Java: Sprachmerkmale

Johannes Rössel

2006-11-14

Zusammenfassung

Wie alle Programmiersprachen, so ist auch Java eine, die einer Einführung bedarf. Sicherlich ist es bei der Komplexität nicht einfach, einen sinnvollen Überblick in akzeptabler Kürze zu geben, daher faßt dieses Dokument die wichtigsten Sprachmerkmale zusammen und geht auch mit Beispielen auf die meisten Aspekte von Java ein. Jedoch werden keine Details der Klassenbibliothek oder ähnlichem besprochen, auch bleibt die Einführung in OOP relativ kurz, was hier der oberen Grenze des Umfanges dieser Arbeit geschuldet ist.

Inhaltsverzeichnis

Java

Java ist unter recht bestimmten Gestaltungsaspekten geschaffen worden. Zunächst einmal sollte die Sprache grundsätzlich objektorientiert sein, es gibt an der Stelle zwar Ausnahmen, die historisch durch Performancegründe bedingt sind, aber insgesamt verfolgt Java dieses Paradigma doch recht erfolgreich. Java sollte plattformunabhängig sein, bzw. bestehender Java-Code sollte auf jeder Plattform laufen können, ohne das beständige Neukompilieren, wie es mit Sprachen wie C oder C++ immer nötig ist. Java erfüllt dies, indem es selbst eher seine eigene Plattform darstellt. Man kann Software für Windows schreiben, für Unix oder andere Betriebssysteme ... oder eben für die JVM, die *Java Virtual Machine.*

Java hat aufgrund dieser Tatsache eine beträchtlich größere Standardbibliothek, als das sonst der Fall ist; es bietet praktisch alles, was der Programmierer braucht. Möglichkeiten, grafische Oberflächen zu gestalten, XML zu verarbeiten, Verschlüsselung, Datenbankzugriff und eine ganze Reihe anderer Anwendungsfälle, für die man bei anderen Sprachen meist entweder direkt auf die API des darunterliegenden Betriebssystems oder aber auf Bibliotheken von anderen Quellen zurückgreift. In diesem Sinne kann sich der Programmierer bei Java meist ganz auf das konzentrieren, was er eigentlich schreiben will und nicht auf die Punkte, wo die Sprache nicht mit seinen Wünschen Schritt hält. Viele kennen diese Einstellung zum Programmieren auch von diversen Skriptsprachen, wo eben jenes oft auch ein Grund für deren Beliebtheit ist.

Die Syntax von Java ist größtenteils an C++ angelehnt, was es Programmierern recht einfach machen sollte, auf Java umzusteigen oder die grundlegende Syntax zu erlernen. Allerdings haben Java und C++ unter der Syntax nicht mehr viel miteinander gemein. Javas Objektmodell entstammt nämlich in gewissen Zügen der Sprache Smalltalk und nicht C++.

Smalltalk ist eine objektorientierte Programmiersprache, die 1972 entwickelt wurde und ein völlig anderes Objektmodell als das damals zehn Jahre alte Simula, an das sich C++ und andere Sprachen anlehnen. In Smalltalk haben Objekte keine *Methoden*, die aufgerufen werden können, sondern erhalten *Nachrichten.* Diese müssen nicht implementiert sein, man kann sie auch weiterreichen, was einem eine ganze Reihe interessanter Möglichkeiten eröffnet und die Sprache sehr flexibel macht. Aber der wesentliche Unterschied, der hier auch für Java relevant ist, ist die Tatsache, daß Objekte „wissen“, daß sie Objekte sind.

Das ist auch im Grunde der Unterschied zwischen Java und C++. Objekte in C++ sind lediglich Datenstrukturen, an denen Operationen hängen. Ein Objekt weiß nicht, daß es ein Objekt ist, wie es heißt, wo es in der Klassenhierarchie abgelegt ist, welche Methoden es implementiert, etc. Was in Java Reflection heißt und einem Zugriff auf gewisse Metainformationen von Objekten erlaubt, ist ein Teil dessen, was ein Objekt in Smalltalk ausmacht. Man findet sogar das Nachrichtenmodell in kleinen Ansätzen wieder, auch wenn Java hier bei weitem nicht so flexibel ist, wie Smalltalk und eher das Augenmerk auf Methoden als Möglichkeit der Interaktion mit Objekten legt.

Letztendlich wollte man Java auch einfach genug halten, um die Sprache für möglichst viele Programmierer attraktiv zu machen, dies erreicht man wohl weder mit sehr speziellen Features noch mit der Möglichkeit, die Syntaxregeln bis zum Letzten auszunutzen und Code zu schreiben, den außer dem Autor keiner mehr lesen kann.

Ein weiter Designaspekt von Java war es, dem Programmierer Operatorüberladungen bewußt nicht zu erlauben. Man mag argumentieren, daß Operatorüberladungen prinzipiell eine nette Sache sind und den Code eher lesbarer machen. Beispielsweise kann man ja mal die folgenden beiden Codefragmente vergleichen:

```
MyVector a, b, c;
```

```
a = b + c;
```

sowie

```
MyVector a, b, c;
a = b.add(c);
```

Aussagekräftiger ist wohl sicherlich das erste. Allerdings gibt es auch Beispiele, die deutlich machen, daß überladene Operatoren nicht immer zu besser lesbaren Code führen, ein Beispiel ist hier der Output Stream bei C++:

```
cout << "Hello World!";
```

Stellt sich die Frage, was ein bitweiser Shiftoperator bei Streams und Zeichenfolgen zu suchen hat. Diese Art „Mißbrauch“ von Operatoren wollte man bei Java vermeiden, damit ein Entwickler sich nicht erst durch den halben Code graben muß, um herauszufinden, ob ein Operator nun für zwei Typen überladen ist und in welcher Weise. Nicht immer ist ein + das, was man denkt, was es sein sollte – in Java (von einer kleinen Ausnahme abgesehen) kann man sich darauf verlassen, daß man immer auf die selbe Bedeutung schaut. Im Grunde ist es allerdings eher eine Gewöhnungsfrage.

Hello World!

Das erste, was wohl nahezu jeder in einer neuen Sprache schreibt, ist ein „Hello World“-Programm. Bekanntermaßen tut es nichts weiter, als den Text „Hello World!“ auf dem Bildschirm auszugeben und natürlich geht das auch in Java:

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

Ich werde hier nicht im Detail auf die einzelnen Bestandteile eingehen, das folgt in den kommenden Abschnitten. Das Beispiel dient lediglich ein wenig zur Einleitung.

1 Variablen und ähnliches

1.1 Variablen

Variablen sind in Java Container für zweierlei Dinge. Sie können einerseits direkt einen Wert enthalten, das ist bei primitiven Datentypen der Fall. Ansonsten enthalten sie lediglich Referenzen auf Objekte. Variablendeklarationen erfolgen im Grunde in der folgenden Form:

```
Typ Bezeichner[, Bezeichner[, ...]]
```

Was bedeutet, zunächst kommt ein Typ, gefolgt von einem oder mehreren Bezeichnern, die dann Variablen eben dieses Typs sind. Variablenbezeichner beginnen in Java immer mit einem Kleinbuchstaben.

Beispiele:

```
int i;
char[] blubb;
double x, y;
java.lang.Object o;
String s;
```

Bezeichner in Java berücksichtigen Groß-Klein-Schreibung, daher sind `a` und `A` nicht die gleichen Bezeichner, wie es in einigen Sprachen (wie z. B. Pascal oder BASIC) wäre. Diese Regeln gelten auch für alle Bezeichner in der Sprache, d. h. auch für Klassen- und Methodennamen (anders als beispielsweise in PHP, wo bei Funktionsnamen *nicht* auf Groß-Klein-Schreibung geachtet wird).

1.2 Konstanten

Konstanten werden in Java durch den Deklarationsmodifikator **`final`** eingeleitet, der Rest der Deklaration entspricht der von normalen Variablen. **`final`** sagt dem Compiler, daß der Wert dieser Variable sich nicht mehr ändert, also ähnlich wie **`const`** in C oder C++, allerdings ist es in Java nicht möglich, diese Variablen trotz der Deklaration doch noch zu ändern (wie beispielsweise in C++ über **`const_cast`**<> möglich ist).

Beispiele:

```
final double pi = 3.14;
final String foo = "bar";
```

2 Datentypen

Zunächst gibt es im Grunde zwei verschiedene Arten von Typen, die im vorherigen Abschnitt schon kurz angesprochen wurden: Primitive Datentypen und Objekte. Diese beiden sind auch tatsächlich grundverschieden, wiewohl der Unterschied (zumindest für den Programmierer) gerade mit Java 5 zunehmend an Bedeutung abnimmt.

2.1 Primitive Datentypen

Wie eingangs schon erwähnt, handelt es sich hierbei *nicht* um Objekte, sondern tatsächlich um skalare Datentypen, die jeweils genau einen Wert enthalten. Diese Typen sind im Vergleich zu Objekten einigermaßen langweilig: Sie speichern exakt einen Wert, bilden keine Datenstruktur und können keine Methoden haben. Es ist eben tatsächlich nicht mehr als ein Wert.

Es gibt allerdings zu jedem dieser Typen eine Klasse, die eben diesen Typ beinhaltet und kapselt, dazu weiter unten mehr.

Java bietet eine ganze Reihe von primitiven Datentypen an, die eigentlich das meiste umfassen, was man aus anderen Sprachen kennt und auch benötigt:

2.1.1 Ganzzahl-Typen

Es gibt in Java vier verschiedene Ganzzahl-Typen: **`byte`**, **`short`**, **`int`** sowie **`long`**. Diese unterscheiden sich nur in ihrer Länge, d. h. in ihrem Wertebereich, der im folgenden einmal tabellarisch dargestellt ist:

Datentyp	Bits	Wertebereich
`byte`	8	$-128 \ldots 127$
`short`	16	$-32768 \ldots 32767$
`int`	32	$-2147483648 \ldots 2147483647$
`long`	64	$-9223372036854775808 \ldots 9223372036854775807$

Ganzzahl- oder auch Integertypen in Java sind grundsätzlich *vorzeichenbehaftet* und werden intern in Zweierkomplementdarstellung gespeichert. Es gibt keine vorzeichenlosen Zahlen in Java. Im Normalfall stört das auch nicht so sehr, es sei denn, man möchte den verfügbaren Wertebereich so gut wie möglich ausnutzen und selbst **`long`** ist einem noch zu kurz. Aber ab irgendeinem Punkt gibt es immer keine andere Abhilfe als Bignums[1], die Frage ist dann lediglich noch, wo.

Eine leider etwas unschöne Eigenschaft von Ganzzahlen in Java ist, daß sie keinerlei Anhaltspunkt darüber geben, ob eine Operation einen Überlauf verursacht hat oder nicht. Wer weiß, wie Zahlen in Computern gespeichert werden, wird wissen, daß sich die Zahlen quasi wie ein Kreis verhalten, d. h. die kleinste Zahl ist ein direkter Nachfolger der größten. Folglich ergibt

```
byte a = 127 + 1;
```

den Wert -128. In Java ist es (meines Wissens nach) unmöglich, sicher herauszufinden, ob ein solcher Überlauf stattgefunden hat oder nicht. Folgendes Beispiel

```
int i = 1000000;
System.out.println(i * i);
long l = i;
System.out.println(l * l);
```

liefert beispielsweise die Ausgabe

```
-727379968
1000000000000
```

und

```
int i = Integer.MAX_VALUE;
System.out.println(i * i)
```

liefert lediglich eine 1.

Man kann Ganzzahlen auf verschiedene Weise im Quelltext darstellen:

```
int decVal = 26;   // 26, dezimal
```

[1] üblicherweise langsamere Implementation von beliebig großen Zahlen. Sämtliche Arithmetik muß hierzu in Software realisiert werden. Java hat für so etwas die Klassen `BigInteger` bzw. `BigDecimal`

```
int octVal = 032;  // 26, oktal
int hexVal = 0x1a; // 26, hexadezimal
```

Die dezimale Schreibweise ist sicherlich die gebräuchlichste, aber je nachdem, was man so schreiben will, bietet sich eventuell auch eine der anderen beiden Darstellungen an. Oktalzahlen werden mit einer führenden Null geschrieben, wohingegen Hexadezimalzahlen am Anfang ein `0x` stehen haben.

