Java Kurs

Mainzer Straße 35
D-66111 Saarbrücken

Tel: +49-(0)-681 / 685 0100
Fax: +49-(0)-681 / 685 0101
kurt@huwig.de
http://www.huwig.de/

Erstellt mit LaTeX 2ε unter Linux

2. September 2000

Inhaltsverzeichnis

Kapitel 1
Einführung

Seitdem Java im Jahre 1995 vorgestellt wurde, erfreut es sich wachsender Beliebtheit und Verbreitung. Waren es am Anfang hauptsächlich Java-Applets auf Webseiten, so findet man heutzutage immer mehr Java-Applikationen, die unabhängig von einem Web-Browser laufen. Viele Entwickler schätzen mittlerweile die Eleganz und Plattformunabhängigkeit von Java, die sie von vielen Problemen bei der Programmentwicklung befreit.

Java wurde ursprünglich dazu entwickelt, Konsumentenprodukte, wie z.B. Videorekorder und Waschmaschinen, zu programmieren. Als Folge davon hat Java viele eingebaute Sicherheitsmechanismen, die Programme fehlertoleranter und weniger fehlerträchtig machen. Als Nachteil nimmt man etwas geringere Geschwindigkeit in Kauf. Bei den meisten Anwendungen, wie etwa einer Waschmaschine, kommt es weniger auf Geschwindigkeit, sondern auf Fehlerfreiheit an, denn wer will schon eine "nicht behebbare Schutzverletzung" im Schleuderprogramm?

Ganz nebenbei eignet sich die Sprache auch für "herkömmliche" Anwendungen, wie z.B. einer Textverarbeitung. Hier freut man sich über stabile Programme - Unterroutinen können abstürzen, ohne dass die Anwendung abstürzt - Internetfähigkeit und die Unabhängigkeit vom Betriebssystem und Prozessor.

1.1 Was ist Java?

In einem frühen Dokument über die Sprache beschrieb Sun ihr Java als

- einfache
- objektorientierte
- verteilte
- interpretierte
- robuste
- sichere
- architekturneutrale
- portable
- hoch-performante
- multithreaded
- dynamische

Sprache. Dies sind einige Schlagwörter, die das ganze recht gut beschreiben. Im Einzelnen:

einfach

Java ist sehr einfach zu lernen, da es sich in den Sprachkonstrukten an C und C++ anlehnt, wobei einige der komplizierteren Eigenschaften von C++ weggelassen wurden.

objektorientiert

Moderne Programmiersprachen sind nicht mehr prozedural, sondern objektorientiert aufgebaut. Eine genaue Erklärung dieses Begriffes finden sie in Kapitel 7.

verteilt

Java ist netzwerkfähig; es spielt keine Rolle, ob sie Daten von der Festplatte oder aus dem Internet laden, die Programmierung ist die gleiche.

interpretiert

Java wird in einer *vituellen Maschine* (JVM) ausgeführt. Dies bedeutet, dass Java-Programme nicht in den Binärcode eines bestimmten Prozessors, z.B. eines *Pentiums* übersetzt wird, sondern in den der JVM. Für viele Betriebssysteme und Prozessoren gibt es Implementationen der JVM, so dass Java-Programme unverändert darauf laufen können.

robust

Java verfügt über Sprachelemente zur Ausnahmenbehandlung, z.B. ein Fehler beim Lesen einer Datei. Bei manchen Systemaufrufen wird man von der Sprache dazu gezwungen den Fehler abzufangen, so dass man sich als Programmierer darüber Gedanken machen muss.

sicher

Bei Netzwerkzugriffen sind Sicherheitsmechanismen wichtig. Im Gegensatz zu anderen Internetsprachen, wie z.B. *ActiveX*, ist es bei Java nicht möglich zerstörenden Code auf eine Arbeitsstation zu schleusen und Zugriffe auf das System können sehr genau kontrolliert werden. Durch digitale Unterschriften können sie die Sicherheit noch weiter erhöhen.

architekturneutral

Java läuft auf jedem Gerät, für das es eine JVM gibt. Dies fängt bei 8-Bit Mikroprozessoren an, geht über PDAs wie den *PalmPilot* über 32-Bit Intel-Prozessoren, 64-Bit Alpha-Prozessoren bis zu HighEnd-Unix-Servern. Sie können also ihre Hardware an ihre Leistungsanforderungen anpassen und müssen nicht ihre Programme an die Hardware anpassen.

portabel

Java-Programme laufen unverändert auf jeder kompatiblen JVM. Sie können z.B. ein Programm unter Linux entwickeln und ihre Kunden benutzen es unter Windows.

hoch-performant

Durch die Verwendung von Just-In-Time-Kompilern erreicht Java die Geschwindigkeit von C++ und ist damit auch für rechenintensive Anwendungen geeignet.

multithreaded

Multithreaded ist “multi-tasking innerhalb einer Anwendung”. Ein Beispiel hierfür ist ein Download mit *Netscape Navigator*. Während sie eine Datei herunterladen, können sie weiter im Internet surfen. Mit Java sind sie in der Lage, solche Anwendungen leicht und sicher zu programmieren.

dynamisch

Netzwerkverbindungen sind immer zu langsam! Java Anwendungen werden nicht zu Beginn komplett in den Speicher geladen, sondern einzelne Teile können zur Laufzeit bei Bedarf geladen werden. Sie können sogar einzelne Teile im Betrieb austauschen um z.B. von Deutsch auf Englisch unzuschalten.

Kapitel 2
Installation

In diesem Kapitel lernen sie, wie sie

- das Java-Development-Kit (JDK) und den Jikes installieren
- ein Java-Programm kompilieren und ausführen

2.1 Installation des JDK

Das Java-Development-Kit von Sun beinhaltet alles Notwendige, um Java-Applikationen und -Applets zu schreiben. Es ist kostenlos erhältlich und darf auch kommerziell genutzt werden. Zur Installation unter Windows gehen sie wie folgt vor:

1. Installationsprogramm starten
   Die minimale Installation umfasst
   - Program Files (Programmdateien)
   - Library and Header Files (Bibliotheken)

> Als Installationsverzeichnis geben Sie bitte `C:\JDK1.3` an, sonst stimmen die Pfadangaben in diesem Kapitel nicht immer!

2. JDK in den Pfad aufnehmen
   Starten sie "Sysedit" mit "Start - Ausführen" und gehen sie dann in die Datei `C:\AUTOEXEC.BAT`. Tragen sie dort folgendes ein:

```
SET CLASSPATH=C:\JAVA
SET JIKESPATH=%CLASSPATH%;C:\JDK1.3\JRE\LIB\rt.jar
DOSKEY /INSERT
```

   Der erste Befehl wird für das Finden der Java-Klassen, der zweite für *Jikes* benötigt und der dritte erleichtert die Arbeit mit einer DOS-Eingabeaufforderung.
   Wenn sie das *JDK 1.1* verwenden wollen, so muss der
   `JIKESPATH` folgendermaßen lauten:

```
SET JIKESPATH=%CLASSPATH%;C:\JDK1.1\LIB\CLASSES.ZIP
```

3. JIKES installieren
   Starten sie das Jikes-Installationsprogramm und geben sie dort als Zielverzeichnis `C:\JDK1.3\BIN` an. `jikes` ist ein etwa 30x schnellerer Ersatz für den Javakompiler `javac` von Sun.
4. Erstellen sie das Verzeichnis `C:\JAVA`
5. Starten sie Windows neu

Wenn sie Java-Programme nur verwenden aber nicht schreiben wollen, brauchen sie nicht das JDK zu installieren, sondern er reicht das Java-Runtime-Environment (JRE), das sie mit ihren Programmen zusammen kostenlos ausliefern dürfen. Das JRE ist deutlich einfacher zu installieren als das JDK und wird im Abschnitt 13.1 ausführlich beschrieben.

2.2 Hello Web!

Sie werden jetzt ihr erstes Java-Programm schreiben, kompilieren und ausführen:

1. Öffnen sie den Editor und geben sie folgendes Programm ein. Achten sie dabei auf die Groß-/Kleinschreibung!

```
public class HelloWeb {
    public static void main( String[] argv ) {
        System.out.println( "Hello Web!" );
    }
}
```

2. Speichern sie die Datei im Verzeichnis `C:\JAVA` unter dem Namen `HelloWeb.java`. Achten sie auch hierbei auf die Schreibweise! Wenn sie 'Notepad' verwenden, geben sie als Dateityp "Alle Dateien" an!
3. Starten sie eine DOS-Eingabeaufforderung und wechseln sie in das Verzeichnis `C:\JAVA`
4. Kompilieren sie ihr Programm

   ```
   C:\JAVA> jikes HelloWeb.java
   ```

   Auch hier auf die Schreibweise achten! Alternativ können sie statt `jikes` den `javac` benutzen um den Geschwindigkeitsunterschied zu sehen.
5. Starten sie ihr Programm

   ```
   C:\JAVA> java HelloWeb
   ```

Sie erhalten die Ausgabe:

```
Hello Web!
```

2.3 Was ist passiert?

Jikes ist ein *Kompiler*, der Textdateien mit Java-Quellcode in *Bytecode* für die JVM umwandelt. Hierbei wird der Quellcode auf syntaktische und semantische Fehler überprüft und ggf. eine Fehlermeldung ausgegeben.

Auf der Festplatte findet sich nach dem Kompilieren eine Datei mit Endung `.class`, die den Bytecode enthält und der von der JVM ausgeführt werden kann. Hierzu muss man dem Programm `java` den Namen der Klasse angeben, die gestartet werden soll.

Sie dürfen nicht `java HelloWeb.class` eingeben, da sonst die JVM nach einer Klasse mit Namen 'HelloWeb.class' sucht, die Klasse aber 'HelloWeb' heißt.

Kapitel 3
Erste Schritte

In diesem Kapitel lernen sie

- Details über das Beispiel des letzten Kapitels
- die Datentypen von Java kennen
- wie sie Variablen definieren und benutzen
- Kontrollstrukturen kennen

3.1 Ein Java-Programm analysiert

Schauen wir uns noch einmal das Beispiel aus dem letzten Kapitel an.

```
1  public class HelloWeb {
2      public static void main( String[] argv ) {
3          System.out.println( "Hello Web!" );
4      }
5  }
```

Die erste Zeile gibt an, dass sich hier die öffentliche Klasse “HelloWeb” befindet. Klassen werden im Kapitel 7 genauer erläutert; jetzt müssen sie nur wissen, dass der Dateiname dem Namen der Klasse entsprechen muss und jede Klasse muss in einer eigenen Datei stehen. Auf diese Weise kann der Kompiler notwendige Klassen finden und ggf. kompilieren.

Die öffnende geschweifte Klammer gibt den Anfang der Klassendefinition an; die schliessende in Zeile 5 das Ende. Ein Stück Programmcode zwischen zwei geschweiften Klammern heißt Block; Blöcke können geschachtelt werden, wie man im Beispiel sieht. Es ist üblich den Code innerhalb eines Blockes einzurücken, damit man leicht die Länge des Blockes erkennen kann.

In Zeile 2 beginnt eine Methode , die in Zeile 4 endet. Methoden werden in Kapitel 6 erläutert. Hier ist nur wichtig, dass `main` die Methode ist, die bei Programmstart aufgerufen wird.

In Zeile 3 steht eine Anweisung, die den Text “Hello Web!” ausgibt. Es handelt sich hierbei um einen Methodenaufruf mit einer Zeichenkette als Parameter. Alle Anweisungen in Java müssen mit einem Strichpunkt (;) abgeschlossen werden. Es ist üblich nur eine Anweisung pro Zeile zu schreiben, obwohl Zeilenvorschübe, Tabulatoren und Leerzeichen vom Kompiler als “White-space” ignoriert werden; sie dienen nur der besseren Lesbarkeit.

Die Methode `System.out.println` gibt den Parameter als Text mit Zeilenvorschub auf der Standardausgabe aus, was normalerweise die DOS-Eingabeaufforderung unter Windows, bzw. die Shell unter Unix ist. Wenn Sie keinen Zeilenvorschub haben wollen, müssen sie `System.out.print` verwenden.

Sie können mehrere Texte ausgeben, indem sie ein Pluszeichen (+) dazwischenschreiben, wobei Zahlen automatisch in Zeichenketten gewandelt werden:

```
    System.out.print( "Java" + "kurs" );
    System.out.println( "Java" + 3 );
```

Beispiel 3.1.1: Textausgabe

3.2 Ein erstes Applet

Ein Applet ist eine „kleine Anwendung (Application)". Damit bezeichnet man die ganzen Helferlein, z.B. um die Uhrzeit am PC einzustellen. Unter Java ist ein Applet eine kleine Java-Anwendung, die meist auf einer Webseite läuft. Man kann es sich so vorstellen, als wenn ein Programmfenster (ohne Rahmen) in eine Seite eingebaut wurde. Obwohl es vielen anders vorkommt, stellen Applets nur einen kleinen Teilaspekt dessen dar, was mit Java möglich ist. Der erste Teil dieses Kurses wird mit Applets arbeiten, da mit ihnen vieles leichter zu erklären ist. Später werden sie lernen, wie sie eigenständige Anwendungen - so wie das Beispiel aus dem letzten Kapitel - erstellen können, die ohne einen Browser lauffähig sind. Um den Einstieg so leicht wie möglich zu gestalten, finden sie auf der Kurs-CD eine Datei namens `KursApplet.class`. Kopieren sie diese bitte in ihrem `Java`-Verzeichnis, da sonst die Beispiele nicht funktionieren werden.

Eine Linie

Das erste Beispiel zeichnet nur eine einfache Linie auf den Bildschirm und bietet ihnen die Möglichkeit, sich mit Applets vertraut zu machen.

```
public class Beispiel1 extends KursApplet {
  void zeichnen() {
    linie( 5, 10, 50, 60 );
  }
}
```

Beispiel 3.2.1: Eine Linie `Beispiel1.java`

Natürlich brauchen sie noch eine HTML-Datei, die das Applet enthält.

```
<HTML>
<HEAD>
   <TITLE>Beispiel 1</TITLE>
</HEAD>
<BODY>
<APPLET code="Beispiel1.class" width=350 height=200></APPLET>
</BODY>
</HTML>
```

Beispiel 3.2.2: HTML-Datei `Beispiel1.html`

Es ist für diesen Kurs nicht wichtig, dass sie HTML können; für alle weiteren Beispiele können sie diese HTML-Datei als Vorlage nehmen. Im Abschnitt 11.5 finden sie eine ausführlichere Erläuterung des verwendeten HTML-Tags.

Der Entwicklungszyklus dieser einfachen Beispiele ist immer der gleiche:

1. Programmcode schreiben/ändern
2. Programmcode kompilieren
3. Browser/Appletviewer starten

Es empfiehlt sich den im JDK enthaltenen *Appletviewer* an Stelle eines Browsers, wie z.B. *Netscape*, zu verwenden, da nach jeder Änderung eines Applets der Browser neu gestartet werden muss und der Appletviewer hierbei deutlich schneller ist. Für dieses Beispiel schreiben sie

```
C:\JAVA> jikes Beispiel1.java
C:\JAVA> appletviewer Beispiel1.html
```

Achten sie bitte unbedingt auf Groß-/Kleinschreibung, da es sonst nicht funktioniert! Wenn sie alles richtig gemacht haben, sollten sie eine Linie von links oben nach rechts unten erhalten.

Java-Grafiksystem

Die folgende Beschreibung wird etwas mathematisch, ist aber recht intuitiv und sollte mit normalen Schulkenntnissen leicht verständlich sein. Das Java-Grafiksystem verwendet ein kartesisches Koordinatensystem mit Nullpunkt in der linken oberen Ecke. Die X-Achse geht nach rechts und die Y-Achse nach unten. Punkte in diesem Koordinatensystem sind durch ihre X- und Y-Koordinate genau bestimmt, wobei immer zuerst die X-Koordinate angegeben wird. Um eine Linie zu zeichnen, muss ein Start- und ein Endpunkt angegeben werden. Die Anweisung `linie( 5, 10, 50, 60 )` zeichnet also eine Linie vom Startpunkt $(5,10)$ zum Endpunkt $(50,60)$, wobei sich der Startpunkt 5 Pixel rechts und 10 Pixel unterhalb und der Endpunkt 50 Pixel rechts und 60 Pixel unterhalb der linken oberen Ecke befindet.

3.3 Datentypen

Java kennt zwei verschiedene Datentypen

1. integrale Typen
2. Klassen

integrale Datentypen

Java kennt acht verschiedene integrale Datentypen:

`boolean`	1 Bit, wahr (`true`) oder falsch (`false`)
`byte`	8 Bit, von -128 bis 127
`short`	16 Bit, von -32768 bis 32767
`int`	32 Bit, von -2147483648 bis 2147483647
`long`	64 Bit, von -9223372036854775808 bis 9223372036854775807
`char`	16 Bit, von '\u0000' bis '\uffff' (0 bis 65535)
`float`	32-bit, IEEE 754, 1.40239846e-45 bis 3.40282347e+38
`double`	64-bit, IEEE 754, 4.94065645841246544e-324 bis 1.79769313486231570e+308

Tabelle 3.1: integrale Datentypen

In Java gibt es das Schlüsselwort `unsigned` nicht; alle Datentypen (bis auf `char`) sind vorzeichenbehaftet. Der Typ `char` ist dazu gedacht, ein Zeichen zu enthalten, wobei als Zeichensatz *Unicode* verwendet wird, ein standardisierter Zeichensatz, mit dem sich die Schriftzeichen aller Sprachen darstellen lassen.

