Eine kurze Einführung in die Programmiersprache

Java (Version 1.2)

basierend auf dem Java Developer's Kit der Firma Sun

von Christoph Moder

http://www.skriptweb.de

Hinweise (z.B. auf Fehler) bitte per eMail an uns: mail@skriptweb.de - Vielen Dank.

Inhaltsverzeichnis

Was ist Java?

Mit dem Ziel, einfache und möglichst fehlerfreie Programme für elektronische Geräte schreiben zu können, die plattformunabhängig auf beliebigen Hardwareachitekturen laufen können, wurde bei Sun die Programmiersprache *Oak* entwickelt (aus der Erfahrung heraus, dass dies mit C++ schwierig zu realisieren ist). Mit der Verbreitung des Internet stellte Sun fest, dass diese Eigenschaften Oak ideal für das Internet machen, und benannte die Sprache nach *Java* um.

Die offizielle Definition von Sun benutzt folgende Schlüsselwörter, um Java zu charakterisieren:

- **Einfach**: Java ist stark an C++ angelehnt, wegen der Verbreitung von C++. Allerdings fehlen die Mehrfachvererbung von Klassen, Operatorenüberladung, automatische Konvertierung von Datentypen , das Präprozessor-System, Pointer, mehrdimensionale Arrays und Strukturen - weil diese Dinge entweder zu kompliziert (z.B. Mehrfachvererbung), unlogisch, veraltet (Präprozessor), hardwarenah (Pointer) oder generell sehr fehlerträchtig sind. Java verzichtet auf jene Teile von C++, die hardwareabhängiges und fehlerträchtiges Programmieren fördern.
- **Objektorientiert**: Daten (= *Felder*) und Programmcode (= *Methoden*) sind in *Objekten* kombiniert, dadurch sind die Daten *gekapselt* (sie können nicht von überall verändert werden, sondern nur über definierte Methoden des Objekts (die Methoden des Objekts können von einem anderen Objekt aufgerufen werden)) - man behält leichter den Überblick, von wem die Daten verändert werden und erkennt so Fehler einfacher; nicht jeder beliebige Programmteil kann jede beliebige Variable verändern).
  Dies führt zu *Abstraktion*: ein fertiges Objekt kann als Black Box gesehen werden, über dessen innere Funktionsweise man nichts wissen muss (*information hiding*, nur die öffentlichen Methoden interessieren), und zu *Modularität* (= Wiederverwendbarkeit und Erweiterbarkeit). Methoden definieren also die Verbindung eines Objekts zur „Außenwelt“.
  Außerdem bedeutet Objektorientierung auch *Vererbung* und damit *Wiederverwendbarkeit*, d.h. eine Klasse erbt die Eigenschaften einer anderen Klasse, man muss diese nicht noch einmal programmieren. Während eine *Superklasse* ziemlich allgemein sein kann, sind die *Subklassen*, die die Methoden der *Superklasse* erben, spezialisierter: sie erweitern und konkretisieren die Methoden der Oberklasse, fügen Methoden hinzu, und falls nötig überschreiben sie Methoden (d.h. definieren Methoden mit den selben Namen wie in der Oberklasse, so dass die Methoden der Oberklasse ersetzt werden).
  Man kann mehrere Methoden gleichen Namens definieren, die unterschiedliche Parameter akzeptieren. Der Computer entscheidet anhand der Anzahl und des Typs der Parameter, welche dieser gleichnamigen Methoden benutzt wird. Man kann also Methoden schreiben, die mit vielen verschiedenen Variablentypen umgehen können (für jeden Typ eine eigene Methode, alle haben den selben Namen), dies nennt man *Überladung* von Methoden.
  Subklassen einer gegebenen Klasse werden immer noch als gleicher Typ wie die Superklasse angesehen, das bezeichnet man als *Polymorphie*. Dann kann man Operationen mit verschiedenen Klassentypen durchführen, falls diese Klassen eine gemeinsame Superklasse haben; z.B. kann man bei einer Methode, die als Parameter eine Instanz der Superklasse erwartet, auch eine Instanz der Subklassen der Superklasse angeben. (Um das erreichen zu können, definiert man in der Superklasse oft Methoden, obwohl man die dort überhaupt nicht implementieren kann und die deshalb abstrakt bleiben müssen - aber die Subklassen, die diese Methoden überschreiben, können die Methoden dann auch mit anderen Subklassen verwenden, weil sie in der gemeinsamen Superklasse bereits deklariert wurden.)
  Eine *Klasse* ist ein Prototyp, in dem die Variablen und Methoden definiert werden - aber sonst nichts, es wird kein Speicher reserviert, eine Klasse ist nur eine abstrakte, allgemeine Definition, eine Schablone. Ein *Objekt* dagegen ist die *Instanz* einer Klasse, eine spezifische Kopie der Klasse, d.h. die konkrete Umsetzung dieser Definition; eine Klasse kann beliebig viele Instanzen haben (z.B. Klasse „Fahrrad“ definiert, was ein Fahrrad allgemein ist und welche Eigenschaften es hat, die Instanzen der Klasse bezeichnen jeweils genau ein bestimmtes Fahrrad). Bei der Erzeugung einer Instanz wird die abstrakte Klasse „zum Leben erweckt“, es wird Speicher

belegt, das Objekt ist funktionsfähig.