2.1.2 Zeichen

`char` ist im Grunde ebenfalls ein Integer-Typ, in dem Sinne, daß er als 16-Bit-Zahl gespeichert wird. Allerdings repräsentiert er einzelne Zeichen aus der Unicode-BMP[2] und ist damit vorzeichenlos und für den Programmierer selten als Zahl zu behandeln. Der Wertebereich liegt damit zwischen U+0000 und U+FFFF.

Grundsätzlich werden Literale vom Typ **`char`** in Hochkommata geschrieben:

```
char capitalC = 'C';
```

Es gibt eine Reihe von Escapes für verschiedene Steuerzeichen, zum Beispiel:

```
char tab = '\t';
```

Die komplette Liste dieser Zeichen ist folgende:

Escape-Sequenz	Zeichen
`'\b'`	Backspace (BS)
`'\t'`	Tabulator (HT)
`'\n'`	Zeilenvorschub (LF)
`'\f'`	Form Feed (FF)
`'\r'`	Wagenrücklauf (CR)
`'\"'`	Doppeltes Anführungszeichen "
`'\''`	Einfaches Anführungszeichen '
`'\\'`	Backslash \

Es gibt noch zwei Möglichkeiten (mehr oder weniger) beliebige Zeichen direkt zu erzeugen, einmal über Oktalzahlen (eine Variante, die eigentlich nur noch aus Kompatibilität zu C existiert):

```
char lowercaseA = '\141';
```

Hierbei folgen dem Backslash zwischen 1 und 3 oktale Ziffern, mit denen sich allerdings nur Zeichen zwischen U+0000 und U+00FF darstellen lassen. Da dies aber ohnehin nur aus Kompatibilitätsgründen existiert, sollte man eher auf die direkten Unicode-Escapes zurückgreifen:

```
char xi = '\u03be';
```

[2] Basic Multilingual Plane – die erste der 17 sogenannten „Planes“, in denen die Unicode Code Points liegen und damit die ersten 65536 Code Points.

Hier wird direkt ein *Unicode Code Point* angegeben. Er wird als Hexadezimalzahl direkt nach dem `\u` angegeben und geht, wie oben schon erwähnt, von 0000 bis FFFF. Im angegebenen Beispiel enthielte die Variable `xi` also ein ξ.

2.1.3 Wahrheitswerte

Nun kommen wir zuweilen in die Verlegenheit, den Wahrheitsgehalt einer Aussage bestimmen zu wollen. Sicherlich kann man sich dafür mit ganzen Zahlen behelfen (wie in hier nicht näher genannten Programmiersprachen), aber der hübschere Weg ist eigentlich, einen eigenen Typ für solche Ergebnisse zu haben. Zu Ehren von George Boole[3] auch liebevoll **`boolean`** genannt, hat er genau zwei mögliche Werte:

```
boolean wahr   = true;
boolean falsch = false;
```

Es gelten die bekannten Regeln einer Booleschen Algebra. Dieser Datentyp hat besondere Bedeutung bei einigen Kontrollstrukturen, da beispielsweise Bedingungen, Schleifenabbrüche, etc. grundsätzlich ein Resultat vom Typ **`boolean`** verlangen.

2.1.4 Gleitkomma-Datentypen

Java bringt zwei Gleitkommadatentypen unterschiedlicher Länge mit, nämlich **`float`** und **`double`**. Diese sind getreu der Norm *IEEE 754* 32 bzw. 64 Bit lang und bestehen jeweils aus Vorzeichen, Exponent und Mantisse sorgfältig festgelegter Länge. Weiterhin existieren neben „normalen“ Zahlenwerten, die diese Typen annehmen können noch verschiedene spezielle Bitmuster, die ∞, $-\infty$, $+0$, -0 und NaN *(Not a Number)* entsprechen:

```
double a = 1.0 / 0.0;   // ergibt +Inf
double b = 1.0 / -0.0;  // ergibt -Inf
double c = 0.0 / 0.0;   // ergibt NaN
```

Die positive und negative Null sind für arithmetische Vergleiche tatsächlich identisch und nicht verschieden:

```
boolean x = 0.0 > -0.0; // ergibt false
```

Tests auf Gleichheit mit NaN sind grundsätzlich **`false`**, das schließt folgenden Ausdruck mit ein:

```
boolean y = (c == c);   // ergibt false
```

Weiterhin ist $\infty - \infty$ ebenfalls ein unbestimmter Ausdruck, welcher in NaN resultiert.

Fließkommazahlen können auf recht verschiedene Weise repräsentiert werden:

```
float d = 1e1f;
float e = 2.f;
float f = 6.022137e+23f;
```

[3] 1815–1864

Hier fällt auf, daß die **float**-Literale alle ein f am Ende benötigen. Das kommt daher, weil Java grundsätzlich für Gleitkommaliterale annimmt, daß es ein **double** sein soll. Das f am Ende sagt jedoch explizit, daß wir einen **float**-Wert haben wollen (**double** könnten wir einem **float** nicht direkt zuweisen, wie wir in Abschnitt 2.5 noch sehen werden).

```
double g = .3;
double h = 1e-9d;
double i = 1e137;
double j = Double.longBitsToDouble(0x400921FB54442D18L);
```

Wie hier zu sehen ist, entfällt für **double**s der Buchstabe f am Ende, allerdings kann man optional ein d anhängen, wie hier in der zweiten Zeile gezeigt. Der dritte Wert überschreitet den Wertebereich von **float** doch schon um einiges und paßt daher nur in einen **double**. Die letzte Zeile ist eine interessante andere Methode, einen **double** mit Werten zu füllen. Hier wird eine hexadezimale Repräsentation eines **double** in selbigen umgewandelt. Der Wert, der sich hier ergibt, ist übrigens π.

2.2 Gekapselte primitive Typen

Wie am Anfang schon erwähnt, existiert für jeden primitiven Datentyp eine sogenannte *Wrapperklasse*, die den jeweiligen Datentyp kapselt, sowie diverse nützliche Methoden und Konstanten für das Arbeiten mit dem jeweiligen Typ bereitstellt. Selbst wenn man mit den primitiven Typen arbeitet, so bemüht man zuweilen dennoch Methoden der zugehörigen Klasse, da sie Funktionalität wie beispielsweise das Umwandeln von Zeichenfolgen in Zahlen und andersherum bereitstellen.

Die primitiven Typen und ihre jeweiligen Wrapperklassen sind im Folgenden noch einmal aufgelistet:

Primitiver Typ	Klasse
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Konvertierungen zwischen Typ und zugehörigem Objekt sind üblicherweise recht unschön zu schreiben, das sieht dann beispielsweise folgendermaßen aus (hier am Beispiel von **int** und Integer):

```
int i = 3;
// Konvertierung von int zu Integer
Integer j = new Integer(i);
// Konvertierung von Integer zu int
int k = j.intVal();
```

Hat man nun in einem Ausdruck eine Hin- *und* Rückkonvertierung, weil eine Methode unbedingt **int** (oder Integer) haben will und auch zurückgibt, so wird der Ausdruck schon länger und sieht in etwa folgendermaßen aus:

```
int i = 3;
i = doSomething(new Integer(i)).intVal();
```

Sonderlich schön lesbar ist das sicher nicht. Java hat mit der Version 1.5 (Java 5) ein Feature namens *Autoboxing* eingeführt, welches eben solche Konvertierungen zwischen den Basistypen und ihren zugehörigen Kapselobjekten für den Programmierer transparent macht, so daß man sich nicht mehr darum kümmern muß. Man sollte allerdings wissen, *daß* es passiert, da jede solche Konvertierung auch einen Performanceverlust mit sich bringt; ist man also darauf angewiesen, daß das Programm möglichst schnell läuft, so sollte man auf so etwas ein wenig aufpassen.

2.3 Zeichenfolgen

Eine der häufigsten Dinge, mit denen Programme arbeiten, gerade, wenn sie auf irgendeine Weise mit dem Benutzer interagieren, sind Zeichenfolgen oder auch *Strings*. Java bietet hierfür ähnlich wie C++ eine eigene Klasse an, die fast alles bietet, was man sich in diesem Zusammenhang wünschen kann: `java.lang.String`. Ebenfalls bieten Strings in Java eine der wenigen Inkonsistenzen der Sprache, nämlich überladene Operatoren. Neue `String`-Objekte werden automatisch erzeugt, indem man Zeichenfolgen in Anführungszeichen einschließt:

```
String s = "Hallo";
```

Ein wesentlicher Aspekt von Strings in Java ist, daß sie unveränderlich sind. Das heißt, sie sind (zumindest für den Programmierer) nicht einfach wie in C ein Array von Zeichen; man kann in Java nicht einfach einzelne Zeichen in einem String direkt ändern. Jede Methode, die scheinbar einen String ändert, gibt in Wirklichkeit einen neuen String zurück. Ebenso sind Strings in Java *nicht* null-terminiert, d. h. sie werden nicht mit dem Nullzeichen (`'\u0000'`) abgeschlossen, wie in C/C++. Folglich erfordern sie eine ganz andere Handhabung als in den beiden anderen Sprachen.

Weiterhin hat *jedes* Objekt in Java eine Methode `toString()`, welche überschrieben werden kann und üblicherweise eine menschenlesbare Textrepräsentation eines Objektes liefert. Für Ein- oder Ausgabemethoden sowie für Verkettung von Strings mit anderen Objekten wird diese Methode implizit aufgerufen.

Die Verkettung zweier Strings kann auf verschiedene Weise bewerkstelligt werden, wenngleich zumindest die zweite Variante vom Compiler in die erste überführt wird:

```
"Hello, ".concat("World!");
"Hello, " + "World" + "!";
String fs = String.format("Hello, %s!", "World");
```

Dem aufmerksamen Beobachter wird nicht entgangen sein, daß die letzte Variante nicht wirklich eine Stringverkettung ist, sondern eher an die Syntax von `snprintf()` erinnert. Das ist eigentlich eine etwas mächtigere Variante der Stringkonstruktion, aber fürs erste beschränken wir uns hier eher auf die zweite Variante, welche ich persönlich eigentlich für am lesbarsten halte.

Aus der Tatsache, daß Strings in Java unveränderlich sind, folgt eine Reihe von häufigen Fehlern, die einem gern unterlaufen. Eine einfache Routine, die eine (Text-)Datei in einen String einliest, mag man vielleicht naiv in etwa folgendermaßen implementieren (mehr oder weniger Pseudocode):

```
s = "";
while (!f.eof())
  s = s + f.readLine();
```

D. h. es wird zeilenweise gelesen, solange man die Datei noch nicht vollständig gelesen hat und die gelesene Zeile wird an einen String angehängt. Was hier nun passiert, ist, daß für jede Zeile ein neuer String erzeugt wird, der Inhalt des alten umkopiert und dann die gelesene Zeile angehängt wird. Verbindet man das mit einem weiteren Feature von Java, nämlich der *Garbage Collection*, die automatisch nicht mehr verwendeten Speicher wieder freigibt, so kommen wir auf (selbst für eine mittelmäßig kleine Datei) immense Datenmengen, die im Speicher alloziert, umkopiert und wieder freigegeben werden. Verketten von Zeichenfolgen, gerade bei langen solchen, ist also sicherlich keine Sache, die man innerhalb von oft laufenden Schleifen machen sollte. Java hat hierfür noch eine Klasse `StringBuilder`, die ermöglicht, daß man auch veränderliche Zeichenfolgen hat.[4] Darauf soll hier aber nicht weiter im Detail eingegangen werden. Dieses Problem findet sich allerdings auch in anderen Sprachen mit gewissen Eigenschaften wieder.