Klassen

Klassen sind komplexere Datentypen, die mehrere integrale Datentypen und andere Klassen, sowie die dazugehörenden Methoden enthalten. Klassen werden später in Kapitel 7 detailliert behandelt. Hier nur soviel, dass Klassen per Referenz (by reference) an Methoden übergeben werden, diese also die enthaltenen Werte verändern können.

3.4 Variablen - ein Rechteck

Als nächsten wollen wir ein Rechteck zeichnen, wofür vier Linien notwendig sind: von links oben nach rechts oben, von rechts oben nach rechts unten usw. Wenn man das ganze programmiert fällt auf, dass der Endpunkt jeder Linie gleich dem Startpunkt der jeweils nächsten Linie ist. Die Angabe dieser Koordinaten ist mühselig, fehleranfällig und man muss sie für jedes Rechteck neu eingeben. Besser wäre es, wenn man nur die wesentlichen Angaben zu machen braucht und der Rest automatisch geht.

Um ein Rechteck genau zu bestimmen, benötigt man nur zwei Koordinaten, z.B. die der linken oberen und der rechten unteren Ecke - alle anderen Koordinaten kann man daraus bestimmen.

Jede dieser Koordinaten ist durch seine beiden Werte *x* und *y* bestimmt. Da wir zwei Koordinaten benötigen, nennen wir sie x_1, y_1 und x_2, y_2.

Die Start- und Endpunkte des zu zeichnenden Rechtecks lassen sich jetzt durch diese Koordinaten genau beschreiben, ohne dass der genaue Wert der Koordinate bekannt sein muss. Durch Verändern dieser Koordinaten kann man dann jedes beliebige Rechteck zeichnen. Die Linien gehen

- von x_1, y_1 nach x_2, y_1
- von x_2, y_1 nach x_2, y_2
- von x_2, y_2 nach x_1, y_2
- von x_1, y_2 nach x_1, y_1

Um das ganze in Java zu schreiben, benötigen wir *Variablen*. Variablen sind „benannte Speicherbereiche", Speicher im Computer, der sich über einen Namen - wie z.B. `x1` - ansprechen lässt. Variablen lassen sich überall dort verwenden, wo man einen Wert auch direkt angeben könnte, insbesondere also bei `linie()`.

Variablen in Java enthalten entweder einen integralen Datentyp oder eine Klasse. Java ist streng typisiert, d.h. eine Variable hat immer den gleichen Datentyp; Zuweisungen mit anderen Datentypen erzeugen (meistens) einen Fehler. Zeiger auf Variablen (Pointer) gibt es in Java nicht und damit auch keine Zeigerarithmetik.

Damit Java den Typ einer Variable kennt, muss er bei der Definition durch Voranstellen angegeben werden. Wollen sie mehrere Variablen vom gleichen Typ haben, so können sie diese mit Komma getrennt angeben. Als Variablennamen können sie beliebig viele Unicode-Zeichen verwenden, wobei sie nicht mit Zahlen beginnen dürfen und natürlich "Buchstaben und Zahlen" verwenden müssen. Variablendefinitionen müssen ebenfalls mit einem Strichpunkt abgeschlossen werden.

```
int a;
int x1, y1, x2, y2;
double dSumme, dÜbertrag, dMwst16;
```

Beispiel 3.4.1: Variablendefinitionen

Ein Fehler in *Jikes* erlaubt es zur Zeit nicht, Umlaute in Variablen oder Methodennamen zu verwenden, die von anderen Klassen aus angesprochen werden. Lokal verwendete Variablen und Methoden sind davon nicht betroffen.

In diesem Skript werden Variablen meistens in der sogenannten „ungarischen Notation" angegeben, d.h. mit einem Präfix, der den Typ der Variablen angibt. Diese erleichtert die Suche von Fehlern, die auf falschen Typen (z.B. `int` statt `float`) beruhen.

Eine Variable ist innerhalb des Blocks bekannt, in dem sie definiert wurde. Innerhalb eines Blocks muss jeder Variablenname eindeutig sein, d.h. sich in mindestens einem Zeichen von allen anderen Namen unterscheiden, sowohl Variablennamen, als auch Methodennamen.

Initialisierung

Variablen dürfen bei ihrer Definition gleich mit einem Wert vorbelegt werden. Dies geschieht durch ein Gleichheitszeichen gefolgt vom gewünschten Wert; dies muss bei jeder Variable einzeln erfolgen.

```
int x1 = 0, y1 = 0, x2, y2;
double dSumme, dÜbertrag = 0.00, dMwst16;
```

Beispiel 3.4.2: Variablendefinitionen

Übung

Schreiben sie ein Programm, das ein Rechteck zeichnet.

3.5 Zuweisungen

Variablen können sie beliebige (gültige) Werte zuweisen, indem sie nach dem Variablennamen ein Gleichheitszeichen und dann den Wert schreiben. Sie können den zuzuweisenden Wert auch aus verschiedenen Werten oder Variablen berechnen. Hierzu stehen ihnen die 4 Grundrechenarten Addition (+), Subtraktion (-), Multiplikation (*) und Division (/), die Modulo-Operation (%), sowie Klammern zur Verfügung, wobei Java die Regel “Punkt vor Strich” beachtet.

```
int a, b;
a = 3;
a = 3 * ( 5 + 1 );
a = a + 1;
b = 2 * a + 3;
```

Beispiel 3.5.1: Zuweisungen

Für verschiedene häufig vorkommende Zuweisungen gibt es Abkürzungen

- `a *= 3` entspricht `a = a * 3`
- `a++` entspricht `a = a + 1`

Wenn sie ganzzahlige Variablen (`byte`, `int` usw.) dividieren, dann erfolgt die Division auch ganzzahlig abgerundet, d.h. `5 / 3` ergibt `1`.

Casting

Variablen unterschiedlichen Typs kann man nur bedingt einander zuweisen. Der Typ `boolean` lässt sich z.B. keinem anderen Datentyp zuweisen. Dagegen kann man immer einen Datentyp in einen mit “höherer Genauigkeit” zuweisen. Man kann daher eine `int`-Variable einer `long`- oder einer `double`-Variablen zuweisen, aber nicht umgekehrt, da man dabei Daten verlieren könnte.

Java lässt es jedoch zu, die Umwandlung mittels *Casting* zu erzwingen. Hierbei gibt man den Zieldatentyp in Klammern vor dem Quelldatentyp an:

```
long l   = 123456789;
double d = 12345.789;
int i;
i = (int) l;
i = (int) d;
```

Beispiel 3.5.2: Casting

Der Kompiler geht dann davon aus, dass man weiß, was man tut. Es funktionieren jedoch nur “sinnvolle” Casts - einen `boolean` kann man damit immer noch nicht umwandeln.

3.6 Kontrollstrukturen

Um den Programmfluss zur Laufzeit zu verändern, gibt es Kontrollstrukturen, mit denen sich das Verhalten in Abhängigkeit von Variablen verändern lässt. In allen Kontrollstrukturen werden Ausdrücke auf ihren Wahrheitsgehalt geprüft. Ihnen stehen mehrere Operatoren zur Verfügung, um Ausdrücke zu schreiben (Tabelle 3.2).

Zusätzlich können sie mit dem Ausrufungszeichen (`!`) eine Bedingung invertieren. Sie können mehrere Ausdrücke mit logischen Verknüpfungen zusammenfassen (siehe Tabelle 3.3).

Sie können Klammern verwenden, um eine bestimmte Reihenfolge zu erzwingen (siehe Beispiel 3.6.1).

Gleichheit	`==`
Ungleichheit	`!=`
größer als	`>`
größer, gleich	`>=`
kleiner als	`<`
kleiner, gleich	`<=`

Tabelle 3.2: Vergleiche

logisches Und	`&&`
logisches Oder	`||`

Tabelle 3.3: logische Verknüpfungen

```
a == 3
a > b
( a > b ) || ( a == 3 )
( a > b ) || !( a < 10 )
```

Beispiel 3.6.1: logische Ausdrücke

Bedingungen

Die einfachste Kontrollstruktur ist eine Bedingung: Wenn (`if`) etwas wahr ist, dann mache etwas, sonst (`else`) mache etwas anderes, wobei der `else`-Zweig auch weggelassen werden kann. Hinter `if` und `else` darf jeweils nur eine Anweisung stehen; wollen sie mehrere Anweisungen ausführen, müssen sie diese in einen Block schreiben.

```
if( a < 0 ) a = 0;

if( a > 255 ) {
    System.out.println( "Bereichsüberschreitung" );
    a = 255;
}

if( a > 128 ) {
    System.out.println( "groß!" );
    a = 255;
} else {
    System.out.println( "klein!" );
    a = 0;
}
```

Beispiel 3.6.2: `if-else`

`while`-Schleifen

In einer `while`-Schleife werden die Anweisungen solange ausgeführt, wie die angegebene Bedingung erfüllt ist. Alle Schleifen, auch die folgenden, können mit `break` abgebrochen werden; der Befehl `continue` springt wieder zum Schleifenanfang.

`do-while`-Schleifen

Die `do-while`-Schleife unterscheidet sich von der `while`-Schleife nur dadurch, dass die Bedingung erst geprüft wird, nachdem die Anweisungen ausgeführt wurden.

```
a = 1000;
while( a > 255 ) a -= 255;

a = 0;
while( a < 10 ) {
    System.out.println( a );
    a++;
}
```

Beispiel 3.6.3: `while`

```
a = 1000;
do a -= 255; while( a > 255 );

a = 0;
do {
    System.out.println( a );
    a++;
} while( a < 10 );
```

Beispiel 3.6.4: `do-while`

`for`-Schleifen

Eine `for`-Schleife ist eine `while`-Schleife mit zwei zusätzlichen Anweisungen, die

- vor dem ersten Ausführen der Schleife
- nach jedem Schleifendurchlauf

ausgeführt werden.

```
for( a = 1; a < 10; a++ ) {
    System.out.println( a );
}
```

entspricht

```
a = 1;
while( a < 10 ) {
    System.out.println( a );
    a++;
}
```

Beispiel 3.6.5: `for`

Verzweigungen

Wenn sie viele Fälle zu unterscheiden haben, bietet sich die `switch`-Anweisung an. Hier können sie einen Ausdruck angeben, der einen Integer liefern muss. Abhängig vom Wert dieses Integers wird dann in den entsprechenden Zweig gegangen. Die einzelnen Zweige werden mit `case` angegeben. Die `case`-Anweisungen verstehen sich als Sprungadressen, denn ein weiteres `case` bedeutet nicht das Ende der Verzweigung, sondern nur ein weiterer Einsprungpunkt. Um eine Verzweigung zu beenden, müssen sie den Befehl `break` angeben. Wie in Beispiel 3.6.6 gezeigt, kann man dies benutzen, um mehrere Integer abzudecken.

Sollte keine der Zahlen bei `case` zutreffen, so wird der Zweig `default` ausgeführt.

- Achten sie immer darauf alle Zweige mit einem `break` zu beenden.
- Fehlt der `default`-Zweig und trifft kein `case` zu, so gibt es keinen Fehler, sondern die Ausführung wird nach dem `switch` fortgesetzt.

```
switch( iTag ) {
case 0:
    System.out.println( "Montag" );
    break;
case 1:
    System.out.println( "Dienstag" );
    break;
case 2:
    System.out.println( "Mittwoch" );
    break;
case 3:
    System.out.println( "Donnerstag" );
    break;
case 4:
    System.out.println( "Freitag" );
    break;
case 5:
case 6:
    System.out.println( "Wochenende" );
    break;
default:
    System.out.println( "Unbekannter Tag" );
    break;
}
```

Beispiel 3.6.6: `switch`

3.7 Kommentare

Nicht jedes Stück Programmcode ist auf Anhieb verständlich. Deshalb schreibt man *Kommentare* in den Quellcode, die diesen beschreiben. Kommentare sind speziell markierte Textstellen, die vom Kompiler ignoriert werden.

Es gibt zwei Arten von Kommentaren, die man am schnellsten an Beispiel 3.7.1 versteht:

```
// Das ist ein Kommentar über eine einzelne Zeile

/* Will man Kommentare über mehrere Zeilen laufen
   lassen, so braucht man diese Konstruktion */
```

Beispiel 3.7.1: `Kommentare`

Einzeilige Kommentare werden mit `//` eingeleitet und gehen bis zum Zeilenende. Mehrzeilige Kommentare muss man in `/*` und `*/` einfassen; alles zwischen diesen beiden Symbolen wird vom Kompiler ignoriert.

Mehrzeilige Kommentare dürfen nicht geschachtelt werden, d.h. eine Konstruktion wie ist unzu-

```
/* hier ist ein Kommentar
    /* und hier der nächste */
   hier soll der erste enden
*/
```

Beispiel 3.7.2: Falscher Kommentar

lässig und führt zu einem Fehler! Einzeilige Kommentare sind hingegen erlaubt:

Im Abschnitt 13.4 lernen sie, wie sie spezielle Kommentare dazu einsetzen können, ihr Programm automatisch zu dokumentieren.

```
/* hier ist ein Kommentar
    // hier ein einzeiliger
   // hier das Ende */
```

Beispiel 3.7.3: Korrekter Kommentar

3.8 Übungen

1. Welches Zeichen muss nach jeder Anweisung stehen?

2. Welchen Datentyp brauchen Sie, um die Zahlen von 1 - 200 darzustellen?

3. Warum gibt `System.out.println( 3 + 5 )` nicht 35, sondern 8 aus?

4. Wir kann man erreichen, dass stattdessen 35 ausgegeben wird?

5. Schreiben sie ein Programm, das die Quadrate der (natürlichen) Zahlen von 1 bis 10 ausgibt! (das Quadrat einer Zahl `n` berechnen sie am einfachsten, indem sie die Zahl mit sich selbst multiplizieren, also `n * n`)
6. *Schreiben sie ein Programm, das eine Multiplikationstabelle für die (natürlichen) Zahlen von 1 bis 10 ausgibt.
7. Mit `oval( x, y, breite, hoehe )` können sie ein Oval zeichnen; sind Breite und Höhe gleich, so entsteht ein Kreis. Schreiben sie ein Programm, das 10 Kreise nebeneinander Zeichnet.
8. *Erweitern sie das obere Programm so, dass statt einem Kreis jeweils 5 konzentrische Kreise mit unterschiedlichen Radien gezeichnet werden (”Zielscheiben”);
9. *Zeichnen sie ein Gitter mit 10 auf 10 Linien.
10. Mit `farbe( ... )` können sie die Farbe des Zeichenstiftes für alle folgenden Zeichenoperationen umschalten (”Stiftwechsel”). Als Farben stehen ihnen SCHWARZ, WEISS, ROT, GELB, GRUEN, BLAU, CYAN, DUNKELGRAU, GRAU, HELLGRAU, MAGENTA, ORANGE und ROSA in genau dieser Schreibweise zur Verfügung. Erweitern sie das obere Programm so, dass der 3. Kreis gelb gezeichnet wird.

3.9 Zusammenfassung

In diesem Kapitel haben sie gelernt

- was das Beispiel im vorherigen Kapitel bedeutet
- welche Datentypen Java hat
- wie sie Variablen definieren und benutzen
- welche Kontrollstrukturen sie benutzen können

Sie sind jetzt in der Lage kleinere Programme selbst zu schreiben. In den folgenden Kapiteln lernen sie mehr über die Vorteile objektorientierter Programmierung, Netzwerkzugriff und Grafikausgabe.

Kapitel 4

Etwas Übung

Wir werden jetzt das Gelernte an ein paar Beispielen vertiefen. Hierbei ist es leider notwendig etwas vorzugreifen, damit die Beispiele nicht zu trocken werden. Wenn sie also etwas nicht verstehen, dann keine Sorge - es wird später alles genau erklärt.

4.1 Einfache Ein- / Ausgabe

`System.out` haben sie schon kennengelernt, die Standardausgabe für Texte. Das gleiche gibt es auch zum Lesen von Zeichen: `System.in`. Leider unterstützt `System.in` nicht mehr als das einzelne Lesen von Zeichen mittels `read()`. `System.in.read()` liefert einen `int` zurück, der das nächste Zeichen enthält, oder `-1`, wenn die Datei zu Ende ist.

Leider kann man `System.in.read()` nicht so einfach verwenden, wie `System.out.println()`, denn Java fordert hier ein Abfangen von eventuellen Lesefehlern. Wie das genau geht, erfahren sie im Abschnitt 7.10; die Anweisungen hierzu lauten `try` und `catch`.