- **Dezentral**: Auf Objekte kann nicht nur lokal, sondern auch z.B. über das Internet zugegriffen werden.
- **Kompiliert**: Java wird zu einem Bytecode kompiliert, d.h. zu einer symbolischen Maschinensprache. Dieser Bytecode wird von der *Java virtual machine* (JVM) interpretiert. Vorteil: durch die Kompilierung hat man einen kompakten, leicht zu interpretierenden und damit schnellen Code, der soweit aufbereitet ist, wie die verschiedenen Hardwarearchitekturen Gemeinsamkeiten haben (z.B. vier Register, Stack, Heap, 32-Bit-Adressen). Trotzdem bleibt der Code maschinenunabhängig, weil dieser Code von der maschinenspezifischen JVM ausgeführt wird.
- **Stabil**: Durch starke Typenüberprüfung können viele Fehler bereits während des Kompilierens gefunden werden, außerdem hat Java kaum Zugang zu Hardware und Systeminterna (und kann dort nichts manipulieren) - deshalb ist es sehr unwahrscheinlich, dass Java-Programme abstürzen.
- **Sicher**: Wegen der JVM kann und wird auch zur Laufzeit eine Fehlerüberprüfung durchgeführt, und wegen des sehr eingeschränkten Zugriffs auf Hardware und Speicher kann ein Java-Programm das System praktisch nicht schädigen.
- **Architekturneutral**: Java-Programme laufen auf jeder Hardware und jedem Betriebssystem (für das es eine JVM gibt).
- **Portierbar**: Die Datentypen sind standardisiert, z.B. ein *int* hat immer 32 Bit (im Gegensatz zu C).
- **Leistungsfähig**: Java-Programme sind im Normalfall kaum langsamer als C++-Programme, weil die Umsetzung von Bytecode in Maschinencode ziemlich einfach ist.
- **Multithreading**: Java unterstützt *Multithreading*, d.h. mehrere Aufgaben können parallel erledigt werden (z.B. durch Multitasking oder Multiprocessing).
- **Dynamisch**: Zugehörige Klassen werden erst zur Laufzeit eingebunden (*dynamisches Linken*).

Was soll dieser Text?

Dieser Text ist als kurze Einführung in die Java-Programmierung gedacht, systematisch aufgebaut und verständlich geschrieben, ohne ausschweifenden Text und ohne zig Beispiele, damit man sich nicht durch hunderte von Seiten lesen muss, nur um das Wichtige zu erfahren. Hier wird praktisch nur die nackte Sprachdefinition erklärt, abstrakte Definitionen (Was ist eine Hochsprache? Was ist ein Algorithmus?) sind weggelassen, und das Programmieren an sich wird auch nicht erklärt, man sollte also schon irgendwie einmal etwas mit einer prozeduralen oder objektorientierten Programmiersprache zu tun gehabt haben. Zusammen mit der Java-API-Dokumentation (beschreibt ausführlich die zu Java dazugehörenden Klassen) und dem Java-Tutorial (Beispiele zu allen möglichen Fragen und Anwendungsgebieten en masse) hat man dann alles, was man zum Java-Programmieren braucht.

Installation und Benutzung des SDKs

Das Java-SDK kann man bei Sun downloaden (aktuell: http://java.sun.com/j2se/index.html). Enthalten sind:

- *javac*, der Java-Compiler. Bedienung: *javac Quellcodedatei.java*. Man kann mehrere Quellcodedateien angeben, dass wird v.a. dann wichtig, wenn diese voneinander abhängen.
- *java*, der Java-Bytecode-Interpreter. Bedienung: *java Programmdatei.class*
- *appletviewer*, der Java-Applets in HTML-Seiten anzeigt (als Alternative zu den Java-fähigen Browsern Navigator und Internet Explorer. Bedienung: *appletviewer HTML-Datei.html*

Diese Programme finden sich im *bin*-Unterverzeichnis; sinnvollerweise setzt man die *PATH* -Umgebungsvariable auf dieses Verzeichnis. Eine weitere Umgebungsvariable für Java ist *CLASSPATH*, sie enhält alle Pfade, in denen sich Java-Klassendateien befinden.

Bestandteile eines Java-Programms

Bei einem Java-Programm unterscheidet der Compiler folgende Token (= Elemente, in die der Code zerlegt werden kann):

- *Schlüsselwörter*: sind fest definierte, reservierte Begriffe.
- *Bezeichner*: Das sind die Namen für Objekte, Methoden, Variablen, Konstanten. Der Name darf bis zu 65536 (Unicode-) Buchstaben enthalten (wobei mit Buchstabe auch das Dollarzeichen '$' und der Underscore '_' gemeint sind), darf natürlich kein reserviertes Schlüsselwort sein und auch nicht `true`, `false` oder `null`. Der erste Buchstabe darf keine Ziffer sein (die anderen aber schon), und innerhalb des Namens dürfen keine Leerzeichen enthalten sein. Groß- und Kleinschreibung wird unterschieden, ebenso Buchstaben mit / ohne Akzent usw. Der Name muss innerhalb des Gültigkeitsbereichs eindeutig sein.
- *Literal*: Das ist der Wert einer Variablen oder Konstanten. Das kann ein Buchstabe (in Hochkomma eingeschlossen), ein String (in Anführungszeichen eingeschlossen), eine Zahl (jeder Art, hexadezimal, Gleitkommaschreibweise usw.) oder ein boole'scher Wert sein (true oder false), auf jeden Fall einer der acht Basis-Datentypen oder ein String.
- *Trennzeichen*: `( ) { } [ ] ; , .`
- *Operator*: Gibt eine Operation mit einer oder zwei Variablen oder Konstanten an: z.B. `+ - * /`
- *Leerzeichen*: Dazu zählt bei Java das normale Leerzeichen, Tabulatoren, Zeilenumbrüche (Carriage Return und Line Feed) und der Seitenvorschub (Form Feed).
- *Kommentar*: Dieser Bereich wird vom Java-Compiler ignoriert; ein kommentierter Quelltext liest sich einfacher.

