2 Sprach-Grundlagen

In diesem Kapitel möchte ich Sie mit den Grundlagen von Java vertraut machen, die für die weiteren Kapitel benötigt werden. Sie sollten zumindest die Grundlagen der Programmierung als Basis-Wissen mitbringen.

2.1 Zeichensatz in Java-Programmen

Die meisten Programmiersprachen erlauben im Quell-Code nur einen begrenzten Zeichensatz, meist ISO-8859-1. Java geht hier einen Schritt weiter, denn es ist grundsätzlich *Unicode*-basiert. Dieser Zeichensatz kodiert jedes Zeichen mit 2 Bytes und umfasst damit wesentlich mehr Zeichen als zum Beispiel der ASCII-Zeichensatz, bei dem nur 7 Bits verwendet werden. Dabei ist die Codierung der ersten 256 Bytes kompatibel mit dem ISO-8859-1 Zeichensatz. Die Darstellung eines Unicode-Zeichens in Java erfolgt durch Voranstellen von `\u` vor den hexadezimalen Zeichen-Code. Ein Beispiel:

```
...
String myVar = "Üund\u03A9 erlaubt";
```

Mit Ausnahme von `\u03A9` sind alle Zeichen des Programmcodes mit dem ISO-8859-1 Code kompatibel und können daher direkt eingegeben werden. Der Unicode `\u03A9` bedeutet das griechische Omega-Zeichen `Ω` und kann nicht über die Tastatur eingegeben werden, da es nicht mit einem, sondern mit zwei Bytes kodiert wird.

Hinweis Der Unicode-Zeichensatz wird auch bei Strings und einzelnen Zeichen vom Typ `char` verwendet. Selbst die Namen von Variablen und Methoden sowie von Klassen können als Unicode-Zeichen angegeben sein (davon möchte ich jedoch abraten).

2.2 Identifier

Unter dem Begriff *Identifier* versteht man den *Bezeichner* bzw. den Namen einer Variablen, einer Methode oder einer Klasse.

In Java muss ein Identifier mit einem Unterstrich oder einem Unicode-Zeichen beginnen, der einen Buchstaben repräsentiert (eine Ziffer am Anfang ist also nicht

erlaubt). Alle weiteren Zeichen eines Identifiers können beliebige Unicode-Zeichen mit Ausnahme von *White Space* sein (das sind alle Zeichen, die auf einem Ausdruck auf Papier weiß bleiben, wie zum Beispiel das Leerzeichen).

Die Länge von Identifiern ist in Java unbegrenzt. Die Zeichen sind case-sensitive, zwischen Klein- und Großschreibung wird also unterschieden.

Ein selbst definierter Identifier darf kein reserviertes Wort sein (auch die Zeichenketten `true`, `false` und `null` sind verboten, obwohl sie nicht zu den reservierten Wörtern gehören).

2.3 Kommentare

Java unterstützt drei verschiedene Arten von Kommentaren:

```
// Das ist ein einzeiliger Kommentar, der sich
// bis zum Ende der Zeile erstreckt.

/* Kommentar, der
   sich über mehrere Zeilen
   erstrecken kann.
*/

/**
* Spezieller Kommentar für Inline-Dokumentation.
* Man kennzeichnet ihn dadurch, dass zwei Asterisk-
* Zeichen nach dem Slash folgen.
* Er wird vom Programm javadoc interpretiert, mit
* dessen Hilfe man HTML-Dokumentation aus der
* Inline-Doku erstellen kann.
*/
```

Grundsätzlich dürfen gleiche Kommentar-Arten nicht geschachtelt werden, eine Mischung davon jedoch schon:

```
/* Mischung aus verschiedenen
   Kommentaren. // einzeiliger Kommentar
   Und wieder normaler mehrzeiliger Kommentar.
*/
```

2.4 Datentypen

Java unterscheidet grundsätzlich zwei verschiedene Arten von Datentypen: Einfache Daten (primitive Datentypen) und Objekte, die als komplexe Daten in Klassen definiert werden. Objekte werden im nächsten Kapitel ausführlich behandelt, deshalb wollen wir uns hier auf die primitiven Datentypen beschränken.

Datentyp	Speicher-Bedarf	Wertebereich	Default-Wert
`byte`	1	-27 .. 27 – 1	0
`short`	2	-215 .. 215 – 1	0
`int`	4	-231 .. 231 – 1	0
`long`	8	-263 .. 263 – 1	0
`float`	4	±1.4E-45 .. ±3.4028235E38	0.0
`double`	8	±4.9E-324 .. ±1.7976931348623157E308	0.0
`boolean`	1	`false` .. `true`	`false`
`char`	2	`\u0000` .. `\uFFFF`	`\u0000`

Tabelle 2.1: Übersicht der primitiven Datentypen

Hinweis Wenn Sie die Tabelle genau betrachten, stellen Sie fest, dass sowohl Speicherbedarf als auch Wertebereich nicht von der verwendeten CPU abhängen. Java ist wohl die einzige Sprache, die für jeden Datentyp einen konstanten Wertebereich vorgibt, egal, ob in dem Computer eine 32-Bit CPU oder einer 1024-Bit CPU eingebaut ist. Ein `int`-Wert ist also immer 4 Byte breit und hat auf jedem Computer der Welt denselben Wertebereich.

Des Weiteren ist bemerkenswert, dass es in Java keinen vorzeichenlosen Datentyp für Zahlen gibt.

Sehen wir uns nun die einzelnen primitiven Datentypen etwas näher an.

2.4.1 Der Datentyp boolean

Dieser Datentyp ist schnell erklärt, denn er kennt nur zwei Werte: `false` oder `true`. Beide Werte sind in Java zwar keine reservierten Wörter, können aber bei Zuweisungen wie solche verwendet werden. Da ich in diesem Buch voraussetze, dass Sie wis-

sen, was logisch *unwahr* bzw. *wahr* bedeutet, möchte ich nicht näher auf die boolesche Algebra eingehen.

2.4.2 Der Datentyp char

Wie eingangs bereits erwähnt haben die Designer von Java von Anfang an Unicode als Basis für alle Zeichen und Zeichenketten gewählt, ein einzelnes Zeichen vom Datentyp `char` wird also immer mit 2 Bytes kodiert. Als einziger der primitiven Datentypen, mit denen Zahlen verarbeitet werden können, besitzt ein `char`-Wert kein Vorzeichen.

`char`-Literale (das sind Zeichen, die als Konstanten im Programmcode stehen), müssen in *einfache* Quotes gesetzt werden. Beispiel:

```
char c = 'A';

// falsch wäre
char c = "A";
// doppelte Quotes sind Strings vorbehalten
```

Wie in allen Programmiersprachen üblich, unterstützt Java einige Zeichen mit besonderer Bedeutung (*Escape-Sequenzen*), die in der folgenden Tabelle aufgeführt sind:

Escape-Sequenz	Bedeutung
`\n`	Zeilenvorschub (Unix-Zeilenende)
`\r`	Wagenrücklauf (MAC-Zeilenende)
`\t`	Tab
`\b`	Backspace
`\f`	Formfeed
`\"`	Doppeltes Quote
`\'`	Einfaches Quote
`\\`	Backslash
`\`*nnn*	Oktal kodiertes Zeichen
`\u`*xxxx*	Unicode-Zeichen

Tabelle 2.2: Übersicht der Escape-Sequenzen in Zeichen-Konstanten

Verwendet man oktal kodierte Zeichen, dann ist der größte Zeichen-Kode 377. Es müssen zwar keine führenden Nullen angegeben werden, ich rate aber trotzdem dazu, damit zeigt man dem Leser deutlicher, was gemeint ist:

```
char c = '\060'; // die Ziffer '0'
// es ginge auch so:
char c = '\60';
```

Achtung Vorsicht ist bei Unicode-Zeichen geboten. Der Java-Compiler ersetzt die Kodierung `\uxxxx`, bevor der Programmcode interpretiert wird. Ich möchte Ihnen die Auswirkung an einem Beispiel zeigen:

```
// Wir wollen ein einfaches Quote literal verwenden
char c = '''; // Fehler
char c = '\''; // OK
char c = '\047'; // OK
char c = '\u0027'; // Fehler
```

Ich glaube, der Fehler im ersten Beispiel ist einleuchtend. Dass jedoch auch bei der letzten Variante ein Compiler-Fehler auftritt, ist nicht sofort ersichtlich. Hier muss man wissen, dass der Compiler als Erstes alle Unicode-Zeichen-Codes durch deren Zeichen-Äquivalent ersetzt, danach erst wird der Programmcode interpretiert. Nach der Ersetzung von `\u0027` haben wir aber wie bei der ersten Variante wiederum drei Quotes im Programm stehen. Bei der oktalen Kodierung erhalten wir jedoch keinen Fehler, weil die Umwandlung des Zeichen-Codes hier erst nach dem Interpreter-Lauf erfolgt.

2.4.3 Der Datentyp String

Im Gegensatz zu den hier besprochenen primitiven Datentypen sind Zeichenketten (Strings) in Java Objekte. Strings werden ausführlich im Kapitel über wichtige Klassen besprochen.

2.4.4 Numerische Datentypen

Insgesamt bietet Java vier Integer- und zwei Fließkomma-Datentypen an. Allen gemeinsam ist die Tatsache, dass es in Java keine *unsigned*-Zahlen gibt (Zahlen ohne Vorzeichen).

Hinweis Java ist sehr konsequent, was die Portabilität von Zahlen angeht. Egal, welche CPU in Ihrem Rechner eingebaut ist, hat jeder numerische Datentyp einen fest vorgegebenen Wertebereich (unabhängig davon, ob Sie eine 8-, 16-, 32- oder 64-Bit CPU Ihr Eigen nennen).

Ich möchte Sie hier nicht mit den Details der Zahlen-Darstellungen langweilen, vielmehr versuche ich, praktische Hinweise zu geben, die Ihnen beim Programmieren helfen können. Deshalb fällt die Übersicht der numerischen Datentypen relativ kurz aus:

byte

Ein `byte`-Wert nimmt im Hauptspeicher genau 1 Byte in Anspruch und hat aufgrund des immer vorhandenen Vorzeichens einen Wertebereich von -128 bis +127.

short

`short`-Werte belegen im Hauptspeicher genau 2 Bytes und können Zahlen im Wertebereich von -32768 bis +32767 erreichen.

int

`int`-Werte sind genau 4 Bytes lang und können somit Werte im Bereich von -2147483648 bis +2147483647 annehmen. Wenn Sie im Programm numerische Literale verwenden, so werden diese standardmäßig als `int`-Werte behandelt:

```
System.out.println( "Ergebnis: " + ( 3 + 5 ) );
// Sowohl 3 als auch 5 werden zunächst in int-Werte
// umgewandelt, bevor die Summe gebildet (ebenfalls
// int-Wert) und ausgegeben wird.
```

long

`long`-Werte sind genau 8 Bytes lang und können Werte im Bereich von -9223372036854775808 bis +9223372036854775807 erreichen. `long`-Literale werden im Programm durch das Zeichen »`L`« oder »`l`« gekennzeichnet (ich empfehle ein »`L`«, da es besser lesbar ist):

```
long lg = 36578899999999L;
```

Ganzzahlige Literale können in den drei Zahlensystemen Oktal, Dezimal oder Hexadezimal durch Voranstellen einer führenden »`0`« (Oktal-System) bzw. von »`0x`« (Hexa-

dezimal-System) angegeben sein. Ohne Präfix nimmt der Compiler an, dass es sich um eine Zahl im Dezimal-System handelt. Hier einige Beispiele:

```
17 // Dezimal-Zahl
0x7f // Hex-Zahl
037  // Oktal-Zahl

0x123456789012L // long-Hex-Zahl
```

float

`float`-Werte belegen genau 4 Bytes im Hauptspeicher und sind Fest- oder Fließkomma-Zahlen. Literale `float`-Zahlen können durch den Buchstaben »`F`« bzw. »`f`« explizit als solche gekennzeichnet werden (fehlt die explizite Kennzeichnung, dann fasst der Compiler die Literale als `double`-Werte auf). Sie dürfen sowohl als Festkomma-Literale als auch mit einem Exponenten zur Basis 10 im Programm angegeben sein. Als Dezimalpunkt muss das Zeichen »`.`« verwendet werden, das Komma ist nicht erlaubt. Allerdings ist die erreichbare Genauigkeit recht gering, so dass sie eher selten benutzt werden. Der Wertebereich geht von ±1.4E-45 bis ±3.4028235E38.

Ein paar Beispiele für `float`-Literale:

```
3.14F
-3.20E9f
17.4e-18F
.37F

7F // Diese Notation ist zwar erlaubt, ich empfehle
   // aber stattdessen
7.0F // da man hier sofort sieht, dass es sich nicht
     // um eine Hex-Zahl handelt.
```

double

`double`-Werte belegen genau 8 Bytes im Hauptspeicher und können daher den Wertebereich von `±4.9E-324` bis `±1.7976931348623157E308` durchlaufen. Literale Fest- oder Gleitkomma-Zahlen ohne eine explizite Typ-Angabe am Ende werden standardmäßig als `double`-Werte interpretiert. Allerdings kann man auch `double`-Literale explizit als solche durch Anhängen von »`D`« bzw. »`d`« kennzeichnen. Auch hierzu einige Beispiele:

```
-13.4
1.17E-130
-9.88888e30
-17E200d
.0005e-10D

1D  // OK, ich rate aber davon ab und empfehle
1.0
```

Wem die Wertebereiche der primitiven numerischen Datentypen nicht ausreichen, kann auf die im JDK enthaltenen Klassen `java.math.BigInteger` und `java.math.BigDecimal` zurückgreifen, die beliebig große Zahlen verarbeiten können (allerdings auf Kosten der Performance).