`System.out` ist für die "normale" Ausgabe eines Programmes vorgesehen und heißt "Standardausgabe"; `System.err` ist die "Standardfehlerausgabe" und ist für Fehlermeldungen vorgesehen. Alle Ausgaben auf `System.err` erscheinen normalerweise auf dem Bildschirm, genauso wie `System.out`.
Es ist möglich, eine der beiden Ausgaben in eine Datei umleiten zu lassen, so dass man z.B. nur Fehlermeldungen angezeigt bekommt. Aus diesem Grund sollte man sich angewöhnen, Fehlermeldungen nach `System.err` zu schreiben.

Beispiel 4.1.1 zeigt ein kleines Programm, das dies alles beachtet und seine Eingabe auf die Ausgabe kopiert. Wichtig ist hier der Cast nach `char`, weil sonst die Unicode-Werte der Zeichen als Zahlen ausgegeben werden.

```
public class Cat {
    public static void main( String[] argv ) {
        int c;
        try {
            while( ( c = System.in.read() ) != -1 ) {
                System.out.print( (char) c );
            }
        } catch( Exception e ) {}
    }
}
```

Beispiel 4.1.1: ein einfaches 'cat'

Um das Programm zu testen, empfiehlt es sich, die Eingabe dafür aus einer Datei zu lesen, statt selbst über die Tastatur einzugeben:

```
C:\JAVA> java Cat < Cat.java
```

mit dem <-Zeichen leiten sie hierbei den Inhalt der Datei `Cat.java` als Eingabe an das Programm weiter, so als wenn sie den Inhalt über die Tastatur eingeben.

Übung

Schreiben sie jetzt ein Programm, das die Anzahl der Buchstaben, die der Großbuchstaben und die der Kleinbuchstaben in einem Text zählt. Beachten sie hierbei, dass es Sprachen gibt, die nicht zwischen Groß- und Kleinbuchstaben unterscheiden (z.B. Arabisch) und achten sie auf die korrekte Behandlung von koreanischen und japanischen Schriftzeichen.

Hinweis: Sie programmieren in eine Sprache, die für internationale Zwecke bestimmt ist. Verwenden sie also die Methoden `Character.isLetter()`, `Character.isUpperCase()` und `Character.isLowerCase()`, die alle einen `char` haben wollen und `true` oder `false` zurückliefern.

```
    char ch = 'ü';
    if( Character.isLowerCase( ch ) ) {
        System.out.println( "Alles in Ordnung" );
    } else {
        System.out.println( "Die JVM ist kaputt!!!" );
    }
```

Beispiel 4.1.2: Beispiel für `isLowerCase`

4.2 Bessere Ein- / Ausgabe

Das zeichenweise Einlesen von Eingabezeichen ist etwas unkomfortabel, wenn man Benutzereingaben verarbeiten will. Von daher hier noch etwas mehr "Voodoozauber", den sie später noch besser verstehen werden. Beispiel 4.2.1 zeigt, wie man in Java eine Zeichenkette einliest und ausgeben kann. Beachten sie die Zeile mit `import` am Anfang! Was sie bedeutet wird in Abschnitt 8.6 erläutert; hier reicht es zu wissen, das man ohne sie die Ein-/Ausgabefunktionen nicht (leicht) nutzen kann.

```
import java.io.*;

public class WerBistDu {
    public static void main( String[] argv ) {
        try {
            BufferedReader br = new BufferedReader(
                new InputStreamReader( System.in ) );

            System.out.println( "Wie heißt Du?" );
            String sName = br.readLine();
            System.out.println( "Hallo " + sName + "!" );

        } catch( Exception e ) {}
    }
}
```

Beispiel 4.2.1: Java ist freundlich

4.3 Passwortabfrage

`String`s sind keine integralen Variablen, wie `int` oder `double`; deshalb kann man sie nicht mit == vergleichen. Es gibt in Java dafür die Methode `equals`:

```
    if( sName.equals( "Kurt" ) ) {
      // ...
    }
```

Beispiel 4.3.1: `equals`

Schreiben sie mit diesem Wissen ein Programm, das eine Passwortabfrage vornimmt und bei Eingabe des richtigen Passworts ein kleines Geheimnis ihrer Wahl preisgibt!

4.4 Zeichenketten in Zahlen umwandeln

Von der Tastatur kann man in Java nur Zeichenketten lesen. Will man Zahlen einlesen, so muss man diese Zeichenketten nach dem Einlesen umwandeln. Hierfür ist wieder etwas “Zauber” notwendig:

```
String s = "1234.5678";
double d = new Double( s ).doubleValue();
```

Beispiel 4.4.1: Zeichen in Zahlen umwandeln

Schreiben sie jetzt einen Eurorechner, der DM-Beträge in Euro umrechnen kann. Er soll einen DM-Betrag von der Tastatur einlesen und ihn in Euro umrechnen. Verwenden sie dabei den Kurs 1 Euro = 1,95583 DM.

Kapitel 5
Arrays

Arrays sind Variablen, die mehrere Objekte enthalten können. Auf jedes Objekt kann mit einem *Index* zugegriffen werden.

In diesem Kapitel lernen sie

- wie sie Arrays erzeugen und initialisieren
- wie sie auf Arrayelemente zugreifen
- wie sie die Länge eines Arrays bestimmen

5.1 Erzeugung von Arrays

Es gibt in Java zwei Möglichkeiten, Arrays zu erzeugen. Die erste verwendet `new` und gibt an, wie groß das Array sein soll:

```
int aiZahlen[] = new int[ 20 ];
```

Die Elemente eines Arrays werden mit den Standardwerten für den Variablentyp initialisiert; bei `int` ist dies `0` und bei `boolean` ist er `false`. Die einzelnen Elemente eines Arrays müssen also getrennt initialisiert werden.

Die zweite Möglichkeit ein Array zu erzeugen ist ein *Initialisierer*:

```
int[] aiZahlen = { 1, 2, 3, 4, 5 };
```

Bei einer Neubelegung muss man allerdings folgende Syntax verwenden (erst ab Java 1.1 möglich):

```
int[] aiZahlen = { 1, 2, 3, 4, 5 };
aiZahlen = new int[] { 6, 7, 8 };
```

5.2 Mehrdimensionale Arrays

In Java ist ein mehrdimensionales Array ein “Array von Arrays”, d.h. die Elemente der ersten Dimension sind Arrays mit einer Dimension weniger. Sie definieren ein mehrdimensionales Array, indem sie einfach mehrere eckige Klammern hintereinander schreiben.

```
int aiMehrZahlen[][] = new int[ 10 ][ 20 ];
```

In Java ist es nicht notwendig, dass jedes Element einer Dimension die gleiche Anzahl an Elementen hat

Sie können in Java bei Variablendefinitionen die eckigen Klammern sowohl hinter den Datentyp (`int[] aiZahlen`), als auch hinter die Variable schreiben (`int aiZahlen[]`), oder auch beides mischen; alle diese Verwendungen sind erlaubt und identisch.

```
int aiZahlen[ 3 ][] =
    { new int[ 10 ], new int[ 20 ], new int[ 30 ] };
```

5.3 Zugriff auf Arrayelemente

Um auf ein Element eines Arrays zuzugreifen, geben sie einfach einen Index in jeder Dimension an, der das Element beschreibt - die Indizes beginnen wie in C/C++ bei 0!

```
int a = aiZahlen[ 0 ][ 1 ];
int[] aiWenigerZahlen = aiZahlen[ 2 ];
```

Die JVM überprüft bei jedem Zugriff auf ein Array, ob der Zugriff innerhalb der Arraygrenzen liegt. Sollte ein Fehler auftreten, so kommt es zu einer *Ausnahme*; Ausnahmen sind in Abschnitt 7.10 beschrieben.

5.4 Länge eines Arrays

Sie können die Länge eines Arrays herausfinden, indem sie hinter den Namen des Arrays `.length` schreiben:

```
int[] a = new int[ 5 ];
System.out.println( "Die Länge von a[] ist: " +
                    a.length );
```

5.5 Übungen zu Arrays

1. Schreiben sie ein Programm, das ausgibt, wieviele Parameter ihm übergeben wurden. Hinweis: Schauen sie sich die Definition von `main` an.
2. Schreiben sie ein Programm, das aus dem Array `{ 3, 2, 7, 9, 1, 5, 4 }` die größte und die kleinste Zahl heraussucht und ausgibt.
3. *Schreiben sie ein Programm, das ein Array von `int` aufsteigend sortiert.

Kapitel 6
Methoden

In diesem Kapitel lernen sie

- was Methoden sind
- was lokale Variablen sind
- wie sie Methodenparameter angeben
- wie sie Rückgabewerte zurückliefern
- wie sie Methoden überladen

6.1 Was sind Methoden?

In der Programmierung steht man häufiger vor dem Problem, dass man die gleichen Anweisungen an mehreren Stellen des Programms benötigt. Ein Beispiel hierfür ist das Rechteck aus dem Abschnitt 3.4. Will man mehrere Rechtecke ausgeben, so musste man bisher den Programmcode kopieren. In kleinen Programmen kann man sich mit Kopieren und Einfügen etwas Arbeit ersparen, aber es ist nicht sonderlich elegant, zumal auch deutlich mehr Anweisungen notwendig sein können.

Um dieses Problem zu beheben, gibt es “Methoden”. Analog zu Variablen, die „benannter Speicher” sind, sind Methoden „benannter Programmcode”. Man gibt einer Folge von Anweisungen einen Namen und kann dann diese über den Namen aufrufen, wie in Beispiel 6.1.1 gezeigt.

```
void zeichnen() {
    meinlogo();
}

void meinLogo() {
    linie( 5, 5, 20, 30 );
    linie( 20, 30, 25, 20 );
    linie( 25, 20, 30, 40 );
}
```

Beispiel 6.1.1: Eine einfache Methode

Es ist in Java üblich, den Namen von Methoden klein zu schreiben, wobei man bei zusammengesetzten Namen die jeweils ersten Buchstaben der hinteren Teile groß schreibt, z.B. `meineMethode` oder `schreibeErstenDatensatz`.

Man sieht hier in `zeichnen()` den Aufruf der Methode `meinLogo()`. Alle Methoden müssen innerhalb einer Klasse definiert werden (man sagt im “Klassenkontext”); es ist nicht möglich eine Methode innerhalb einer anderen Methode zu definieren.

Eine einmal definierte Methode lässt sich beliebig oft im Programm verwenden, wobei eine Methode auch andere Methoden aufrufen darf. Auf diese Weise erspart man sich nicht nur sehr viel Tipparbeit, sondern Änderungen können viel leichter und weniger fehleranfällig durchgeführt werden, weil man das Programm nur noch an einer und nicht an vielen Stellen ändern muss.

Im Gegensatz zu C/C++ oder Pascal braucht man eine Methode nicht zu deklarieren, wenn man sie verwenden will, bevor sie definiert wurde, d.h. wenn die Verwendung im Programmtext vor der Definition steht. Eine Methode gehört zur Klasse und ist deshalb überall verfügbar, egal wann sie definiert wird.

6.2 lokale Variablen

Technisch gesehen unterscheidet die Methode `zeichnen()` von `meinLogo()` nur die Konvention, dass `zeichnen()` beim Neuzeichnen des Applets ausgeführt wird. Wir können also in `meinLogo()` alles machen, was wir auch in `zeichnen()` gemacht haben, insbesondere eigene Variablen definieren und verwenden. Variablen, die innerhalb einer Methode definiert sind, heißen *lokale Variablen*.

Lokale Variablen sind nur innerhalb der Methode bekannt, in der sie definiert wurden und sie sind unabhängig von allen Variablen, die in anderen Methoden definiert werden, selbst wenn diese den gleichen Namen haben. Wenn sie also in `zeichnen()` eine Variable `a` benutzen und in `meinLogo()` ebenfalls eine Variable mit Namen `a` definieren, so haben diese keinen Einfluss aufeinander, d.h. eine Veränderung von `a` in `meinLogo()` verändert den Wert von `a` in `zeichen()` nicht.

Wir werden später noch Variablen kennenlernen, die innerhalb einer Klasse definiert sind und für alle Methoden gleich sind, d.h. alle Methoden können auf diese Variablen zugreifen und es sind immer die gleichen Variablen.

6.3 Parameter

Die Klammern hinter `meinLogo` lassen vermuten, dass man den Methoden Werte übergeben kann und genauso ist es. Als Beispiel wollen wir eine Methode schreiben, die ein Rechteck ausgibt, wenn man die Koordinaten von der linken oberen und der rechten unteren Ecke angibt, also z.B.

```
meinRechteck( 5, 10, 150, 100 );
```

Man muss sich zuerst überlegen, welche Parameter man dieser Methode übergeben will, wobei dies hier sehr einfach ist: die beiden Koordinaten jeweils mit $x-$ und y-Wert In Beispiel 6.3.1 sehen sie, wie man so etwas implementieren kann.

```
void zeichnen() {
    meinRechteck( 5, 10, 150, 100 );
}

void meinRechteck( int x1, int y1, int x2, int y2 ) {
    linie( x1, y1, x2, y1 );
    // usw.
}
```

Beispiel 6.3.1: Methode `meinRechteck()`

Benötigt eine Anweisung mehr Zeichen, als in eine Zeile passen, so ist es üblich, diese so auf mehrere Zeilen aufzuteilen, dass zusammengehörende Teile in der gleichen Spalte anfangen.

Man sieht, dass beim Methodenaufruf die Parameter mit Komma getrennt angeben werden. Hierbei ist es egal, ob man Konstanten oder Variablen übergibt, wie man oben sieht. Man hätte also die `5` auch direkt angeben können.

Die Definition der Parameter bei der Methodendefinition sieht so ähnlich aus, wie die Definition von Variablen. Hierbei ist nur zu beachten, dass bei jedem Parameter der Typ einzeln anzugeben ist, d.h. man kann mehrere Parameter mit dem gleichen Typ nicht zusammenfassen.

Parameter wirken innerhalb einer Methode wie lokale Variablen, werden aber mit dem übergebenen Wert initialisiert. Die Zuordnung von übergebenen Wert zum Namen des Parameters

erfolgt hierbei über die Position, d.h. der erste übergebene Wert wird in den ersten Parameter geschrieben, der zweite in den zweiten usw.

Eine Methode kann den Wert eines Parameters verändern, allerdings hat dies keinen Einfluss auf das aufrufende Programm; man sagt, die Parameter werden "by value" übergeben, d.h. nur der Wert, nicht die Variablen selbst wird weitergegeben (siehe Beispiel 6.3.2). Dies ist auch ganz logisch, da man, wie man oben gesehen hat, auch Konstanten übergeben kann, die nicht verändert werden können.

```
    int a = 5;
    System.out.println( a );
    meth( a );
    System.out.println( a );  // a ist immer noch 5!

void meth( int x ) {
    x = 3;
}
```

Beispiel 6.3.2: Parameterübergabe als Wert

6.4 Rückgabewerte

Bisher stand vor den Methodennamen immer ein `void` ("leer"). Dieses `void` gibt an, dass die Methode keinen Wert zurückliefert. Man kann allerdings auch Methoden schreiben, die einen beliebigen Wert zurückliefern. Hierzu muss man nur den Typ des Rückgabewertes anstelle von `void` schreiben und am Ende der Methode mit `return` den Wert angeben, der zurückgeliefert werden soll. Im Beispiel 6.4.1 sehen sie eine Methode, die das Quadrat der übergebenen Zahl zurückliefert.

```
    void zeichnen() {
        text( "Das Quadrat von 5 ist " + quadrat( 5 ), 20, 10 );
    }

    int quadrat( int i ) {
        return i * i;
    }
```

Beispiel 6.4.1: Methode "quadrat"

Die Anweisung `return` beendet die Methode sofort und liefert den angegebenen Wert zurück. Sie kann auch innerhalb eines Blocks, z.B. einer `if`-Anweisung stehen. Wichtig ist nur, dass irgendwie immer ein `return` ausgeführt wird, bevor das Ende der Methode erreicht ist. Man kann immer nur einen Wert zurückliefern. Wir werden in Kapitel 9 sehen, was man machen muss, um diese Einschränkung zu umgehen.

Der Aufruf der Methode kann überall dort stehen, wo auch ein Wert stehen kann, d.h. auch in Zuweisungen oder Methodenaufrufen. Man kann den Rückgabewert auch ignorieren, indem man die Methode einfach so aufruft.

6.5 Überladung

Was passiert, wenn man Parameter des falschen Typs angibt, also z.B.

```
text( "das war wohl nix", "20", 30 );
```

Man erhält eine Fehlermeldung vom Kompiler, dass es keine Methode `text` gibt, die für die ersten beiden Parameter einen `String` und als dritten einen `int` haben will. Man könnte also auf die Idee kommen genau so eine Methode zu schreiben und das funktioniert sogar!