Kommentare

Es gibt drei Arten von Kommentaren in einem Java-Programm: mehrzeilige Kommentare (wie in C), einzeilige Kommentare (wie in C++), und Javadoc-Kommentare.

Mehrzeilige Kommentare

Mehrzeilige Kommentare beginnen mit `/*` und enden mit `*/` . Alles dazwischen wird als Kommentar betrachtet, auch wenn es sich über mehrere Zeilen erstreckt. Beispiel:

```
/* Dies ist ein Kommentar. */
```

Einzeilige Kommentare

Einzeilige Kommentare beginnen mit `//` und enden am Ende der Zeile. Beispiel:

```
// Dies ist ein Kommentar.
```

Javadoc-Kommentare

Javadoc-Kommentare sind eine Spezialität von Java. Sie beginnen mit `/**` und enden mit `*/` , also ähnlich wie die mehrzeiligen Kommentare, und können spezielle Tags enthalten. Das Programm `javadoc` extrahiert diese Kommentare aus der Quelltextdatei und erzeugt daraus eine HTML-Programmdokumentation (man hat also bei Java die Möglichkeit, in einem Quelltext sowohl den Programmcode als auch die dazugehörige Dokumentation zu schreiben). Daher ist es auch sinnvoll, innerhalb von Javadoc-Kommentaren HTML-Elemente zu verwenden.

Man kann (und soll) auch spezielle Tags einfügen, die den Kommentar logisch gliedern:

`@see Name_der_Klasse`	Erzeugt einen Hyperlink mit dem Text „See also“ auf die angegebene Klasse.

`@see fq_Name_der_Klasse`	Dito, mit dem 'fully qualified classname' (z.B. `java.lang.Object`).
`@see fq_Klassenname#Methode`	Ein Hyperlink auf die angegebene Methode der angegebenen Klasse, ansonsten s.o.
`@author Name des Autors`	Erzeugt einen Eintrag mit dem Namen des Autors.
`@version Versionstext`	Erzeugt einen Eintrag mit dem Versionstext (ohne festes Format).
`@param Parametername Beschreibung`	Erzeugt einen Abschnitt namens „Parameters" mit der Beschreibung eines Parameters einer Methode.
`@return Beschreibung`	Erzeugt einen Abschnitt namens „Returns" mit der Beschreibung des Rückgabewerts einer Methode.
`@exception fq_Klassenname`	Erzeugt einen Eintrag namens „Throws", der den Namen der Exception enthält, die von der Methode ausgelöst werden kann.

Wichtig ist auch der erste Satz des Kommentars; javadoc benutzt diesen als Indexeintrag, daher sollte der erste Satz prägnant und aussagekräftig sein.
JavaDoc-Kommentare sollten unmittelbar vor der kommentierten Klasse oder Methode stehen, nicht innerhalb des Blocks.

Beispiel:

```
/**
* This method calculates the nPrime-th prime number. Because prime numbers are
* not easy to determine, this may take some time for very high numbers.
* @param nPrime The Number of the desired prime number; attention: the first
* prime number is 2.
* @return the desired number if nPrime was valid (i.e. greater than 0),
* otherwise 0.
*/
```

Variablen

Jede Variable hat einen *Namen* (*identifier*), einen *Typ* (*type*) und einen *Gültigkeitsbereich* (*scope*). Durch die *Deklaration* wird der Variablenname und Typ festgelegt; der Gültigkeitsbereich ergibt sich daraus, wo im Programm die Variable deklariert wird.

```
[Zugriffsrechte] [static] [final] [transient] [volatile] Typ Name;
```

Beispiel:

```
int anzahl;
```

Variablenname und Gültigkeitsbereich:

Per Konvention beginnen Variablen immer mit einem Kleinbuchstaben. Weil Variablennamen Bezeichner sind, gilt das oben gesagte.

Lokale Variablen sind nur innerhalb des innersten einschließenden Blocks (= zwischen den geschweiften Klammern) gültig, ansonsten innerhalb der Klasse. Es ist nicht erlaubt, in einem inneren Block eine Variable mit gleichem Namen wie im umgebenden Block zu definieren, so dass beide Variablen gültig sind (man nennt das Verschattung: die innere Variable verschattet die äußere, auf die man nicht mehr zugreifen kann). Hinter dem inneren Block kann man sehr wohl so eine Variable im äußeren Block definieren, weil die Variable im inneren Block dann nicht mehr gültig ist und es auch nicht mehr wird. Beispiel:

```
{
        // Variable meineVariable ist ab jetzt gültig, bis ans Ende des äußeren Blocks
        int meineVariable = 5;

        for( int i = 1; i < 5; i++ )
        {
                int meineVariable = 3;          // NEIN! Verschattet die Variable im äußeren Block
                float deineVariable = 7.14;     // ja, hier wird nix verschattet
```

```
        }

        // Variable deineVariable ist ab jetzt gültig; die Variablen im inneren Block sind wieder
ungültig
        float deineVariable = 2.3;
}
```

Basis-Datentypen:

`byte`	ein einzelnes Byte	8-Bit-Binärzahl
`short`	Short integer	16-Bit-Binärzahl
`int`	Integer	32-Bit-Binärzahl
`long`	Long integer	64-Bit-Binärzahl
`float`	Gleitkommazahl einfacher Genauigkeit	32-Bit IEEE 754
`double`	Gleitkommazahl doppelter Genauigkeit	64-Bit IEEE 754
`char`	einzelner Buchstabe	16-Bit-Unicode-Zeichen
`boolean`	eine boolesche Variable (Wahrheitswert; wahr oder falsch)	*true* oder *false*

Alle numerischen Basis-Datentypen sind vorzeichenbehaftet! Die Darstellung negativer Zahlen erfolgt dabei durch Zweierkomplement-Arithmetik (d.h. eine negative Zahl wird dargestellt, indem der maximale Wert des Datentyps plus 1 hinzuaddiert wird: bei einer 8-Bit-Zahl wäre –1 also 256 + 1 = 255).