Vordefinierte numerische Konstanten

Das JDK bietet mit den so genannten Wrapper-Klassen für die primitiven Datentypen unter anderem einige Konstanten an, von denen ich Ihnen hier die wichtigsten vorstellen möchte:

- `Byte.MIN_VALUE, Byte.MAX_VALUE`
- `Short.MIN_VALUE, Short.MAX_VALUE`
- `Integer.MIN_VALUE, Integer.MAX_VALUE`
- `Long.MIN_VALUE, Long.MAX_VALUE`
- `Float.MIN_VALUE, Float.MAX_VALUE`
- `Double.MIN_VALUE, Double.MAX_VALUE`
- `Float.NEGATIVE_INFINITY, Float.POSITIVE_INFINITY`
- `Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY`
- `Float.NaN, Double.NaN`

Die Wrapper-Klassen `Byte`, `Short`, `Integer` und `Long` sowie `Float` und `Double` sind im Kapitel über *Objektorientierte Programmierung* beschrieben. Ich glaube, die Konstanten sind einigermaßen selbst erklärend. Auf der CD-ROM finden Sie in der Datei *MinMax.java* ein Beispiel für die Konstanten der Wrapper-Klassen.

Erklärungsbedürftig könnte vielleicht die Konstante `NaN` sein, die in den Wrapper-Klassen für Fließkomma-Zahlen definiert ist. Die Abkürzung `NaN` steht für *Not-a-Number* und wird dann zurückgegeben, wenn die Division `0.0/0.0` ausgeführt wird. Auch die Konstanten für die negative bzw. positive Unendlichkeit möchte ich

kurz erklären. Sie werden bei der Division -0.0/0.0 bzw. +0.0/0.0 zurückgegeben. Allerdings ist die Benutzung dieser speziellen Konstanten nicht ganz so einfach. Hierzu folgendes Beispiel:

```
double d = 0.0 / 0.0; // Division durch 0 ergibt NaN
if ( d == Double.NaN )
    System.out.println( "Division durch Null" );
else
    System.out.println( "OK" );
```

Die Preisfrage lautet: »Was wird ausgegeben?«

Ich denke, jeder Politiker, der »Division durch Null« zu seinem Wahlversprechen erklärt hätte, würde gewählt werden, oder? Tja, meist wird man nach der Wahl enttäuscht. So auch hier, denn der Programmcode gibt brav »OK« aus.

Jetzt gehen wir noch einen Schritt weiter in der Verwirrung und lassen folgenden Programmcode laufen:

```
System.out.println( "0 / 0 = " + 0 / 0 );
```

Als Ergebnis bekommen wir eine saftige RuntimeException in Form einer ArithmeticException (Wer mit diesen Begriffen nichts anfangen kann, sei auf das Kapitel *Exceptions* verwiesen, hier soll genügen, dass es sich dabei um Laufzeitfehler handelt, die von der Virtual Machine ausgelöst werden). Probieren Sie es aus, es stimmt.

Damit nicht genug, wenn wir den Programmcode geringfügig ändern und einen der beiden Operanden der Division zu einem double-Literal machen, erhalten wir keine Fehlermeldung mehr:

```
System.out.println( "0 / 0 = " + 0 / 0.0 );
```

Stattdessen gibt die Methode println() nun den Text "0 / 0 = NaN" aus.

Ziemlich verwirrend, was?

Nun wollen wir die Sache nach und nach aufklären:

Achtung Eine Division durch Null mit Integer-Werten führt zu einer Fehlermeldung mit Abbruch des Programms, wenn der Fehler nicht abgefangen wird, bei Gleitkomma-Werten jedoch tritt kein Fehler auf. Stattdessen enthält das Ergebnis der fehlerhaften Division je nach Wert und Vorzeichen des Zählers entweder `NaN` (Zähler ist `0.0`), `POSITIVE_INFINITY` (Zähler ist positiv und größer als `0.0`) oder `NEGATIVE_INFINITY` (Zähler ist negativ und sein absoluter Wert ist größer als `0.0`).

Bei Gleitkomma-Operationen muss das Ergebnis mit den Methoden `Double.isNaN()` bzw. `Double.isInfinite()` überprüft werden, etwa so wie im folgenden Beispiel:

```
...
double x = 0.0;
double y = 0.0;
double d = x / y;
if ( Double.isNaN( d ) || Double.isInfinite() ) {
    // Division durch Null
}
```

Der direkte Vergleich des Ergebnisses z.B. mit der Konstanten `NaN` führt jedoch nicht zum gewünschten Ergebnis:

```
...
if ( d == Double.NaN ) {
    // Der Vergleich ist false und kann somit nicht
    // verwendet werden.
}
```

2.5 Programmblöcke

Ein Java-Programm besteht nicht nur aus einem Block Programmcode, sondern enthält vielmehr unterschiedliche Arten von Blöcken, die teilweise voneinander abhängen. So setzt sich ein einfaches Hauptprogramm in Java mindestens aus zwei Programmblöcken zusammen:

```
public class MyProg {
    public static void main( String[] args ) {
        System.out.println( "Griaß Gott" );
    }
}
```

Der Beispielcode enthält zwei Blöcke, nämlich der Klasse `MyProg` und einer darin befindlichen Methode (für alte C-Hasen: *Funktion*) `main`. Jeder Programmblock wird in Java durch ein Pärchen geschweifter Klammern umrahmt, die dessen Grenzen kennzeichnen. Normalerweise haben diese Programmblöcke einen Namen und eine vorbestimmte Bedeutung (`MyProg` ist durch das Schlüsselwort `class` eindeutig eine Java-Klasse, während `main` aufgrund der runden Klammern eindeutig als Methode identifiziert ist). Es ist sogar möglich, willkürlich Programmblöcke ohne einen Namen festzulegen, solche Blöcke nennt man dann *anonyme Programmblöcke*:

```
{
    int i = 5;
    {
        int j = 6;
    }
}
```

Vielleicht wird sich jetzt der eine oder andere fragen, wofür dieses komplizierte Gebilde notwendig ist. Sie werden weiter unten mehr dazu erfahren, wenn ich über Geltungsbereiche spreche. Hier soll genügen, dass jeder Programmblock, auch ein anonymer Block, einen neuen Geltungsbereich beginnt (und mit der schließenden geschweiften Klammer wieder beendet).

Programmblöcke findet man in einem Java-Programm sehr häufig, z.B.:

```
if ( i < 5 ) {
    // Programmblock
    System.out.println( "i < 5" );
}

while ( i-- < 5 ) {
    // Programmblock
    System.out.println( i );
}

for ( int i = 0; i < 5; i++ ) {
    // Programmblock
}
```

2.6 Variablen

In Java hat grundsätzlich jede Variable einen festen Datentyp, der bereits zur Compile-Zeit überprüft wird (Ausnahmen gibt es jedoch, wie wir bei der Besprechung von Arrays noch sehen werden). Das bedeutet, dass man bei einer Variablen-Deklaration im Programm immer den Datentyp mit angeben muss, der sich später nicht mehr ändern lässt. Dieser Datentyp muss entweder der Name eines der primitiven Typen sein oder ein Klassen-Name.

- Primitive Variablen

  Eine primitive Variable hat als Datentyp einen der insgesamt 8 einfachen Typen `byte`, `short`, `int`, `long`, `float`, `double` oder `boolean`. Variablen dieses Typs werden deshalb *primitiv* genannt, weil der darin gespeicherte Wert direkt im Hauptspeicher als einfacher Wert abgelegt wird.

- Referenz-Variablen

  Eine Referenz-Variable enthält keinen primitiven Datenwert, sondern die Speicheradresse, wo ein Objekt als komplexes Gebilde im Hauptspeicher abgelegt ist. Solche Variablen entsprechen in etwa den Zeigern, die man aus anderen Programmiersprachen kennt.

Der Name einer Variablen kann grundsätzlich aus allen alphanumerischen Unicode-Zeichen bestehen (Ausnahme: *White Space*-Zeichen sind verboten). Er muss mit einem Buchstaben (auch das griechische Omega ist ein Buchstabe) oder einem Unterstrich beginnen und darf beliebig lang sein.

2.6.1 Deklaration von Variablen

Bevor eine Variable benutzt werden kann, muss sie dem Compiler durch eine Deklaration bekannt gemacht werden. Wie bereits erwähnt, muss man dabei immer den Datentyp der Variablen angeben. Bereits bei der Deklaration einer Variablen kann diese initialisiert werden. Dies ist wie gesagt eine Kann-Regel, die Initialisierung ist also nicht zwingend erforderlich. Der Compiler führt eine automatische Initialisierung durch, wenn man dies bei der Deklaration nicht selbst tut. Handelt es sich um eine Referenz-Variable, dann sollte man es sich angewöhnen, diese immer bei der Deklaration zu initialisieren. Hierzu ein paar Beispiele:

```
int i; // nur Deklaration, der Compiler führt eine
       // automatische Initialisierung mit 0 durch.
```

```
double d = 1.5; // explizite Initialisierung durch eine
                // Deklaration einschließlich Wert-
                // Wert-Zuweisung.

double g; // nur Deklaration, der Compiler initialisiert
          // die Variable mit dem Wert 0.0

long l;           // Deklaration mit anschließender
l = 78999900012L; // Zuweisung eines Werts.

double d1 = d + i; // Eine Variable kann auch mit einem
                   // Ausdruck initialisiert werden.

boolean b; // nur Deklaration, der Compiler
           // initialisiert die Variable automatisch
           // mit dem Wert false.

String s; // Deklaration einer Referenz-Variablen ohne
          // Initialisierung, der Compiler führt keine
          // automatische Initialisierung durch.

String s1 = null; // besser, wir initialisieren die
                  // Referenz-Variable explizit mit dem
                  // Sonder-Wert null (d.h. s1 enthält
                  // keine Referenz auf ein Objekt).

String s2 = s1; // s2 enthält eine gültige Referenz
                // auf s1 (obwohl diese wiederum keine
                // gültige Referenz besitzt).
```

Wie bereits erwähnt, muss eine Variable deklariert werden, bevor man sie benutzt. Wo sich die Deklaration im Programmcode befindet, bleibt dem Programmierer überlassen, sie muss nur vor deren erstem Gebrauch stehen. Damit bleibt einem also erspart, dass man alle benutzten Variablen grundsätzlich am Anfang des Programmteils (z.B. des Hauptprogramms) deklarieren muss.

Namenskonvention für Variablen

Nahezu für jede Programmiersprache existiert ein so genannter *Style Guide*, in welchem nachzulesen ist, welche Regeln man beim Schreiben von Programmcode einhalten sollte. Leider hält sich nur ein geringer Prozentsatz aller Entwickler an diese Konventionen. Ich möchte Sie ausdrücklich bitten, sich den auf der CD-ROM enthaltenen Style Guide nicht nur durchzulesen, sondern die dort beschriebenen Vorga-

ben auch einzuhalten. Auch bezüglich Variablen-Namen gibt es eine Konvention, die besagt, dass der Name einer normalen Variablen mit einem Kleinbuchstaben beginnen soll. Besteht der Name aus mehreren logischen Wort-Teilen, dann ist jeweils der erste Buchstabe eines logischen Teils großzuschreiben. Unterstriche für diese Trennung soll man nicht verwenden. Eine Ausnahme von dieser Regel gilt für Konstanten, deren Name grundsätzlich komplett aus Großbuchstaben bestehen soll. Hier dürfen Unterstriche verwendet werden.

Beispiele für Variablen-Namen:

```
String thisIsAString;
int loopCount;

// schlechter Stil
int loop_count;

// ebenfalls gegen die Regeln, da erster Buchstabe groß
int LoopCount;

// Konstanten-Definition
public final String VERSION = "1.1";
public final int MAX_COUNT = 1000;
```

2.6.2 Geltungsbereich (Scope)

Variablen besitzen eine Lebensdauer, d.h. sie sind nur innerhalb eines bestimmten Bereichs gültig. Diesen Bereich nennt man *Geltungsbereich* oder neudeutsch *Scope*. Eine Variable beginnt mit ihrer Deklaration zu leben und ist so lange gültig, bis der Geltungsbereich, in dem sie deklariert wurde, verlassen wird.

Java teilt in punkto Geltungsbereich die Variablen in drei verschiedene Arten ein:

- Lokale Variablen

  Variablen, die in einer Methode (so nennt man in Java eine Funktion) deklariert sind, werden als lokale Variablen bezeichnet. Sie gelten nur innerhalb dieser Methode und leben daher nur für die Dauer des Methodenaufrufs. Außerhalb der Methode sind sie völlig unsichtbar.