Man nennt diesen Vorgang “Überladung”. Hierbei definiert man mehrere Methoden, die sich in Anzahl und/oder Typ der Parameter unterscheiden. Eine sinnvolle Anwendung dafür wäre z.B. eine weitere Methode `meinRechteck`, die nur drei statt vier `int` bekommt und damit ein Quadrat zeichnet mit dem dritten Parameter als Kantenlänge.

```
void meinRechteck( int x, int y, int iSeitenLänge ) {
    linie( x, y, x + iSeitenLänge, y );
    // usw.
}
```

Bei diesem Ansatz macht man sich doppelte Arbeit. Einfacher ist es, wenn man die Aufrufe der Methoden miteinander „verkettet”, d.h. dass beispielsweise Methoden mit weniger Parametern gleichnamige Methoden mit mehr Parametern aufrufen und die fehlenden Parameter durch Standardwerte ersetzen, oder wie in diesem Fall, die fehlenden Parameter errechen.

```
void meinRechteck( int x, int y, int iSeitenLänge ) {
    meinRechteck( x, y, x + iSeitenLänge, y + iSeitenLänge );
}
```

Dieser Ansatz ist deutlich eleganter und spart zudem noch einiges an Tipparbeit. Und wer weniger tippt, macht weniger Fehler!

6.6 Übungen

1. Schreiben sie eine Methode `doppletRechteck()`, die die gleichen Parameter bekommt, wie `meinRechteck()` bekommt, sowie einen zusätzlichen, der einen Abstand in Pixeln angibt. Die Methode soll dann zwei Rechtecke zeichenen, wobei das eine mit dem angegebenen Abstand innerhalb des anderen liegen soll.
2. Benutzen sie Überladung, um eine Version von `doppeltRechteck()` zu schreiben, die genausoviele Parameter bekommt, wie `meinRechteck` und immer einen Abstand von 2 Pixeln verwendet.
3. Schreiben sie einige Aufgaben aus den vorherigen Kapiteln so um, dass diese Methoden verwenden.

Kapitel 7
Klassen

In diesem Kapitel lernen sie

- wie sie Klassen definieren
- wie sie Instanzen erzeugen und benutzen
- wie sie Konstruktoren definieren
- wie sie Methoden überladen
- wie sie Klassenfelder und Klassenmethoden definieren
- wie sie Arrays erzeugen und verwenden
- wie sie Ausnahmen abfangen und erzeugen

7.1 Einführung in Klassen

Die reale Welt besteht nicht aus `int` und `char`, sondern aus Objekten, z.B. einem Computer. Diese Objekte haben Eigenschaften oder Zustände: 64MB RAM, ausgeschaltet, grau usw. Mit diesen Objekten kann man verschiedene Dinge tun, einschalten, umherschieben usw. Die objektorientierte Programmierung versucht dies in eine Programmiersprache abzubilden.

Ein Objekt in Java heißt *Klassse* und besteht aus zwei Teilen:

- Feldern
- Methoden

Ein Feld enthält einen beliebigen Datentyp, z.B. `int` oder `String`; die Felder einer Klasse speichern Ihren Zustand. Methoden sind Programmcode, der zu diesem Objekt gehört, z.B. zeigt die Methode `show` eines `Windows` dieses am Bildschirm an. Idealerweise wird der Zustand einer Klasse nur über Methoden und nicht über direkte Zugriffe auf die Felder geändert, denn dies erlaubt eine Prüfung der Änderungen.

In Java muss sich jedes Stück Programmcode innerhalb einer Klasse befinden; globale Prozeduren oder Variablen, wie in Pascal, C oder C++, gibt es nicht. Im Beispiel aus Kapitel 2 haben sie deshalb die Klasse “HelloWeb” definieren müssen und die eine Methode “main” innerhalb der Klasse angegeben.

7.2 Eine Fensterklasse

Beispiel 7.2.1 zeigt (im Ansatz), wie eine Klasse für ein Fenster aussehen könnte. Beachten sie bitte, dass sich die gesamte Klassendefinition innerhalb der geschweiften Klammern befindet und eine Definition ausserhalb, wie z.B. in C++, nicht erlaubt ist.

Nach dem Schlüsselwort `class` folgt der Name der Klasse, für den die gleichen Einschränkungen wie für Variablennamen gelten, gefolgt von einer öffnenden geschweiften Klammer. Beendet wird die Klassendefinition mit einer schließenden geschweiften Klammer, wobei hier kein Strichpunkt folgt.

Danach kommt als erstes die Definition der Felder, die wie Variablen definiert werden. Es ist üblich, aber nicht notwendig, die Felder an den Anfang einer Klasse zu schreiben.

Die Zeile mit den zwei Schrägstrichen ist ein Kommentar, d.h. alles nach den Schrägstrichen bis zum Zeilenende wird vom Kompiler ignoriert. Genau wie in C oder C++ gibt es in Java Kommentare über mehrere Zeilen; diese beginnen mit `/*` und enden mit `*/`.

Als letztes kommt die Definition der Methoden, die aus

```
public class Window {
    // Felder der Klasse Window
    int iXKoordinate, iYKoodinate;
    int iBreite, iHöhe;
    boolean isVisible;

    // Methoden der Klasse Window
    void show() { isVisible = true; }
    void hide() { isVisible = false; }
    void setBreite( int breite ) { iBreite = breite; }
    int getBreite() { return iBreite; }
}
```

Beispiel 7.2.1: die Klasse `Window`, teilweise implementiert

- Rückgabetyp
- Methodenname
- Parameter
- Programmcode

bestehen. Als Rückgabetyp dürfen sie alle Variablentypen angeben; wollen sie nichts zurückliefern, so müssen sie den Typ `void` (leer) angeben. Für den Methodennamen gilt das gleiche, wie für Variablen und Klassennamen. Die Parameter einer Methode werden ähnlich wie lokale Variablen definiert, wobei sie hier allerdings bei *jedem* Parameter den Typ angeben müssen. Den Programmcode müssen sie in geschweifte Klammern fassen, nach denen, genau wie bei Klassen, kein Strichpunkt folgt.

Objekte sind Instanzen einer Klasse

Nachdem wir nun (zumindest teilweise) die Klasse `Window` definiert haben, wollen wir auch etwas mit ihr anfangen. Mit der Window-Klasse selbst können wir nichts anstellen, sie entspricht einer Bauanleitung und keinem konkreten Objekt. Wir benötigen ein bestimmtes Fenster, mit dem gearbeitet werden soll, eine *Instanz* der Klasse.

Durch die Definition der Klasse `Window` haben wir einen neuen Variablentyp erzeugt, mit dem wir Variablen definieren können:

```
   Window w;
```

Aber diese Variable `w` ist nur ein Name, der auf eine Fensterobjekt *verweist*, er ist nicht das Objekt selbst (er handelt sich hierbei genaugenommen um einen *Zeiger* auf ein Objekt, obwohl sie damit im Gegensatz zu C/C++ keine Zeigerarithmetik machen können). Bei Java müssen alle Objekte dynamisch erzeugt werden. Dies geschieht nahezu immer mit dem Schlüsselwort `new`:

```
   Window w;
   w = new Window();
```

Hiermit haben wir eine *Instanz* der Klasse `Window` erzeugt und sie der Variable `c` zugewiesen. Solange dies noch nicht geschehen ist, hat `c` den Wert `null`, der anzeigt, dass noch keine Instanz der Variablen zugewiesen wurde. Sie können den Wert `null` jeder Variablen zuweisen und damit den Verweis auf eine Instanz ungültig machen; ebenso können sie eine Variable mit `null` vergleichen, um herauszufinden, ob bereits eine Instanz zugewiesen wurde.

Zugriff auf Felder

Nachdem wir nun ein Objekt erzeugt haben, können wir auf dessen Felder zugreifen. Die Syntax sollte C-Programmierern vertraut sein:

```
Window w = new Window();
w.iBreite = 300;
w.iHöhe   = 100;
```

```
Window w = new Window();
w.setBreite( 300 );
w.show();
```

Verwendung von Methoden

Die Verwendung von Methoden geschieht genauso, wie der Zugriff auf Felder:

Das interessante hierbei ist, dass wir nicht

```
show( w );
```

geschrieben haben, sondern

```
w.show();
```

Aus diesem Grund spricht man von “objektorientierter Programmierung”: das Objekt bildet den Mittelpunkt, nicht der Funktionsaufruf. Durch die Angabe des Objektes vor dem Methodennamen weiß die Methode immer, mit welchem Objekt sie arbeiten soll, d.h. `setBreite` setzt immer die Breite des angebenen Objektes und nicht irgendeines anderen.

`this`

Woher weiß die Methode, mit welcher Klasse sie arbeiten soll? Sie bekommt beim Aufruf einen impliziten Parameter `this` mitgegeben, der auf das Objekt verweist. Der Programmcode

```
void setBreite( int breite ) { iBreite = breite; }
```

lautet in Wirklichkeit

```
void setBreite( int breite ) { this.iBreite = breite; }
```

Der Kompiler nimmt diese Umsetzung automatisch vor, um dem Programmierer Schreibarbeit zu sparen. Wir können `this` aber verwenden, um den obigen Programmcode etwas eleganter zu schreiben:

```
void setBreite( int iBreite ) { this.iBreite = iBreite; }
```

Sehen sie? Der Name des Parameters ist identisch mit dem des Klassenfeldes. Parameter und lokale Variablen überdecken Klassenfelder, so dass man diese nicht mehr direkt verwenden kann. Mit Hilfe von `this` kann man auf verdeckte Klassenfelder zugreifen und braucht sich daher bei der Wahl der Parameternamen nicht unnötig viele Gedanken zu machen.

7.3 Übungen zu Klassen

1. Vervollständigen sie die Klasse `Window`, speichern sie das ganze in der Datei `Window.java` und kompilieren sie diese.
2. Versuchen sie die Klasse zu starten. Welche Fehlermeldung erscheint?

3. Warum funktioniert es nicht?

4. Schreiben sie analog zu `HelloWeb.java` eine Klasse `WindowTest.java`, die eine Instanz von `Window` erzeugt, die Fensterbreite setzt und dann die Fensterbreite aus der Instanz ausliest und ausgibt.

7.4 Objekterzeugung

Bei der Erzeugung einer Instanz schreibt man nach dem Namen der Klasse immer ein Paar runder Klammern - es sieht aus wie ein Methodenaufruf:

```
Window w = new Window();
```

In der Tat handelt es sich hierbei um den Aufruf einer speziellen Methode, dem *Konstruktor*. Jede Klasse in Java besitzt mindestens einen solchen Konstruktor; wird er nicht angegeben, so erhält sie einen Standardkonstruktor, der keine Argumente bekommt und nichts besonderes macht.

Einen Konstruktor definieren

Unsere Window-Klasse sollte bei der Erzeugung sinnvolle Startwerte haben. Beispiel 7.4.1 zeigt, wie eine solche Initialisierung aussehen kann.

```
class Window {
    // ...ursprünglicher Code...

    Window() {
        iXKoordinate = 0;
        iYKoordinate = 0;
        iBreite = iHöhe = 0;
        isVisible = false;
    }
}
```

Beispiel 7.4.1: ein Konstruktor für die Klasse `Window`

Der Konstruktor ist eine Methode mit dem Namen der Klasse und ohne Rückgabetyp. An dem Namen erkennt ihn der Kompiler und der Rückgabetyp ist implizit immer eine Instanz der Klasse selbst, darf deshalb *nicht* angegeben werden.

Mehrere Konstruktoren

Manchmal wollen sie ein Objekt auf mehrere Arten initialisieren, z.B. ein Fenster gleich mit den Koordinaten und der richtigen Größe. Aus diesem Grund können sie mehrere Konstruktoren angeben. Damit der Kompiler sie unterscheiden kann, müssen sie sie allerdings im Typ und/oder Anzahl der Parameter unterscheiden (Beispiel 7.4.2).

Die hier angewandte Technik nennt man *Überladung*. Sie können beliebige Methoden überladen, solange sich diese in Typ und/oder Anzahl der Parameter unterscheiden. Dies wird zum Beispiel bei den Dateizugriffsmethoden verwendet, bei denen man mit `write()` unterschiedliche Datentypen ausgeben kann.

Es reicht *nicht* aus, wenn sich überladenen Methoden im Typ des *Rückgabewertes* unterscheiden.

Wollen sie innerhalb eines Konstruktors einen anderen Konstruktor der gleichen Klasse aufrufen, so können sie das mit dem `this`-Schlüsselwort machen (siehe Beispiel 7.4.3). Dies muss allerdings als erster Befehl im Konstruktor stehen!

7.5 Übungen zur Überladung

1. Implementieren sie die vorgestellten Konstruktoren.
2. Erweitern sie `WindowTest.java` so, dass die neuen Konstruktoren verwendet werden.

```
class Window {
    // ...ursprünglicher Code...

    Window() {
        // ...
    }

    Window( int iX, int iY, int iHöhe, int iBreite ) {
        // ...
    }

    Window( int iX, int iY, int iHöhe, int iBreite, boolean sichtbar ) {
        // ...
    }
}
```

Beispiel 7.4.2: mehrere Konstruktoren für die Klasse `Window`

```
class Window {
    // ...

    Window( int iX, int iY, int iBreite, int iHöhe, boolean sichtbar ) {
        this( iX, iY, iBreite, iHöhe);
        isVisible = sichtbar;
    }
}
```

Beispiel 7.4.3: Verwendung von `this()` in einem Konstruktor

7.6 Statische Elemente

Unsere Klasse `Window` besitzt mehrere Felder, die Daten enthalten. Diese Daten "gehören der Instanz", d.h. jedes Objekt hat ihre eigenen Daten und eine Veränderung wirkt sich nur auf dieses aus, die anderen bleiben unverändert. Manchmal braucht man aber eine Variable, von der es nur eine Kopie gibt, die also mit der Klasse und nicht einem Objekt verbunden ist. Solche Felder heißen *statische Felder* oder *Klassenfelder*.

Klassenfelder

Um ein Feld als Klassenfeld zu definieren, schreibt man das Schlüsselwort `static` davor. Als Beispiel wollen wir zählen, wieviele Fenster bereits erzeugt wurden (siehe Beispiel 7.6.1).

```
class Window {
    static int iAnzahlFenster = 0;

    // ...

    Window() {
        iAnzahlFenster++;
    }

    // ...
}
```

Beispiel 7.6.1: Statisches Klassenfeld

In Java ist es erlaubt, Felder und Variablen mit Werten vorzubelegen, in diesem Fall mit `0`. Bei Klassenfeldern ist dies besonders wichtig, da man nie weiß, ob das Feld schon initialisiert wurde, oder nicht.

Ohne dass Schlüsselwort `static` würde das Feld `iAnzahlFenster` in jeder Instanz den Wert `1` enthalten, da jedesmal eine eigenständige Kopie davon erzeugt wurde. Da das Feld jetzt ein Klassenfeld ist, erhöht jeder Aufruf des Konstruktors das Feld um eins und zählt somit die Anzahl der erzeugten Fenster.

Um auf ein Klassenfeld zuzugreifen, brauchen sie keine Instanz der Klasse. Man kann also statt einem Zugriff über eine Instanz

```
a =  w.iAnzahlFenster;
```

auch über den Namen der Klasse zugreifen

```
a =  Window.iAnzahlFenster;
```

Hierbei ist es nicht notwendig, dass vorher bereits eine Instanz der Klasse erzeugt wurde.

Klassenmethoden

Genau wie Felder können sie auch Methoden als statisch definieren. Statische Methoden haben eine Einschränkung gegenüber normalen Methoden: sie dürfen nur statische Felder und andere statische Methoden der eigenen Klasse benutzen. Der Grund hierfür ist klar, die Methode "gehört zur Klasse", bekommt daher *keine* `this`-Variable mitgeliefert und kann deshalb nur über den Klassennamen zugreifen. Umgekehrt dürfen allerdings normale Methoden auf statische Methoden und Felder zugreifen.

Welchen Vorteil haben dann statische Methoden? Man benötigt keine Instanz, um sie aufrufen zu können! Ein Beispiel für statische Methoden ist die Sinus-Funktion in Java - diese ist statisch definiert und kann deshalb direkt verwendet werden:

```
double d =  Math.sin( 0 );
```

Dies ist auch der Grund dafür, dass die Methode `main` als `static` definiert wurde: nur so kann sie beim Programmstart direkt aufgerufen werden.

7.7 Übungen zu Klassenfeldern und -methoden

1. Erweitern sie `Window.java` so, dass alle Konstruktoren den Zähler erhöhen.
2. Erweitern sie `Window.java` so, dass sie auch direkt ausgeführt werden kann und das Funktionieren des Zählers überprüft.

7.8 Objektfreigabe

Nachdem sie jetzt gesehen haben, wie sie Objekte erzeugen und verwenden, ist die nächste Frage, wie diese Objekte wieder entfernt und der davon benötigte Speicherplatz wieder freigegeben wird.