Wenn bei einer Variablen ein Überlauf eintritt, so wird ganz normal der untere Teil der Zahl zurückgegeben, was übergelaufen ist, fällt weg.

Java besitzt keinen `typedef`-Befehl wie C, mit dem man eigene Datentypen erschaffen kann; es gibt nur diese acht Basis-Datentypen. Klassen ähneln in vieler Hinsicht Datentypen (sind aber komplexer), und Klassen kann und muss man selbst schreiben - die Beschränkung auf diese Basis-Datentypen ist also keine echte Einschränkung.

Zugriffsrechte

Man hat die Wahl zwischen...

- *private*: nur die Klasse, in der die Variable deklariert ist, darf darauf zugreifen
- *protected*: außerdem dürfen Unterklassen oder Klassen im selben Paket darauf zugreifen (aber nicht Unterklassen, die in einem anderen Paket sind)
- *public*: jeder darf darauf zugreifen
- *package*: nur die eigene Klasse sowie Klassen aus dem selben Paket dürfen darauf zugreifen

static:

Bedeutet, dass nicht für jede Instanz der Klasse eine eigene, unabhängige Kopie der Variable erzeugt werden soll, sondern alle Instanzen auf die selbe Variable zugreifen.

final:

Bedeutet, dass die Variable nicht geändert werden kann: es ist eine Konstante.

transient:

Wird bei der Serialisation von Objekten verwendet.

Wertzuweisung

Einer Variablen kann man einen Wert bereits bei der Deklaration zuweisen:

```
int anzahl = 3;
```

entspricht

```
int anzahl;  anzahl = 3;
```

Das geht auch bei einer als *final* deklarierten Variable; trotzdem darf so einer Variablen nur einmal ein Wert zugewiesen werden!

Arrays

Ein Array ist ein Datenfeld: eine indizierte Sammlung von Objekten des gleichen Typs. Die Indizierung beginnt immer mit 0, und der Index muss vom Typ int sein. Die Wertzuweisung erfolgt in geschweiften Klammern, die die Einzelwerte, durch Komma getrennt, enthalten. Arrays kann man auch verschachteln, die Wertzuweisung ist dann ebenfalls verschachtelt. Im Unterschied zu den Basis-Datentypen muss bei Arrays (wie auch bei Objekten) der new-Operator verwendet werden. Lange Rede, kurzer Sinn, hier sind die Beispiele:

```
long meineNoten[] = { 2, 1, 3, 5, 2, 3 };          // Array mit 6 Einträgen
int schiffeVersenkenBrett[][] = { { 0, 0, 1, 1 }, { 0, 1, 0, 2 }, { 0, 1, 0, 2 }, { 2, 2, 0, 0 } };
float meineZahlen[] = new float[1000];              // Speicher für 1000 Zahlen reservieren
long deineZahlen[300] = { 5, 8, 2 };                // nur die ersten drei werden initialisiert
```

Operatoren

Die Java-Operatoren sind fast identisch mit denen in C++. Arithmetische Operatoren berechnen aus zwei Integer- oder Gleitkomma-Operanden einen Wert, dessen Format mindestens so groß wie der größere der beiden Operanden ist (aus `int` und `long` wird also `long`, aus `int` und `int` wird `long`, wenn die Zahl zu groß wird; analog mit `float` und `double`). Logische Operatoren haben als Operanden und als Ergebnis boole'sche Werte. Vergleichsoperatoren haben Zahlen als Operanden und boole'sche Werte als Ergebnis. Die bitweise arithmetischen Operatoren betrachten nicht die ganze (Ganz-)Zahl, sondern nur die einzelnen Bits an zueinander entsprechenden Positionen. Und die Zuweisungsoperatoren kombinieren die Zuweisung mit anderen Operatortypen. Hier eine kurze Zusammenfassung (in der Rangfolge der Operatoren, z.B. „Punkt vor Strich“):

Präfix- und Postfix-Operatoren

`[]`	Um Arrays zu deklarieren, sie zu erzeugen, und um auf sie zuzugreifen.
`.`	Um auf Elementfunktionen eines Objekts oder einer Klasse zuzugreifen.
`()`	Funktionsaufruf: in den Klammern werden die Parameter übergeben.
`++op1`	Präfix-Inkrement: zuerst wird op1 um 1 erhöht, dann wird op1 zurückgeliefert.
`--op1`	Präfix-Dekrement: zuerst wird op1 um 1 erniedrigt, dann wird op1 zurückgeliefert.
`op1++`	Postfix-Inkrement: zuerst wird op1 ausgewertet (und zurückgeliefert), und dann wird op1 um 1 erhöht.
`op1--`	Postfix-Dekrement: zuerst wird op1 ausgewertet (und zurückgeliefert), und dann wird op1 um 1 erniedrigt.