2.4 Arrays

Arrays sind, wie in anderen Sprachen auch, ein ein- oder mehrdimensionales Feld eines bestimmten Typs. Allerdings, anders als beispielsweise in Pascal, gehört in Java die Länge des Arrays nicht zum Typ. Folglich kann eine Array-Variable Referenzen auf Arrays beliebiger Länge aufnehmen. Arrays werden in Java folgendermaßen deklariert:

```
int[] intArr;
String[] stringArr;
```

Mehr als eine Dimension wird kenntlich gemacht, indem man einfach mehr als ein eckiges Klammerpaar anfügt:

```
int[][] matrix;
```

Array-Instanzen, also die Enden der Referenzen, haben allerdings grundsätzlich eine konstante Länge, die einmal festgelegt wird:

```
intArr = new int[4];
```

erzeugt ein Array vom Typ **`int`** der Länge 4. Arrays sind in Java grundsätzlich nullbasiert, d. h. der erste Index eines Arrays ist 0 und der letzte *Länge* -1. Im obigen Beispiel deckt das Array also das Intervall $[0,3]$ ab.

Es ist bei der Deklaration von Arrays egal, ob man die eckigen Klammern, die ein Array kennzeichnen, nach dem Typ oder nach dem Bezeichner schreibt. Folgende Deklarationen sind also äquivalent:

```
int[][] foo;
int foo[][];
```

[4] Alternativ gäbe für genau diese Anwendung noch eine Variante, indem man zunächst nach der Länge der Datei schaut und sich einen Puffer exakt dieser Größe anlegt, in den man dann den Inhalt mit einem Mal einliest.

```
int[] foo[];
```

Man sollte allerdings zugunsten der Lesbarkeit bei einer Variante bleiben und diese konsistent durchhalten. Es sei denn, man möchte Spielereien der folgenden Art machen:

```
float[] arr1, arr2, matrix[];
```

welche zwei eindimensionale, sowie ein zweidimensionales Array (eine Matrix) in einer Zeile deklariert. Aber in den meisten Fällen sollte es nicht nötig sein, auf solche Konstrukte zurückzugreifen.

Bei vorgegebenem Inhalt von Arrays ist die Größe auch gleich implizit festgelegt. Man kann Arrays folgendermaßen gleich bei der Deklaration initialisieren:

```
int[] fak = { 1, 1, 2, 6, 24, 120, 720, 5040 };
```

Hierbei ist die Länge des Arrays offenbar 8 und man erhält durch `fak[i]` für $i \leq 7$ die i-te Fibonacci-Zahl. Dies ist eine bequeme Art, Felder ähnlicher Werte ohne viel Codeaufwand zu initialisieren.

2.5 Konvertierungen

Java kennt verschiedene Arten von Konvertierungen eines Datentyps in einen anderen, man sollte darüber Beischeid wissen, es erspart einem so manche unliebsame Überraschung.

2.5.1 Explizite Konvertierung (Casting)

Betrachten wir folgendes kleine Beispiel:

```
int i = 12.5;
```

Offenbar versuchen wir hier, einem **`int`** einen Wert vom Typ **`double`** zuzuweisen. Intuition wie auch der Compiler sagen einem, daß dies nicht möglich ist. Wenn wir eine solche Zuweisung benötigen, dann müssen wir die Konvertierung explizit durchführen, da bei der Konvertierung möglicherweise Information verlorengeht – in diesem Falle die 0,5 der 12,5 Wenden wir also den *Cast-Operator* an, um folgendes zu erhalten:

```
int i = (int) 12.5;
```

Nun wird die 12,5 vor der Zuweisung erst in einen **`int`** konvertiert. Hierbei gehen die Nachkommastellen verloren, aber da wir selbst die Konvertierung befohlen haben, geht Java davon aus, daß der Programmierer sich über die Konsequenzen im Klaren war.

2.5.2 String-Konvertierung

Dies ist wahrscheinlich eine der am häufigsten unbewußt genutzen Konvertierungen. Wir hatten weiter oben den Operator + auch für die Stringverkettung kennengelernt. Dieser funktioniert auch, wenn man Strings mit anderen Typen oder Objekten verkettet. In diesem Falle wird im Hintergrund automatisch die `toString()`-Methode des Objektes aufgerufen (bei primitiven Datentypen wird der Typ zunächst in ein Objekt konvertiert, auf dem diese Methode dann aufgerufen wird). Die Zeile

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

ergibt mit unserem oben festgelegten `i` die Ausgabe `i = 12`.

2.5.3 Widening Conversion

In Ermangelung einer sinnvollen deutschen Bezeichnung für dieses Konzept nenne ich es hier einmal ebenso wie im Englischen. Grundlage ist, daß ein (numerischer) Datentyp automatisch in einen mit größerem Wertebereich umgewandelt werden kann, wenn es der Kontext erfordert. Wenn wir uns an die nötige explizite Konvertierung erinnern, so wissen wir noch, daß

```
int i = 12.5;
```

nicht möglich war. Stattdessen mußten wir

```
int i = (int) 12.5;
```

schreiben.

```
float f = i;
```

ist hingegen problemlos möglich, da hier der Wertebereich von **`float`** den von **`int`** übersteigt (auch wenn die Genauigkeit abnimmt, wie wir in Kürze sehen werden). Die hierbei möglichen automatischen Konvertierungen sind im Folgenden noch einmal tabellarisch dargestellt.

Ausgangstyp	Konvertierungstypen
`byte`	**`short`**, **`int`**, **`long`**, **`float`** oder **`double`**
`short`	**`int`**, **`long`**, **`float`** oder **`double`**
`char`	**`int`**, **`long`**, **`float`** oder **`double`**
`int`	**`long`**, **`float`** oder **`double`**
`long`	**`float`** oder **`double`**
`float`	**`double`**

Nun kann es passieren, daß eine solche Konvertierung mit Genauigkeitsverlust abläuft. Solange nur zwischen Ganzzahl- oder nur zwischen Gleitkommatypen konvertiert wird, ändert sich der Wert nicht, allerdings kann es bei der Konvertierung von einem Ganzzahl- zu einem Gleitkommatyp passieren, daß die Zahl hinterher nicht mehr die gleiche ist. In folgendem Beispiel

```
int big = 1234567890;
float approx = big;
System.out.println(big - (int) approx);
```

wird `-46` ausgegeben, da **`float`** eben nicht die Genauigkeit von **`int`** erreicht bei der Größe der Zahl.

2.5.4 Numeric Promotion

Promotion oder „Beförderung“ ist ein Konzept, welches die Implementierung von Operatoren vereinfacht, indem immer nur gleiche Typen miteinander verknüpft werden können. Hierbei wird bei ge-

mischttypigen Ausdrücken der Typ mit dem geringeren Wertebereich in den anderen überführt, bevor die Operation ausgeführt wird:

```
int i = 12.5;
float f = i;
f = i * f;
```

In der letzten Zeile wird zunächst `i` in einen **`float`** umgewandelt, da der zweite Operand (`f`) hier vom Typ **`float`** ist, was nach obiger impliziter Konvertierungstabelle ein geeigneter Zieltyp für eine Konvertierung von **`int`** ist. Nach der Konvertierung wird wird die Multiplikation mit zwei **`float`**s ausgeführt.

2.5.5 Konvertierung beim Methodenaufruf

Diese Konvertierungsform ist ähnlich zur vorhergehenden, allerdings wird sie beim Aufruf von Methoden und nicht beim Anwenden von Operatoren durchgeführt. Sie dient hier auch nicht zur Angleichung der Operanden sondern zum Angleichen der Parameter an die Typen, die eine Methode akzeptiert.

Beispiel:

```
float f = 3.0f;
double d = Math.sin(f);
```

Hier wird `f` automatisch in einen **`double`** umgewandelt, da `Math.sin()` lediglich **`double`** als Argument akzeptiert.

3 Operatoren

Java kennt insgesamt 37 Operatoren, die sich wie folgt kategorisieren lassen:

Vergleich	>, <, <=, >=, !=, ==
Boolesche Operatoren	!, &&, ||
Bitweise	~, &, |, ^, >>, <<, >>>
Arithmetisch	+, -, *, /, %, ++, --
Bedingung	? und :
Zuweisung	=, +=, -=, *=, /=, &=, |=, ^=, %=, <<=, >>=, >>>=

Im Folgenden betrachten wir uns diese mal ein wenig genauer.

3.1 Vergleichsoperatoren

Sämtliche der Vergleichsoperatoren sind nur auf numerischen Datentypen definiert und sie liefern immer ein Resultat vom Typ **`boolean`**.

Zunächst gibt es den Test auf numerische (Un-)Gleichheit. Hierfür dienen die Operatoren == sowie !=:

```
boolean a = (5 == 5);   // true
boolean b = (.3 != .2); // true
boolean c = (1 == 4);   // false
```

Weiterhin gibt es die Operatoren für den numerischen Vergleich: >, <, >= sowie <=:

```
boolean d = (3 > 4);        // false
boolean e = (8 <= 10);      // true
boolean f = (2 < 2);        // false
boolean g = (1.5 >= 1.25); // true
```

Zu den Relationen selbst werde ich hier nichts mehr sagen, sie dürften ausreichend selbsterklärend sein

3.2 Boolesche Operatoren

Diese sind nur auf dem Typ **boolean** definiert und liefern ihrerseits wieder ein Resultat vom Typ **boolean**. Die folgenden Ergebnisse sollten an sich nicht überraschen. Es gibt zunächst ein logisches *Nicht:*

```
boolean a = !true;  // false
boolean b = !a;     // true
```

Weiterhin ein logisches *Und:*

```
boolean c = a && b;  // false
boolean d = a && !c; // true
```

sowie ein logisches *Oder:*

```
boolean e = true || false;  // true
boolean f = false || false; // false
boolean g = true || true;   // true
```

Im Grunde also nicht sonderlich verwunderlich, die Operatorzeichen wurden auch direkt so aus C übernommen, so daß sich jeder, der C bzw. C++ beherrscht, sofort heimisch fühlen sollte.

3.3 Bitweise Operatoren

Nun kann man mit den booleschen Operatoren zwar interessante Dinge anstellen aber für mehr als Aussagenlogik reicht es auch nicht. Manchmal möchte man vielleicht Bits in Zahlen direkt manipulieren und während das zwar schwer zu durchschauen für den gelegentlichen Leser des Quelltextes sein mag, so geht es doch und zwar mit eben den Operatoren `!`, `&`, `|`, `^`, `<<`, `>>` sowie `>>>`.

Diese Operatoren funktionieren nur mit Ganzzahltypen und sindliefern auch wieder einen solchen zurück. Dank Promotion ist das Ergebnis vom größeren der beiden Eingangstypen. Hilfreich ist auch bei den Operatoren, sich vor Augen zu halten, daß Ganzzahlen in Java grundsätzlich als Zweierkomplement dargestellt werden.

Zunächst gibt es ein bitweises Nicht, ~, welches einfach nur sämtliche Bits kippt:

```
int a = ~0;    // -1
int b = ~-846; // 845
```

Die zweite Zeile ist hier interessant und bietet und eine alternative Möglichkeit, die numerische Negation nachzubilden:

```
int c = ~b + 1; // -845
```

Diese Variante nutzt die Eigenschaften des Zweierkomplements, bzw. die entsprechende Rechenvorschrift.

Natürlich können wir auch nur die jeweils gleichen Bits aus zwei Zahlen übernehmen, hierfür das bitweise Und: &

```
int d = a & 7;    // 7
int e = 231 & 14; // 6
//   11100111
// & 00000110
// = 00000110
```

Weiterhin haben wir ein bitweises Oder: |

```
int f = 13 | 24;  // 29
//   001101
// | 011000
// = 011101
```

sowie ein ausschließendes Oder: ^

```
int g = 13 ^ 24; // 21
//   001101
// ^ 011000
// = 010101
```

Mit dem ausschließenden Oder können wir noch andere lustige Dinge machen. Beispielsweise werden wir merken, daß der Ausdruck

```
x ^ -1;
```

genau das gleiche ergibt wie

```
~x;
```

Dies liegt daran, daß die Zahl -1 in Zweierkomplementdarstellung nur aus gesetzten Bits besteht. Somit kehrt ^ einfach jedes Bit der Zahl um.