Die Antwort lautet: Gar nicht! In Java werden nicht mehr benutzte Objekte vom *Garbage Collector* automatisch freigegeben. Der Garbage Collector läuft ständig mit niedriger Priorität im Hintergrund und durchsucht den Speicher nach nicht mehr verwendeten Objekten und gibt diese frei. Dies bedeutet, dass sie sich als Java-Programmierer nicht mehr mit Speicherlöchern (ein Programm braucht immer mehr Speicher, weil vergessen wurde ihn freizugeben) befassen müssen. Auch viele der Gründe für "Nicht behebbare Schutzverletzung" - freigegebener und danach trotzdem verwendeter Speicher - entfallen damit komplett. Das ganze frisst natürlich Rechenzeit, aber dafür laufen Java-Programme deutlich stabiler!

```
Window awFenster[] = new Window[ 10 ];
Window awFenster[] = { null, new Window(), null };
```

7.9 Arrays von Objekten

Array von Objekten unterscheiden sich nur wenig von Array mit integralen Datentypen.

Weil die Erzeugung eines Arrays nicht zu dessen Initialisierung führt, wird kein Konstruktor angegeben. Der Standardwert bei Objekten ist `null`.

Übungen zu Arrays

- Schreiben sie ein Programm, dass mehrere Fenster erzeugt und in einem Array ablegt. Weisen sie den Fenstern unterschiedliche Größen zu und lassen sie danach alle Fenstergrößen ausgeben.

7.10 Ausnahmen

Was passiert, wenn man die Arraygrenzen überschreitet, z.B. in einem Array von 10 Integern auf den 11. zugreift? In C wird der Zugriff ausgeführt, was gut gehen kann oder aber "nicht behebbare Schutzverletzungen" und "blaue Bildschirme" erzeugt. In Java wird jeder Zugriff auf ein Array zur Laufzeit auf seine Gültigkeit überprüft. Dies verlangsamt zwar die Ausführung eines Javaprogrammes, aber die Praxis zeigt, dass Programmierer viel Zeit auf der Suche nach solchen Fehlern verbringen und Programme selten frei davon sind.

Tritt in Java ein fehlerhafter Zugriff auf, so wird eine *Exception* (Ausnahme) erzeugt, die vom Programm abgefangen werden kann. Eine Exception ist eine Instanz der Klasse `Exception` (genaugenommen einer Unterklasse davon, aber dazu in Abschnitt 8 mehr), die vom Programm ausgewertet werden kann. Das Abfangen geschieht mit einem `try` - `catch`-Block, der beliebige Programmzeilen einschließen kann (siehe Beispiel 7.10.1).

```
try {
    int[] aiZahlen = { 1, 2, 3, 4 };
    int a = 0;
    while( true ) {
        System.out.println( aiZahlen[ a++ ] );
    }
} catch( Exception e ) {
    System.err.println( e.getMessage() );
    e.printStackTrace();
}
```

Beispiel 7.10.1: `try-catch`

Dieses Beispiel zeigt, wie man nicht programmieren sollte. Der Programmierer zählt den Index immer weiter hoch, ohne auf die Arraygrenze zu achten. Am Ende des Arrays tritt eine Ausnahmen auf, die abgefangen wird und danach läuft das Programm normal weiter. Man kann sich hiermit die Vergleiche sparen; allerdings braucht eine Ausnahme auch etwas Zeit, so dass sich dies erst bei großen Arrays lohnt.

Da es unterschiedliche Exceptions gibt, kann man in einem `try` - `catch`-Block auch beliebig viele `catch`-Blöcke verwenden. Die JVM nimmt immer den, der dazu passt.

`throw`

Natürlich sind Bereichsüberschreitungen nicht die einzigen Fehlerquellen - nicht gefundene Dateien, unzureichender Speicher, vergessene Variableninitialisierung usw. ärgern sowohl Programmierer als auch Anwender. Zwar liefern Funktionen in C Fehlercodes zurück, aber oft hält man sich an das Motto "never check for errorconditions that you cannot handle!" ("Prüfe nie auf Fehler, die man sowieso nicht behandeln kann!"), seit es aus Faulheit, oder weil man denkt, dass das niemals schiefgehen kann.

In Java hat man nicht die Wahl, ob man Fehlerzustände abfrägt und behandelt oder nicht - wenn eine Methode einen Fehler erzeugen kann, dann muss er abgefangen werden! Alle Lesemethoden eines `InputStream` können z.B. `IOException`s erzeugen - Lesefehler. Wenn sie eine solche Methode aufrufen, *müssen* sie die entsprechende Exception mit `try-catch` abfangen, oder der Kompiler wird ihr Programm nicht akzeptieren!

Wie gibt eine Methode jetzt an, dass sie einen Fehler erzeugen kann? Schreiben sie einfach das Schlüsselwort `throws` gefolgt von der Exception hinter den Namen der Klasse (siehe Beispiel 7.10.2). Wie man dort sieht erzeugt man die Exception mit dem Schlüsselwort `throw` gefolgt von einer Instanz der Exception.

```
public int liesInteger() throws IOException {
    // Methodencode
    if( fehler ) {
        throw new IOException( "Lesefehler bei..." );
    }
}
```

Beispiel 7.10.2: `throws`

`finally`

Zusätzlich oder alternativ zu einem `catch`-Block können sie auch einen `finally`-Block schreiben. Dieser Block wird immer ausgeführt, egal ob der Block normal beendet wurde oder durch `break`, `continue`, `return` oder einer Exception abgebrochen wurde. Er eignet sie deshalb ideal dafür “Aufräumarbeiten” am Ende einer Routine zu erledigen (siehe Beispiel 7.10.3).

```
try {
    // Programmcode
} finally {
    // Aufräumen
}
```

Beispiel 7.10.3: `finally`

7.11 Übungen zu Ausnahmen

1. Machen sie ihr Array-Programm sicher, indem sie auftretende Fehler abfangen.
2. Greifen sie absichtlich auf undefinierte Arrayelemente oder nicht mit `new` initialisierte Variablen zu.

7.12 Zusammenfassung

In diesem Kapitel haben sie gelernt

- wie sie Klassen definieren
- wie sie Instanzen erzeugen und benutzen
- wie sie Konstruktoren definieren
- wie sie Methoden überladen
- wie sie Klassenfelder und Klassenmethoden definieren
- wie sie Arrays erzeugen und verwenden
- wie sie Ausnahmen abfangen und erzeugen

Sie sind jetzt in der Lage eigene Klassen selbst zu definieren und mit diesen objektorientiert zu programmieren. In den weiteren Kapiteln werden sie ihre Kenntnisse vertiefen und weitere Prinzipien der objektorientierten Programmierung kennenlernen.

Kapitel **8**

Erbung, Pakete und Sichtbarkeit

In diesem Kapitel lernen sie

- wozu Erbung gut ist
- wie Variablen überdeckt werden
- wie Methoden überschrieben werden
- was Polymorphie ist
- wofür der `CLASSPATH` gut ist
- wie sie die Sichtbarkeit von Klassenelementen verändern
- wie sie finale Elemente definieren

8.1 Wozu Erbung?

Bei der Entwicklung von heutiger Software kann man auf die Arbeit von vielen Bibliotheken zurückgreifen, so dass niemand mehr "das Rad neu erfinden" muss. Häufig es so, das bestehende Programmroutinen erweitert und angepasst werden, um so schneller zum Ziel zu kommen.

Als Beispiel wollen wir unsere `Window`-Klasse so erweitern, dass sie einen Fenstertitel besitzt. Wir haben jetzt mehrere Möglichkeiten

1. wir verändern die `Window`-Klasse, indem wir die neuen Fähigkeiten hinzuschreiben
2. wir kopieren die Klasse in eine neue Datei und verändern diese neue Datei

Beide Möglichkeiten haben Vor- und Nachteile. Bei der ersten Möglichkeit blähen wir den Code auf, so dass wir mehr Speicher brauchen, als eigentlich notwendig; bei der zweiten Möglichkeit müssen wir alle Fehlerkorrekturen an der Originalklasse in die Kopie übertragen. Durch *Erbung* können wir diese Nachteile vermeiden und den Code sowohl einfach als auch erweiterbar halten.

Durch einen Übersetzungsfehler wird in anderer Literatur meistens von *Vererbung* gesprochen. Es ist allerdings so, dass etwas von einer anderen Klasse geerbt wird und nicht diese einen beerbt. Der Erbe ist also der Aktive - wie im richtigen Leben...

8.2 Unterklassen

Wir wollen also unsere Window-Klasse um einen Fenstertitel erweitern. Der dazu notwendige Code sieht z.B. so aus (Beispiel 8.2.1).

```
class TitleWindow extends Window {
    String sTitel = "";

    void setTitle( String s ) {
        sTitel = s;
    }
}
```

Beispiel 8.2.1: Erweiterung der Klasse `Window`

Wie sie sehen, steht dort das Schlüsselwort `extends`, das den Namen der Klasse angibt, die erweitert wird. Die Klasse `TitleWindow` heißt *Unterklasse* der Klasse `Window`, die man als *Oberklasse* bezeichnet. Von dieser Klasse werden alle Felder und Methoden geerbt, d.h. wir können jetzt schreiben (siehe Beispiel 8.2.2).

```
TitleWindow tw = new TitleWindow();
tw.setTitle( "Mein Fenster" );
```

Beispiel 8.2.2: Benutzung der Klasse `TitleWindow`

Konstruktoren

Sie können alle Methoden und Felder der Originalklasse verwenden, ohne diese neu implementieren zu müssen. Wenn sie die Konstruktoren der Oberklasse verwenden wollen, brauchen sie allerdings eine neue Syntax, die der Verwendung von `this` (siehe Beispiel 7.4.3) in Konstruktoren sehr ähnlich ist: `super` (siehe Beispiel 8.2.3).

```
TitleWindow() {
    super();
}

TitleWindow( String s, int iX, int iY, int iBreite, int iHöhe ) {
    super( iX, iY, iBreite, iHöhe );
    sTitel = s;
}
```

Beispiel 8.2.3: Verwendung von `super` in Konstruktoren

Auch hier muss `super` in der ersten Zeile stehen!

8.3 Überdeckung und Überschreibung

Sollten sie in einer Unterklasse neue Felder oder Methoden mit dem gleichen Namen einführen, so sind diese nicht mehr direkt sichtbar. Das Verhalten ist bei Feldern und Methoden unterschiedlich, so dass diese hier getrennt behandelt werden.

Überdeckung von Feldern

Felder einer Unterklasse, die den gleichen Namen wir Felder der Oberklasse haben, überdecken diesen. Hierbei ist es nicht notwendig, dass der Typ des Feldes identisch ist. Um auf das überdeckte Feld der Oberklasse zuzugreifen, müssen sie das Schlüsselwort `super` verwenden:

```
a = super.iBreite;
```

Sie sollten es möglichst vermeiden, Felder zu überschreiben, da ihre Programme dadurch unübersichtlich werden.

Überschreibung von Methoden

Im Gegensatz zur Überdeckung von Feldern macht das Überschreiben von Methoden einen Sinn und wird sogar sehr häufig verwendet. Um eine Methode zu überschreiben, definieren sie diese in einer Unterklasse genauso, wie in ihrer Oberklasse. Um auf die Methode der Oberklasse zuzugreifen, müssen sie ebenfalls das Schlüsselwort `super` verwenden:

```
a = super.getBreite();
```

Wo liegt jetzt der Vorteil einer überschriebenen Methode?

8.4 Polymorphie

Sie können Instanzen von Unterklassen an Variablen vom Typ der Oberklasse zuweisen, also

```
Window w = new TitleWindow();
```

Obwohl sie jetzt eine Variable vom Typ `Window` haben, "weiß" das Objekt, dass es in Wirklichkeit ein `TitleWindow` ist. Wenn sie jetzt eine überschriebene Methode auf diesem `Window`-Objekt aufrufen, dann wird nicht die Methode aus `Window`, sondern die aus `TitleWindow` aufgerufen! Dieses Verhalten nennt man *Polymorphie* und erleichtert die Programmierung von grafischen Oberflächen sehr stark.

Stellen sie sich vor, sie wollen ein Programm schreiben, das viele unterschiedliche Arten von Fenstern verwaltet, welche mit Titel, welche ohne Titel, mit Scrollbalken, fester Größe usw. Jedes dieser Fenster besitzt eine eigene Klasse, die jeweils von `Window` abgeleitet ist und jede Klasse besitzt die Methode `show`. Alle Fenster verwalten sie in einer Liste vom Typ `Window`. Um jetzt alle Fenster auf dem Bildschirm darzustellen, braucht ihr Programm nur bei jedem Fenster in der Liste die Methode `show` aufzurufen, ohne zu wissen, um welche Art von Fenster es sich handelt; die richtigen Methoden werden automatisch von der JVM herausgesucht und aufgerufen. Selbst wenn nach der Fertigstellung des Programmes neue Fenstertypen hinzugefügt werden, brauchen sie an ihrem Programmcode nichts zu ändern!

Casting

Wenn man auf die erweiterten Methoden einer Unterklasse zugreifen will, die zuvor an eine Variable vom Typ der Oberklasse zugewiesen wurde, muss man die Instanz *Casten*, d.h. umwandeln:

```
Window w = new TitleWindow();
TitleWindow tw = (TitleWindow) w;
```

Man schreibt den Namen der gewünschten Klasse in Klammern vor die Variable, die diese (hoffentlich) enthält. Ist die gewünschte Klasse nicht in der Oberklasse enthalten, so tritt ein Laufzeitfehler auf.

`instanceof`

Um solche Laufzeitfehler zu vermeiden, sollte man herausfinden, ob eine Umwandlung überhaupt möglich ist. Hierfür gibt es in Java den `instanceof`-Operator. Er liefert `true` oder `false` zurück, jenachdem ob die angegebene Variable eine Instanz der angegebenen Klasse, bzw. einer Unterklasse davon, enthält.

```
if( w instanceof TitleWindow ) {
  TitleWindow tw = (TitleWindow) w;
  // ...
}
```

8.5 Übungen zur Polymorphie

1. Fügen sie in `Window` und `TitleWindow` die Methode `show()` ein, die den Klassennamen als Text ausgibt.
2. Erzeugen sie in `WindowTest.java` jeweils eine Instanz der beiden Klassen und weisen sie diese an Variablen vom Typ `Window` zu.
3. Rufen sie auf beiden die Methode `show()` auf.

4. Versuchen sie beide Instanzen nach `TitleWindow` zu casten. Welche Fehlermeldung tritt auf?

8.6 Packages und der CLASSPATH

Im Gegensatz zu "herkömmlichen" Programmen besteht ein Java-Programm nicht aus einer "EXE-Datei", sondern aus einer Sammlung von `.class`-Dateien. Ab einer gewissen Anzahl ist man geneigt, diese Dateien in Unterverzeichnisse zu ordnen, damit das ganze nicht unübersichtlich wird. Damit die JVM die Klassen in den Unterverzeichnissen findet, muss man angeben, zu welchem Verzeichnis eine Klasse gehört, sowohl bei der Definition, wie auch beim Aufruf.

Wir nehmen jetzt einmal an, dass die Klassen `Window` und `TitleWindow` ins Unterverzeichnis `gui\window` sollen, während `WindowTest` im Hauptverzeichnis bleibt, also

```
WindowTest.java
gui\window\Window.java
gui\window\TitleWindow.java
```

`package` und `import`

Um dem Kompiler die neuen Postitionen mitzuteilen, muss in der ersten Zeile von `Window.java` und `TitleWindow.java` folgendes stehen:

```
package gui.window;
```

Die Klasse `WindowTest` muss jetzt natürlich auch geändert werden, damit die Klassen an der neuen Position gefunden werden: jedes Auftreten von `Window` muss in `gui.window.Window` geändert werden, ebenso jedes Auftreten von `TitleWindow`. Da dies sehr viel Tipparbeit ist, kann man mit dem Schlüsselwort `import` Verzeichnisse bzw. Pakete angeben, auf deren Klassen wie auf lokale Klassen zugegriffen werden soll:

```
import gui.window;
```

Jetzt kann man die alte Schreibweise beibehalten, obwohl sich die Klassen in einem Unterverzeichnis befinden.

`import` erlaubt nur die Abkürzung der Klassennamen. Im Gegensatz zu `include` aus C/C++ wird kein Code eingelesen, Variablen definiert o.ä.!

Der CLASSPATH

Da wir jetzt mit Unterverzeichnissen arbeiten, ist es nicht mehr egal, in welchem Verzeichnis wir uns beim Starten der JVM befinden. Für "normale" ausführbare Dateien gibt es den `PATH`, der Verzeichnisse angibt, die auf der Suche nach einem Programm durchforscht werden sollen. Nur deshalb kann man in jedem Verzeichnis Befehle benutzen, die sich in anderen Verzeichnissen befinden.

Das Java-Äquivalent dazu ist der `CLASSPATH`, der den Pfad zu den Klassen angibt. Deshalb mussten sie im Kapitel 2 den `CLASSPATH` entsprechend setzen. Wie sie im Kapitel 13.3 noch erfahren werden, können sie auch komprimierte ZIP und JAR-Dateien in den `CLASSPATH` aufnehmen, aus denen heraus Java-Programme laufen können.