Unäre Operatoren

`+op1`	Plus-Vorzeichen	
`-op1`	Minus-Vorzeichen	
`~op1`	bitweise Negation (alle 1 werden 0, alle 0 werden 1)	bitweise arithmetisch
`!op1`	logische Negation (aus einer wahren Aussage wird eine falsche und umgekehrt)	logisch

Cast und Erzeugung

`(Typ) op1`	Konvertiert op1 in Typ.
`new op1`	erzeugt ein neues Objekt namens op1

Multiplikation

`op1 * op2`	Multiplikation	arithmetisch
`op1 / op2`	Division	arithmetisch
`op1 % op2`	Modulo-Division (liefert den Rest der Division)	arithmetisch

Addition

`op1 + op2`	Addition von op1 und op2	arithmetisch
`op1 - op2`	Substraktion	arithmetisch

'Shift'-Operatoren

`op1 >> op2`	verschiebt die Bits von op1 um op2 Stellen nach rechts (jede Stelle entspricht einer Division durch 2)	bitweise arithmetisch
`op1 << op2`	verschiebt die Bits von op1 um op2 Stellen nach links (jede Stelle entspricht einer Multiplikation mit 2)	bitweise arithmetisch
`op1 >>> op2`	genauso wie op1 >> op2, aber op1 wird als vorzeichenlos betrachtet	bitweise arithmetisch

Relationale Operatoren

`op1 > op2`	Ist op1 größer als op2? Wenn ja, wird true zurückgeliefert.	Vergleich
`op1 < op2`	Ist op1 kleiner als op2? Wenn ja, wird true zurückgeliefert.	Vergleich
`op1 >= op2`	Ist op1 größer oder gleich op2? Wenn ja, wird true zurückgeliefert.	Vergleich
`op1 <= op2`	Ist op1 kleiner oder gleich op2? Wenn ja, wird true zurückgeliefert.	Vergleich
`op1 instanceof op2`	Ist (das Objekt) op1 eine Instanz der Klasse op2? Wenn ja, wird true zurückgeliefert.	Vergleich

Gleichheitsoperatoren

`op1 == op2`	Ist op1 gleich op2? Wenn ja, wird true zurückgeliefert.	Vergleich
`op1 != op2`	Ist op1 ungleich op2? Wenn ja, wird true zurückgeliefert.	Vergleich

Bitweises AND

`op1 & op2`	Wenn sowohl das n-te Bit von op1 als auch das n-te Bit von op2 gleich 1 sind, dann wird das n-te Bit des Ergebnisses gleich 1, ansonsten 0. Dies wird mit allen Bits von op1 gemacht.	bitweise arithmetisch

Bitweises XOR (exklusives OR)

`op1 ^ op2`	Wenn n-te Bit von op1 oder das n-te Bit von op2 gleich 1 ist, aber nicht beide, dann wird das n-te Bit des Ergebnisses gleich 1, ansonsten 0. Dies wird mit allen Bits von op1 gemacht.	bitweise arithmetisch

Bitweises OR (inklusives OR)

`op1 | op2`	Wenn n-te Bit von op1 oder das n-te Bit von op2 oder beide gleich 1 ist / sind, dann wird das n-te Bit des Ergebnisses gleich 1, ansonsten 0. Dies wird mit allen Bits von op1 gemacht.	bitweise arithmetisch

Logisches AND

`op1 && op2`	Wenn sowohl op1 als auch op2 gleich true sind, dann wird true zurückgeliefert, ansonsten false. Wenn op1 bereits false ist, dann wird op2 nicht mehr überprüft.	logisch

Logisches OR

`op1 || op2`	Wenn entweder op1 oder op2 oder beide gleich true ist / sind, dann wird true zurückgeliefert, ansonsten false. Wenn op1 bereits true ist, dann wird op2 nicht mehr überprüft.	logisch

Der Bedingungsoperator

`op1 ? op2 : op3`	Wenn op1 true ist, dann wird op2 zurückgeliefert, ansonsten op3. Dies ist also ein verkürztes if/else.

Die Zuweisungsoperatoren

`op1 = op2`	op1 erhält den Wert op2.
`op1 += op2`	Zu op1 wird op2 hinzuaddiert. Entspricht op1 = op1 + op2.
`op1 -= op2`	Von op1 wird op2 subtrahiert. Entspricht op1 = op1 - op2.
`op1 *= op2`	op1 wird mit op2 multipliziert, op1 speichert das Ergebnis. Entspricht op1 = op1 * op2.
`op1 /= op2`	op1 wird durch op2 dividiert, op1 speichert das Ergebnis. Entspricht op1 = op1 / op2.
`op1 %= op2`	op1 wird durch op2 modulo-dividiert, op1 speichert das Ergebnis. Entspricht op1 = op1 % op2.
`op1 &= op2`	op1 wird mit op2 bitweise AND-verknüpft, op1 speichert das Ergebnis. Entspricht op1 = op1 & op2.
`op1 ^= op2`	op1 wird mit op2 bitweise XOR-verknüpft, op1 speichert das Ergebnis. Entspricht op1 = op1 ^ op2.
`op1 |= op2`	op1 wird mit op2 bitweise OR-verknüpft, op1 speichert das Ergebnis. Entspricht op1 = op1 | op2.
`op1 |= op2`	op1 wird mit op2 bitweise OR-verknüpft, op1 speichert das Ergebnis. Entspricht op1 = op1 | op2.
`op1 <<= op2`	op1 wird um op2 bitweise nach links verschoben, op1 speichert das Ergebnis. Entspricht op1 = op1 << op2.
`op1 >>= op2`	op1 wird um op2 bitweise nach rechts verschoben, op1 speichert das Ergebnis. Entspricht op1 = op1 >> op2.
`op1 >>>= op2`	op1 wird um op2 bitweise nach links verschoben, op1 speichert das Ergebnis. Entspricht op1 = op1 >>> op2.