- Klassen-Variablen

  Eine Variable, die statisch in einer Java-Klasse deklariert wird (wie das funktioniert, lernen wir im Kapitel über *Objektorientierte Programmierung*), ist von dem Moment an gültig, wo die Klasse von der Virtual Machine geladen wird. Ihr Geltungsbereich endet erst dann, wenn die Klasse wieder verworfen (entladen) wird.

- Instanz-Variablen

  Variablen, die in einer Java-Klasse ohne das reservierte Wort `static` deklariert werden, sind so genannte Instanz-Variablen, auch Member-Variablen genannt. Solche Variablen beginnen ihr Leben damit, dass eine Instanz der Klasse erzeugt wird (ein Objekt der Klasse wird instanziert). Mit jedem neu erzeugten Objekt der Klasse wird eine eigenständige Kopie aller in der Klasse deklarierten Instanz-Variablen angelegt, die so lange gültig bleibt, bis das Objekt, zu dem sie gehört, vom Garbage Collector aus dem Hauptspeicher entfernt wird. Wer sich jetzt schwer tut mit den Begriffen, der sollte im Kapitel *Objektorientierte Programmierung* nachlesen.

Hinweis Neben den hier vorgestellten Variablen gibt es noch die so genannten formalen Parameter-Variablen, die bei der Deklaration von Java-Methoden angegeben sind. Auf diese Art werde ich im Abschnitt über Methoden zu sprechen kommen.

Verdecken von Variablen

Unter diesem etwas seltsamen Ausdruck versteht man die Deklaration einer Variablen, die unter demselben Namen in einem weiter außen liegenden Programmblock bereits deklariert wurde. Ich demonstriere dies am besten anhand eines (nicht erlaubten) Beispiels:

```
int i = 5;
for ( int i = 1; i <= 10; i++ ) {
    System.out.println( i );
}
```

Im Beispiel deklariere ich zunächst eine Variable mit dem Namen `i`. Danach wird durch die `for`-Schleife (zu der wir noch kommen) erneut eine Variable mit demselben Namen deklariert. Wenn wir uns innerhalb der `for`-Schleife befinden, existiert aus dieser Sicht ein außen liegender Programmblock, in dem die Variable `i` bereits definiert wurde. Durch die erneute Deklaration einer Variablen mit demselben Namen in der `for`-Schleife versuchen wir, die äußere Variable `i` zu verdecken. Dies erlaubt Java

jedoch nicht und teilt uns das klar und unmissverständlich durch eine Compiler-Meldung mit. Hätten wir es jedoch anders herum programmiert, würde keine Fehlermeldung entstehen:

```
for ( int i = 1; i <= 10; i++ ) {
    System.out.println( i );
}
int i = 5;
```

Nun deklarieren wir die Variable `i` im äußeren Programmblock, nachdem die `for`-Schleife zu Ende ist. Dies ist erlaubt, weil der Geltungsbereich für die innere Variable `i` bereits verlassen wurde und damit `i` nicht mehr existiert.

Demnach sind folgende Konstrukte erlaubt:

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

for ( int i = 1; i <= 10; i++ ) {
    System.out.println( i );
}

{ int i = 5; }
{ int i = 6; }

int i = 10;

// Auch das ist erlaubt (try/catch-Blöcke werden
// im Kapitel über Exceptions ausführlich behandelt).
try {
    double a = 7.0 / 1.0;
} catch ( ArithmeticException ex ) {
    ...
} catch ( RuntimeException ex ) {
    ...
}
```

Die Variable `ex` wird in zwei aufeinander folgenden `catch`-Blöcken mit demselben Namen deklariert. Es folgt kein Compiler-Fehler, weil beide Blöcke in derselben Ebene liegen, im zweiten `catch`-Block wurde der Geltungsbereich des ersten bereits verlassen.

Doch Vorsicht: Man kann hier viele Fehler machen:

```
// Vorsicht: Der folgende Code führt zu einer
// Fehlermeldung, weil ArithmeticException aus
// RuntimeException abgeleitet ist. Mehr hierzu
// im Kapitel über Objektorientierte Programmierung.
try {
    double a = 7.0 / 1.0;
} catch ( RuntimeException ex ) {
    ...
} catch ( ArithmeticException ex ) {
    ...
}
```

Auch der folgende Code wird vom Compiler nicht übersetzt, weil nun sowohl in einem äußeren als auch in einem inneren `catch`-Block derselbe Variablen-Name verwendet wird.

```
try {
    ...
} catch ( Exception ex ) {
    try {
        ...
    } catch ( Exception ex ) {
        ...
    }
}
```

Lokale Variablen von Methoden dürfen im Gegensatz zum bisher Gesagten weiter außen liegende Variablen durchaus verdecken. Auch das möchte ich an einem kurzen Beispiel erläutern:

```
public class MyProg {
    int var = 10;

    public void myMethod() {
        int var = -10;
    }
}
```

Auch in diesem Beispiel existieren zwei Programmblöcke, der äußere für die Klasse `MyProg`, der innere enthält den Programmcode der Methode `myMethod()`. In beiden Programmblöcken wird eine Variable mit gleichem Namen deklariert. Diesmal erfolgt keine Fehlermeldung des Compilers, sondern die lokale Variable `var` in der Methode `myMethod` verdeckt die äußere Variable `var`, die als Instanz-Variable der Klasse `MyProg` deklariert wurde. Es handelt sich dabei um zwei völlig unterschiedliche Variablen, die allerdings denselben Namen tragen. Solange man sich in der Methode `myMethod` befindet, kann man nur auf die lokale Variable zugreifen (und deren Wert verändern), ohne dass dies irgend eine Auswirkung auf die äußere Variable hat. Kehrt der Programmfluss wieder in den äußeren Programmblock zurück, so existiert nur noch die äußere Variable.

Eine weitere Möglichkeit, Variablen zu verdecken, besteht in der Parameterliste von Methoden. Auch hierzu ein (vorgreifendes) Beispiel:

```
class MyClass {
    private int i = 1;
    private static int j = 2;

    public MyClass( int i, int j ) {
        this.i = i;
        MyClass.j = j;
    }
}
```

Das Beispiel zeigt eine sehr einfache Klasse mit nur einer Instanz-, einer statischen Klassen-Variablen und einem Konstruktor, durch den diese Variablen gesetzt werden können. Die formalen Methodenparameter haben denselben Namen wie die in der Klasse deklarierten Variablen und verdecken diese somit innerhalb des Konstruktors, dies bedeutet, dass man auf normalem Wege nicht auf die äußeren Variablen zugreifen kann.

Instanz-Variablen dürfen aber auch absolut adressiert werden, indem man vor den Namen der Variablen das reservierte Wort `this` setzt und die beiden Bestandteile durch einen Punkt voneinander trennt. Java bietet mit der vordefinierten Variablen `this` eine Referenz auf die Instanz des aktuellen Objekts an, die man in allen Instanz-Methoden verwenden kann.

Äquivalent hierzu kann man direkt auf statische Klassen-Variablen zugreifen, indem man, durch einen Punkt getrennt, den Klassen-Namen vor die Variable stellt.

2.7 Referenzen

Bei der Besprechung der Variablen haben wir uns bisher meist auf einfache Variablen konzentriert. Nun wollen wir uns den Referenz-Variablen zuwenden, die beim Programmieren in Java eine wichtige Rolle spielen.

Jede Java-Variable, die keinen primitiven Datentyp hat, ist automatisch eine Referenz-Variable. Sie enthält also keinen einfachen Datenwert, sondern die Speicheradresse auf ein anderes Objekt, dessen komplexe Struktur vor dem Programmierer verborgen bleibt.

Hinweis Während es in anderen Programmiersprachen einen Dereferenzierungs-Operator gibt, der nur für Referenz-Variablen verwendet wird, greift man in Java immer mit derselben Syntax auf Variablen zu, gleichgültig, ob es sich um eine primitive oder eine Referenz-Variable handelt. Es existiert kein Dereferenzierungs-Operator in Java.

Da in Java auch Strings und Arrays wie Objekte behandelt werden, sind Variablen auf Strings oder Arrays automatisch ebenfalls Referenz-Variablen. Vor allem bei Arrays ist deshalb besonders dann Vorsicht geboten, wenn Array-Variablen als Parameter von Methoden-Aufrufen verwendet werden. Doch ich greife vor, lassen Sie uns zunächst die wichtigsten Merkmale von Referenzen erörtern.

2.7.1 Zuweisung einer Referenz

Sehen wir uns dazu folgenden Code an:

```
01 StringBuffer v1 = new StringBuffer( "hallo" );
02 StringBuffer v2 = s1;
03
04 v1.append( ", Welt" );
05 v1 = null;
06 System.out.println( "v2 = " + v2 );
07
08 StringBuffer v3 = null;
```

Bei der ersten Zuweisung wird zunächst eine neue Instanz der Klasse `StringBuffer` erzeugt und diese mit dem Wert `hallo` belegt. Anschließend erfolgt eine Zuweisung des so entstandenen neuen Objekts an die Referenz-Variable `v1`. Dabei wird nicht der

Wert `hallo` an die Variable übergeben, sondern eine Referenz auf das Objekt, dessen Wert `hallo` ist. Prägen Sie sich diesen Unterschied gut ein.

Die zweite Zuweisung kopiert die Referenz `v1` in die neue Variable `v2`. Wohl gemerkt, es wird die Referenz kopiert, nicht das Objekt, auf das die Variable `v1` zeigt. Das bedeutet, dass nun beide Variablen auf dasselbe Objekt zeigen. Eine Änderung (im Beispiel hänge ich den String `", Welt"` hinten an) über die Referenz-Variable `v1` ist gleichfalls über die Referenz `v2` sichtbar (Die Ausgabe über `v2` mit `println()` in Zeile 06 ist `"hallo, Welt"`.).

Mit der Zuweisung in Zeile 05 lösche ich die Referenz von `v1`, die Variable zeigt anschließend auf NICHTS (dargestellt durch den Pseudo-Wert `null`). Das `StringBuffer`-Objekt selbst ist davon nicht betroffen und wird dadurch nicht verändert oder gelöscht (weil über `v2` noch eine weitere Referenz auf das Objekt besteht, doch mehr dazu gleich).

Man kann (und sollte) bei der Deklaration einer Referenz-Variablen diese mit `null` (oder mit einer gültigen Referenz auf ein Objekt) initialisieren, dies beugt so mancher Warnung des Compilers beim Übersetzen von Programmen vor. Ich habe eine solche Deklaration mit expliziter Initialisierung in Zeile 08 implementiert.

Hinweis Java besitzt einen so genannten *Garbage Collector*, der im Hintergrund läuft und nicht mehr benötigten Hauptspeicher automatisch freigibt. Im Wesentlichen sucht der Garbage Collector nach Objekten, denen keine Referenz mehr zugeordnet ist, und entfernt diese Objekte aus dem Hauptspeicher. Besteht jedoch noch mindestens eine gültige Referenz auf ein Objekt, dann verbleibt es im Hauptspeicher. In unserem Code-Beispiel würde das `StringBuffer`-Objekt nicht gelöscht werden, weil zwar die Referenz über `v1` gelöscht ist, jedoch besteht nach wie vor die Referenz über die Variable `v2`. Erst wenn auch diese mit `null` belegt wird, löscht der Garbage Collector das Objekt.

Ein Hinweis für Performance-Hungrige: Wenn man genau weiß, dass eine Referenz-Variable nicht mehr benötigt wird und der damit verbundene Hauptspeicher freigegeben werden kann, sollte man dieser Variablen explizit den Wert `null` zuweisen, der Garbage Collector kann dann umgehend seine Zerstörungstat vollbringen.

Achtung Vorsicht ist beim Vergleich von Referenz-Variablen geboten. Hierzu ein Beispiel:

```
01 String s1 = "hallo";
02 String s2 = "hallo";
03 if ( s1 == s2 ) {
04     System.out.println( "s1 = s2" );
05 }
06 else {
07     System.out.println( "s1 != s2" );
08 }
09
10 s2 = s1;
11 if ( s1 == s2 ) {
12     System.out.println( "s1 = s2" );
13 }
14 else {
15     System.out.println( "s1 != s2" );
16 }
```

Zunächst definieren wir zwei unterschiedliche Referenz-Variablen mit demselben Wert "`hallo`". Danach fragen wir ab, ob beide Variablen gleich sind. Das Ergebnis ist evtl. nicht wie erwartet, denn die String-Werte sind zwar identisch, es handelt sich aber in beiden Zuweisungen um jeweils eigenständige und vor allem um unterschiedliche `String`-Objekte. Damit enthält `s1` eine andere Referenz als `s2`, die Variablen sind also ungleich (nur die darin gespeicherten Werte sind identisch).

Der Code-Teil ab Zeile 10 verhält sich anders, hier wird der Variablen `s2` die Referenz, auf welche `s1` zeigt, zugewiesen, beide Variablen enthalten also dieselbe Referenz, und die Abfrage liefert `true` und gibt "`s1 = s2`" aus.

2.8 Arrays

Ich gehe davon aus, dass Ihnen der Begriff *Array* vom Grundsatz her bekannt ist. Wie in anderen Programmiersprachen auch enthält ein Java-Array mehrere Elemente, auf die man über einen numerischen Index zugreifen kann, der beim ersten Element mit `0` beginnt. Auch Java-Arrays besitzen einen festen Datentyp (mit einer kleineren Abweichung bei vererbten Objekten, wie wir im Kapitel über *Objektorientierte Programmierung* sehen werden).

Der große Unterschied bei Java besteht in der Tatsache, dass Arrays von Java als Objekte mit spezieller Sprach-Unterstützung behandelt werden (ähnlich wie Strings, die ebenfalls spezielle Java-Objekte sind). Die Folge davon ist, dass in Java Arrays zwar zur Übersetzungszeit auf ihren Typ hin überprüft, aber erst zur Laufzeit angelegt wer-

den. Allerdings kann anschließend die Größe eines einmal angelegten Arrays nicht mehr verändert werden.

Hinweis Halten wir also fest: Java-Arrays sind Zwittergestalten. Ein Array kann zwar zur Laufzeit dynamisch erzeugt werden (die Größe des Arrays muss also nicht als Konstante im Source-Code angegeben sein, sondern kann auch dynamisch über eine Variable angegeben werden), ist das Array aber einmal angelegt worden, kann dessen Größe nachträglich nicht mehr verändert werden.

2.8.1 Deklaration von Arrays

Array-Variablen werden durch eckige Klammern hinter dem Array-Typ gekennzeichnet:

```
int[] ids;        // Array-Variable vom Typ int
String[] strings; // Array-Variable vom Typ String
```

Häufig findet man auch folgende Deklarationen:

```
int ids[];
String strings[];
```

Diese Variante ist zwar (noch) erlaubt (jedoch wurde sie als *deprecated* markiert und sollte nicht mehr verwendet werden), ich empfehle Ihnen die erste Methode (Klammern nach dem Array-Typ).

Hinweis Wie wir bereits an den Deklarationen sehen, müssen alle Elemente eines Arrays denselben Datentyp besitzen, nämlich den bei der Array-Variablen verwendeten. Eine Ausnahme möchte ich Ihnen im Vorgriff auf *Objektorientierte Programmierung* nicht verheimlichen: Wird bei der Deklaration einer Array-Variablen eine Klasse angegeben, dann müssen alle Elemente zwar einen dazu kompatiblen Typ haben, es muss aber nicht zwingend derselbe sein wie bei der Deklaration angegeben.

Ein Array-Element kann auch ein Objekt einer Kind-Klasse sein oder eines Objekts, das ein kompatibles Interface implementiert. Sie werden den Aha-Effekt in den Kapiteln über *Objektorientierte Programmierung* bzw. *Interfaces* erleben.

2.8.2 Initialisierung von Arrays

Grundsätzlich initialisiert man Arrays mit dem reservierten Wort `new`, das allgemein für die Erzeugung neuer Objekte verwendet wird. Nach `new` folgen der Datentyp und die Anzahl der Elemente des Arrays in eckigen Klammern:

```
int[] ids = new int[ 5 ];

int len = 100;
double[] darray = new double[ len ];
```

Hinweis Durch die Initialisierung eines Arrays wird nur Speicherplatz für die angegebene Anzahl von Elementen bereitgestellt, die Array-Elemente selbst werden entweder mit dem Default-Wert initialisiert (primitive Datentypen) oder mit `null` vorbelegt, wenn das Array aus Objekten besteht.

Literale Initialisierung von Arrays

Wenn man bereits bei der Deklaration eines Arrays die Elemente selbst initialisieren möchte, kann man dies ohne das reservierte Wort `new` in Form von geschweiften Klammern tun:

```
boolean[] flags = { false, true, false, };
String[] msgs = { "A", "B", };
int initVal = 1;
int[] ids = { initVal, initVal, };
```

Als Trennzeichen für die einzelnen Elemente wird das Komma verwendet. Man kann natürlich auch `new` in Verbindung mit geschweiften Klammern benutzen:

```
int[] ids = new int[] { 1, 2, 3, };
```

Bei der Benutzung von `new` in Verbindung mit geschweiften Klammern muss die Anzahl der Elemente in den eckigen Klammern entfallen, weil sie durch die Elemente in den geschweiften Klammern vorgegeben ist.

Hinweis In den Beispielen für die literale Initialisierung von Arrays habe ich auch nach dem letzten Element ein Komma gesetzt. Dies ist erlaubt und durchaus gewollt, weil man damit Fehler vermeidet, wenn das Array im Laufe der Programmentwicklung später erweitert wird.

2.8.3 Zugriff auf Array-Elemente

Auf ein einzelnes Element in einem Array greift man sowohl zum Lesen als auch zum Schreiben über dessen numerischen Index zu, der in eckigen Klammern angegeben wird:

```
int[] ids = { 1, 2, 3, };
System.out.println( ids[ 0 ] );
ids[ 1 ] = 5;
int ind = 2;
int i = ids[ ind ];
```

Hinweis Der numerische Index von Array-Elementen wird in Java als `int`-Wert behandelt. Allerdings sind auch `byte`, `short` und sogar `char`-Werte zulässig, sie werden implizit in `int`-Werte umgewandelt.

Versucht man, auf ein Element zuzugreifen, dessen Index gar nicht existiert (zum Beispiel negativer Index), dann wird von der Virtual Machine eine `ArrayIndexOutOfBoundsException` ausgelöst.

2.8.4 Das Attribut length

Eine Besonderheit von Java-Arrays ist, dass automatisch das vordefinierte Attribut `length` angeboten wird, das die Anzahl der Array-Elemente enthält:

```
String[] strings = { "a", "b", "c", };
for ( int i = 0; i < strings.length; i++ ) {
    System.out.println(
        "strings[ " + i + " ] = " + strings[ i ]
    );
}
```

Achtung Das vordefinierte Attribut `length` von Arrays darf nicht verwechselt werden mit der Instanz-Methode `length()`, die in vielen Klassen, zum Beispiel auch in der Klasse `String`, zur Verfügung steht.

2.8.5 Mehrdimensionale Arrays

In Java sind mehrdimensionale Arrays recht straightforward, man muss einfach mit jeder Dimension ein Paar eckiger Klammern hinzufügen (bei literaler Initialisierung sind natürlich auch mehrere Pärchen geschweifter Klammern notwendig):

```
// zweidimensionales Array
int[][] ids;

// hier mit Initialisierung
int[][] is = {
    { 1, },
    { 1, 2, },
    { 1, 2, 3 },
};
```

Hinweis Wie man an diesem Beispiel sieht, müssen die Sub-Arrays nicht dieselbe Länge besitzen (dies ist z.B. in C zwingend notwendig). Jedes Unter-Array kann eine beliebige Länge haben.

2.9 Casting

Häufig steht man vor dem Problem, dass man das Ergebnis einer Operation in einer Variablen abspeichern möchte, das nicht genau denselben Datentyp hat wie die Operanden selbst. In einem solchen Fall müssen die Datentypen so konvertiert werden, dass sie zu dem des Ergebnisses passen (kompatibel sind). Diese Konvertierung erfolgt mit dem so genannten *Cast*-Operator, der in Java aus einem Paar runder Klammern besteht, in denen der gewünschte Ziel-Datentyp steht. Nehmen wir gleich ein paar Beispiele (allesamt mit primitiven Datentypen) zur Erläuterung:

```
01 byte b = 0x09;
02 short s = ( short ) b;
03 int i = ( int ) s;
04 long l = ( long ) i;
```

```
05
06 int i1 = ( int ) b;
07 long l1 = ( long ) b;
08 long l2 = ( long ) s;
09 long l3 = ( long ) i;
10
11 int i2 = b + s;
12 long l4 = i * b * s;
13
14 long l5 = ( long ) i + s;
15
16 long l6 = ( long ) ( i + s );
```

Bis zur Zeile 09 habe ich die einzelnen Werte explizit mit Hilfe des Cast-Operators in den gewünschten Ziel-Datentyp umgewandelt. Dem aufmerksamen Leser wird auffallen, dass dabei der Wertebereich des Ziel-Datentyps immer größer ist als der des ursprünglichen Datentyps, dies nennt man *erweiterndes Casting*. Bei dieser Art von Konvertierung hat man in der Regel keine Probleme, weil sichergestellt ist, dass der zu konvertierende Wert in jedem Fall in die Ziel-Variable passt.

Es gibt jedoch auch den umgekehrten Fall, den man *einschränkendes Casting* nennt. Dabei wird ein Wert in einen Datentyp umgewandelt, dessen Wertebereich kleiner ist, als der ursprüngliche Datentyp zulässt:

```
int i = 20;
byte b = ( byte ) i;
```

Hier ist nicht sichergestellt, dass das Ergebnis auch tatsächlich innerhalb des Wertebereichs des Ziel-Datentyps liegt. Java schneidet in diesem Fall alle Bits ab, die außerhalb des Wertebereichs für den Ziel-Datentyp liegen:

```
int i = 0x123456;
byte b = ( byte ) i;
```

Das Ergebnis, das in der Variablen `b` abgelegt wird, ist `0x56` (dezimal `86`), da alle weiteren Bits abgeschnitten werden. Dieses Verhalten ist übrigens unabhängig von der verwendeten CPU und sowohl bei Little-Endian (z.B. SPARC-Prozessoren) als auch bei Big-Endian (z.B. INTEL-Prozessoren) CPUs gleich.

Ab Zeile 11 wird es etwas mystisch. Sowohl dort als auch in der darauf folgenden Zeile überlassen wir es der Intelligenz des Compilers bzw. der Virtual Machine, die Konvertierung der Datentypen durchzuführen. Diese reicht zumindest so weit, dass er bzw. sie erweiternde Konvertierungen automatisch durchführt, wir hätten jedoch Zeile 11 auch so schreiben können:

```
11 int i2 = ( int ) ( ( short ) b + s );
```

Die Addition hat zwei unterschiedliche Datentypen, also konvertieren wir den kleineren Datentyp in den nächstgrößeren, das ist `short`. Das Ergebnis wiederum müssen wir in den Datentyp `int` konvertieren.

In Zeile 14 muss man sich fragen, worauf sich die Konvertierung bezieht. Wird die Variable `i` umgewandelt und anschließend die Addition durchgeführt, dessen Ergebnis wiederum (implizit) von der Virtual Machine in `long` umgewandelt wird? Oder wird zunächst die Addition durchgeführt und anschließend das Ergebnis umgewandelt?

Die Antwort findet man entweder dadurch, dass man sich die Prioritäten der Operatoren genau durchliest, oder man schreibt die einzelnen Variablen und Ausdrücke in runde Klammern, um die Prioritäten selbst festzulegen (was ich in Zeile 16 getan habe). Ich empfehle die zweite Methode. Schreiben Sie bitte immer genau hin, in welcher Reihenfolge Sie was bearbeitet haben möchten, auch wenn Sie dabei mehr Klammern setzen müssen.

Zeile 16 sagt dem Compiler: »Bilde zuerst die Summe zweier Variablen, dabei überlasse ich dir die Typ-Konvertierung. Anschließend möchte ich das Ergebnis explizit in den Datentyp `long` umwandeln«.

Wie Casting mit Objekten funktioniert, sehen Sie im Kapitel *Objektorientierte Programmierung*.

Hinweis Java erlaubt keine Konvertierung eines Strings in einen numerischen Datentyp, auch dann nicht, wenn sichergestellt ist, dass der String eine gültige Zahl enthält (allerdings bieten die Wrapper-Klassen für die primitiven Datentypen hierfür Methoden an):

```
String sd = "2.0";
// Die folgende Anweisung liefert einen Fehler
double d = ( double ) sd;
```

Im folgenden Bild möchte ich noch einmal einen Überblick über das Casting der primitiven Datentypen geben.

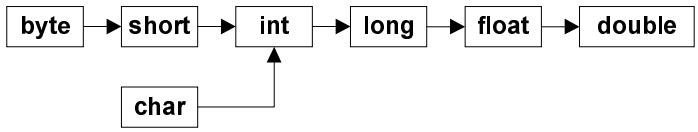

Abbildung 2.1: Casting von primitiven Datentypen

Solange man sich bei der Konvertierung eines Datentyps in Pfeilrichtung bewegt (*erweiterndes Casting*), kann der Casting-Operator entfallen (Bei der Konvertierung können auch ein oder mehrere Datentypen übersprungen werden, zum Beispiel, wenn ein `byte`-Wert nach `long` konvertiert werden soll). Sie dürfen natürlich explizit den Operator angeben, wenn Sie wollen. Bewegt man sich jedoch entgegen der Pfeilrichtung (*einschränkendes Casting*), dann muss der Casting-Operator zwingend angegeben werden, sonst erhalten Sie eine Compiler-Warnung.

Der Compiler führt eine erweiternde Typ-Konvertierung automatisch in folgenden Fällen für Sie durch:

- Bei Zuweisungen, wenn der Variablen-Typ des Ergebnisses nicht mit dem Ausdruck übereinstimmt.
- In arithmetischen Ausdrücken, wenn die Operatoren unterschiedliche Datentypen besitzen.
- Bei Array-Indizes, deren Datentyp kleiner ist als `int`.
- In Methoden-Aufrufen, wenn der Typ eines formalen Parameters nicht mit dem zugehörigen Aufruf-Parameter übereinstimmt.

Hinweis Interessanterweise bietet Java keinen Datentyp für vorzeichenlose Bytes an. Sollen also Bytes ohne Berücksichtigung ihres Vorzeichens einschließlich des höchstwertigen Bits verarbeitet werden, dann muss man vorher den `byte`-Wert in `short` oder `int` umwandeln. Hier kommt es aber zu Problemen, wenn der `byte`-Wert größer ist als `127`, denn damit ist er im Datentyp `byte` negativ. Das Vorzeichen aber wird bei der Typ-Konvertierung mit berücksichtigt. Folgender Code führt also zu Überraschungen:

```
byte b = 0xff;
int i = b;
```

Hier erleben wir gleich zwei Überraschungen, denn die erste zeigt sich sofort beim Übersetzen des Programmcodes, weil der Compiler eine Fehlermeldung ausgibt. Dies liegt daran, dass er numerische Konstanten (hier `0xff`) vor der Verarbeitung in `int`-Werte umwandelt. Diesen weisen wir ohne Cast-Operator direkt einer `byte`-Variablen zu (einschränkendes Casting), deshalb der Fehler.

Als Nächstes ist zu beachten, dass aus `0xff` nicht etwa `255` wird, sondern `-1`, weil im höchstwertigen Bit das Vorzeichen steht. In der Variablen `i` steht also ebenfalls `-1`.

Das erste Problem lässt sich einfach durch den Casting-Operator lösen:

```
byte b = ( byte ) 0xff;
```

Etwas komplizierter wird es, wenn wir versuchen, den `byte`-Wert als vorzeichenlosen Wert zu behandeln:

```
int i = ( b & 0x7f ) + ( ( b < 0 ) ? 128 : 0 );
```

Sieht schwierig aus, was? Leider geht es nicht anders, hier hat Java eindeutig eine Design-Schwäche. Übrigens, wenn Sie jetzt nicht wissen, wie die letzte Code-Zeile funktioniert: Sie werden nach der Besprechung der Operatoren dazu in der Lage sein.

2.10 Operatoren

Operatoren sind das Salz in der Suppe einer Programmiersprache, man trifft sie überall in Programmen an. Die meisten Operatoren sind so trivial, dass man sie gar nicht mehr direkt wahrnimmt:

```
    |------------ Operand
    |     |------- Operand
    |   |---|
    |   |--------- Operand
    |   |   |----- Operand
int i = 7 + 9;
      |   |------- Additions-Operator
      |----------- Zuweisungs-Operator
```

In diesem Beispiel haben wir insgesamt zwei Operatoren. Mit dem +-Operator werden die beiden Operanden 7 und 9 durch eine Addition verknüpft. Das Ergebnis dieser Operation wird wiederum als Operand für den Zuweisungs-Operator benutzt. Bei diesem muss man wissen, dass er den rechten Operanden evaluiert (berechnet) und das Ergebnis dem linken Operanden zuweist, der Zuweisungs-Operator hat also eine bestimmte Richtung (rechts nach links).

Manche Operatoren wiederum besitzen nur einen Operanden:

```
 ||---------- Auto-Increment-Operator
i++;
|------------ Operand


        |---- Minus-Operator
int j = -i;
         |--- Operand
```

Operatoren, die nur einen Operanden erwarten, bezeichnet man als *unäre* Operatoren, während die Mehrzahl der Operatoren zwei Operanden haben, diese Operatoren nennt man *binäre* Operatoren. Ab und zu trifft man auch auf einen Ausreißer, der gleich 3 Operanden besitzt:

```
                      |--------------- ?-Operator
boolean j = ( i > 0 ) ? true : false;
            |       |    |       |---- Operand
            ---------    |------------ Operand
                |--------------------- Operand
```

Operatoren wie zum Beispiel der Auto-Increment- oder Auto-Decrement-Operator verhalten sich unterschiedlich, je nachdem, wo ihr Operand steht:

```
i++ // Der Operator erhöht nach der Verwendung von i
    // deren Wert.

++i // Der Operator erhöht vor der Verwendung von i
    // deren Wert
```

Verwendet man mehr als nur einen Operator in Ausdrücken, muss man wissen, in welcher Reihenfolge etwas berechnet wird, man muss also die Prioritäten der Operatoren kennen (oder Klammern setzen):

```
+ 5 * 2 = 13, nicht etwa 16, weil Punkt vor Strich.
( 3 + 5 ) * 2 = 16, weil wir Klammern verwenden.
```

Java arbeitet mehrere Operatoren grundsätzlich von links nach rechts ab, wenn diese gleiche Prioritäten besitzen (Ausnahme: Zuweisungs-Operatoren, diese werden von rechts nach links bearbeitet). Sind die Prioritäten der Operatoren in einem Ausdruck unterschiedlich, dann wird der Operator mit der höchsten Priorität zuerst verarbeitet. Nur deshalb dürfen wir zum Beispiel folgenden Programmcode ohne Überraschungen schreiben:

```
int i = 3 + 2;
```

Der Additions-Operator hat eine höhere Priorität als der Zuweisungs-Operator, deshalb wird zunächst die Summe berechnet, anschließend wird das Ergebnis der Addition an die Variable `i` zugewiesen. Wäre es umgekehrt, dann würde zunächst der Wert `3` unserer Variablen zugewiesen und anschließend dieser Wert mit `2` addiert, ohne die Variable `i` erneut zu verändern. Das Ergebnis der Addition (`5`) schließlich würde im Nirwana enden.

So, nun wissen wir, dass Operatoren doch nicht ganz so trivial sind, wie man meinen könnte, es gibt durchaus den einen oder anderen Fallstrick, den man sich vor allem selbst durch entsprechende Programmierung legen kann, ohne es zu bemerken.

An dieser Stelle möchte ich Sie eindringlich bitten, dass Sie immer dann Klammern setzen, wenn mehrere Operatoren beteiligt sind und die Funktion des Ausdrucks aufgrund unterschiedlicher Prioritäten nicht sofort ersichtlich ist. Machen Sie sich außerdem ruhig die Mühe, zwischen Operatoren und Operanden Leerzeichen zu setzen, der Code wird dadurch wesentlich verständlicher. Ich habe schon häufig Programmcode in der folgenden Art gesehen:

```
double d=x+y/z*a;
```

Der Ausdruck ist schlicht unverständlich, da man ihn aufgrund der kompakten Schreibweise ohne Leerzeichen schlecht lesen kann, außerdem wurde die Kenntnis der Prioritäten vorausgesetzt. Ich hätte diesen Ausdruck so geschrieben:

```
double d = x + ( y / z * a );
```

Hier sind zwar die Klammern eigentlich überflüssig, machen aber dem Leser des Programms unmissverständlich deutlich, was gemeint ist, ohne dass er in einem Buch die Prioritäten nachlesen muss.

So viel zu den Grundlagen. Kommen wir nun zu den von Java angebotenen Operatoren selbst. Ich habe die einzelnen Operatoren in Funktionsgruppen eingeteilt, die wir nun nach und nach besprechen wollen.

2.10.1 Arithmetische Operatoren

Arithmetische Operatoren werden zur Verarbeitung von Zahlen verwendet (sowohl ganze Zahlen als auch Gleitkomma-Zahlen). Ich glaube, hier kann ich mich auf eine Übersicht der vorhandenen Operatoren beschränken:

Operator	Name	Bedeutung
+	Positives Vorzeichen	Unärer Operator, der den rechts von ihm stehenden Operanden unverändert evaluiert.
-	Negatives Vorzeichen	Unärer Operator, der das Vorzeichen des rechts stehenden Operanden umkehrt.
+	Addition	Binärer Operator für die Summen-Bildung
-	Subtraktion	Binärer Operator für die Differenz-Bildung
*	Multiplikation	Binärer Operator für die Produkt-Bildung
/	Division	Binärer Operator für die Quotient-Bildung
%	Modulo-Division (Rest)	Binärer Operator für die Bildung des ganzzahligen Rests einer Division. `5 % 3` liefert also 2, weil dies der Rest aus der Division ist.

Tabelle 2.3: Arithmetische Operatoren

Operator	Name	Bedeutung
++	Auto-Increment	Unärer Operator, der den Wert des rechts oder links stehenden Operanden um 1 erhöht. o++ erhöht den Wert von o nach dessen Verwendung (Post-Increment), ++o erhöht erst den Wert, dann wird o verwendet (Pre-Increment).
--	Auto-Decrement	Unärer Operator, der den Wert des rechts oder links stehenden Operanden um 1 verkleinert. o-- verringert den Wert von o nach dessen Verwendung (Post-Decrement), --o verringert erst den Wert, dann wird o verwendet (Pre-Decrement).

Tabelle 2.3: Arithmetische Operatoren (Forts.)

2.10.2 Logische Operatoren

Die logischen Operatoren dienen der Verknüpfung von `boolean`-Werten. Ich gehe davon aus, dass ich Sie mit den Grundlagen der booleschen Algebra nur langweilen würde, und verzichte daher auf eine Seiten füllende Beschreibung.

Für die Grundfunktionen UND sowie ODER stehen jeweils zwei Operatoren zur Verfügung, einer mit und einer ohne *Short-Circuit-Evaluierung*. »Was ist das denn?«, werden sich nun die meisten fragen. Hier die Antwort:

```
true ODER x // ergibt immer true, egal, welchen Wert x
            // hat.
false UND x // ergibt immer false, egal, welchen Wert
            // x hat.
```

Bei der *Short-Circuit-Evaluierung* wird der rechte Operand überhaupt nicht mehr ausgewertet, wenn das Endergebnis bereits durch den ersten Operanden feststeht.

In Java kann man mit dem verwendeten Operator wählen, ob man diese schnellere Variante der Short-Circuit-Evaluierung benutzen möchte oder ob auch der rechte Operand evaluiert werden soll.

Operator	Name	Bedeutung
!	NOT	Unärer Operator, der den Operanden logisch negiert
&&	AND Short-Circuit	Binärer Operator, der beide Operatoren mit der UND-Funktion logisch verknüpft. Ist der linke Operator `false`, dann entfällt die Evaluierung des rechten Operanden.
&	AND	Wie vorher, jedoch werden beide Operanden in jedem Fall evaluiert.
||	OR Short-Circuit	Binärer Operator, der beide Operatoren mit der ODER-Funktion logisch verknüpft. Ist der linke Operator `true`, dann entfällt die Evaluierung des rechten Operanden.
|	OR	Wie vorher, jedoch werden beide Operanden in jedem Fall evaluiert.
^	EXOR	Binärer Operator, der beide Operanden mit der Exklusiv-ODER-Funktion logisch verknüpft

Tabelle 2.4: Logische Operatoren

2.10.3 Bitweise Operatoren

Die bisher vorgestellten arithmetischen und logischen Operatoren haben die Operanden jeweils als Zahl bzw. als logische Werte verarbeitet. Java stellt mit den bitweisen Operatoren Verknüpfungen auf Bit-Ebene zur Verfügung, die Operanden werden dabei als Bit-Folge betrachtet und nicht als ganze Einheit.

Operator	Name	Bedeutung
~	Einer-Komplement	Unärer Operator, der alle Bits des Operanden invertiert (umkehrt)
&	AND	Binärer Operator, der eine bitweise UND-Verknüpfung der Operanden durchführt
|	OR	Binärer Operator, der eine bitweise ODER-Verknüpfung der Operanden durchführt
^	EXOR	Binärer Operator, der eine bitweise Exklusiv-ODER-Verknüpfung der Operanden durchführt

Tabelle 2.5: Bitweise Operatoren

Operator	Name	Bedeutung
>>	Rechts-Shift (mit Vorzeichen)	Binärer Operator `a >> b`, der die Bits des Operanden `a` um so viele Stellen nach rechts schiebt, wie im zweiten Operanden `b` angegeben ist. Ist `a` negativ (das höchstwertige Bit von `a` ist gesetzt), dann wird mit jeder Schiebe-Operation ein 1-Bit von links nachgeschoben.
>>>	Rechts-Shift (ohne Vorzeichen)	wie vorher, jedoch wird mit jeder Schiebe-Operation ein `0`-Bit von links nachgeschoben.
<<	Links-Shift	Binärer Operator a << b, der die Bits des Operanden a um so viele Stellen nach links schiebt, wie im zweiten Operanden b angegeben ist. Bei jeder Schiebe-Operation wird ein `0`-Bit von rechts nachgeschoben.

Tabelle 2.5: Bitweise Operatoren (Forts.)

2.10.4 Vergleichs-Operatoren

Vergleichs-Operatoren sind immer binär und vergleichen die Werte der beiden Operanden. Sie liefern einen logischen Wert zurück, der sich aufgrund des Vergleichs ergibt.

Operator	Name	Bedeutung
==	Gleichheit	Evaluiert `true`, wenn beide Operanden gleich sind, ansonsten `false`. Vorsicht bei Referenzen: Es werden nicht die Objekte selbst miteinander verglichen, sondern nur die Referenzen. In diesem Fall bedeutet `true`, dass beide Referenzen auf dasselbe Objekt zeigen.
!=	Ungleichheit	Evaluiert `true`, wenn beide Operanden ungleich sind, ansonsten `false`. Vorsicht bei Referenzen. In diesem Fall bedeutet `true`, dass die Referenzen auf unterschiedliche Objekte zeigen.
<	Kleiner	Evaluiert `true`, wenn der linke Operand kleiner als der rechte ist.
<=	Kleiner gleich	Evaluiert `true`, wenn der linke Operand kleiner ist oder gleich.

Tabelle 2.6: Vergleichs-Operatoren

Operator	Name	Bedeutung
>	Größer	Evaluiert `true`, wenn der linke Operand größer als der rechte ist.
>=	Größer gleich	Evaluiert `true`, wenn der linke Operand größer ist oder gleich.

Tabelle 2.6: Vergleichs-Operatoren (Forts.)

2.10.5 Zuweisungs-Operatoren

Operator	Name	Bedeutung
=	Einfache Zuweisung	`a = b` weist `a` den Wert von `b` zu und evaluiert `b`.
+=	Additions-Zuweisung	`a += b` weist `a` den Wert `a + b` zu und evaluiert `a + b`.
-=	Subtraktions-Zuweisung	`a -= b` weist `a` den Wert `a - b` zu und evaluiert `a - b`.
*=	Multiplikations-Zuweisung	`a *= b` weist `a` den Wert `a * b` zu und evaluiert `a * b`.
/=	Divisions-Zuweisung	`a /= b` weist `a` den Wert `a / b` zu und evaluiert `a /b`.
%=	Modulo-Zuweisung	`a %= b` weist `a` den Wert `a % b` zu und evaluiert `a % b`.
&=	AND-Zuweisung	`a &= b` weist `a` den Wert `a & b` zu und evaluiert `a & b`.
|=	OR-Zuweisung	`a |= b` weist `a` den Wert `a | b` zu und evaluiert `a | b`.
^=	EXOR-Zuweisung	`a ^= b` weist `a` den Wert `a ^ b` zu und evaluiert `a ^ b`.
<<=	Links-Shift-Zuweisung	`a <<= b` weist `a` den Wert `a << b` zu und evaluiert `a << b`.
>>=	Rechts-Shift-Zuweisung mit Vorzeichen	`a >>= b` weist `a` den Wert `a >> b` zu und evaluiert `a >> b`.
>>>=	Rechts-Shift-Zuweisung ohne Vorzeichen	`a >>>= b` weist `a` den Wert `a >>> b` zu und evaluiert `a >>> b`.

Tabelle 2.7: Zuweisungs-Operatoren

2.10.6 Sonstige Operatoren

Operator	Name	Bedeutung
(*params*)	Liste	Komma-separierte Liste von Parametern
(*type*)	Casting	Typ-Konvertierung
`?:`	Konditional	Abkürzung für if-else
`[]`	Array	Kennzeichen für ein Array
`.`	Punkt	Trennung von Packages und Klassen-Methoden bzw. Attributen
`New`	new	Instanzierung neuer Objekte
`Instanceof`	instanceof	Klassen-Abfrage

Tabelle 2.8: Sonstige Operatoren

Einige Operatoren fallen ein wenig aus der Reihe, deshalb wurden sie in einer eigenen Gruppe zusammengefasst. An dieser Stelle möchte ich nur zwei vorstellen, die Beschreibung der anderen finden Sie in den Kapiteln der jeweiligen Themengebiete.

Der Konditional-Operator ?:

Weiter unten lernen wir die Kontroll-Anweisung `if-else` kennen. Für einfache Abfragen wurde eine Abkürzung entwickelt, die häufig eingesetzt wird, der Fragezeichen-Operator, auch *Konditional-Operator* genannt. Hier die Syntax des Operators:

```
expr ? trueExpr : falseExpr
```

Dies ist der einzige Operator mit drei Operanden. Links vom Fragezeichen muss ein Ausdruck *expr* stehen, der einen `boolean`-Wert zurückliefert (evaluiert). Wenn *expr* `true` evaluiert, dann wird der Ausdruck *trueExpr* links vom Doppelpunkt evaluiert, ansonsten der Ausdruck *falseExpr* rechts vom Doppelpunkt. Ich glaube, hier tut ein Beispiel gut:

```
int i;
...

// Lösung mit einer einfachen if-else-Abfrage
int sign;
```

```
if ( i >= 0 ) sign = 1;
else sign = -1;

// Lösung mit ?:
int sign = ( i >= 0 ) ? 1 : -1;
```

Wie man deutlich sieht, ist der Code mit dem Konditional-Operator wesentlich kürzer. Genau das war auch im Sinne des Erfinders.

Der Punkt-Operator .

Ein Operator, den man gar nicht als solchen wahrnimmt, ist der Punkt-Operator ».«. Wir haben ihn bereits viele Male in den Beispielen verwendet, ohne ein Wort darüber zu verlieren. Auch hier möchte ich nicht viel Zeit damit verschwenden. Der Punkt-Operator wird benutzt, um den Variablen-Namen einer Referenz-Variablen auf ein Objekt vom Namen der aufzurufenden Methode oder des Attributs zu trennen. Bei statischen Methoden und Attributen trennt er den Klassen-Namen vom Methoden- bzw. Attribut-Namen. Hierzu einige Beispiele:

```
// Aufruf der Methode println(), die in der Klasse
// System mit dem Attribut Objekt out zur Verfügung
// gestellt wird.
System.out.println();
|      |   |---------- Instanz-Methode println()
|      |-------------- Attribut out der Klasse System
|---------------------- Klassen-Name

// Wir speichern den Inhalt der lokalen Variablen
// attr (rechts vom Gleichheitszeichen) in der
// Instanz-Variablen attr ab, auf die wir über die
// vordefinierte Variable this zugreifen.
this.attr = attr;
|    |------------ Attribut attr der Variablen this
|----------------- spezielle Instanz-Variable
```

Der instanceof-Operator

Mit dem unären Operator `instanceof` kann man den Datentyp einer Referenz-Variable dynamisch zur Laufzeit abfragen. Ein Beispiel hierfür finden Sie im Kapitel *Objektorientierte Programmierung*.

2.10.7 Operator-Prioritäten

Gruppe	Operatoren
Postfix-Operatoren	`[]`, `.`, `(` *params* `)`, *expr*`++`, *expr*`--`
Unäre Operatoren	`++`*expr*, `--`*expr*, `+`*expr*, `-`*expr*, `~`, `!`
Spezielle Objekt-Operatoren	`new`, `(` *type* `)`
Multiplikative Operatoren	`*`, `/`, `%`
Additive Operatoren	`+`, `-`
Shift-Operatoren	`<<`, `>>`, `>>>`
Vergleichs-Operatoren	`<`, `>`, `<=`, `>=`, `instanceof` `==`, `!=`
Bitweises AND	`&`
Bitweises EXOR	`^`
Bitweises OR	`|`
Logisches AND	`&&`
Logisches EXOR	`^`
Logisches OR	`||`
Konditional	`?:`
Zuweisung	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `>>>=`

Tabelle 2.9: Operator-Prioritäten

2.11 Expressions

Wenn Sie Konstanten und Variablen mit Operatoren verknüpfen oder Methoden aufrufen, dann erhalten Sie einen Ausdruck, den man im Englischen *Expression* nennt. Solche Expressions bilden eine Programmeinheit. Mehrere solcher Einheiten wiederum gruppieren sich zu Anweisungen (*Statements*), die dann als ausführbarer Programmcode in der CPU abgearbeitet werden.

Definition einer Expression

Unter einer Expression versteht man eine Serie von Konstanten, Variablen, Operatoren und Methoden-Aufrufen, die einen einfachen Rückgabe-Wert evaluiert.

Beispiele für Expressions:

```
7
3 + 4
i = j + k
Integer.parseInt( "134" )
```

2.12 Statements

Wenn man eine oder mehrere Expressions mit einem Semikolon abschließt, dann erhält man eine Anweisung, die im Englischen als *Statement* bezeichnet wird. Ein Statement ist die kleinste Programmeinheit, die in der Regel als Ganzes von der CPU in einem Stück ausgeführt wird. Man unterscheidet folgende Arten von Statements:

2.12.1 Ausdrucks-Statements

Diese Art von Statements, die man im Englischen als *Expression Statements* bezeichnet, besteht aus Expressions der folgenden Art:

- Zuweisungs-Ausdrücke
- Auto-Increment oder Auto-Decrement-Ausdrücke
- Methoden-Aufrufe
- Instanzierungen von Objekten

Daneben gibt es noch den Sonderfall einer leeren Anweisung, die nur aus einem Semikolon besteht. Man benötigt dieses Statement in seltenen Fällen, wenn die Sprach-Syntax eine Anweisung vorschreibt, das Programm aber gar nichts tun soll.

Hier einige Beispiele für Expression-Statements:

```
i = 5;
i++;
System.out.println( i );
```

2.12.2 Deklarations-Statements

Immer dann, wenn Sie eine Variable deklarieren (und ggf. initialisieren), haben Sie ein Deklarations-Statement verwendet. Hier ein paar Beispiele:

```
int i;
double d;
String s = new String( "bla" );
```

Das letzte Deklarations-Statement ist übrigens gleichzeitig auch ein Ausdruck-Statement.

2.12.3 Kontroll-Statements

Kontroll-Statements dienen der Steuerung des Programmflusses. Die folgende Tabelle gibt Ihnen einen Überblick der in Java erhältlichen Kontroll-Statements:

Statement-Typ	verfügbare Statements
Abfragen	`if-else, switch-case`
Exceptions	`try-catch-finally, throw`
Schleifen	`for, while, do-while`
Sprünge	`break, continue, label:, return`

Tabelle 2.10: Kontroll-Statements in Java

Bevor wir nun zu den einzelnen Kontroll-Statements kommen, möchte ich Ihnen die allgemeine Definition eines Statements nicht vorenthalten:

Definition eines Statements

```
// einfaches Statement
// Alle Angaben in eckigen Klammern sind optional und
// können daher auch entfallen.
expr[ expr ...];

// zusammengesetztes Statement (compound statement)
// Alle Angaben in eckigen Klammern sind optional und
// können daher auch entfallen.
{
    statement[
    statement
    ...]
}
```

Wie wir sehen, gibt es zwei verschiedene Arten von Statements, einfache, die nur aus einer Reihe von Ausdrücken (Expressions) bestehen, die mit einem Semikolon abgeschlossen werden, und zusammengesetzte (*compound statements*), die mit geschweiften Klammern gekennzeichnet werden und eine beliebige Anzahl von Statements enthalten können (die ihrerseits wiederum zusammengesetzte Statements sein dürfen).

Zu Deutsch bedeutet dies: Ein einfaches Statement ist eine einzelne Anweisung, abgeschlossen durch ein Semikolon. Ein zusammengesetztes Statement besteht aus mehreren Einzel-Statements, die in geschweifte Klammern gesetzt werden.

if-else

Das `if-else`-Statement ist eines der am häufigsten benutzten Mittel, um zur Laufzeit Programmverzweigungen durchzuführen. Sehen wir uns die Syntax an (Angaben in eckigen Klammern sind optional und können daher auch entfallen):

```
if ( expr ) statement1[
else statement2]
```

Der Ausdruck *expr* wird zunächst evaluiert und muss einen `boolean`-Wert zurückliefern. Ist das Ergebnis `true`, dann wird *statement1* ausgeführt. Falls nicht, dann kann man im `else`-Zweig der Abfrage Programmcode hinterlegen, der dann ausgeführt wird, wenn die `if`-Abfrage `false` evaluiert hat.

So weit zur unverdaulichen Theorie. Wollen wir nun das Ganze anhand von Beispielen mit Leben füllen:

```
...
int i = 3;
int k = 4;

if ( i == k ) System.out.println( "Gleichheit" );
if ( i < k ) System.out.println( "i < k" );
if ( i > k ) System.out.println( "i > k" );

// dasselbe, aber anders programmiert
if ( i == k ) {
    System.out.println( "Gleichheit" );
}
else {
    if ( i < k ) {
```

```
        System.out.println( "i < k" );
    }
    else {
        System.out.println( "i > k" );
    }
}
```

Wie immer führen viele Wege nach Rom. Ich persönlich verwende grundsätzlich keine einfachen Statements in `if-else`-Abfragen, sondern zusammengesetzte Statements mit geschweiften Klammern (Ausnahmen können in diesem Buch durchaus vorkommen, um Platz zu sparen). Der Grund ist ganz einfach. Meist fügt man zu einem späteren Zeitpunkt weitere Statements im `if`-Block oder im `else`-Block hinzu. In diesem Fall muss man jedoch ein zusammengesetztes Statement mit geschweiften Klammern bilden, da dies bei mehreren Statements notwendig ist. Häufig vergisst man dies aber, und schon hat man sich selbst eine Falle gestellt:

```
// Ursprünglicher Code
if ( i > 10 )
    j = 1;

// Veränderter Code
if ( i > 10 )
    j = 1;
    System.out.println( "i > 10" );
```

Wir haben im Laufe der Zeit ein Debug-Statement hinzugefügt und damit den `if`-Block um ein Statement erweitert. Dieses wird aber nicht nur dann ausgeführt, wenn die Variable `i` größer als `10` ist, sondern immer, unabhängig vom aktuellen Wert von `i`. Damit die Sache funktioniert, müssen wir geschweifte Klammern setzen:

```
// Ursprünglicher Code
if ( i > 10 ) {
    j = 1;
}

// Veränderter Code
if ( i > 10 ) {
    j = 1;
    System.out.println( "i > 10" );
}
```

switch-case

Lassen Sie mich es deutlich ausdrücken: Ich halte nicht viel vom `switch`-Statement. Es gibt wohl nur sehr seltene Fälle, in denen man dieses Statement wirklich benötigt. Als Fingerzeig gilt: Braucht man es öfter, dann hat man falsch programmiert. Aber sei's drum, hier die Syntax (wie immer sind alle Angaben in eckigen Klammern optional und können daher entfallen):

```
switch ( expr ) {
    case constant1:
        statement
        ...[
        break;]

    case constant2:
        ...[
        break;]
    ...[

    default:
        statement
        ...]
}
```

Der Ausdruck *expr* wird nacheinander mit den Konstanten in den `case`-Zeilen verglichen. Ist *expr* mit einer Konstanten identisch, dann wird der Programmcode dieses `case`-Blocks (ausnahmsweise dürfen hier die geschweiften Klammern fehlen) ausgeführt. Am Ende jedes `case`-Blocks kann man optional mit der `break`-Anweisung dafür sorgen, dass das `switch`-Statement verlassen wird. Fehlt die `break`-Anweisung, dann fährt die Virtual Machine mit dem nächsten `case`-Block fort. Ebenfalls optional kann die `switch`-Anweisung einen `default`-Block enthalten, der in jedem Fall ausgeführt wird, wenn vorher nicht mit `break` das `switch`-Statement verlassen wurde.

Aus der Sicht der Programmlogik ist die `switch`-Anweisung nichts anderes als eine geschachtelte `if`-Abfrage. Sie hat allerdings einen gravierenden Nachteil, denn nach dem reservierten Wort `case` dürfen nur Konstanten stehen, keine Variablen. Zudem müssen die Konstanten und der Ausdruck `expr`, der nach `switch` steht, kompatibel zueinander sein. Damit Sie sehen, was ich von diesem Statement halte, verzichte ich hier auf ein Beispiel.

for-Schleife

Ein sehr häufig benutztes Vehikel in der Programmierung sind Schleifen, in denen Programmcode mehr als nur einmal ausgeführt werden kann. Ein wichtiger Vertreter dieser Gattung ist die `for`-Schleife, die meist dann verwendet wird, wenn die Anzahl der Schleifen-Durchläufe vorher bekannt ist (obwohl sehr viele Programmierer sie für Endlos-Schleifen benutzen). Hier ist die Syntax der `for`-Schleife (Angaben in eckigen Klammern sind optional und können entfallen):

```
for ( [init]; [condition]; [postProcess] )
    statement
```

Eine `for`-Schleife besteht aus einem Kopf-Teil und einem Rumpf-Teil. Im Schleifen-Rumpf steht der Programmcode, der x-mal ausgeführt werden soll, er kann beliebig viele zusammengesetzte Statements enthalten. Der Schleifen-Kopf besteht aus dem reservierten Wort `for` und drei weiteren Bestandteilen, die durch runde Klammern umrahmt sind. Der *init*-Teil wird vor dem Eintritt in den ersten Schleifen-Durchlauf einmalig ausgeführt und dient meist der Initialisierung von Schleifen-Variablen. Es dürfen mehrere Variablen initialisiert werden, jeweils durch ein Komma getrennt. Allerdings müssen alle Variablen im Datentyp zueinander kompatibel sein.

condition ist ein Ausdruck, der einen `boolean`-Wert zurückliefern muss und eine Abfrage darstellt, ob der Programmcode im Schleifen-Rumpf ausgeführt werden soll. *condition* wird jedes Mal ausgeführt, bevor ein neuer Schleifen-Durchlauf beginnt. Ergibt die Abfrage `false`, dann wird die `for`-Schleife beendet. Der *postProcess*-Teil schließlich wird jedes Mal am Ende eines Schleifen-Durchlaufs ausgeführt, also immer nach dem letzten Statement des Schleifen-Rumpfs. Er kann auch leer sein. Man darf mehrere Ausdrücke durch Komma getrennt angeben.

Ich glaube, ein paar Beispiele für die `for`-Schleife tun gut (Das in manchen Beispielen verwendete `break`-Statement lernen wir gleich kennen.):

```
// for-Schleife, die 10-mal durchlaufen wird
for ( int i = 0; i < 10; i++ ) {
    System.out.println( "i = " + i );
}

// dasselbe, diesmal jedoch ist der Initialisierungsteil
// leer.
int i = 0;
```

```
for ( ; i < 10; i++ ) {
    System.out.println( "i = " + i );
}

// Im nächsten Beispiel wird der Schleifen-Zähler
// im Schleifen-Rumpf erhöht.
for ( int i = 0; i < 10; ) {
    System.out.println( "i = " + i );
    i += 2;
}

// Nun lassen wir den Abfrage-Teil weg und verlassen
// die Schleife aus dem Schleifen-Rumpf heraus.
for ( int i = 0; ; i++ ) {
    if ( i >= 10 ) break;
    System.out.println( "i = " + i );
}

// Typische Endlos-Schleife
for ( ; ; ; ) {
    // nicht vergessen: Die Schleife muss per
    // Programmlogik aus dem Schleifen-Rumpf heraus
    // abgebrochen werden, da man sonst tatsächlich
    // eine endlos ablaufende Schleife gebaut hat.
    if ( ... ) break;
}

// Ausgabe eines zweidimensionalen Arrays mit
// geschachtelten for-Schleifen
int[][] ar = new int[ 5 ][ 5 ];
for ( int i = 0; i < ar.length; i++ ) {
    for ( int j = 0; j < ar[ i ].length; j++ ) {
        System.out.println(
            "ar[ " + i + " ][ " + j + " ] = " +
            ar[ i ][ j ]
        );
    }
}

// Verwendung mehrerer Schleifen-Variablen
for ( int i = 0, j = i + 1; i < 5; i++, j += 2 ) {
    System.out.println( "i = " + i + ", j = " + j );
}
```

```
// Bei der Initialisierung der Schleifen-Variablen
// darf der Datentyp nur einmal angegeben werden,
// alle verwendeten Variablen müssen also denselben
// Datentyp haben.
```

while-Schleife

Neben der `for`-Schleife wird häufig die `while`-Schleife benutzt, und zwar immer dann, wenn die Anzahl der Schleifen-Durchläufe nicht von vornherein bekannt ist und sich das Abbruch-Kriterium somit erst im Schleifen-Rumpf ergibt. Die `while`-Schleife hat folgende Syntax:

```
while ( expr ) statement
```

expr dient demselben Zweck wie bei der `for`-Schleife, nämlich vor jedem Eintritt in den Schleifen-Rumpf zu prüfen, ob dieser überhaupt ausgeführt werden soll. Der Schleifen-Rumpf selbst besteht wiederum aus beliebig vielen zusammengesetzten Statements (oder auch aus einem einzelnen einfachen Statement). Zur Veranschaulichung wieder ein paar Beispiele:

```
int i = 10;
while ( i > 0 ) {
    System.out.println( "i = " + i );
    i--;
}

// typische Endlos-Schleife
int i = 0;
while ( true ) {
    // nicht vergessen: Irgendwann muss die Schleife
    // aus dem Schleifen-Rumpf heraus beendet werden.
    if ( i == 10 ) break;
    System.out.println( "i = " + i );
    i++;
}
```

Ein wesentlicher Unterschied zur `for`-Schleife ist, dass ein Schleifen-Zähler (wenn vorhanden) im Schleifen-Rumpf explizit erhöht (oder verringert) werden muss, da dies nicht im Schleifen-Kopf möglich ist.

do-Schleife

Die do-Schleife arbeitet fast genauso wie die while-Schleife, jedoch wird nicht vor Eintritt in den Schleifen-Rumpf überprüft, ob dieser ausgeführt werden soll, sondern am Ende des Schleifen-Rumpfs (nach der letzten Anweisung im Schleifen-Rumpf). Das hat zur Folge, dass bei der do-Schleife die Statements des Schleifen-Rumpfs mindestens einmal ausgeführt werden. Syntax der do-Schleife:

```
do statement
while ( expr );
```

Wiederum besteht der Schleifen-Rumpf meist aus zusammengesetzten Anweisungen, die in geschweifte Klammern gesetzt werden.

Achtung Das Semikolon am Ende der do-Schleife ist kein Tippfehler meinerseits, sondern zwingend vorgeschrieben.

break und continue

Wie wir in den Beispielen für Schleifen bereits gesehen haben, ist es manchmal nötig, aus dem Schleifen-Rumpf heraus die Entscheidung zu treffen, ob die Schleife beendet werden oder sofort der nächste Schleifen-Durchlauf begonnen werden soll. Hierfür stellt Java zwei Statements zur Verfügung: break und continue.

Die Bedeutung der reservierten Wörter ist eindeutig, break führt zum sofortigen Abbruch einer Schleife, continue zum sofortigen Beenden des aktuellen Schleifen-Rumpfs und zum Eintritt in den nächsten Durchlauf, ohne dass die folgenden Anweisungen des Schleifen-Rumpfs ausgeführt werden. Vorher wird bei for-Schleifen natürlich der Programmcode der *PostProcess*-Ausdrücke und die Abbruch-Bedingung ausgeführt. Das break-Statement haben wir ja bereits bei den Endlos-Schleifen kennen gelernt. Hier ein Beispiel für das continue-Statement:

```
// Ausgabe aller ungeraden Zahlen
for ( int i = 0; i < 10; i++ ) {
    // nächsten Schleifen-Durchlauf beginnen, wenn
    // der Zähler eine gerade Zahl enthält
    if ( ( i % 2 ) == 0 ) continue;

    System.out.println( "i = " + i );
}
```

Achtung Die Statements `break` und `continue` dürfen nur innerhalb von Schleifen verwendet werden. Beide Statements wirken sich ohne weitere Angaben nur auf die den Schleifen-Rumpf umgebende Schleife aus.

Schleifen-Labels

Bei geschachtelten Schleifen möchte man des Öfteren eine weiter außen liegende Schleife beenden, obwohl man sich in einer inneren Schleife befindet:

```
// Die folgende geschachtelte Schleife soll beendet
// werden, wenn das Produkt von i und j größer als 50
// ist.
for ( int i = 0; i < 10; i++ )
    for ( int j = i + 1; j < 10; j++ ) {
        if ( ( i * j ) > 50 ) break;
    }
```

Wenn wir den Programmcode ausführen, erzielen wir leider nicht das gewünschte Ergebnis, denn die `break`-Anweisung führt nur zum Abbruch der inneren Schleife, wir wollen aber die äußere Schleife beenden.

Zu diesem Zweck bietet Java die Möglichkeit, eine Schleife mit einem Namen, dem so genannten *Label*, zu versehen. Damit kann man gezielt eine bestimmte Schleife entweder beenden (`break`) oder mit deren nächstem Schleifen-Durchlauf fortfahren (`continue`). Die Syntax eines Labels ist wie folgt:

```
label: Schleife
```

Um die so gekennzeichnete Schleife anzusprechen, gibt man nach dem `break`- bzw. `continue`-Statement deren Name *label* an:

```
loop1:
for ( int i = 0; i < 10; i++ ) {
    for ( int j = i + 1; j < 10; j++ ) {
        if ( ( i * j ) > 50 ) break loop1;
    }
}
```

return

Die `return`-Anweisung gehört ebenfalls zu den Kontroll-Statements und wird in Methoden benutzt, um die aktuelle Methode sofort zu verlassen, womit der Programmfluss wieder zum aufrufenden Programmcode zurückkehrt. Es sind zwei Varianten möglich:

```
return;
return expr;
```

Die erste Variante wird in Methoden vom Typ `void` verwendet, während bei der zweiten Variante dem aufrufenden Programmteil ein Rückgabewert *expr* übergeben wird. Dieser muss kompatibel zum Datentyp der Methode sein. Mehr hierzu sehen Sie im Abschnitt über Methoden.

assert

Das Kontroll-Statement `assert` wurde mit dem JDK 1.4 neu eingeführt. Man benutzt es, um ein Programm mit einem harten Fehler abzubrechen, wenn bestimmte Annahmen über den Zustand von Variablen oder Programmzuständen nicht erfüllt sind.

Hinweis Falls Sie noch ein Neuling in Java sind, können Sie die Beschreibung des `assert`-Statements zunächst überspringen, denn für das Verständnis der Anweisung sind einige Dinge Voraussetzung, die erst in den folgenden Kapiteln erläutert werden.

Die Syntax des `assert`-Statements ist wie folgt:

```
assert expr;
assert expr : msg;
```

expr muss ein Ausdruck sein, der einen `boolean`-Wert evaluiert. Solange *expr* einen `true`-Wert evaluiert, passiert überhaupt nichts, das Programm verhält sich so, als wäre das `assert`-Statement gar nicht vorhanden. In dem Moment, wo *expr* einen `false`-Wert evaluiert, löst die Virtual Machine einen `AssertionError` aus. Das ist eine besondere Ausnahme, die nicht von der Klasse `java.lang.Exception` abgeleitet wird, sondern von `java.lang.Error`.

In der zweiten Variante kann man mit *msg* einen Ausdruck angeben (meist ein String), der Zusatz-Informationen über den Fehler enthält. Aus technischer Sicht wird *msg* im Konstruktor von `java.lang.AssertionError` übergeben.

Wenn Sie sich den Text genau durchgelesen haben, stellen Sie vielleicht fest, dass man dasselbe Verhalten auch mit normalen Abfragen und Exceptions erreichen kann. Assertions besitzen jedoch den Vorteil, dass sie zur Laufzeit über ein Kommandozeilen-Argument der Virtual Machine ein- oder ausgeschaltet werden können. Standardmäßig sind sie abgeschaltet. In diesem Fall werden die `assert`-Statements von der Virtual Machine überhaupt nicht beachtet. Auch läuft das Programm damit etwas schneller, als hätte man statt Assertions normale Abfragen eingebaut.

Damit Sie auch in der Praxis etwas mit dem Begriff *Assertions* anfangen können, möchte ich die Wirkungsweise anhand eines Beispiels verdeutlichen:

```
01 import java.util.*;
02
03 public class AssertTest {
04     public static void main( String[] args ) {
05         MyClass cl = new MyClass();
06         for ( int i = 0; i < 10; i++ ) {
07             String s = "Item " + i;
08             if ( i == 9 ) {
09                 s = null;
10             }
11
12             cl.addItem( s );
13         }
14     }
15 }
16
17 class MyClass {
18     private Vector items = new Vector();
19     private int maxItems = 50;
20     private int nItems = 0;
21
22     public void addItem( Object o ) {
23         assert nItems < maxItems : "Speicher voll";
24         assert o != null : "NULL object";
25
26         items.addElement( o );
27         nItems++;
28     }
29 }
```

Das Hauptprogramm benutzt eine einfache Klasse `MyClass`, um Objekte in einem Speicher abzulegen. Hierzu habe ich eine Methode `addItem()` implementiert. Sie verwendet das `assert`-Statement, um sicherzustellen, dass die maximale Anzahl von Elementen nicht überschritten werden kann (Zeile 23) und kein `null`-Objekt aufgenommen wird (Zeile 24). In der abgebildeten Version des Programms wird ein Laufzeitfehler aufgrund eines `null`-Objekts ausgelöst. Wenn Sie in Zeile 19 den Wert für die maximale Anzahl von Elementen auf einen Wert ändern, der kleiner als 10 ist, erhalten Sie eine Assertion mit dem Text "`Speicher voll`".

Damit Sie das Programm übersetzen können, benötigen Sie in jedem Fall mindestens die Version 1.4.0 des JDK und müssen dem Compiler mit dem Kommandozeilen-Argument »`-source 1.4`« mitteilen, dass er Assertions unterstützen soll. Tun Sie das nicht, dann ernten Sie eine Fehlermeldung. Hier die Kommandozeile für den Compiler-Lauf:

```
C:\temp>javac -source 1.4 AssertTest.java
```

Damit Assertions zur Laufzeit unterstützt werden, müssen sie dies der Virtual Machine über den Schalter »`-ea`« mitteilen:

```
C:\temp>java -ea AssertTest
```

Probieren Sie sowohl die Variante mit als auch ohne Schalter »`-ea`« aus und vergleichen Sie die Resultate.

Man hätte alternativ auch normale Exceptions verwenden können:

```
01 import java.util.*;
02
03 public class AssertTest1 {
04     public static void main( String[] args ) {
05         MyClass cl = new MyClass();
06         for ( int i = 0; i < 10; i++ ) {
07             String s = "Item " + i;
08             if ( i == 9 ) {
09                 s = null;
10             }
11
```

```
12              try {
13                  cl.addItem( s );
14              } catch ( Exception ex ) {
15                  System.err.println( ex );
16                  System.exit( 1 );
17              }
18          }
19      }
20  }
21
22  class MyClass {
23      private Vector items = new Vector();
24      private int maxItems = 50;
25      private int nItems = 0;
26
27      public void addItem( Object o ) throws Exception
28      {
29          if ( nItems >= maxItems )
30              throw new Exception( "Speicher voll" );
31          if ( o == null )
32              throw new Exception( "NULL object" );
33          items.addElement( o );
34          nItems++;
35      }
36  }
```

Der große Unterschied zum Beispiel mit Assertions besteht darin, dass der Programmcode für die Überprüfung immer ausgeführt wird und nicht abgeschaltet werden kann.

Achtung Assertions sind aus anderen Programmiersprachen bekannte Konstrukte, die harte Laufzeitfehler produzieren, wenn sich aufgrund von Programmier-Fehlern inkonsistente Programmzustände ergeben. Auch in der JDK-Version 1.0 war dieses Feature vorgesehen, wurde jedoch aus zeitlichen Gründen zurückgestellt und erst mit der Version 1.4 ins JDK integriert.

Das bedeutet natürlich, dass auf jedem Rechner, auf dem Java-Programme mit Assertions laufen sollen, mindestens das JDK 1.4 installiert sein muss. Außerdem sind Assertions standardmäßig ausgeschaltet. Um sie zu aktivieren, muss die Virtual Machine mit dem Kommandozeilen-Argument »`-ea`« gestartet werden (die Lang-Form dieses Schalters ist »`-enableassertions`«).

Auch dem Compiler muss explizit über das Kommandozeilen-Argument »`-source 1.4`« gesagt werden, dass er Assertions berücksichtigen soll. Vergisst man dies, dann erntet man einen Compiler-Fehler, der auf höfliche Art mitteilt, dass `assert` seit Version 1.4 ein reserviertes Wort ist.

Da es erfahrungsgemäß weit mehr als ein Jahr dauert, bis sich eine neue Java-Version etabliert hat, sollte man Assertions derzeit gezielt nur dann einsetzen, wenn klar ist, dass dieses Feature auch auf allen benutzten Rechnern unterstützt wird.

2.13 Methoden

Ich möchte hier nicht in die Details von Funktionen einsteigen, da ich davon ausgehe, dass Sie wissen, was eine Funktion oder was der Unterschied zwischen formalen Parametern und lokalen Variablen ist. Vielmehr will ich Ihnen mit diesem Abschnitt zeigen, worauf man beim Entwurf und bei der Implementierung von Methoden (so heißen Funktionen in Java) achten muss.

Ein paar grundlegende Dinge sollten aber zum gemeinsamen Verständnis nicht fehlen, deshalb zunächst zur allgemeinen Deklaration einer Methode (alle Angaben in eckigen Klammern sind optional und können entfallen):

```
[perms][ static][ final] type name([ params ])[
    throws exList]
{
    Methoden-Rumpf[

    return expr;]
}
```

perms kann, falls vorhanden, einer der drei folgenden Strings sein: `public`, `private` oder `protected`. Was die einzelnen Strings bedeuten, werde ich im Kapitel *Objektorientierte Programmierung* beschreiben. Ebenso die Bedeutung der reservierten Wörter `static` und `final`. Das Schlüsselwort `throws` ist im Kapitel *Exceptions* näher erläutert.

Hier von Bedeutung sind *type*, mit dem man den Datentyp (und damit den `return`-Value) der Methode festlegt, sowie *params*, mit dem die formalen Parameter der Methode als Komma-separierte Liste angegeben werden.

2.13.1 Der Datentyp von Methoden

In Java muss jede Methode mit einem festgelegten Datentyp deklariert werden, der auch den `return`-Wert bestimmt. Dieser muss entweder einer der acht primitiven Datentypen sein oder der Klassen-Name eines Objekts, das die Methode zurückgibt. Hat eine Methode keinen `return`-Value, der an den Aufrufer übergeben wird, dann muss der spezielle Datentyp `void` angegeben werden. In jedem Fall muss der Rückgabe-Wert, den die Methode an den aufrufenden Programmcode übergibt, kompatibel zu dem in der Deklaration angegebenen sein. Falls der Datentyp `void` ist, gibt die Methode gar keinen `return`-Value zurück, sondern beendet sich einfach dadurch, dass entweder das letzte Statement des Methoden-Rumpfs ausgeführt wurde, oder durch ein leeres `return`-Statement ohne Argument. Hierzu ein paar Beispiele:

```
// Methode vom Typ int
// Es muss zwingend ein return-Statement vorhanden
// sein, mit dem ebenfalls ein int-Wert an den
// Aufrufer zurückgegeben wird.
public int sum( int a, int b ) {
    return a + b;
}

// Methode, die keinen return-Value besitzt, sondern
// einfach irgendetwas tut.
public void pr( String msg ) {
    System.out.println( msg );
}

// dasselbe, aber diesmal wird ein return-Statement
// verwendet, um den Programmcode zu vereinfachen.
// Normalerweise hätte man eine geschachtelte
// if-Abfrage implementieren müssen, wenn man
// der Maxime folgt, dass eine Methode grundsätzlich
// nur einen Eingang und nur einen Ausgang haben soll.
public void countNumbers( int num ) {
    if ( num < 0 ) return;

    if ( num < 10 ) {
        ones++;
        return;
    }

    if ( num < 100 ) {
        tens++;
```

```
        return;
    }

    if ( num < 1000 ) {
        hundreds++;
        return;
    }
}

// Methode vom Typ String, die eine Instanz-Variable
// mit dem Argument erweitert und dann zurückgibt.
// Das reservierte Wort this wird benötigt, um den
// formalen Parameter msg vom Variablen-Namen des
// Instanz-Attributs unterscheiden zu können.
protected String doLog( String msg ) {
    this.msg += msg;
    return this.msg;
}

// Beispiel, bei dem zwar mit dem return-Statement
// ein anderer Datentyp zurückgegeben wird, als
// im Datentyp der Deklaration angegeben ist, dieser
// aber aufgrund der Kompatibilität vom Compiler
// akzeptiert wird, weil die Klasse Vector das
// Interface List implementiert und somit kompatibel
// zu diesem Datentyp ist.
public List add( int[] ar ) {
    Vector v = new Vector();
    for ( int i = 0; i < ar.length; i++ ) {
        v.addElement( new Integer( ar[ i ] ) );
    }

    return v;
}
```

2.13.2 Die Parameterliste von Methoden

In Java werden alle Aufruf-Parameter von Methoden als Kopie übergeben (*call-by-value*), da es keine Zeiger-Variablen oder Adress-Operatoren wie zum Beispiel in C gibt. Was passiert aber nun, wenn einer Methode ein Objekt übergeben wird?

Nun, jedes Objekt wird über eine Referenz-Variable angesprochen, die man beim Aufruf einer Methode als Parameter ganz normal wie andere Variablen auch als Kopie übergibt. Die Referenz-Variable enthält jedoch nicht das Objekt selbst, sondern nur

dessen Adresse im Hauptspeicher. Also wird der Methode beim Aufruf eine Kopie dieser HauptSpeicheradresse übergeben. In der Methode kann man deshalb den Wert der Referenz-Variablen selbst nicht ändern, die übergebene Referenz-Variable behält also in jedem Fall ihren ursprünglichen Wert.

Das Objekt, auf das die Referenz-Variable zeigt, kann aber vom Programmcode der Methode geändert werden, obwohl die Referenz-Variable selbst als Kopie an die Methode übergeben wurde.

Hat Ihnen das weitergeholfen? Wenn nicht, hier ein Beispiel:

```
01 // Datei ArgTest01.java
02
03 // Das Programm zeigt, dass man auf der Hut sein
04 // sollte, wenn man Objekte als Parameter von
05 // Methoden-Aufrufen benutzt.
06 public class ArgTest01 {
07     public static void main( String[] args ) {
08         int[] ar = { 1, 2, 3, };
09         myMethod( ar );
10         printArray( ar );
11     }
12
13     public static void myMethod( int[] ar ) {
14         ar[ 0 ] = 10;
15         ar = null;
16     }
17
18     public static void printArray( int[] ar ) {
19         if ( ar == null ) {
20             System.out.println( "null" );
21             return;
22         }
23
24         for ( int i = 0; i < ar.length; i++ ) {
25             System.out.println(
26                 "ar[ " + i + " ] = " + ar[ i ]
27             );
28         }
29     }
30 }
```

Im Hauptprogramm wird zunächst ein Array aus `int`-Werten definiert, das wir anschließend als Argument für den Methoden-Aufruf benutzen. Nachdem sich die Methode beendet hat, geben wir den Inhalt unseres Arrays aus.

In der Methode `myMethod()` ändern wir das erste Element des übergebenen Arrays, danach löschen wir das Array, indem wir die Referenz auf `null` setzen.

Wenn wir das Programm ausführen, dann erhalten wir folgende Ausgabe:

```
ar[ 0 ] = 10
ar[ 1 ] = 2
ar[ 2 ] = 3
```

Die Ausgabe beweist: Die Methode ist zwar in der Lage, das über die Referenz-Variable übergebene Objekt zu ändern (nicht vergessen: Arrays sind Objekte). Die Referenz-Variable selbst bleibt jedoch unangetastet, weil die Methode ja nur eine Kopie der Variablen erhält. Um auch dies zu ermöglichen, müsste man eine Referenz auf die Referenz übergeben, und das ist in Java aufgrund des fehlenden Adress-Operators nicht möglich.