aufgerufen, bevor eine Applet-Instanz dauerhaft gestoppt wird. In dieser Methode sollten „Aufräumarbeiten“ stattfinden.

Da keine dieser Methoden eine abstrakte Methode darstellt, ist es nicht zwingend notwendig, sie bei der Erstellung eines Applets zu implementieren. Oft genügt es, nur die Methode `init()` zu überschreiben. Applets, die lediglich eine Graphik anzeigen, deklarieren häufig lediglich eine modifizierte Fassung der Methode `paint()`.

[13] Eine URL kann als Name für Ressourcen aufgefaßt werden, die im WWW zur Verfügung stehen.

8.5.3 Anordnung von Interaktionsobjekten

Aus Benutzersicht läuft ein Applet innerhalb eines rechteckigen, rahmenlosen Ausschnitts der Seite, die der HTML-Browser anzeigt. Daher ist die Klasse `java.applet.Applet` eine Subklasse der Klasse `java.awt.Panel` (s.o.). Interaktionsobjekte können somit auf die gleiche Art und Weise über Layout-Manager angeordnet werden wie bereits besprochen. Während jedoch bei Java-Applikationen Anweisungen zur Anordnung der Interaktionsobjekte meist innerhalb des Konstruktors stattfinden, müssen sie bei Applets in der Methode `init()` erfolgen.
Als Programmierer überschreibt man dazu die Methode `init()` und schreibt die Anweisungen zum Anordnen der Interaktionsobjekte in deren Rumpf.

Ein kleines, aber funktionsfähiges Applet sieht z.B. wie folgt aus:

Beispiel 62:

```
import java.applet.*;
import java.awt.*;

public class Minimal extends Applet {
  public void init() {
    add(new Label("Hallo Kursteilnehmer"));
  }
}
```

Die Aufruf `appletviewer Minimal.html` führt zu folgender Bildschirmausgabe:

8.5.4 Reaktion auf Ereignisse

Benutzerinteraktionen können genauso an Methoden eines Applets geknüpft werden, wie es im Zusammenhang mit Frames bereits gezeigt wurde. Dazu kann beispielsweise der Aufruf von Methoden der konkreten Unterklasse von `java.applet.Applet` durch das Betätigen von Buttons oder das Drücken der `ENTER`-Taste innerhalb eines Textfelds ausgelöst werden. In diesem Fall muß die konkrete Unterklasse der `Applet`-Klasse die entsprechenden Schnittstellen wie z.B. `ActionListener` implementieren.

Das folgende Applet implementiert einen Zähler. Beim Betätigen des Buttons wird der angezeigte Wert um eins erhöht.

Beispiel 63:

```
import java.awt.*;
import java.awt.event.*;

public class Counterapplet extends java.applet.Applet
  implements ActionListener {
  Button incr_button;
  byte value = 1;

  public void init(){
    incr_button = new Button( String.valueOf( value ) );
    incr_button.addActionListener( this );
    add( incr_button);
```

```
  }

  public void actionPerformed( ActionEvent e ){
    incr_button.setLabel( String.valueOf( ++value ) );
  }
}
```

8.5.5 Zugriff auf persistente Daten über das Internet

Zur Lösung der zuvor gestellten Aufgabe ist es notwendig, in Dateien persistent abgespeicherte Objekte vom Typ `Abteilung` über das Internet zu lesen.
Zum Zugriff auf persistente Daten über das Internet bietet das Package `java.net` eine Reihe nützlicher Klassen an. Darunter befindet sich die Klasse `java.net.URL`. Bei der Konstruktion eines Objekts der Klasse `java.net.URL` muß der String angegeben werden, der die URL repräsentiert.

Beispiel 64:

```
URL url1=new URL("http://www.in.tum.de")
URL url2=new URL("file:/<verzeichnis>/<datei>"))
```

Nach der Instanziierung eines `URL`-Objekts erhält man mit Hilfe der Methode `openStream()` eine Referenz auf einen `InputStream`, der einem `ObjectInputStream`-Objekt vorgeschaltet werden kann. Das Lesen von Objekten erfolgt dann genauso wie zuvor bereits beschrieben.
Zum weiteren Vorgehen bei der Lösung der Programmieraufgabe wäre es z.B. möglich, sämtliche Abteilungen persistent in einem Verzeichnis zu speichern, das über einen Web-Server vom Internet aus zugreifbar ist. Man könnte dann die Abteilungen in diesem Verzeichnis mit ihrem Abteilungsnamen als Dateinamen abspeichern. Wenn der Benutzer in das entsprechende Textfeld den Abteilungsnamen eingegeben hat, kann der URL-Bezeichner für das Verzeichnis mit dem eingegebenen Namen konkateniert und damit ein `URL`-Objekt erzeugt werden.

Es ist auch möglich, den URL-Bezeichner nicht absolut in das Applet zu programmieren, sondern relativ zur URL des Applets anzugeben. Beispielsweise könnte man ein Unterverzeichnis `abteilungen` im selben Verzeichnis anlegen, in dem sich auch das Applet befindet.
Man benötigt dazu eine Methode, die die URL des Verzeichnisses liefert, in der sich der Bytecode des Applets befindet. Die Klasse `java.applet.Applet` stellt dafür die Methode `getCodeBase()` zur Verfügung.

Zum Abschluß wird auszugsweise eine mögliche Implementierung des Zugriffs auf persistente Objekte vom Typ `Abteilung` über eine URL präsentiert:

Beispiel 65:

```
private void oeffne() {
  try {
    ObjectInputStream in = new ObjectInputStream(
      new URL(getCodeBase() + "/abteilungen/" + textfieldabteilung.getText()
                ).openStream());
    Abteilung abteilung = (Abteilung) in.readObject();
    setAbteilung(abteilung);
    in.close();
  } catch (Exception e) {
    System.out.println(e);
  }
}
```

Die Methode `oeffne()` wird innerhalb der Applet-Klasse deklariert, die die zu Beginn des Abschnitts gestellte Aufgabe lösen soll. Über `textfieldabteilung` (Objekt des Typs `java.awt.TextField`, Einbindung in Applet weggelassen) gibt der Benutzer den Namen der Abteilung an, deren Inhalt er anschauen will. Die Methode `setAbteilung()` (Deklaration ebenfalls weggelassen) besetzt die entsprechenden Interaktionsobjekte des Applets anhand des Inhalts des neu eingelesenen Objekts vom Typ `Abteilung`.

Index

Literatur

[Bis98] Judy Bishop: *Java Gently - Programming Principles Explained*
Second Edition, Addison Wesley, 1998

[Lem96] L. Lemay: *Java in 21 Tagen*
Markt&Technik, 1996

[CW98] M. Campione, K. Walrath: *The Java Tutorial - Object Oriented Programming for the Internet*
Addison-Wesley, 2nd Edition 1998

[Fla97] D. Flanagan: *Java in a Nutshell - A Desktop Quick Reference*
O'Reilly & Associates, 2nd Edition 1997

[Kr97] G. Krüger: *Java 1.1 lernen*
Addison-Wesley, 1997

[Kue96] R. Kühnel: *Die Java-Fibel*
Addison-Wesley, 1996

[GJS96] James Gosling, Bill Joy, Guy Steele: *The Java Language Specification*
Addison-Wesley, 1996