Die Auswertungsreihenfolge

Im Allgemeinen sind die Operatoren in Java linksassoziativ, d.h. bei gleicher Rangfolge mehrerer Operatoren wird von links nach rechts ausgewertet (wie als würde mit dem Assoziativgesetz rekursiv jeweils die beiden linken Operanden geklammert); Ausnahmen sind die unären Operatoren, die Prä- und Postinkrementoperatoren, der Cast-Operator, die Zuweisungsoperatoren und der ternäre Bedingungsoperator, diese sind alle rechtsassoziativ (just to let you know...).

Der Cast-Operator

Mit dem Cast-Operator kann man Variablentypen konvertieren (aber nur zwischen Basis-Datentypen oder zwischen Referenztypen, aber nicht gemischt; Ausnahme: alle Typen können in den Referenztyp `String` gecastet werden), denn oft liegen Variablen nicht in dem Format vor, in dem sie gebraucht werden. Z.B. für Arrays als Indexvariable braucht man `int`, wenn die gewünschte Variable aber als `long` vorliegt, muss man konvertieren. In manchen Fällen kann Java selbst die Datentypen konvertieren (z.B. von `int` auf `long`), wenn aber die Konvertierung unklar ist, oder ein Datenverlust auftritt, weil in einen begrenzteren Datentyp konvertiert werden muss, so muss man das selbst explizit festlegen. Der gewünschte Typ wird in Klammern vor die Variable geschrieben:

```
long meinLong = 10;
int meinInt = 0;

meinInt = (int) meinLong;
```

Aber Achtung! Dabei wird die Variable einfach in einen neuen Typ gepresst, die `long`-Zahl heißt jetzt `int` und muss sich mit einem kleineren Speicherplatz begnügen, da kann ein Datenverlust

auftreten. Klar wird es an diesem Beispiel:

```
float meinFloat = 1.7;
int meinInt = (int) meinFloat;
```

`meinInt` hat danach den Wert 1, weil beim Wechsel auf das Integer-Format alle Kommastellen einfach wegfallen, wie bei der Gauß'schen Klammerfunktion. Um das zu umgehen, müsste man hier eine Funktion zum Runden aus der Klasse `java.lang.Math` verwenden, z.B. `round` ('richtig' runden), `ceil` (aufrunden) oder `floor` (abrunden).

Schlüsselwörter

Die 57 reservierten Schlüsselwörter von Java lauten:

`abstract boolean break byte byvalue case cast catch char class const continue default do double else extends final finally float for future generic goto if implements import inner instanceof int interface long native new null operator outer package private protected public rest return short static super switch synchronized this throw throws transient try var void volatile while`

Einige dieser Wörter sind zwar reserviert, werden aber (noch?) nicht unterstützt: `byvalue, cast, const, future, generic, goto, inner, operator, outer, rest, var`.

Verzweigungen (Auswahlanweisungen)

If/else-Verzweigung

Grundstruktur:

```
if ( erste_Bedingung )
{
        Ergebnis_1;
}
else if ( zweite_Bedingung )
{
        Ergebnis_2;
}
else
{
        Ergebnis_3;
}
```

Natürlich kann man die `else if`- und die `else`-Teile weglassen, wenn sie nicht benötigt werden. Und auch beliebig viele `else if`-Abschnitte verwenden; für diesen Fall ist aber eventuell `switch` besser geeignet.

Switch-Verzweigung

Beispiel:

```
switch ( variab )
{
        case 1: Anweisung_1;
        case 2: Anweisung_2; break;
        case 5: Anweisung_3; Anweisung_4; break;
        default: Anweisung_x; break;
}
```

Der Unterschied zur `if`-Verzweigung ist, dass es sich bei `switch` um Gleichungen handelt (`Anweisung_3` wird also ausgewertet, wenn gilt: `variab == 5`), man kann keine Ungleichungen verwenden, und die Bedingungen nicht mit `&&` oder `||` verknüpfen, sondern muss sich auf eine Bedingung beschränken. Die Konstanten müssen vom Typ `byte`, `short`, `char` oder `int` sein.

Was hinter `default` steht, wird ausgewertet, wenn kein anderer `case`-Asudruck passt (entspricht dem `else` bei der `if`-Verzweigung).

Die `break`-Anweisungen dahinter bewirken, dass die `switch`-Verzweigung verlassen wird, anstatt die Anweisungen hinter dem nächsten `case` auszuführen; in diesem Beispiel: ist `variab` gleich 2, so wird `Anweisung_2` ausgewertet und dann hinter der schließenden geschweiften Klammer weitergemacht, ist `variab` gleich 1, so werden `Anweisung_1` und `Anweisung_2` ausgeführt, erst das `break` hinter `Anweisung_2` führt dazu, dass die Verzweigung verlassen wird.

Natürlich ist das break hinter dem letzten case oder default überflüssig, aber es ist guter Programmierstil.