Der `CLASSPATH` wird unter Windows und Unix unterschiedlich definiert. Unter Windows ist der Backslash (\) das Pfadtrennzeichen und mit dem Semikolon (;) werden einzelne Pfade getrennt. Unter Unix wird dafür der Forwardslash (/), sowie der Doppelpunkt (:) verwendet. Der Grund hierfür ist, dass sich Java an der Definition des `PATH` für ausführbare Dateien orientiert, der unter Windows anders als unter Unix definiert wird.

8.7 Sichtbarkeit

Bis jetzt war es immer so, dass man auf alle Felder und Methoden einer Klasse von aussen zugreifen konnte. Normalerweise ist dies nicht erwünscht, z.B. braucht `TitleWindow` keinen Zugriff auf `iAnzahlFenster`. Ausserdem soll niemand von aussen auf die Felder direkt zugreifen dürfen, sondern nur auf die entsprechenden Methoden, damit die Werte auf Fehler (z.B. negative Breite) geprüft werden können.

Aus diesem Grund gibt es die Schlüsselwörter `public`, `protected` und `private`, die die Zugriffsmöglichkeiten *anderer Klassen* einschränken und einfach vor die entsprechende Methode oder Feld geschrieben werden.

public

Diese Zugriffsmethode bedeutet, dass von aussen ohne Einschränkung auf die entsprechende Methode oder Feld zugegriffen werden darf.

protected

Nur Unterklassen und Klassen im gleichen Verzeichnis dürfen darauf zugreifen.

private

Niemand ausser der eigenen Klasse darf darauf zugreifen.

“package”

Wird keines der obigen Schlüsselwörter angeben, so wird die Standardsichtbarkeit verwendet, die nur Zugriffe von Klassen innerhalb des aktuellen Verzeichnisses (*package*) erlaubt.

Damit eine Klasse von aussen sichtbar ist, müssen sie diese auch als `public` definieren, ansonsten ist sie nur im eigenen Verzeichnis sichtbar.

Beispiel

Die Definition

```
private int iBreite;
```

erlaubt nur der eigenen Klasse den Zugriff auf `iBreite`.

8.8 Übungen zur Sichtbarkeit

1. Nach dem Verschieben in die Unterverzeichnisse lassen sich ihre Klassen nicht mehr kompilieren oder ausführen. Warum?

2. Machen sie die Klassen wieder lauffähig und verstecken sie alle nicht notwendigen Methoden und Felder.

3. Zwei Klassen in unterschiedlichen Verzeichnissen haben den gleichen Namen. Wie können sie Instanzen von beiden innerhalb einer Klasse erzeugen, ohne Namenskonflikte zu bekommen?

4. Welchen Grund kann es dafür geben, dass die Sichtbarkeit einer überschriebenen Methode in einer Unterklasse nicht eingeschränkter sein darf, als die der Methode in der Oberklasse? (Hinweis: Polymorphie)

8.9 Finale Elemente

"Manche Dinge ändern sich nie!" - in Java haben sie die Möglichkeit, solche Tatsachen festzuschreiben. Wenn sie das Schlüsselwort `final` vor ein Feld oder eine Klasse schreiben, dann darf dieses nicht mehr verändert werden.

Finale Felder

Ein als `final` definiertes Feld muss bei der Initialisierung mit einem Wert belegt werden, der dann nicht mehr verändert werden kann, entspricht also dem einer mit `const` markierten Variablen in C.

```
   final int iPort = 4444;
```

Finale Methoden

Auch Methoden können als `final` definiert werden - diese können dann in Unterklassen nicht mehr überschrieben werden. Ein Vorteil hiervon ist, dass die JVM finale Methoden schneller aufrufen kann, da sie nicht überprüfen muss, ob sie in einer Unterklasse überschrieben wurde.

8.10 Abstrakte Methoden und Klassen

Nicht immer will oder kann man eine Implementierung einer Methode direkt angeben. Ein Beispiel hierfür wäre eine `Window`-Klasse, die alle Grundeigenschaften eines Fensters (Höhe, Breite usw.) definiert und die Oberklasse für alle Arten von Fenstern ist, selbst aber nicht am Bildschirm dargestellt werden kann, weil sie zu allgemein ist.

Diese Klasse kann die Methode `show` nicht sinnvoll implementieren, zugleich soll sie aber vorhanden sein, damit man durch Überschreibung und Polymorphie jede Art von Fenster als `Window` verwalten und darstellen kann.

Statt jetzt bei `Window` eine unsinnige Methode anzugeben, läßt man diese einfach weg und definiert die Methode als `abstract`.

```
public abstract class Window {
    // ...

    public abstract void show();
}
```

Man sieht, dass hier statt den geschweiften Klammern nur ein Strichpunkt angegeben ist. Da man die Methode `show` auf einer Instanz von Window aufrufen kann, muss man die ganze Klasse als

`abstract` definieren. Jede Klasse, die mindestens eine abstrakte Methode besitzt, muss selbst als abstrakt definiert werden!

Eine abstrakte Klassen kann nicht mehr instanziiert werden, d.h.

```
Window w = new Window();   // geht nicht!
```

ergibt eine Fehlermeldung beim Kompilieren. Implementiert eine Unterklasse alle abstrakten Methoden, kann mal allerdings schreiben

```
Window w = new TitleWindow();
```

Hierdurch kann man alle Vorteile der Polymorphie verwenden und verhindert gleichzeitig, dass ein Programmierer eine allgemeine Oberklasse instanziiert und dann zur Laufzeit einen Fehler bekommt.

8.11 Interfaces

Sind alle Methoden einer Klasse abstrakt, so kann man die Klasse auch als *Interface* (Schnittstelle) definieren

```
public interface Window {
    // ...

    public abstract void show();
}
```

Beispiel 8.11.1: `Window` als Interface

Ein Interface beschreibt alle Methoden und Felder, die eine Unterklasse besitzt, ohne dafür eine Implementation zu liefern. Ein Beispiel hierfür ist die Klassse `java.io.DataInput`, die allgemeine Methoden zur Dateneingabe definiert.

Eine Klasse kann sagen, dass sie ein Interface implementiert, also alle Methoden des Interfaces zur Verfügung stellt.

```
public class TitleWindow implements Window {
    // ...
}
```

Beispiel 8.11.2: `TitleWindow` implementiert das Interface `Window`

Hiermit wird sichergestellt, dass alle Methoden von `Window` von der Klasse `TitleWindow` zur Verfügung gestellt werden; sollte eine Methode vergessen werden, gibt der Kompiler eine Fehlermeldung aus.

Eine Klasse kann beliebig viele Interfaces implementieren; diese werden dann mit Komma getrennt hinter dem Schlüsselwort `implements` angegeben.

Anwendungen

Sie können ein Interface als Parametertyp für eine Methode angeben. Auf diese Weise stellen sie sicher, dass übergebene Objekte bestimmte Eigenschaften haben müssen (eine bestimmte Schnittstelle aufweisen), die sie in ihrer Methode benötigen.

8.12 Mehrfacherbung

In Java kann eine Klasse nur eine Oberklasse haben. Wollen sie weiterem mehreren Klassen ableiten, so müssen diese Interfaces sein, die ihre Klasse implementiert.

```
public class TitleWindow extends SuperWindow
                         implements Window, Serializable {
    // ...
}
```

8.13 Übungen

1. Was müssen sie machen, um die Methoden einer abstrakten Klasse verwenden zu können?

8.14 Zusammenfassung

In diesem Kapitel haben sie gelernt

- wozu Erbung gut ist
- wie Variablen überdeckt werden
- wie Methoden überschrieben werden
- was Polymorphie ist
- wofür der `CLASSPATH` gut ist
- wie sie die Sichtbarkeit von Klassenelementen verändern
- wie sie finale Elemente definieren
- was abstrakte Methoden und Klassen sind

Sie sind jetzt in der Lage, bestehende Klassen weiterzuverwenden und ihre Projekte auf Unterverzeichnisse aufzuteilen. Klassenfelder können sie vor Zugriffen von “aussen” schützen und so die Gültigkeit der Daten sicherstellen.

Kapitel 9
Innere Klassen

In diesem Kapitel lernen sie

- welche Arten innerer Klassen es gibt
- wie sie innere Klassen definieren
- wofür sie innere Klassen einsetzen können

9.1 Was sind innere Klassen?

Ab Java 1.1 ist es erlaubt, Klassen innerhalb von Klassen zu definieren. Damit ist es möglich, Klassen noch weiter zu unterteilen, als es Packages erlauben und private Klassen nur dort zu definieren, wo sie gebraucht werden. Alle Klassen, die wir bis jetzt kennengelernt haben, bezeichnet man als “Toplevel-Klasse”.

Eine spezielle inneren Klasse, die “anonyme innere Klasse”, passt sehr gut zum neuen Event-Modell, das mit AWT 1.1 eingeführt wurde. Es erlaubt die einfache Implementierung eines Callback-Prinzips, obwohl es in Java keine Zeiger auf Methoden gibt.

Alle inneren Klassen dürfen (mit kleinen Einschränkungen) auf Variablen und Methoden der sie enthaltenden Klasse zugreifen.

9.2 Elementklassen

Elementklassen sind innere Klassen, die innerhalb des Klassenkontextes, d.h. so wie eine Methode, definiert werden:

```
public class Aussen {
    public class Innen {
        // ...
    }

    // ...
}
```

Elementklassen sind von aussen über den Namen der umschließenden Klasse erreichbar, also in diesem Fall als `Aussen.Innen`.

Eine Hauptanwendung von Elementklassen ist die Kapselung von Daten, die z.B. in einen `Vector` geschrieben werden sollen. `Vector` erlaubt nur die Angabe eines Objektes, also muss man mehrere Objekte zu einem zusammenfassen, will aber nicht immer eine eigene Toplevel-Klasse dafür anlegen.

9.3 Lokale Klassen

Lokale Klassen werden innerhalb eines Codeblocks, d.h. wie eine lokale Variable, definiert:

Lokale Klassen können, wie Elementklassen auch, auf Variablen und Methoden der umschließenden Klasse zugreifen. Zusätzlich können sie auch auf Variablen und Parameter der umschließenden Methode zugreifen, die als `final` deklariert sind.

```
public class Aussen {
    public void methode() {
         public class Innen {
             // ...
         }
    }
}
```

```
public class Aussen {
    public void methode( int nicht_zugreifbar, final int zugreifbar ) {
        int auch_nicht_zugreifbar;
        final int auch_zugreifbar = 1;
        public class Innen {
            // ...
        }
    }
}
```

Dies ist eine Einschränkung, die in der Praxis manchmal sehr störend sein kann, wenn der Wert der Variablen erst berechnet werden muss. Hier kann sich helfen, indem man den Wert zuerst berechnet und dann einer finalen Variable zuweist:

```
public class Aussen {
    public void methode() {
        int nicht_zugreifbar;

        // Berechnung von 'nicht_zugreifbar'

        final int zugreifbar = nicht_zugreifbar;

        public class Innen {
            // ...
        }
    }
}
```

9.4 Anonyme innere Klassen

Anonyme innere Klassen sind diejenigen, die in der Praxis am häufigsten verwendet werden. Betrachten wir folgendes Beispiel:

```
public class Aussen {
    public void methode( int nicht_zugreifbar, final int zugreifbar ) {
        public class Innen extends AndereKlasse {
            public void andereMethode() {
                // überschreibender Code
            }
        }

        weitereMethode( new Innen() );
    }
}
```

Hier wird die Klasse `AndereKlasse` zur Klasse `Innne` erweitert und dabei die Methode `andereMethode` überschrieben. Diese neue Klasse wird instatiiert und als Argument an `weitereMethode` übergeben.

Mit einer anonymen inneren Klasse lässt sich dieser Programmcode vereinfachen:

```
public class Aussen {
    public void methode( int nicht_zugreifbar, final int zugreifbar ) {
        weitereMethode( new AndereKlasse {
            public void andereMethode() {
                // überschreibender Code
            }
        } );
    }
}
```

Wie man sieht, bekommt die innere Klassen keinen Namen zugewiesen, weshalb sie “anonym” genannt wird. Dies hat den Vorteil, dass man sich für solche Klassen keinen Namen auszudenken braucht.

Anwendung anonymer Klassen

Wofür braucht man anonyme Klassen? Wie oben bereits erwähnt, erlaubt Java keine Zeiger auf Methoden, sondern nur Instanzen von Klassen. Damit steht man vor dem Problem, wie man einen “Callback” realisieren soll.

Callback bedeutet, dass man sich bei einer anderen Klasse “registriert” und bei bestimmten Ereignissen benachrichtigt wird. In C/C++ wird hierzu ein Zeiger auf die Methode übergeben, die aufgerufen werden soll, wenn das Ereignis eintritt.

Da dies in Java nicht möglich ist, muss man für jeden Callback eine eigene Klasse definieren, diese instanziieren und an die andere Klasse übergeben. Mit anonymen inneren Klassen ist dies sehr einfach möglich, wie man an dem folgenden Beispiel sieht. Wir greifen hierbei auf das Kapitel 10 vor, in dem die Grafikbibliothek von Java beschreiben wird.

Es geht darum, einen Knopf zu realisieren, der das Java-Programm beendet, wenn er gedrückt wird:

```
        Button butBeenden = new Button( "Beenden" );

        butBeenden.addActionListener( new ActionListener() {
            public void actionPerformed( ActionEvent e ) {
                System.exit();
            }
        } );
```

Per Konvention der Bibliothek wird bei einem Druck auf den Knopf die Methode `actionPerformed` aller registrierten `ActionListener` aufgerufen. Mit einer anonymen inneren Klasse kann man sehr leicht Programmcode angeben, der bein Druck auf den Knopf ausgeführt werden soll.

9.5 Neue Syntax für innere Klassen

Mit den inneren Klassen wurde gleichzeitig die Syntax von `this`, `new` und `super` erweitert, um Mehrdeutigkeiten zu vermeiden. Mehrdeutigkeiten können auftreten, wenn umschließende und umschlossene Klasse die gleichlautende Methoden oder Variablen besitzen.

```
public class Aussen {
    int iMenge;

    private class Innen {
        int iMenge;

        int getMengeInnen() {
            int iMenge;  // das wollen wir nicht
            return this.iMenge;
        }

        int getMengeAussen() {
            return Aussen.this.iMenge;
        }
    }
}
```

Neue `this`-Syntax

`this` erlaubt es anzugeben, dass man nicht auf eine lokale Variable, sondern auf die gleichlautende Variable im Klassenkontext zugreifen will. Will man jetzt auf eine gleichlautende Variable in der umschließenden Klasse zugreifen, muss man die neue Syntax benutzen:

Vor das `this` wird einfach der Name der umschließenden Klasse geschrieben, auf deren Variable man zugreifen will.

Neue `new`-Syntax

Der `new`-Befehl erlaubt jetzt die Angabe der umschließenden Instanz:

```
        Aussen a = new Aussen();
        Aussen.Innen ai = a.new Innen();
```

Dies erzeugt eine Instanz `ai` von `Innen` innerhalb der Instanz `a` von `Aussen`.

Neue `super`-Syntax

Das gleiche gibt es auch für den Aufruf von Konstruktoren. Auch hier kann man die umschließende Instanz angeben:

```
        public Innen( Aussen a ) {
            a.super();
        }
```

9.6 Übungen

1. Schreiben sie ein Programm, das in einem Array Namen und Altern von Personen verwaltet. Verwenden sie zur Speicherung eine innere Klasse

Kapitel 10
AWT

Das *AWT* (Abstract Windowing Toolkit) ist eine Klassenbibliothek, mit der unter Java grafische Benutzeroberflächen (GUI) erzeugt werden. Sie erlaubt das Erzeugen von Fenstern, Knöpfen, Eingabefeldern usw. (dies bezeichnet man als *Widgets*), sowie die Beschreibung der Aktionen (*Events*), die durch das Verwenden der Widgets ausgelöst werden.

Die Widgets sind hierbei der jeweiligen Plattform angepasst (*Look & Feel*), unter der die JVM läuft, d.h. unter MS-Windows hat man Windows-Knöpfe, unter Linux bekommt man Motif-Knöpfe und unter Macintosh eben Macintosh-Knöpfe - jeder Benutzer fühlt sich direkt "heimisch".

10.1 Layout-Manager

Unter AWT werden grafische Elemente anderen zugewiesen, um so eine Gliederung zu erreichen. Zum Beispiel kann ein `Frame` (Fenster) einen `Button` (Knopf) und ein `Panel` (Fläche) enthalten, und das `Panel` kann seinerseits wieder einen oder mehrere `Buttons` enthalten. Das Ergebnis ist eine baumartige Struktur, die das Verhältnis "Was ist worin enthalten" der Komponenten untereinander angibt. Alle Komponenten hierbei Unterklassen von `Component`.

Damit allein ist es nicht getan, denn für eine Darstellung am Bildschirm muss das AWT noch wissen, wie diese Elemente angeordnet werden sollen. Diese Aufgabe übernehmen die *Layout-Manager*. Es gibt mehrere Layout-Manager, die unterschiedliche Anordnungen zur Verfügung stellen.