Es gibt die drei Schiebeoperatoren: <<, >> und >>>. Sie verschieben das Bitmuster um eine gegebene Anzahl Bits nach links oder rechts. Zu beachten ist hierbei, daß es für das Schieben nach rechts zwei Operatoren gibt, von denen der eine (>>>) das Vorzeichen nicht beachtet und tatsächlich das echte Bitmuster weiterschiebt. Bei >> würde das erste Bit vervielfacht werden, wohingegen bei >>> Nullen nachgeschoben werden:

```
int h = 31 >> 3;   // 3
int i = -67 >> 3;  // -3
int j = 31 >>> 3;  // 3
int k = -67 >>> 3; // 536870903
int l = 3 << 4;          // 48
int m = 1000000000 << 2; // -294967296
```

Wie zu sehen ist, ergibt der >>>-Operator für negative Zahlen andere Ergebnisse, bei positiven ist es allerdings egal. Was hier auch dargestellt wurde, war, daß der <<-Operator eine Zahl in den negaiven Bereich „schiebt“, wie im letzten Beispiel zu sehen.

Ein vielleicht nicht auf den ersten Blick sichtbares Feature (und auch eines, welches man vielleicht nicht sonderlich häufig benötigt), ist, daß eine Shift-Operation um einen negativen Betrag in der jeweils umgekehrten Shift-Operation resultiert.

3.4 Arithmetische Operatoren

Nun kommt man häufiger in die Verlegenheit, etwas berechnen zu wollen. Auch hier bietet Java einem das übliche Spektrum an Arithmetikoperatoren. Diese sind für alle Operationen zwischen numerischen (also Ganzzahl- oder Gleitkommatypen) definiert, mit der Ausnahme des +, da es auch für Strings definiert ist, aber die Bedeutung haben wir schon kennengelernt und ich konzentriere mich hier auf den arithmetischen Zweck dieser Operatoren.

Addition und Subtraktion dürften soweit bekannt sein:

```
int a = 5 + 6;         // 11
float b = a + 3.2f;    // 14.2
int c = a - 36;        // -25
double d = b - c + a; // 50.2
```

Weiterhin gibt es eine Art Kurzform für den Ausdruck `a = a + 1` bzw. `a = a - 1`, nämlich die sogenannten Inkrementoperatoren: `++` und `--`. Diese können entweder vor oder nach einer Variablen stehen und ändern damit ihr Verhalten ein wenig. Steht der Operator *nach* der Variable (`a++`), so erhält dieser Ausdruck den Wert, den `a` *vor* dem Inkrement hatte und *danach* wird der Wert der Variablen erhöht. Steht der Operator hingegen *vor* der Variable, so wird zunächst inkrementiert und der Ausdruck liefert den Wert *nach* der Erhöhung. Analog mit `--`.

```
a = 1;
a++;      // -> a = 2
++a;      // -> a = 3
b = a++;  // -> b = 3, a = 4
b = --a;  // -> b = a = 3
```

Nachdem wir nun jegliche Art und Weise, Zahlen zu addieren und zu subtrahieren, eingehend behandelt haben, schreiten wir nunmehr zu Multiplikation und Division. Auch hier eigentlich wenig überraschendes; bei Division sollte man allerdings hier und da aufpassen: Wenn man beispielsweise zwei Ganzzahlen durcheinander dividiert, wird keine Gleitkommazahl herauskommen, wegen der oben schon angesprochenen Konvertierungen.

```
double x = 14.2 * 13;  // 184.6
int y = 3 / 2;         // 1
float z = 3 / 2;       // 1.0
float z1 = 3f / 2;     // 1.5
```

Weiterhin gibt es noch die Division mit Rest, wie sie sicher noch aus der Grundschule bekannt ist. *Modulo* (%) funktioniert in Java ebenfalls mit allen numerischen Typen, nicht – wie in anderen Sprachen – nur auf Ganzzahltypen. Folgendes sollte schon bekannt sein, das zweite Beispiel vielleicht nicht so sehr:

```
int mod1 = 14 % 8;       // 6
double mod2 = 5.2 % 1.2; // 0.4
```

3.5 Bedingungsoperator

Es kommt häufig vor, daß man Code der folgenden Art hat:

```
int i;
if (a)
  i = 5
else
  i = 6;
```

oder auch

```
int i = 6;
if (a)
  i = 5;
```

Dieses Konstrukt läßt sich mit dem Bedingungsoperator auch folgendermaßen schreiben:

```
int i = a ? 5 : 6;
```

Dieser Operator hat die Syntax *Bedingung* ? *Wert* : *SonstWert* und liefert *Wert*, falls *Bedingung* wahr ist, ansonsten *SonstWert*. Die Bedingung muß logischerweise ein boolescher Ausdruck sein.

3.6 Zuweisungsoperatoren

Das einfachste hierbei ist sicherlich die *einfache Zuweisung*. Diese sieht einfach folgendermaßen aus:

```
a = b;
```

oder

```
x = 5 + b - a;
```

und inzwischen sollte sie uns nicht mehr ungewohnt vorkommen.

Häufig kommt allerdings auch das folgende Muster von Ausdrücken vor: `a = a ∘ b` (∘ hier mal ein Platzhalter für einen beliebigen Operator). Also eine Zuweisung zu einer Variablen, die als ersten Operanden wieder die gleiche Variable hat. Der resultierende Typ muß dabei der gleiche sein wie der der Variable. Dieses Muster resultiert in folgender Kurzschreibweise: `a ∘= b`. Diese Operatoren gibt es natürlich nur für solche Operatoren, wo der Rückgabetyp dem des ersten Operanden entspricht, aber beispielsweise folgende Konstrukte sind denkbar:

```
a += 1;   // a = a + 1, entspricht auch a++ oder ++a
x *= 2;   // x = x * 2
y >>>= 1; // y = y >>> 1
```

usw.

4 Anweisungen und Kontrollstrukturen

Wie große Teile der restlichen Syntax sind auch die Kontrollstrukturen eigentlich unverändert aus C bzw. C++ übernommen worden. Dies macht, wie üblich, den Umstieg auf Java um einiges einfacher.

4.1 Anweisungen und Blöcke

Zunächst ein paar grundsätzliche Dinge zu Anweisungen und Blöcken. Die meisten Kontrollstrukturen führen Anweisungsblöcke aus. Ein solcher Anweisungsblock besteht aus einer oder mehr Anweisungen, mehrere Anweisungen werden dabei durch geschweifte Klammern gruppiert, Anweisungen werden durch ein Semikolon abgeschlossen:

```
a = a + b;
System.out.println("Hello World");
{
  x = Math.sin(y);
  y = x++;
}
```

Da jede einzelne Anweisung für sich genommen auch ohne geschweifte Klammern ein Anweisungsblock ist, kann man also an den Stellen, wo eine Kontrollstruktur einen Block erwartet, man aber nur eine Anweisung ausführen möchte, auch diese eine Anweisung direkt hinschreiben.

4.2 Leere Anweisung

Die leere Anweisung ist wohl mit die langweiligste von allen. Sie tut nichts, dennoch gibt es sie:

```
;
```

Beruhigend allerdings, daß man nach Belieben Semikola zwischen Anweisungen einfügen kann, ohne daß es den Sinn des Programms entstellt (allenfalls vielleicht die Lesbarkeit).

4.3 Bedingungen: If

Wohl eine der häufigsten Kontrollstrukturen ist die **if**-Anweisung. In ihrer allgemeinen Form sieht sie folgendermaßen aus: **if** (*Bedingung*) *DannBlock* **else** *SonstBlock*, wobei der **else**-Teil optional ist, d. h. man darf ihn auch weglassen. Dies führt dann beispielsweise zu folgenden **if**-Anweisungen:

```
if (a == b) doSomething();

if (b == c)
    doSomething()
else
    doSomethingElse();

if (c != d) {
    statement1();
    statement2();
}
```

4.4 Mehrfachauswahl: Switch

Zuweilen möchte man den gleichen Ausdruck auf mehr als einen möglichen Wert überprüfen. Natürlich kann man das in folgender Form machen:

```
if (a == 1)
  method1()
else if (a == 2)
  method2()
else if (a == 3)
  method3()
else
  methodElse();
```

Einige Sprachen (z. B. Python) erlauben auch nur diese Variante, allerdings gibt es in Java eine elegantere Lösung (allerdings mit den gleichen Einschränkungen wie schon zu Zeiten von C):

```
switch (a) {
case 1:
  method1();
  break;
case 2:
  method2();
  break;
case 3:
  method3();
  break;
default:
```

```
    methodElse();
    break;
}
```

Zwar hat diese Schreibweise einige Vorteile gegenüber dem „`if ... else if ...`“-Konstrukt, allerdings ergeben sich auch hier Probleme. Zunächst wird der Vergleichswert (`a` in diesem Beispiel) lediglich einmal geladen und kann dann mehrmals verglichen werden. Weiterhin ist dies eigentlich die bevorzugte Methode für Mehrfachverzweigung, da `switch` genau dafür da ist. Nachteile hierbei sind allerdings, daß nur primitive Datentypen verglichen werden können. Keine Strings, keine anderen Objekte, etc., was dieses Konstrukt für einige Anwendungen wieder sehr unflexibel macht, wo man dann dennoch wieder auf `if` zurückgreifen muß (ein Grund, warum beispielsweise Python eben nur `if` erlaubt, damit solche Codeblöcke konsistent sind). Weiterhin erlaubt der Ursprung in C und die exakte Übernahme in Java einige Unschönheiten und Fehler. Ein gutes Beispiel hierfür ist das sogenannte *Fall-Through:*

```
static void howMany(int k) {
    switch (k) {
    case 1: System.out.print("one ");
    case 2: System.out.print("too ");
    case 3: System.out.println("many");
    }
}

public static void main(String[] args) {
    howMany(3);
    howMany(2);
    howMany(1);
}
```

Was wir hier erhalten, ist folgendes:

```
many
too many
one too many
```

Was wir aber eigentlich haben wollten, wäre vielleicht folgendes gewesen:

```
one
two
many
```

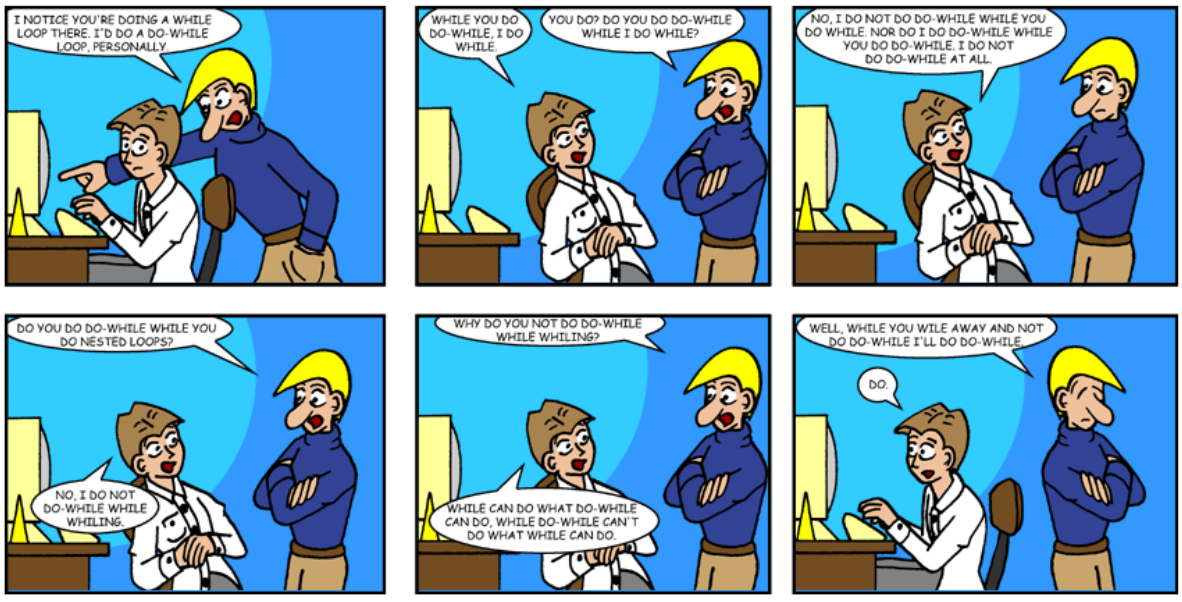
Der Teufel liegt hier im Detail bzw. in der exakten Übernahme der Syntax von `switch` aus C. `switch` springt an der Stelle in den Block hinein, wo ein passendes `case` gefunden wird (oder `default`, falls vorhanden und kein passendes `case`). Danach wird von oben nach unten gesucht. Nach dem Sprung wird die Ausführung dort fortgesetzt und läuft entweder bis zum Ende des `switch`-Blocks oder bis zu einem `break` (welches wir weiter unten noch kennenlernen werden). Vergißt man also ein `break`,

so kann es passieren, daß gleich mehrere **case**-Zweige abgearbeitet werden, angefangen bei dem, der eigentlich zutraf. Manchmal ist dieses Verhalten gewollt, allerdings in meiner Erfahrung eher selten und andere Sprachen haben dem auch Rechnung getragen und den Fall-Through-Fall durch leichter zu lesende und weniger fehleranfällige Konstrukte ersetzt.