Schleifen (Iterationsanweisungen)

Die for-Schleife (Zählschleife)

```
for ( initialization; termination; increment )
{
        statement
}
```

Das ist das Grundgerüst der for-Schleife. Der initialization-Ausdruck wird am Anfang einmal ausgeführt, der termination-Ausdruck ist eine Bedingung, die bei jedem Schleifendurchlauf ausgewertet wird; ist die Bedingung `false`, dann wird die Schleife abgebrochen. Der increment-Ausdruck wird bei jedem Schleifendurchlauf einmal ausgewertet.

Natürlich kann für diese drei Ausdrücke alles mögliche stehen, oder sie können auch weggelassen werden (dann hat man eine Endlosschleife). Typischerweise benutzt man eine for-Schleife aber so:

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

Diese Schleife zählt von 1 bis 10 und schreibt die Zahlen nebeneinander, durch je ein Leerzeichen getrennt, hin.

Die while-Schleife (Bedingungsschleife)

... ist sozusagen eine vereinfachte for-Schleife. Es gibt sie in zwei Versionen:

```
while ( expr )
{
        statement
}
```

entspricht

```
for ( ; expr; )
{
        statement
}
```

Hier wird expr vor dem Schleifendurchlauf ausgewertet; ist der Ausdruck false, dann wird die Schleife nicht durchlaufen.

Die zweite Form lässt sich nicht durch for nachbilden:

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

Hier wird expr erst nach dem Durchlauf ausgewertet; diese Schleife läuft also immer mindestens einmal!

Schleifenabbruch und Labels (= Sprunganweisungen)

Ein Label ist sozusagen eine markierte Stelle im Programm. Man schreibt einfach den Namen des Labels, gefolgt von einem Doppelpunkt:

```
andere_Anweisungen;

NameDesLabels:
andere_Anweisungen;
```

Sinn macht das nur mit den Schlüsselwörtern zum Schleifenabbruch, `break` und `continue`.

`break` beendet eine Schleife (egal, ob `for`-, `while`- oder `do-while`-Schleife). Steht `break` alleine, so springt das Programm direkt hinter das Ende der Schleife und macht dort weiter. Man kann aber auch hinter `break` den Namen eines Labels stellen, das Programm springt dann dorthin (genauer: zur ersten Anweisung hinter dem Label). Beispiel:

```
while( a > 0 )
{
        if( 0 == a )
        {
                break;
        }
}

erste_Anweisung_nach_der_Schleife;
```

Das Programm springt, wenn es auf das Schlüsselwort `break` trifft, an `erste_Anweisung_nach_der_Schleife`.

```
do
{
        if( 0 == a )
        {
                break mein_Label;
        }
} while( a > 0 );

erste_Anweisung_nach_der_Schleife;

mein_Label:
zweite_Anweisung_nach_der_Schleife;
```

Hier springt das Programm an `zweite_Anweisung_nach_der_Schleife`.

`continue` dagegen beendet die Schleife nicht, sondern bricht den aktuellen Schleifendurchlauf ab und springt dorthin, wo die Schleifenbedingung ausgewertet wird:

```
for( i = 0; i < 20; i++ )
{
        if( 13 == i )
        {
                continue;
        }

        mach_was;

} while( a > 0 );
```

Wenn `i` gleich 13 ist, dann wird `mach_was` weggelassen, es geht direkt zum Schleifenanfang.

Zu `continue` kann man auch ein Label abgeben; es funktioniert dann analog zu `break`.

Literale

- *Boole'sche Literale*: haben den Wert `true` oder `false`. Beispiel:

```
boolean bIsIconified = true;
```

- *Integer-Literale*: Sie können als Dezimal-, Oktal- (Wertebereich der Ziffern 0-7) oder Hexadezimalzahl (Wertebereich der Ziffern 0-F, bei den Buchstabenziffern ist Groß-/Kleinschreibung egal) dargestellt werden. Hexadezimalzahlen haben am Anfang ein `0x` (oder `0X`), Oktalzahlen haben eine 0 am Anfang, alles andere ist dezimal. Außerdem kann man durch ein dahintergestelltes `L` (oder `l`) dafür sorgen, dass die Zahl als long interpretiert wird. Beispiele:

```
int meineGewöhnlicheZahl = 38;
int meineOktalzahl = 027;              // entspricht 23 im Dezimalsystem
int meineHexZahl = 0x7F;               //  entspricht  127  im  Dezimalsystem;  kann  man  auch  0X7F
schreiben
x = 20l + 17L;                         // das Ergebnis ist long (obwohl die Zahlen klein genug für
int sind)
```

- *Gleitkommaliterale*: Sie können aus mehreren Teilen bestehen, einem Ganzzahlteil, hinter dem Dezimalpunkt folgt ein gebrochener Teil, hinter einem `e` oder `E` folgt ein positiver oder negativer Exponent, und durch das Typensuffix wird der Typ der Zahl festgelegt (`f` oder `F` für `float`, `d` oder `D` für `double`, wenn das Suffix fehlt, wird `double` angenommen). Es muss mindestens entweder der Ganzzahlteil oder der Bruchteil vorhanden sein. Beispiele:

```
float meinFloat1 = 2;
float meinFloat2 = 2.4;
float meinFloat3 = .5;
float meinFloat4 = 1.3e-2;     // 1,3 * 10^(-2), also 0,013
float meinFloat5 = 3E4F;
double meinDoubl = .3e+4d;     // 0,3 * 10^4 = 3000
```