Die Layout-Manager passen sich dynamisch der Fenster- und Widgetgröße an, so dass man sich um die Größe der Fenster keine Gedanken zu machen braucht; mit `pack()` wird das Fenster auf die minimal notwendige Größe skaliert. Der Benutzer zieht auch einen Vorteil daraus, weil er alle Fenster, z.B. einen Dateidialog, so groß machen kann, wie er ihn braucht und nicht damit leben muss, was der Programmierer als sinnvoll erachtet hat.

Jeder Fläche kann man einen eigenen Layout-Manager zuordnen:

```
Frame f = new Frame( "Mein Fenster" );
f.setLayout( new FlowLayout() );
```

FlowLayout

Das `FlowLayout` kann man als "zentriert mit Zeilenumbruch" beschreiben.

Ein `Label` (Text) wird immer als ganzes aufgefasst, d.h. nicht umbrochen.

Das Hinzufügen geschieht mit `add( Component c )`:

GridLayout

Das `GridLayout` ordnet die Elemente in einem Gitter mit gleich großen Zellen an. Beim Erzeugen des `GridLayouts` muss man die Anzahl der Reihen und Spalten des Gitters angeben; einzelnen Komponenten werden mit `add( Component c )` hinzugefügt.

Abbildung 10.1: FlowLayout

```
Frame f = new Frame( "Mein FlowLayout" );
f.setLayout( new FlowLayout() );
f.add( new Label( "Sie haben neue eMail" ) );
f.add( new Button( "OK" ) );
f.add( new Button( "Abbrechen" ) );
f.pack();
f.show();
```

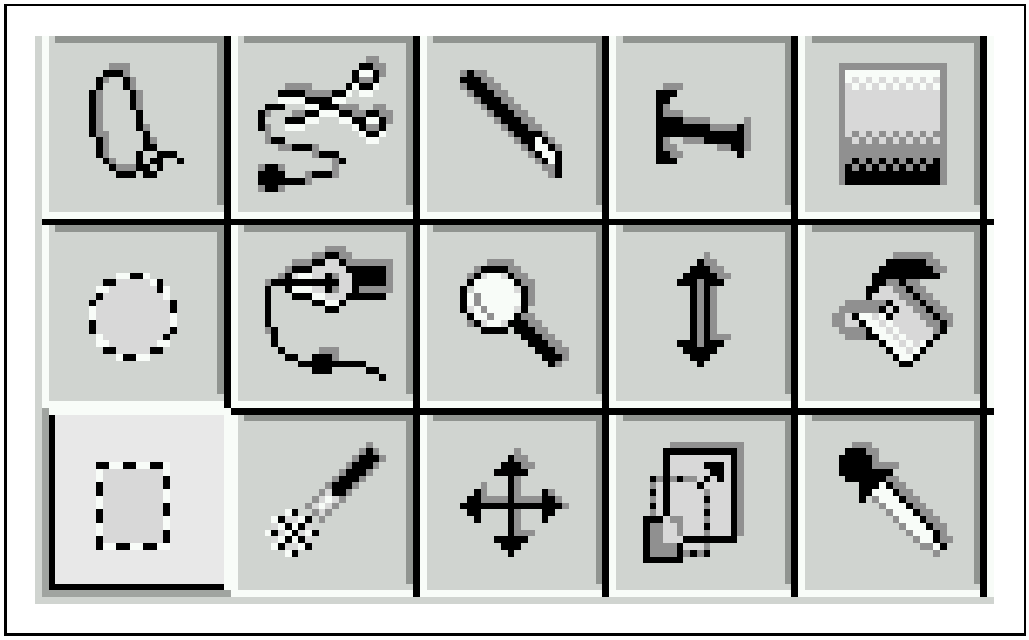

Abbildung 10.2: GridLayout

BorderLayout

Das `BorderLayout` unterteilt eine Fläche in 5 Bereiche:

- “North” (Norden, oben)
- “South” (Süden, unten)
- “East” (Osten, rechts)
- “West” (Westen, links)
- “Center” (Zentrum, mittig)

```
Frame f = new Frame( "Mein GridLayout" );
f.setLayout( new GridLayout( 3, 2 ) );
f.add( new Button( "1" ) );
f.add( new Button( "2" ) );
f.add( new Button( "3" ) );
f.add( new Button( "4" ) );
f.add( new Button( "OK" ) );
f.add( new Button( "Abbrechen" ) );
f.pack();
f.show();
```

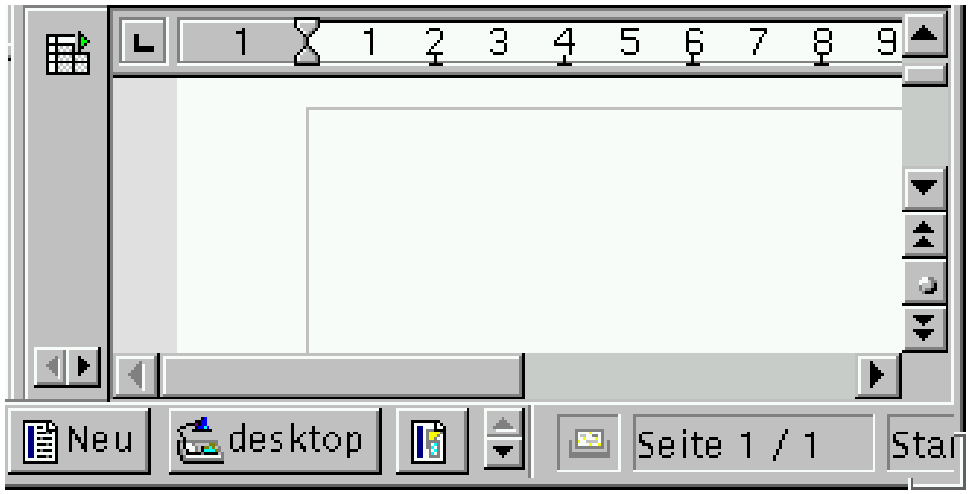

Abbildung 10.3: BorderLayout

Die Bereiche “North” und “South” haben eine konstante Höhe und passen ihre Breite der Fenstergröße an. Die Bereiche “East” und “West” haben eine konstante Breite und passen ihre Höhe der Fenstergröße an. Der Bereich “Center” füllt den Rest in der Mitte aus.

Das Hinzufügen geschieht mit `add( String s, Component c )`:

```
Frame f = new Frame( "Mein BorderLayout" );
f.setLayout( new BorderLayout() );
f.add( "North", new Button( "OK" ) );
f.add( "South", new Button( "Abbrechen" ) );
f.pack();
f.show();
```

Wie man sieht, muss nicht jeder dieser Bereiche ausgefüllt werden.

CardLayout

Das `CardLayout` erlaubt beliebig viele “Karten”, von denen nur jeweils eine sichtbar ist. Wenn die Karten definiert sind, kann man zwischen ihnen vorwärts und rückwärts blättern, sowie einzelne Karten gezielt anwählen.

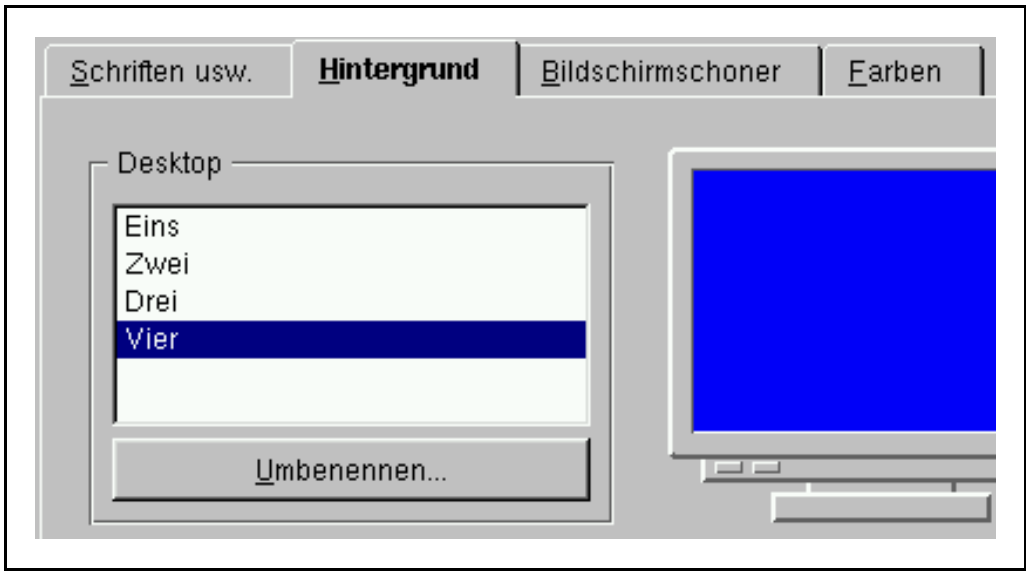

Abbildung 10.4: CardLayout

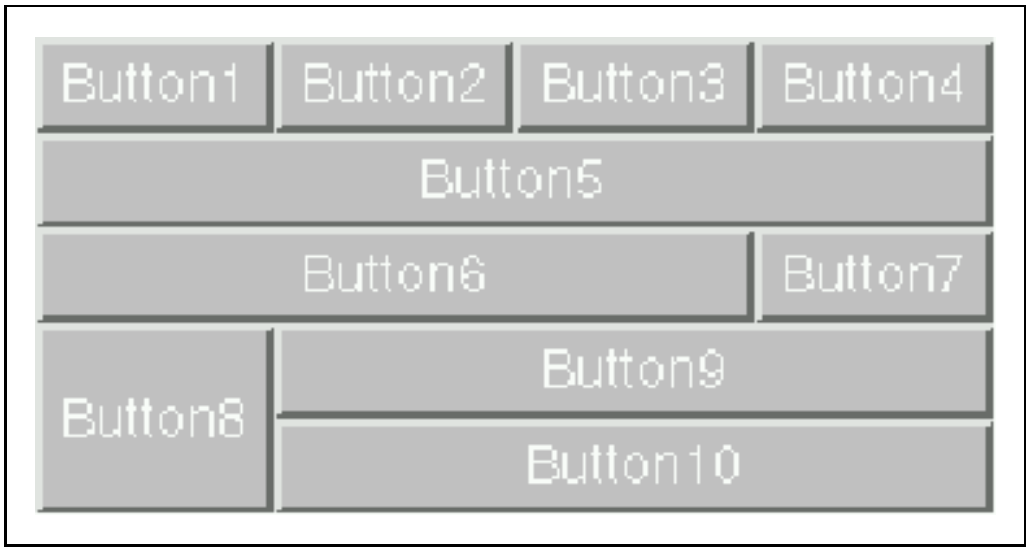

Abbildung 10.5: GridBagLayout

GridBagLayout

Das `GridBagLayout` ist das mächtigste und komplizierteste Layout des AWTs (wahrscheinlich deshalb haben die ersten Java-Entwicklungsumgebungen dieses Layout nicht unterstützt). Es erlaubt die Anordnung der Komponenten innerhalb eines Gitters, wobei die einzelnen Zellen unterschiedlich groß sein dürfen und eine Komponente auch über mehrere Zellen gehen kann.

Jeder Komponenten wird hierbei ein `GridBagConstraints`-Objekt mitgegeben, das alle Layouteigenschaften genau definiert.

Vorteile von Layout-Managern

Ein Java Programmierer muss davon ausgehen, dass seine Programme auf einer Vielzahl von Betriebssystemen und Benutzeroberflächen laufen werden. Er kann sich nicht darauf verlassen, dass sein Programm nur unter Microsoft Windows läuft, sondern eventuell auf einem PalmPilot oder gar einer textbasierten Oberfläche.

Von daher sollte man den von manchen Entwicklungsumgebung zur Verfügung gestellten Layout-Manager `AbsoluteLayout` tunlichst vermeiden. Er erlaubt es, die Positionen aller

Elemente als Pixelpositionen genau anzugeben. Das Ergebnis ist, dass man Fenster nicht mehr beliebig skalieren kann und die Oberfläche auf anderen Plattformen als der des Entwicklers sehr schlecht aussieht und oftmals unbenutzbar wird.

Mit den Standard-AWT-LayoutManagern hat man diese Nachteile nicht und die Benutzer erhalten immer ein optimales Bild.

Swing

Ab Java 2 wird als Alternative zum AWT die Bibliothek *Swing* ausgeliefert. Sie enthält noch mehr Widgets und neue Layout-Manager. Das Look & Feel ist hierbei frei wählbar, d.h. unter MS-Windows kann man auch Widgets im Motif-Aussehen darstellen.

Übungen

Schreiben sie ein Programm, dass eine Frage am Bildschirm darstellt und zwei Knöpfe “Ja” und “Nein” anbietet.

10.2 Ereignisse (Events)

Eine grafische Oberfläche muss auf Aktionen des Benutzers, z.B. der Druck auf einen Knopf, reagieren. Hierzu löst jede Komponente ein Ereignis aus, das vom Programm abgefangen und verarbeitet werden kann.

Die Ereignisbehandlung hat sich mit dem Wechsel von Java 1.0 auf Java 1.1 deutlich verändert, so dass wir sie hier getrennt behandeln werden. Neue Programme sollte man immer für AWT 1.1 erstellen, allerdings kennen viele Browser im Internet nur AWT 1.0, so dass man für Internetanwendungen noch nach dem alten Schema programmieren muss.

AWT 1.0

Jede Komponente besitzt eine `action()`-Methode, die aufgerufen wird, wenn etwas mit der Komponente gemacht wurde. Als Paramter erhält sie eine Referenz auf das auslösende Objekt, sowie ein `Event`-Objekt, das die Art des Ereignisses beschreibt. Standardmäßig macht diese Methode nichts und das Ereignis wird an die Komponente weitergeleitet, in der die auslösende Komponente enthalten ist.

Das Ereignis wird solange an die jeweils enthaltende Komponente weitergeleitet, bis entweder eine Komponente das Ereignis als behandelt bezeichnet, oder bis die oberste Komponente erreicht wird. In diesem Fall wird das Ereignis vernichtet.

Um jetzt auf ein Ereignis zu reagieren, muss man eine Unterklasse der entsprechenden Komponente erzeugen und die `action()`-Methode überschreiben. Diese Methode hat als Rückgabewert einen `boolean`, der angibt, ob das Ereignis als “behandelt” vernichtet werden soll, wobei dies unabhängig davon ist, ob die Methode wirklich etwas gemacht hat.

Da man für jede Komponente eine eigene Unterklasse erzeugen müsste, sammelt man den Code lieber in den enthaltenden Komponenten, z.B. einem Panel. Hier schaut man dann nach, welche Komponenten das Ereignis ausgelöst hat und verzweigt in den entsprechenden Code.

AWT 1.0 hat den Nachteil, dass fast alle Ereignisse durch eine Reihe von Komponenten durchgereicht (propagiert) werden müssen, was das ganze recht langsam macht. Ausserdem ist der behandelnde Code von der Komponente getrennt, was die Lesbarkeit des Quelltextes reduziert. Ausserdem müssen sehr viele Unterklassen geschrieben werden, die ein Projekt unübersichtlich machen.

AWT 1.1

AWT 1.1 vermeidet die Probleme von AWT 1.0 sehr elegant durch das *Ereigniszuhörer*-Modell (Event Listener). Hierbei werden die Ereignisse nicht einfach an die enthaltenden Komponenten weitergereicht, sondern die Objekte, die die Ereignisse behandeln, registrieren sich bei den Komponenten und werden beim Auftreten eines Ereignisses benachrichtigt.

```
public class BeendenButton extends Button {
    public BeendenButton( String s ) {
        super( s );
    }

    public boolean action( Event ev, Object which ) {
        System.exit( 0 );
        return true;
    }
}
```

```
            // ...
            Frame f = new Frame();
            f.add( new BeendenButton( "Beenden" ) );
            f.pack();
            f.show();
            // ...
```

Auf diese Weise müssen nur noch die Ereignisse behandelt werden, die auch wirklich interessieren und sie werden nur noch an die Objekte weitergeben, die etwas damit anfangen können.

Alle Objekte, die bei einem auftretenden Ereignis benachrichtig werden wollen, müssen ein `Listener`-Interface implementieren. Welches sie implementieren müssen, hängt von der Art der Ereignisse ab. Ein `Button` erfordert einen `ActionListener`, während ein `Frame` einen `WindowListener` benötigt, da beide völlig unterschiedliche Ereignisse weiterleiten.

Tritt ein Ereignis auf, so wird bei jedem Listener die entsprechende Methode des Interfaces aufgerufen. Hierbei darf sich kein Listener auf eine bestimmte Reihenfolge verlassen, in der er mit den anderen Listenern aufgerufen wird. Ausserdem kann kein Listener eine Weiterleitung an andere Listener verhindern.