Wie dem auch sei, so hätte der Code aussehen sollen, um die erwartete Ausgabe zu liefern:

```
static void howMany(int k) {
    switch (k) {
    case 1: System.out.println("one");
            break; // exit the switch
    case 2: System.out.println("two");
            break; // exit the switch
    case 3: System.out.println("many");
            break; // not needed, but good style
    }
}
public static void main(String[] args) {
    howMany(1);
    howMany(2);
    howMany(3);
}
```

4.5 Schleifen: While und Do



Wie in obigem Cartoon schon angedeutet, gibt es zwei verschiedene Arten von allgemeinen Schleifen: **do** und **while**. Während **while** *erst* nach einer Bedingung schaut und nur dann weitermacht, wenn diese erfüllt ist, so läuft **do** zunächst durch den Schleifenkörper und schaut danach erst nach der Schleifenbedingung; die Schleife läuft also mindestens einmal durch.

Dies äußert sich auch optisch im Quelltext. Bei **while** kommt die Abbruchbedingung *vor* der eigentlichen Schleife und bei **do** danach:

```
while (i < 5) {
  doSomething();
  i++;
}

do {
  doSomething();
  i++;
} while (i < 5);
```

Im Grunde tun diese beiden Schleifen das gleiche, zumindest solange i anfänglich kleiner als 5 ist. Ist es das nicht, so bricht **while** sofort ab (da Bedingung nicht erfüllt) aber **do** läuft noch einmal durch die Schleife, ehe es merkt, daß die Bedingung nicht erfüllt ist.

Die Bedingung selbst muß immer vom Typ **boolean** sein, folglich kann man sich auch bei **boolean**s sparen, auf **true** oder **false** zu prüfen. Die Schleife führt ihrerseits einen Anweisungsblock aus.

4.6 Gezählte Schleifen: For

Was ebenfalls nicht selten benötigt wird, sind Schleifen, die gezählt werden. Man hat einen Startwert, einen Endwert und eine Iteration, die irgendwann den Endwert liefert. Zumindest ist dies die häufigste Anwendung. Beispielsweise nacheinander alle Zahlen von 1 bis 10 durchlaufen. Für diese Anwendungen gibt es die folgende Syntax:

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

Diese **for**-Anweisung hat drei Bestandteile: Eine Initialisierung, eine Abbruchbedingung sowie eine Anweisung, die nach jedem Schleifendurchlauf ausgeführt wird. Man kann jede **for**-Schleife dieser Art auch als **while**-Schleife schreiben:

```
i = 1;
while (i <= 10) {
  ...
  i++;
}
```

Diese **while**-Schleife ist völlig äquivalent zu obiger **for**-Schleife und das ist im Grunde auch das, was der Compiler im Hintergrund aus einer **for**-Schleife macht. In diesem Sinne ist die **for**-Schleife also lediglich *syntactic sugar* für etwas, was sonst ein wenig mehr Platz wegnehmen würde bzw. schwerer lesbar ist (so wie Schleifen allgemein im Grunde auch nur syntactic sugar für **goto** sind). Wie hier auch zu sehen ist, sind **for**-Schleifen dieser Art durchaus nicht auf einfaches Hochzählen von Zahlen

beschränkt, man kann damit einiges erschlagen, was allerdings nicht selten zu Lasten der Lesbarkeit geht.

Es gibt allerdings ab Java 1.5 noch eine weitere Syntax der **for**-Schleife, die Konstrukte der folgenden Art vereinfacht:

```
for (int i = 0; i < myArr.length; i++) {
  ...
}
```

welches eine Schleife über alle Elemente eines Arrays darstellt. Braucht man innerhalb der Schleife nämlich nur die jeweiligen Werte (und nicht auch noch ihren Index im Array), so kann man das auch folgendermaßen schreiben (angenommen, `myArr` sei hier einmal ein Array von **double**-Werten):

```
for (double d : myArr) {
  ...
}
```

Tatsächlich funktioniert dieses Konstrukt nicht nur auf Arrays, sondern auch auf Collections, Dictionaries, etc. Die Syntax ist jeweils die gleiche: Vor dem Doppelpunkt steht eine Variable des im Container enthaltenen Typs, also im Falle eines String-Arrays wäre sie vom Typ String und nach dem Doppelpunkt kommt das Array oder ein Containerobjekt. Innerhalb der Schleife hat man nun in jeder Iteration jeweils das nächste Element des Containers. Im Falle von Arrays sind diese wahrscheinlich in der gleichen Reihenfolge wie im Array, allerdings muß dies bei anderen Containern nicht der Fall sein, da diese evtl. nicht einmal eine interne Reihenfolge der Elemente haben.

4.7 Steuerung des Kontrollflusses: Break und Continue

Wir hatten **break** oben bei **switch** schon kennengelernt. Die allgemeine Bedeutung ist einfach der Aussprung aus einem Block. **break** allein springt aus dem aktuellen Block, d. h. in folgendem Beispiel

```
for (int i : a) {
  if (i < 50)
    break;
}
```

springt **break** aus dem umgebenden **for**-Block. Dies funktioniert analog auch für **do**, **while** und, wie schon angesprochen, **switch**.

In dieser Form springt break lediglich aus dem direkt umgebenden Block, um genauer zu spezifizieren, aus welchem Block **break** herausspringen soll, hierzu muß man einem Block zunächst einen Namen geben, dies geschieht über ein Label:

```
block: {
  doSomething();
  doSomethingElse();
}
```

Dieser Block hat nun den Namen `block` und wenn wir nun von innerhalb dieses Blockes herausspringen möchten, so müssen wir dies **`break`** nur noch mitteilen:

```
block: {
  for (int i = 100; i > 0; i--) {
    doSomething();
    if (i < 50)
      break block;
    doSomethingElse();
  }
  doSomethingUtterlyElse();  // this won't get executed
}
```

Neben **`break`**, welches eine Schleife ganz verläßt, gibt es auch noch **`continue`**, welches einfach mit der nächsten Iteration fortfährt. Durch diese enge Bindung zu Schleifen ist ein **`continue`** in **`switch`**-Anweisungen natürlich nicht sehr sinnvoll und ist somit nur in **`for`**-, **`do`**- und **`while`**-Schleifen erlaubt:

```
for (int i : a) {
    if (i >= 50)
      continue;
}
```

Ebenso wie **`break`**, kann **`continue`** ein Label erhalten, welches definiert, an welcher Stelle der Programmfluß wieder aufgenommen wird:

```
elsewhere: for (int i = 0; i < 10; i++) {
  for (int j = 0; j < 10; j++) {
    if (i == j)
      continue elsewhere;
    System.out.println("Differenz" + (i - j));
  }
}
```

4.8 Rückgabewerte und Aussprung aus Methoden: Return

`return` hat im Grunde zwei Bedeutungen. Wie auch **`continue`** und **`break`** kann es mit oder ohne Argument auftreten. **`return`** springt (meist) sofort aus einer Methode und kann einen Wert zurückgeben. Im Falle von Methoden ohne Rückgabewert (in Pascal **`procedure`** genannt), ergibt es erstaunlich wenig Sinn, einen Wert zurückgeben zu wollen. Hierfür also die Variante ohne Argument:

```
void PrintNumbersBelowFifty(int i) {
  if (i >= 50)
    return;
  System.out.println(i);
}
```

Ebenfalls ist hier zu sehen, daß **`return`** den Kontrollfluß in der Methode sofort abbricht. Die Ausgabezeile wird nicht mehr ausgeführt. Tatsächlich wird **`return`** in **`void`**-Methoden (also ohne Rückgabewert) häufig benutzt, um unter bestimmten Bedingungen einfach abzubrechen. Hier stört es auch wenig.

Wenden wir uns dem **`return`** mit Argument zu. Jede Methode, deren Typ nicht **`void`** ist, muß einen Wert des deklarierten Typs zurückgeben. Die Verwendung von **`return`** ohne etwas kommt hier also nicht in Frage. Stattdessen erfüllt **`return`** hier die Aufgabe, den Rückgabewert zu setzen und gleich aus der Methode zu springen (im Gegensatz zu beispielsweise Pascal, wo man den Rückgabewert der Funktion setzt ohne selbige mit der gleichen Anweisung zu verlassen):

```
int inc(int i) {
  return i + 1;
}
```

oder auch

```
int saturatedDec(int i) {
  if (i > 0)
    return i - 1
  return 0;
}
```

Hier auch wieder demonstriert, daß **`return`** aus der Methode springt und nachfolgende Zeilen nicht mehr ausgeführt werden.

Nicht gültig wäre allerdings eine Methode, die **`return`** wie folgt nutzt:

```
int test() {
  return "blah";
}
```

da hier ein `String` zurückgegeben werden würde, die Methode allerdings vom Typ **`int`** ist. Dies ist allerdings schon ein Fehler zur Kompilierzeit.

Natürlich können Methoden nicht nur Werte von primitive Typen zurückgeben sondern auch Objekte. Für diesen Fall gibt es noch eine besondere Möglichkeit, neben der Rückgabe eines Objekts des eigentlichen Typs: Eine **`null`**-Referenz. **`null`** ist ein spezieller „Wert“ in Java, der jeder Objektreferenz zugewiesen werden kann. Natürlich stellt **`null`** selbst kein wirkliches Objekt dar und ein Zugriff darauf wird in einem Fehler resultieren, eine sogenannte *Ausnahme* oder *Exception* (siehe hierzu auch Abschnitt 8. Es ist allerdings nicht unüblich, eine Methode **`null`** zurückgeben zu lassen, wenn sie anderweitig keinen gültigen Rückgabewert liefern kann oder ein Fehler aufgetreten ist (wiewohl Exceptions für letzteren Fall eine elegantere Variante darstellen, welche aber nicht überall möglich ist).

5 Objektorientierung – ein schlechter Einstieg

Als Einstieg möchte ich hier exemplarisch die Umsetzung eines Moduls einer normalen prozeduralen Sprache (hier Pascal) in eine Java-Klasse durchführen, ehe ich im Detail darauf eingehe, wie so etwas gebildet wird – nicht unähnlich dem „Hello World“-Beispiel zu Anfang.

Nehmen wir uns eine einfache prozedurale Implementation eines Kellerspeichers (Stack) in Pascal zur Hand:[5]

```
unit stacka;

interface

uses elem;

const maxs = 5;

type STACK = record
               elts : array [1..maxs] of ELEMENT;
               ptr : integer;
             end;

procedure empty(var s : STACK);

procedure pop(var s : STACK);

function top(s : STACK) : ELEMENT;

procedure push(var s : STACK; e : ELEMENT);

implementation

function isempty(s : STACK) : boolean;
begin
  isempty := (s.ptr = 0)
end;

procedure mkErrorst(var s : STACK);
begin
  s.ptr := maxs + 1
end;

function iserror(s : STACK) : boolean;
begin
  iserror := (s.ptr > maxs)
end;

procedure empty(var s : STACK);
begin
```

[5] Dieser Quelltext ist der Vorlesung *Programmierungstechnik I* von Prof. Kirste entnommen. Zu finden sind die Vorlesungsunterlagen sowie auch diese Implementation unter `http://www.informatik.uni-rostock.de/mmis/courses/ws0506/23001/`

```
  s.ptr := 0
end;

function top(s : STACK) : ELEMENT;
begin
  if isempty(s) then begin
    top := errorel;
    writeln('Top: Stack Empty')
  end else if iserror(s) then begin
    top := errorel;
    writeln('Top: Stack broken')
  end else
    top := s.elts[s.ptr]
end;

procedure pop(var s : STACK);
begin
  if isempty(s) then begin
    writeln('Pop: Tried to pop from empty stack. Bang.');
    mkErrorst(s)
  end else if iserror(s) then
    writeln('Pop: Stack Empty') {do nothing}
  else
    s.ptr := pred(s.ptr)
end;

procedure push(var s : STACK; e : ELEMENT);
begin
  if iserrel(e) then
    writeln('Push: Bad element') {do nothing}
  else if iserror(s) then
    writeln('Push: Stack broken') {do nothing}
  else if s.ptr = maxs then begin
    writeln('Push: Stack overflow. Bang.'); {stack full}
    mkErrorst(s)
  end else begin
    s.ptr := succ(s.ptr);
    s.elts[s.ptr] := e
  end
end;

begin
end.
```

Eine (sehr) naive Umsetzung dieses Codes in Java soll hier einmal folgen. Zunächst unsere Datenstruktur (die Typdefinition aus Pascal):

```
public class Stack {
  public static final int maxs = 5;

  public int[] elts = new int[maxs];

  public int ptr;
}
```

Soweit erst einmal nicht viel überraschendes. Weiterhin benötigen wir die Umsetzung des Restes der Unit. Erst die verbleibende Konstante:

```
import Stack;

public class StackClass {
  public static final int errorel = -99;
```

sowie die Umsetzung der einzelnen Funktionen und Prozeduren:

```
  public static boolean isEmpty(Stack s) {
    return s.ptr == 0;
  }

  public static void mkErrorStack(Stack s) {
    s.ptr = maxs + 1;
  }

  public static boolean isError(Stack s) {
    return s.ptr > maxs;
  }

  public static void empty(Stack s) {
    s.ptr = 0;
  }

  public static int top(Stack s) {
    if (isEmpty(s)) {
      System.out.println("Stack empty");
      return errorel;
    } else if (isError(s)) {
      System.out.println("Stack broken");
      return errorel;
    } else
      return s.elts[s.ptr];
  }

  public static void pop(Stack s) {
```

```
      if (isEmpty(s)) {
        System.out.println("Stack empty");
        mkErrorStack(s);
      } else if (isError(s)) {
        System.out.println("Error Stack");
        // do nothing
      } else
        s.ptr--;
    }

    public static void push(Stack s, int i) {
      if (i == errorel) {
        System.out.println("Bad Element");
        // do nothing
      } else if (isError(s)) {
        System.out.println("Stack broken");
        // do nothing
      }
      else if (s.ptr == maxs) {
        System.out.println("Stack Overflow");
        mkErrorStack(s);
      } else {
        s.ptr++;
        s.elts[s.ptr] = i;
      }
    }
  }
```

Nun, da wir diese schwere Arbeit hinter uns haben, stellt sich die Frage: *Ist dies nun eigentlich objektorientiert?* Die Antwort fällt eigentlich nicht sehr schwer, auch wenn man bisher nicht viel in der Richtung getan hat.

Schauen wir uns doch einmal an, wo wir hier mit Objekten gearbeitet haben. Nach sorgfältigem Durchsuchen des obigen Quelltextes wird möglicherweise auffallen, daß wir eigentlich *überhaupt nicht* mit Objekten gearbeitet haben. Unser „Objekt“ (der Stack) ist lediglich eine Datenstruktur, alle Methoden, die damit etwas anstellen, lagern in einer völlig anderen Klasse. Im Grunde haben wir hier also nichts weiter als eine Abbildung der Pascal-Sprachkonstrukte **`type`** sowie **`unit`** auf Javas **`class`** – und das nicht mal sonderlich schön. Hier gibt es also noch einiges zu verbessern.

6 Klassen und Interfaces

6.1 Grundlegendes

Schauen wir uns zunächst an, was Klassen in Java eigentlich sind. Eine Klasse kapselt üblicherweise eine Datenstruktur und ihre zugehörigen Funktionen (*Methoden* genannt). Oben wäre also die eigentlich sinnvolle Variante gewesen, die zum Stack gehörenden Methoden auch in die Klasse `Stack` zu schreiben. Weiterhin stellen Klassen ihrerseits immer einen Typ dar. Man kann also Variablen vom Re-

ferenztyp einer bestimmten Klasse deklarieren. Dies haben wir mit `String` schon getan, wie wir uns vielleicht erinnern. Diesen Variablen kann man neue Objekte diesen Typs zuweisen, dies geschieht mit einem sogenannten *Konstruktor,* den wir uns gleich anschauen werden.

Klassen werden wie folgt deklariert:

```
class MyClass {
  ...
}
```

nach Belieben können vor **`class`** noch einige Modifikatoren stehen, die den Sichtbarkeitsbereich einschränken oder besondere Arten von Klassen deklarieren. Hierzu im Anschluß und auch in Abschnitt 6.4 noch mehr. Klassenbezeichner werden, im Gegensatz zu Bezeichnern von Variablen grundsätzlich am Anfang groß geschrieben.

Im Allgemeinen kann man davon ausgehen, daß nahezu alles, was man in Java benötigt, durch eine Klasse repräsentiert wird. Leicht zu merken und nicht völlig fern der Wirklichkeit ist der Spruch „Klassen in Massen“, der dies illustriert. Selbst wenn man nur eine simple Datenstruktur benötigt, wird man eine neue Klasse anlegen. Dies hat den weiteren Vorteil, daß man zugehörige Methoden gleich dort mit hinein sortieren kann. In diesem Sinne könnten Klassen auch als Ersatz der Module aus anderen Sprachen angesehen werden (**`unit`**s in Pascal beispielsweise). Klassen mitsamt ihrer Packages erzeugen im Grunde einen Namespace, der eindeutigen Zugriff auf Methoden und Member möglich macht.

6.2 Sichtbarkeit

Klassen, sowie ihre Daten, Methoden und Konstruktoren haben einen gewissen Sichtbarkeitsbereich innerhalb von Java. Dies kann und sollte man dazu nutzen, Implementationsdetails vor dem Nutzer der Klasse zu verbergen. Versucht man, außerhalb des deklarierten Sichtbarkeitsbereiches auf eine Klasse oder einen Bestandteil einer Klasse zuzugreifen, so schlägt dies fehl. Standardmäßig sind Klassen und ihre Bestandteile (*Member* genannt) lediglich im gleichen *Package* sichtbar (zu Packages mehr in Abschnitt 7). Weiterhin gibt es den **`public`**-Modifikator, der bewirkt, daß eine Klasse oder ein Member überall sichtbar ist. Wie leicht zu erraten ist, steht natürlich auch das Gegenteil zur Verfügung: **`private`**, welches nur für Member zulässig ist, nicht für Klassen und den Zugriff auf die Klasse einschränkt, in der die Deklaration steht. Außerhalb dieser Klasse ist ein Zugriff nicht möglich. Und schlußendlich gibt es noch **`protected`**, welches sich im Grunde ähnlich wie **`private`** verhält, aber den Zugriff in abgeleiteten Klassen (siehe hierzu auch Abschnitt 6.6) erlaubt.

Beispiele:

```
public class Test { // öffentliche Klasse
  private int i; // nur innerhalb dieser Klasse sichtbar

  public void blah() { // öffentliche Methode
    return;
  }

  protected int getOne() { // geschützte Methode
    return 1;          // sichtbar auch in abgeleiteten Klassen
  }
```

```
}
```

6.3 Member

6.3.1 Statische und nicht statische Member

Klassen können zunächst zweierlei Arten von Membern enthalten: solche, die als **`static`** deklariert wurden und solche, die es nicht wurden. Was zunächst banal und offensichtlich klingt, ist jedoch ein nicht zu unterschätzender Unterschied. Statische Member gehören zur jeweiligen Klasse, nicht-statische hingegen immer zu einer *Instanz* dieser Klasse, also einem Objekt. Statische Methoden können nicht auf nicht-statische Member zugreifen, was an sich klar ist, da eine statische Methode nicht auf ein Objekt festgelegt ist und damit keine dynamischen Daten zum Arbeiten hat, andersherum ist dieser Zugriff allerdings möglich.

6.3.2 Felder

Als *Felder* werden Variablen oder Konstanten innerhalb einer Klasse bezeichnet. Sie sind in den meisten Fällen **`private`** deklariert und dienen zur Speicherung diverser interner Zustände oder anderen Datenstrukturen, die man nicht unbedingt immer mit nach außen tragen muß.

6.3.3 Methoden

Methoden sind im Grunde ähnlich wie Variablen, nur daß sie Code ausführen, statt direkt einen Wert zurückzuliefern. Dieser Vergleich hinkt zwar, das soll uns hier aber nicht weiter stören. Methoden werden grundsätzlich als Member einer Klasse deklariert und zwar in der folgenden Form:

```
    Typ Bezeichner([Parameter])
```

was schon sehr an die in Abschnitt 1.1 gezeigte Syntax erinnert, nur daß man einer Methode auch Parameter übergeben kann. Die Deklaration der Typen dieser Parameter erfolgt in Klammern nach dem Methodennamen. Direkt danach folgt ein Block mit dem sogenannten *Methodemrumpf*. Dieser stellt die Implementation der Methode dar. Über Rückgabewerte und auch den besonderen Typ **`void`** habe ich schon in Abschnitt 4.8 geschrieben. Der Typ des Rückgabewertes zusammen mit den Typen der Parameter bildet die sogenannte *Signatur* der Methode.

Es kann in einer Klasse mehrere Methoden mit gleichem Namen und unterschiedlicher Signatur geben (Polymorphismus):

```
    int incMe(int i) {
      return i + 1;
    }

    double incMe(double d) {
      return d + .1;
    }

    int incOrZero() {
      return 0;
```

```
    }

    int incOrZero(int i) {
      return i + 1;
    }
```

Dies ist unter anderem hilfreich, wenn man verschiedene Typen anders behandeln muß oder aber für verschiedene Aufrufe der gleichen Methode unterschiedliche Parameter nutzen oder gar Teile weglassen kann.

6.3.4 Konstruktoren

Konstruktoren sind besondere Methoden, die dazu da sind, ein Objekt zu erzeugen und zu initialisieren. Tatsächlich muß man sich um die eigentliche Erzeugung des Objektes nicht mehr kümmern, das wird einem abgenommen. Aber man hat in einem Konstruktor schon Zugriff auf die Felder und Methoden des Objektes. Konstruktoren sind damit nie `static`. Weiterhin müssen Konstruktoren grundsätzlich so heißen, wie die Klasse, in der sie deklariert sind und haben keinen Rückgabewert. Allerdings können nach wie vor Sichtbarkeitsmodifikatoren und andere davorgeschrieben werden. Wie auch bei Methoden, so gibt es auch bei den Konstruktoren die Möglichkeit, mehrere Konstruktoren mit unterschiedlicher Signatur zu haben (allerdings bezieht sich hier *Signatur* verständlicherweise nur auf die Parameter, da es keinen Rückgabewert gibt).