Mit den mit Java 1.1 eingeführten inneren Klassen, lassen sich Listener sehr einfach implementieren:

```
        Frame f = new Frame();

        Button butBeenden = new Button( "Beenden" );
        butBeenden.addActionListener( new ActionListener() {
            public void actionPerformed( ActionEvent e ) {
                System.exit( 0 );
            }
        } );

        f.add( new butBeenden() );
        f.pack();
        f.show();
```

Die Kombination AWT 1.1 und anonyme innere Klassen erlaubt es, den behandelnden Code direkt zur Erzeugung der Komponenten zu schreiben, was die Lesbarkeit des Quellcodes stark erhöht. Durch die direkte Übergabe des Ereignisses an den behandelnden Code, reagieren Anwendung mit AWT 1.1 deutlich schneller, als solche mit AWT 1.0-Ereignisbehandlung.

Übungen

Erweitern sie das Programm aus dem vorherigen Abschnitt so, dass es zwei unterschiedliche Texte ausgibt, jenachdem welcher Knopf gedrückt wird. Machen sie dies einmal mit AWT 1.0- und einmal mit AWT 1.1-Ereignisbehandlung.

10.3 Zeichnen

Jede Komponente besitzt eine Methode `paint()`, die die entsprechende Komponente am Bildschirm zeichnet. Hierzu bekommt sie ein Objekt vom Typ `Graphics` übergeben, das verschiedene Zeichenoperationen unterstützt.

Will man jetzt eine eigene Grafik darstellen, muss man diese Methode überschreiben und die entsprechenden Methoden von `Graphic` aufrufen.

```
public class MeineGrafik extends Panel {
    public void paint( Graphics g ) {
        g.setColor( Color.blue );
        g.drawLine( 10, 10, 100, 200 );
    }
}
```

Übungen

Schreiben sie ein Programm, das die Werte eines Arrays als Grafik ausgibt. Bestimmen sie hierzu zuerst den maximalen Wert, um die Grafik richtig zu skalieren.

Kapitel 11
Applets

In diesem Kapitel lernen sie

- was Applets sind
- wie Applets vom Browser verwendet werden
- welchen Einschränkungen Applets unterworfen sind und wie man sie umgehen kann
- wie Applets in HTML-Seiten eingebaut werden

11.1 Was sind Applets?

Auf vielen Internetseiten sieht man Java-Applets, die etwas Leben in die Seiten bringen. Beispiele hierfür sind Laufschriften oder Börsenticker.

Aus Sicht des Java-Programmierers sind Applets Unterklassen von `java.applet.Applet` und unterscheiden sich ansonsten nicht von anderen Java-Klassen.

Im Browser werden Applets als eigenes HTML-Tag eingebunden, das u.a. die Größe des Applets auf der Internetseite angibt. Man kann sich also Applets als Java-Programme vorstellen, die in einem Fenster mit fester Größe innerhalb einer Internetseite laufen.

11.2 Applet-Konventionen

Applets werden nicht über die `main()`-Methode gestartet. Stattdessen erzeugt der Browser zuerste eine Instanz der Klasse. Dann ruft er die `init()`-Methode auf und danach die `start()`-Methode.

Wird die Internetseite verlassen, so wird die `stop()`-Methode aufgerufen. Kommt der Benutzer wieder zurück auf die Internetseite, wird nur noch die `start()`-Methode aufgerufen, sofern die Instanz der Applets noch existiert. Ansonsten wird genauso, wie beim ersten Start vorgegangen.

Es ist möglich, Java-Programme zu schreiben, die sowohl Applets, als auch Applikationen sind. Hierzu muss man nur eine `main()`-Methode implementieren, die ein Fenster und eine Instanz der Klasse erzeugt, diese dem Fenster hinzufügt und dann das Fenster anzeigt.

11.3 Einschränkungen von Applets

Applets unterliegen starken Sicherheitseinschränkungen, wenn sie aus dem Internet heruntergeladen werden. Sie dürfen nicht auf die Festplatte zugreifen oder andere Programme starten. Sie dürfen auch fast keine Systemeinstellungen, z.B. den Benutzernamen, abfragen und dürfen nur Verbindungen zu dem Rechner aufbauen, von dem sie heruntergeladen wurden.

Alles dies hat zur Folge, dass Applets ohne Bedenken auf dem Rechner des Benutzers ausgeführt werden können.

11.4 Unterschriebene Applets

Ab Java 1.1 gibt es die Möglichkeit, Applets digital zu unterschreiben. Unterschriebene Applets dürfen den Benutzer nach bestimmten Privilegien fragen, z.B. danach, ob sie auf die Festplatte schreiben dürfen. Erlaubt der Benutzer dies, so kann das Applet zugreifen.

Unterschriebene Applets werden z.B. von Web-basierten Installationsprogrammen verwendet und können in Intranets benutzt werden.

11.5 Das APPLET-Tag

Das Applet-Tag erlaubt einige Attribute, von denen ich hier nur die wichtigsten an Hand eines Beispiels erkläre:

```
<APPLET CODE="de/huwig/tarif/TarifRechner.class"
        WIDTH=720
        HEIGHT=540>
Sie brauchen einen Java-fähigen Browser, um diese Seite darzustellen
</APPLET>
```

Mit den Attribut `CODE` gibt man die Adresse der Appletklasse auf dem Server an. Mit `WIDTH` und `HEIGHT` gibt man Breite und Höhe des Platzes auf der Internetseite an, der für das Applet reserviert ist. Das Applet kann diese Größe nicht verändern - es kann allerdings neue Fenster öffnen.

Wenn sie viele Klassen für ihre Anwendung benötigen, empfiehlt sich die Verwendung des `ARCHIVE`-Tags, mit dem sie eine Jar-Datei (siehe Abschnitt 13.3) angeben können, die alle Klassen in komprimierter Form enthält. Hierfür muss der Browser nur eine Internetverbindung aufbauen, was das Laden des Applets deutlich beschleunigt.

Eine weitere Beschleunigung erreicht man dadurch, dass fertig initialisierte Klassen auf dem Webserver gespeichert (serialisiert) werden. Diese stehen dann nachdem Laden direkt ohne Initialisierung zur Verfügung.

11.6 Der Appletviewer

Mit dem *Appletviewer* können sich Applets ohne einen Webbrowser starten. Er ist im JDK enthalten.

```
C:\JAVA> appletviewer Applet.html
```

Der Appletviewer öffnet für jedes auf der Seite enthaltene Applet ein eigenes Fenster. Auf der Seite enthaltener Text ausserhalb der Applets wird nicht angezeigt.

Kapitel 12
Threads

12.1 Was sind Threads?

Viele der heutzutage verwendeten Programme unterstützen *Multithreading*, was soviel heißt, wie "Multitasking innerhalb eines Programmes". Ein Beispiel hierfür sind die WWW-Browser, die es erlauben während des Herunterladens einer Datei weiter im Internet zu "surfen". Das Programm hat sich in zwei *Threads* aufgeteilt - einer der die Datei herunterlädt und einer der weiterhin WWW-Seiten anzeigt.

Einen Thread erzeugen

Java unterstützt Threads auf eine sehr einfach zu verwendende Weise: eine Klasse muss das Interface `java.lang.Runnable` implementieren, das eine Methode `run()` enthält. Eine Instanz dieser Klasse kann an den Konstruktor von `Thread` übergeben werden. Dieser legt in der JVM einen neuen Thread an und bereitet ihn auf den Start vor. Sobald auf der Instanz von `Thread` die Methode `start()` aufgerufen wird, startet diese den Thread, indem in der zu Beginn übergebenen Instanz die Methode `run()` aufgerufen wird. In Beispiel 12.1.1 sehen sie ein Programm, das zwei Threads erzeugt, wobei der erste immer wieder das Zeichen 'A' ausgibt und der zweite das Zeichen 'B'.

```
public class ABThreads implements Runnable {
    static boolean bFirst = true;

    public static void main( String argv[] ) {
        new Thread( new ABThread() ).start();
        new Thread( new ABThread() ).start();
    }

    public void run() {
        char cZeichen;

        if( bFirst ) {
            cZeichen = 'A';
            bFirst = false;
        } else {
            cZeichen = 'B';
        }

        while( true ) {
            System.out.print( cZeichen );
            Thread.yield();
        }
    }
}
```

Beispiel 12.1.1: AB-Threads

Das Beispiel erkennt an Hand einer statischen Variable, ob es der erste oder der zweite Thread ist. Der Aufruf der Methode `yield()` gibt Rechnenzeit für andere Threads frei. Da Java auf vielen verschiedenen Plattformen läuft, können sie nicht sicher sein, das das verwendete Betriebssystem *preemptives Multitasking* unterstützt, d.h. ihre Threads nach einer gewissen Zeit die Kontrolle automatisch entzogen bekommen damit andere Threads laufen können. Ohne diesen Befehl würden manche JVM statt `ABABABABAB` nur `AAAAAAAAAA` ausgeben.

12.2 synchronized

Wenn mehrere Threads laufen, kann es passieren, dass mehrere gleichzeitig versuchen das gleiche Objekt zu ändern, was dieses in einen ungültigen Zusand bringen kann. Dies kann z.B. passieren, wenn zwei Threads gleichzeitig ein Array sortieren wollen. Sie können dies verhindern, indem sie Objekte oder Methoden mit dem Schlüsselwort `synchronized` markieren. Die JVM merkt sich intern wenn ein Thread darauf zugreift und hält jeden anderen Thread an, der auch darauf zugreifen will.

`synchronized`-Methoden

Wenn sie das Schlüsselwort `synchronized` vor eine Methode schreiben, kann nur ein Thread diese Methode aufrufen. Alle anderen Threads, die diese Methode aufrufen wollen, werden solange angehalten, bis der erste Thread die Methode verlassen hat.

`synchronized`-Objekte

Sie können auch Objekte innerhalb eines Blockes gegen gleichzeitige Zugriffe mehrerer Threads schützen. Wenn sie das Objekt `a` für kurze Zeit sperren wollen schreiben sie einfach

```
        synchronized( a ) {
            // kritischer Code
}
```

Solange sich ein Thread innerhalb des angegebenen Blockes befindet sind alle Zugriffe anderer Threads auf das Objekt `a` geschützt, solange alle diese Zugriffe auch mit `synchronized` markiert sind. Dies kann der gleiche Code sein oder Code in anderen Klassen - er muss nur ebenfalls das Objekt `a` schützen.

Kapitel 13
Nützliches

13.1 Java Runtime Environment 1.1

Wenn sie Java-Programme nur ausführen wollen, aber selbst keine schreiben, reicht ihnen das *Java Runtime Environment* - kurz *JRE*. Die Installation des JRE ist deutlich einfacher, als die des JDK, da alle notwendigen Einstellungen automatisch vorgenommen werden.

Die Angabe des *CLASSPATH* können sie direkt beim Aufruf machen, z.B.

```
C:\> jre -cp C:\JAVA HelloWeb
```

Unter Windows ist das Programm `jre.exe` ist eine DOS-Anwendung, die beim Starten immer ein DOS-Fenster öffnet. Will man eine reine GUI-Anwendung starten, ist dies etwas störend. Von daher gibt es das Programm `jrew.exe`, das ein Java-Programm startet, ohne ein DOS-Fenster zu öffnen. Unter Unix oder Linux haben sie solche Probleme selbstverständlich nicht.

13.2 Java Runtime Environment 1.2

Mit Java 2 hat sich die Benutzung des JRE geändert. Sowohl im JDK, als auch im JRE heißt die JVM immer `JAVA.EXE`, bzw. `JAVAW.EXE`.

Keine von beiden befindet sich allerdings im Pfad, so dass ein direktes Starten nicht mehr möglich ist. Dies ist auch nicht mehr notwendig, da es ab Java 2 möglich ist, ausführbare *JAR-Dateien* (siehe 13.3) zu erzeugen, die sich mit einem Doppelklick auf die Datei starten lassen.

Für diese ausführbaren Dateien ist auch eine neue Option hinzugekommen: `-jar`. Hiermit lassen sich ausführbare JAR-Dateien direkt starten (sofern sich das JRE im Pfad befindet).

```
C:\> java -jar Anwendung.jar
```

13.3 JAR-Dateien

Größere Programme bestehen aus hundert und mehr Klassen, die alle auf dem Rechner des Anwenders installiert werden müssen. Da dies etwas unhandlich ist, kann man in Java Klassen zu sogenannten *JAR-Dateien* zusammenfassen. Statt 100 Klassen in vielen Verzeichnissen hat man nur noch eine handliche Datei, die ausserdem komprimiert ist und daher nur noch etwa die Hälfte an Platz benötigt.

Die Erzeugung einer JAR-Datei geht mit dem Programm `jar`:

```
C:\JAVA> jar cvf MeineJar.jar de\huwig\programm
```

In diesem Fall werden alle Dateien im Verzeichnis `de\huwig\programm` in die Datei `MeineJar.jar` gepackt.

Manifest-Dateien

JAR-Dateien erlauben es auch ein *Manifest* anzugeben, das bestimmte Eigenschaften einer JAR-Datei definieren kann, z.B.

- digitale Unterschrift
- Versionsnummer
- Name der Klasse, die zum Start aufgerufen werden muss
- Erweiterung des CLASSPATH

Manche dieser Eigenschaften sind erst ab Java 2 (JDK 1.2) verfügbar, z.B. der Name der zu startenden Klasse. Damit kann man JAR-Dateien direkt ausführbar machen, d.h. unter Windows reicht ein Doppelklick auf die Datei und das JRE wird automatisch mit den richtigen Parametern aufgerufen.

In Beispiel 13.3.1 sehen sie, wie eine Manifest-Datei aussehen muss, damit die Klasse `de.huwig.programm.Haupt` beim Starten aufgerufen wird und die Datei `xml4j.jar` in den CLASSPATH aufgenommen wird.

```
Main-class: de.huwig.programm.Haupt
Class-Path: xml4j.jar
```

Beispiel 13.3.1: Manifest-Datei `Mein.manifest`

Erzeugt wird die JAR-Datei mit

```
C:\JAVA> jar cvfm MeineJar.jar Mein.manifest de\huwig\programm
```

Ein Doppelklick auf die so erzeugte JAR-Datei ist dann identisch mit einem Aufruf von

```
C:\> jre -cp MeineJar.jar -cp xml4j.jar de.huwig.program.Haupt
```

13.4 Javadoc

Kaum ein Programmierer schreibt gerne Dokumentation zu seinen Programmen, denn sie muss immer auf dem neuesten Stand gehalten werden und schließlich “sieht man ja” was das Programm macht.

Spätestens beim ersten größeren Projekt ist eine Dokumentation immer notwendig, um auch nach ein paar Monaten das ganze noch zu verstehen. Zum Glück bietet Java die Möglichkeit Klassen automatisch zu dokumentieren. Dies geschieht mit speziellen Kommentaren, die direkt in den Quellcode hineingeschrieben werden. Damit ist es relativ einfach, die Dokumentation auf dem neuesten Stand zu halten.

Um die Funktion einer Klasse zu dokumentieren, schreibt man einen erläuternden Text in einen *Javadoc-Kommentar* direkt vor die Klasse. Javadoc-Kommentare sind normale mehrzeilige Kommentare, die allerdings mit zwei Sternen anfangen:

```
/**
 * Diese Klasse ist das erste Java-Programm,
 * das man normalerweise Anfängern zeigt
 */
public class HelloWeb {
    // ...
}
```

Die Sterne am Zeilenanfang sind überflüssig, erhöhen aber die Lesbarkeit.

```
C:\JAVA> javadoc HelloWeb.java
```

Startet man `javadoc`, so wird eine HTML-Dokumentation der Klasse erzeugt. An Stelle einer oder mehrerer einzelner Java-Quelldateien kann man auch ein Verzeichnis angeben, das dann komplett dokumentiert wird.

Was ist dokumentierbar?

Jede Klasse, Methode und Variable (oder Konstante) im Klassenkontext ist dokumentierbar. Mit einem Kommandozeilenschalter kann man aktivieren, welche minimale Sichtbarkeit notwendig ist, damit das Element in die Dokumentation aufgenommen wird. Lokale Variablen werden niemals in die Dokumentation aufgenommen.

Javadoc-Tags

Javadoc-Kommentare dürfen spezielle Tags (Schildchen) enthalten, die weitere Eigenschaften der Klasse angeben. Die wichtigsten sind

- @author
- @exception
- @param
- @return
- @version

Auch hier sagt Beispiel 13.4.1 mehr als 1000 Worte.

```
/**
 * Eine Klasse, die irgendetwas macht
 *
 * @author Kurt Huwig
 * @version 1.0
 */
public class Irgendetwas {
    /**
     * Liefert irgendetwas zurück
     *
     * @param d Umrechnungsfaktor
     * @param s Fehlermeldung
     * @return Anzahl der Dinger
     * @exception IOException, wenn ein Lesefehler auftritt
     */
    public int liefertWas( double d, String s ) throws IOException {
        // ...
    }
}
```

Beispiel 13.4.1: Eine Javadoc-dokumentierte Klasse

Da die Dokumentation in HTML erstellt wird, sind in den beschreibenden Texten auch HTML-Tags erlaubt.

Index