6.3.5 Finalizer

Mancher mag aus Sprachen wie C++ vielleicht eine weitere besondere Art einer Methode kennen: den *Destruktor.* Ein Konzept wie dieses hat Java nicht direkt, Objekte werden vom Garbage Collector freigegeben, wenn sie nicht mehr referenziert werden und alle ihre Bestandteile lösen sich damit ebenso auf, da selbigen ja nun die Referenz fehlt. Folglich müssen Objekte in Java nicht mehr unbedingt hinter sich aufräumen, wie es in Sprachen mit expliziter Speicherverwaltung nötig ist. Dennoch gibt es etwas ähnliches, eine besondere Methode namens `finalize`, die vom Garbage Collector aufgerufen wird bevor er das Objekt vollends zerlegt. Allerdings hat man auf die Ausführung dieser Methode keinerlei Einfluß. Die Spezifikation sagt lediglich, daß sie zwischen Beendigung des Konstruktors und vor dem Freigeben der vom Objekt belegten Ressourcen aufgerufen wird; wann dies passiert und aus welchem Thread, ist undefiniert. Aus diesem Grunde sollte man die Anforderungen von `finalize` an das Laufzeitverhalten so gering wie möglich halten.

6.3.6 Innere Klassen

Klassen können neben den oben genannten Dingen auch noch selbst Klassen enthalten, die hier lediglich der Vollständigkeit halber kurz genannt werden sollen. Ich werde hier nicht weiter im Detail darauf eingehen, da dies ein einigermaßen fortgeschrittenes Thema ist.

Innere Klassen sind normalerweise an eine Instanz iher äußeren Klasse gebunden. Weiterhin gibt es sogenannte *lokale innere Klassen,* welches Klassen sind, die innerhalb einer Methode deklariert wurden. *Anonyme Klassen* hingegen sind lokale innere Klassen, die nicht benannt wurden.

6.4 Weitere Feinheiten

Klassen können als abstrakt (über das Schlüsselwort **`abstract`**) deklariert werden, was bedeutet, daß diese Klassen unvollständig implementiert sind. Abstrakte Klassen können *abstrakte Methoden* beinhalten, die mit dem gleichen Modifikator gekennzeichnet sind. Abstrakte Methoden dürfen allerdings nur in abstrakten Klassen vorkommen. Abstrakte Klassen dürfen nicht instanziiert werden, d. h. es darf kein Objekt aus einer abstrakten Klasse erzeugt werden. Dies resultiert in einem Kompilierfehler.

Eine andere, fast schon entgegengesetzte, Deklaration für Klassen ist die Möglichkeit, sie **`final`** zu deklarieren. Von Klassen, die in dieser Art deklariert wurden dürfen keine abgeleiteten Klassen erstellt werden.

6.5 Der Stack – revisited

Mit dem nun gerade neu erworbenen Wissen darüber, was Klassen sind und wie man sie zweckmäßigerweise schreibt, können wir nun unser Beispiel vom Anfang wieder hervorholen und es mal etwas sinnvoller schreiben. Diesmal kann unser Stack auch beliebige Objekte beinhalten, nicht nur Ganzzahlen.

Die Klasse `Stack` können wir schon fast so behalten und erweitern sie lediglich um den Kram, der vorher in der Hilfsklasse stand:

```
public class Stack {
  /** Maximale Anzahl an Elementen */
  private int maxs;

  /** Konstante für das Fehlerelement */
  public static final Object errorel = null;

  /** Das Array der Elemente */
  public Object[] elts;

  /** Zeiger auf das oberste Element im Stack */
  public int ptr;

  /**
   * Erstellt einen neuen Stack mit der angegebenen
   * Maximalgröße
   *
   * @param size
   *          die maximale Größe des Stacks
   */
  public Stack(int size) {
    ptr  = 0;
    maxs = size;
    elts = new Object[size];
  }

  /**
```

```
     * Gibt zurück, ob der Stack leer ist.
     *
     * @return true falls der Stack leer ist, sonst false
     */
    public boolean isEmpty() {
      return ptr == 0;
    }

    /** Wandelt den Stack in einen Fehlerstack um */
    private void mkErrorStack() {
      ptr = maxs + 1;
    }

    /**
     * Gibt zurück, ob der Stack ein Fehlerstack ist
     *
     * @return true, falls der Stack ein Fehlerstack ist,
     *         sonst false
     */
    public boolean isError() {
      return ptr > maxs;
    }

    /**
     * Gibt das oberste Element vom Stack zurück
     *
     * @return das oberste Element vom Stack falls der
     *         Stack nicht leer ist, sonst errorel
     */
    public Object top() {
      if (isEmpty()) {
        System.err.println("Stack empty");
        return errorel;
      } else if (isError()) {
        System.err.println("Stack broken");
        return errorel;
      } else
        return elts[ptr];
    }

    /** Entfernt das oberste Element vom Stack */
    public void pop() {
      if (isEmpty()) {
        System.err.println("Stack empty");
        mkErrorStack();
      } else if (isError()) {
```

```
        System.err.println("Error Stack");
        // nichts machen
      } else
        elts[ptr] = null;
        ptr--;
    }

    /**
     * Fügt ein Element in den Stack ein
     *
     * @param elem
     *          das einzufügende Element
     */
    public void push(Object elem) {
      if (elem == errorel) {
        System.err.println("Bad Element");
        // do nothing
      } else if (isError()) {
        System.err.println("Stack broken");
        // do nothing
      }
      else if (ptr == maxs) {
        System.err.println("Stack Overflow");
        mkErrorStack();
      } else {
        ptr++;
        elts[s.ptr] = elem;
      }
    }
  }
```

Ich habe hier auch Gebrauch einer speziellen Kommentarform gemacht. Diese als *JavaDoc* bezeichnete Form ist eine einheitliche Variante, Quelltext zu dokumentieren und kann automatisch in verschiedene Formate umgewandelt werden, beispielsweise HTML. Weiterhin wurde die Größenbeschränkung für den Stack variabel gemacht und kann nun einem Konstruktor übergeben werden. Außerdem wurden Methoden, die nur zur inneren Verwendung da sind, **`private`** deklariert und nun braucht keine Methode mehr als zusätzlichen Parameter den Stack, auf dem sie arbeitet, denn dieser ist implizit schon durch das Objekt gegeben, auf dem die Methode aufgerufen wird (tatsächlich wird das Objekt selbst noch mit übergeben, aber dies passiert hinter den Kulissen und ist für den Programmierer so nicht sichtbar – lediglich in der Existenz der speziellen Referenzvariable **`this`**, welche immer auf das aktuelle Objekt zeigt).

6.6 Vererbung

Vererbung ist ein mächtiges Konzept in der objektorientierten Programmierung und wird auch als *Generalisierung* (zum Beispiel in UML) bezeichnet. Ich habe in vergangenen Abschnitten schon ab und zu

von *abgeleiteten* Klassen geschrieben. Damit ist Vererbung gemeint. Wenn eine Klasse von einer anderen „erbt“, so übernimmt sie alle Felder, Methoden, etc. von der „Elternklasse“ und kann diese wahlweise um weitere Member ergänzen oder aber bestehende überschreiben und damit ihr Verhalten zu ändern.

In Java dürfen Klassen nur von maximal einer anderen Klasse erben. Dies ist eine Designentscheidung und liegt darin begründet, daß Vererbung von mehr als einer Klasse im Allgemeinen sehr komplex zu implementieren ist. Im Quelltext wird Vererbung durch das Schlüsselwort **`extends`** kenntlich gemacht:

```
public class MyClass extends MyOtherClass {
  ...
}
```

In diesem Beispiel würden für `MyClass` sämtliche Methoden, Konstruktoren, etc. zur Verfügung stehen, die in `MyOtherClass` ebenfalls deklariert sind und, solange man sie nicht überschreibt, ist die Implementation sogar identisch. Zum Überschreiben einfach einmal ein kleines Beispiel: Angenommen, wir haben eine Klasse, die natürliche Zahlen repräsentiert, diese werden intern in einem long abgelegt, so haben wir ausreichend Platz. Wir definieren die Methoden `inc` und `dec` für das Inkrementieren und Dekrementieren einer Zahl sowie `zero` als Konstante für die Null. Ebenfalls soll `dec()` auf einer 0 wieder `0` ergeben, der Zahlenbereich ist also nach unten *gesättigt.* Implementieren wir das mal eben:

```
public Class Nat {
  public static final long zero = 0L;

  private long theNumber;

  public Nat() {
    theNumber = 0L;
  }

  public Nat(long l) {
    // Vorsicht mit negativen Zahlen
    if (l < 0)
      theNumber = 0
    else
      theNumber = l;
  }

  public void inc() {
    theNumber++;
  }

  public void dec() {
    // nur dekrementieren, wenn größer 0
    if (theNumber > 0)
      theNumber--;
  }
```

Und damit wir die Zahl auch ausgeben können, folgt noch die überladene Methode `toString()`:

```
    public String toString() {
      return new Long(theNumber).toString();
    }
  }
```

Soweit haben wir jetzt natürliche Zahlen. Rechnen können wir nicht sehr aber das ist auch nicht weiter wichtig. Nun kommt uns in den Sinn, daß es vielleicht schön wäre, auch ganze Zahlen, also den negativen Zahlenbereich zusätzlich noch zu haben. Die restliche Funktionalität soll die gleiche bleiben. Es bietet sich hier also an, die Klasse `Ganz` von `Nat` erben zu lassen:

```
public class Ganz extends Nat {
```

Nun haben wir schon alles, was wir in `Nat` auch schon hatten, wir müssen uns also nur darum kümmern, den negativen Zahlenbereich hinzuzufügen. Wie wir an obiger Implementation sehen, gibt es einen Konstruktor und eine Methode, die jeweils darauf achten, daß `theNumber` keine negativen Zahlen annimmt. Folglich müssen wir diese überschreiben. Weiterhin müssen Konstruktoren von abgeleiteten Klassen immer neu implementiert werden, man kann allerdings mit dem Schlüsselwort **`super`** Methoden und Konstruktoren der „Elternklasse“ aufrufen.

```
    public Ganz() {
      // Aufrufen des übergeordneten Konstruktors
      super();
    }

    // Überschreiben des Konstruktors
    public Ganz(long l) {
      theNumber = l;
    }

    // Überschreiben der Methode
    public void dec() {
      theNumber--;
    }
```

Anders als beispielsweise in Delphi ist es in Java nicht nötig, das Überschreiben von Methoden explizit durch ein Schlüsselwort zu kennzeichnen. Man mag dazu stehen wie man mag, allerdings sind überschriebene Methoden selten eine schwer zu findende Fehlerquelle, insofern erübrigt sich eine explizite Kennzeichnung meistens und schafft oft eher nur Verärgerung, wenn man die Kennzeichnung vergißt.

Nun wollen wir unsere Klasse `Ganz` noch um die unäre Negation erweitern, also das Negieren der gespeicherten Zahl. Dies ist eine Methode, die in `Nat` unsinnig wäre, aber in `Ganz` hat sie durchaus Sinn, also basteln wir sie uns einfach:

```
    public negate() {
      thenumber *= -1;
```

```
    }
}
```

Und sind hier auch einmal am Ende mit der Klasse `Ganz`. Sie bietet die gleichen Methoden wie `Nat` und sogar noch eine weitere, die `Nat` nicht hat. Allerdings ist sie von der Funktionalität her eine *echte Obermenge* von `Nat` (was uns auch ein wenig an die tatsächlichen Zahlenbereiche erinnert). Wenn man also konsequent ist, so kann man `Ganz` auch anstelle von `Nat` einsetzen. Tatsächlich ist so etwas sogar eine recht häufige Anwendung von Vererbung, indem man eine sehr allgemeine Oberklasse erstellt und spezielle Funktionalität dann in abgeleiteten Klassen unterbringt. So kann man überall, wo man eine der Unterklassen haben möchte, auch die Oberklasse hinschreiben und es ist dann üblicherweise egal, welche der Unterklassen man benutzt, da sie alle gleichermaßen funktionieren. Eine kleine Erweiterung dieses Konzeptes wird uns auch noch im folgenden Abschnitt bei den Interfaces begegnen.

6.7 Interfaces

Vererbung bei Klassen hat den (gewollten) Nachteil, daß man lediglich von einer Klasse erben kann. Da dies für einige komplexere Anwendungen nicht ausreicht, kamen noch sogenannte *Interfaces* hinzu. Interfaces sind im Grunde die öffentlich sichtbaren Bestandteile einer Klasse ohne Implementation. Sie ähneln hier ein wenig abstrakten Klassen in dem Sinne, daß man aus Interfaces kein Objekt erstellen kann und die Implementation unvollständig ist. Nur, daß die Implementation bei Interfaces so ziemlich am unvollständigsten ist, was man sich überhaupt vorstellen kann.

Interfaces dürfen, wie schon erwähnt, lediglich öffentliche Bestandteile enthalten. Konkret bedeutet dies, daß man nur öffentliche Konstanten (**`public static final`**) sowie öffentliche Methoden deklarieren darf. Die Methoden allerdings ohne Implementation, wie schon erwähnt:

```
public interface MyInterface {
  // Konstante
  public static final int x = 5;
  // Methodensignatur
  public void doSomething();
}
```

Zwischen Interfaces darf man allerdings Mehrfachvererbung machen:

```
public interface MyInterface extends Int1, Int2, Int3 {
  ...
}
```

Außerdem dürfen Klassen mehrere Interfaces *implementieren.* Man beachte hier, daß es sich von der Terminologie her schon von der Vererbung unterscheidet. Klassen *implementieren* Interfaces, aber *erben* von anderen Klassen. Quelltextmäßig sieht das folgendermaßen aus:

```
class MyClass implements MyInterface {
  // vom Interface deklarierte Methode,
  // muß implementiert werden:
  public void doSomething() {
```

```
        ...
    }

    ...
}
```

Wie schon im Kommentar im Beispiel angedeutet müssen vom Interface definierte Methoden implementiert werden, vergißt man dies, so meldet einem der Compiler einen Fehler.

Interfaces sind eine mächtige, allerdings nicht allmächtige Methode der Mehrfachvererbung. Es gibt einzelne (seltene) Probleme, die sich nur mit „echter“ Mehrfachvererbung aber nicht mit Interfaces lösen lassen.

Java hat einige sogenannte *“tagging interfaces”*, welche im Grunde leer sind und nur eine Art Marke darstellen, welchen Zweck eine Klasse hat. Dies ist vom Standpunkt des Programms oft völlig irrelevant aber es erleichtert die Lesbarkeit, wenn man sie benutzt und sie geben auch eine generelle Menge von Objekten mit ähnlichem Sinn und Zweck an, so daß man sie auch als Typ verwenden kann.

7 Packages

Packages sind neben Klassen die grundelegende hierarchische Ordnungsstruktur in Java. Sie bilden direkt eine Ordnerstruktur im Dateisystem ab, allerdings sind die Komponenten eines Packages durch Punkte getrennt. Aus dem Pfad unterhalb eines Projekts

```
de/hypftier/vortrag/java/test
```

würde beispielsweise der Packagename

```
de.hypftier.vortrag.java.test
```

werden. Hier zeigt sich auch das von Sun vorgeschlagene (oder gar vorgeschriebene) Benennungsschema für öffentliche Packages. Im Grunde ist es zwar egal, wie man seine Packages gliedert und unterteilt, aber sobald man vorhat, den Code öffentlich zu machen, sollte man diesem Schema folgen, um mögliche Kollisionen mit Packages anderer Leute oder Organisationen zu vermeiden. Dieses Schema beginnt mit der Domain desjenigen, der das Package geschrieben hat; und zwar komponentenweise rückwärts geschrieben. D. h. es beginnt mit der Top-level Domain beispielsweise `de`, und geht dann schrittweise rückwärts weiter. Danach käme der normale Domainname, also hätten wir inzwischen zum Beispiel `de.uni_rostock`. Danach kämen noch eventuelle Subdomains und schlußendlich eine eigene, beliebige Ordnung. Sun Microsystems behält sich hier das Package `com.sun` und alle Unterpackages vor (und wenn man portabel zu anderen Java-VMs sein möchte, so sollte man diese Packages auch nicht benutzen).

Sämtliche von Java nativ bereitgestellten Klassen finden sich im `java`-Package, welches noch einmal unterteilt ist, beispielsweise in `java.util` oder `java.lang`, etc.

Festlegen, zu welchem Package eine Klasse gehört, kann man über die Anweisung

```
package de.toll.bin.ich.mypackage
```

vor der Deklaration der Klasse in der Datei. Dies erfordert, daß die Klassendatei sich in genau dem Verzeichnis befindet, das vom Package vorgegeben wird.

Nun ist es nicht immer angenehm, den kompletten Packagepfad vor jede Klasse zu schreiben, die man verwenden möchte, daher kann man Klassen auch *importieren.* Dies bedeutet, daß sie im aktuellen Namespace sichtbar und zugreifbar sind, auch wenn man nicht das Package mit davor schreibt. Dies kann man über die **`import`**-Anweisung machen. Entweder für ganze Packages (d. h. alle Klassen in dem Package):

```
import javax.crypto.*;
```

oder aber für einzelne Klassen:

```
import java.awt.AWTKeyStroke;
```

8 Fehlerbehandlung

Fehlerbehandlung in Java ist C++ entlehnt und verhält sich damit auch ziemlich ähnlich. Ein Fehler wird durch das Auftreten einer sogenannten *Ausnahme* bzw. *Exception.* ausgelöst. Diese unterbrechen den Kontrollfluß des Programms an der Stelle ihres Auftretens. An dieser Stelle gibt es nun zwei Möglichkeiten, mit ihnen umzugehen. Einerseits kann man Ausnahmen behandeln oder aber festlegen, daß sich die Methode, in der sie auftritt nicht darum kümmert und die Ausnahme einfach an die aufrufende Methode weiterreichen.

Als Grundregel gilt, daß Ausnahmen immer entweder behandelt oder weitergereicht werden müssen. Ausgenommen hiervon sind Ausnahmen, die von `RuntimeException` abgeleitet sind, aber auch hier gehört es zum guten Ton, sie zu behandeln. Ausnahmen werden mit dem Schlüsselwort **`throw`** ausgelöst:

```
throw new IndexOutOfBoundsException();
```

Eine Ausnahme wird über eine **`throws`**-Klausel in der Methodendeklaration explizit an die aufrufende Methode weitergereicht:

```
int doSomething() throws IndexOutOfBoundsException {
  ...
  throw new IndexOutOfBoundsException();
  ...
}
```

Diese Regelung hat außerdem den Vorteil, daß Java immer weiß, ob irgendwo unbehandelte Ausnahmen sind, da jede Methode explizit deklarieren muß, wenn sie bei der Ausführung eventuell eine Ausnahme auslösen kann. Manchmal ist es zwar nervig, den halben Code mit dem gleich folgenden **`try-catch-finally`** zu spicken, aber es hilft auf jeden Fall, Programmabstürze zu vermeiden, da es bei diesem Konstrukt keinen Grund gibt, das Programm zu beenden, es fängt sich bei Auftreten eines Fehlers meist selbst wieder und das ist eigentlich genau der Zweck dieser Art der Fehlerbehandlung.

Strukturierte Ausnahmenbehandlung erfolgt über die **`try`**-Anweisung:

```
try {
  // Block, der eine Ausnahme auslösen kann
} catch (Ausnahmetyp e) {
  // Fehlerbehandlung
} catch (AndererAusnahmetyp e) {
  // mehr Fehlerbehandlung
} finally {
  // Aufräumarbeiten
}
```

Wie hier zu sehen ist, schließt der **try**-Block alles ein, was eine Ausnahme auslösen könnte. Manchmal ist es sinnvoll, gleich allen Code, der von einer Anweisung abhängt, die eine Ausnahme auslösen kann, mit in das **try** zu schreiben, da der Code meist im Fehlerfall ohnehin nicht mehr ausgeführt werden kann. Nach dem **try**-Block folgen beliebig viele **catch**-Blöcke, die jeweils einen bestimmten Ausnahmetyp „fangen“. Diese werden von oben nach unten durchlaufen und der erste Block, der vom Typ her paßt, wird ausgeführt. Es empfiehlt sich hier also, mit speziellen Ausnahmen anzufangen und evtl. als letztes als eine Art “catch-all” ein **catch** (`Exception e`) zu verwenden. Anders als bei **switch** benötigt man hier kein **break**, um ein **catch** zu beenden. Dies sind kleine Inkonsistenzen, die es so mit sich bringt, wenn man Syntax von anderen Sprachen übernimmt, die abwärtskompatibel sein wollten, allerdings modernere Konstrukte mit in die Sprache gebracht haben; C++ hat genau dieses Problem.

Schlußendlich kann noch ein **finally**-Block kommen, welcher *immer* ausgeführt wird – nach dem fehlerfreien Durchlaufen des **try**, nach einem **catch** und allgemein immer. Dies bietet einem die Möglichkeit, hinter sich aufzuräumen, falls nötig.

Mit **finally** ergeben sich allerdings eine ganze Reihe interessanter Verhaltensweisen im Zusammenhang mit **break** und **return**. Tritt innerhalb eines **try**-**catch**-**finally**-Konstruktes ein **break** oder **return** auf, welches dieses Konstrukt komplett verlassen würde, so werden von innen nach außen zunächst noch sämtliche **finally**-Blöcke abgearbeitet. Dies kann zu seltsamen Resultaten führen:

```
private int i = 0;

public int puzzle() {
  try {
    return i++;
  } catch (RuntimeException e) {
    i++;
  } finally {
    i++;
    return 5;
  }
}
```

Zugegeben, ein recht konstruiertes Beispiel, aber wer Lust hat, darf gerne einmal raten, was die Methode zurückgibt und welchen Wert `i` hinterher hat.[6]

[6] Es sei hier einmal verraten: Der Rückgabewert ist `5`, da das **return** im **finally** als letztes aufgerufen wird und damit die eigentliche Rückkehr aus der Methode bewirkt und `i` ist nach Aufruf dieser Methode `2`, da der Ausdruck `i++` zweimal ausgeführt wird.

Es gibt neben Exceptions auch noch sogenannte *Errors.* Dies sind meist von der virtuellen Maschine ausgelöste Fehler, von denen sich ein Programm normalerweise nicht mehr erholen kann. Dies sind schwerwiegende Fehler wie beispielsweise `OutOfMemoryError` oder `StackOverflowError`. Sie seien hier nur der Vollständigkeit halber erwähnt, ohne weiter darauf einzugehen.

Literatur

[1] James Goslin, Bill Joy, Guy Steele, Gilad Bracha: *The Java Language Specification.* Addidon-Wesley, Boston 2005. ISBN 0-321-24678-0

[2] Christian Ullenboom: *Java ist auch eine Insel.* Galileo Computing, Bonn 2007. ISBN 978-3-89842-838-5

[3] Sun Microsystems: *The Java Tutorials. Trail: Learning the Java Language.* URL: `http://java.sun.com/docs/books/tutorial/java/index.html` (abgerufen am 28. Januar 2007)

Abbildungsverzeichnis

Worte des Dankes

Ich möchte an dieser Stelle noch einmal einigen einzelnen Leuten für Hilfe zum Thema Java danken: Dr. Elmar Ludwig, Andreas Kohn sowie Andreas Tschritter. Ebenfalls dankend erwähnt sei hier Andreas Dähn für das Beantworten vieler Anfängerfragen zum Thema LaTeX.