- *Zeichenliterale*: einzelne Zeichen werden in Hochkommata eingeschlossen. Eine Besonderheit

sind die Escape-Sequenzen, die man in bis zu drei Formen schreiben kann: oktal (nach einem Backslash kommt die dreistellige Oktalzahl, von 0 bis 377 (oktal)), hexadezimal (nach einem Backslash kommt `u00` und die zweistellige Hexadezimalzahl von 0 bis ff (hexadezimal)). Manche Steuerzeichen lassen sich auch durch Buchstaben darstellen (und das sollte man auch tun, weil z.B. die Zeichen für Zeilenumbruch als Hex- oder Oktaldarstellung bewirken, dass aus der Sicht des Compilers die Quelltextzeile umgebrochen ist, was zu Fehlern führt). Beispiele:

```
'\101'       // der Buchstabe A
'\u0041'     // ebenfalls
'A'  // so schreibt man ihn normalerweise...
'\b' // Steuerzeichen: Backspace
'\t' // Steuerzeichen: Tabulator
'\n' // Steuerzeichen: Line Feed (neue Zeile)
'\r' // Steuerzeichen: Carriage Return (Wagenrücklauf)
'\f' // Steuerzeichen: Form Feed (Seitenvorschub)
'\\' // der Backslash selbst
'\'' // das Hochkomma selbst
'\"' // das Anführungszeichen selbst
```

- *Strings*: sind Zeichenketten; für sie gilt also das Gleiche wie für Zeichenliterale, allerdings werden Strings in Anführungszeichen geschrieben, können mit '+' miteinander (und mit Variablen von anderen Datentypen, die implizit zu Strings umgewandelt werden) verknüpft werden, und die Hexadezimaldarstellung von Zeichen in Escape-Sequenzen ist nicht auf die ASCII-Zeichen beschränkt. Der Datentyp String ist zwar kein Basis-Datentyp, sondern ein Referenzdatentyp, nämlich eine Klasse, hat aber im Unterschied zu anderen Klassen die Besonderheiten der Schreibweise in Anführungszeichen und des Plus-Operators (man kann mit plus also zwei Klassen addieren!). Beispiele:

```
"Ich bin ein String.\n"
"Ich bin ein String." + " Und noch einer... und eine Zahl: " + meineHexZahl + " \n"
"Hier ist ein eingefügtes Unicode-Zeichen: \u096f und ein Wort in \"Anführungszeichen\"\n"
```

Klassen

Eine Klasse sieht z.B. folgendermaßen aus:

```
public class BrettSpiel
{
        public String myname;
        private static final int WIDTH = 10;

        public GameBoard( String gamename )
        {
                board = new int[WIDTH][HEIGHT];
                myname = new String(gamename);
        }

        public synchronized boolean isEmpty( int x, int y )
        {
                if( EMPTY == board[x][y] )
                {
                        return( true );
                }

                return( false );
        }
}
```

Eine Klasse besteht also aus Variablen und Funktionen / Prozeduren. In der objektorientierten Programmierung nennt man die Variablen einer Klasse die *Felder* der Klasse, und die Funktionen heißen *Methoden*.
Übrigens kann man Klassen auch verschachteln (seit Java 1.1): eine Klasse kann neben Feldern und Methoden auch innere Klassen besitzen, die wiederum Felder, Methoden und innere Klassen haben.
Außerdem kann als Datentyp für Felder die Klasse selbst verwendet werden:

```
class Person
{
        private String name;
        private int alter;
        private Person besterFreund;

        Person( String name, int alter, Person besterFreund )
        {
                this.name = name;
                this.alter = alter;
                this.besterFreund = besterFreund;
        }
}

// ...

// der beste Freund von Hans-Jürgen ist Peter (vereinfachend wird angenommen, dass dieser keinen
besten Freund hat)
Person hans = new Person( "Hans-Jürgen", 18, new Person( "Peter", 20, null ) );
```

Dieses Beispiel ist gleichzeitig eine Demonstration des Schlüsselwortes `this`: damit ist die eigene Klasse gemeint (kann meistens weggelassen werden; hier wird es aber gebraucht, weil sonst die Felder mit den gleichnamigen Parametern verwechselt werden).

Die Deklaration einer Klasse

Die komplette Klassendeklaration sieht so aus:

```
modifiers class NameKlasse extends NameSuperklasse implements NameSchnittstellen
```

Unbedingt benötigt wird nur das Schlüsselwort `class` sowie der Name der Klasse `NameKlasse`. Alles andere ist optional. Für den Namen der Klasse gelten die selben Regeln wie für Variablennamen, siehe oben; Klassennamen werden gewöhnlich mit großem Anfangsbuchstaben geschrieben.

Modifier

Es gibt vier Modifier für Klassendeklarationen:

- `public`: Die Klasse kann von allen Objekten benutzt werden, egal zu welchem Paket sie gehören. Damit das funktioniert, muss die Klasse in einer Datei definiert sein, die den Namen `NameKlasse.java` trägt.
- `final`: Die Klasse darf keine Subklassen haben. Ein Grund dafür könnte sein, dass diese Klasse

z.B. ein standardisiertes Netzwerkprotokoll implementiert; die Methoden müssen so erhalten bleiben und dürfen nicht von Subklassen überschrieben (= verändert) werden, sonst funktioniert der Netzwerkverkehr nicht mehr.

- `abstract`: In der Klasse ist mindestens eine Methode unvollständig. Ein Grund dafür könnte sein, dass diese Methode sehr speziell ist, es macht keinen Sinn, sie allgemein zu definieren, sondern sie muss in jedem Fall von den Subklassen überschrieben werden. Also lässt man sie gleich leer; ein Beispiel wäre eine abstrakte Klasse zur Grammatikprüfung, bei der man alle sprachspezifischen Methoden leer lässt; die sprachspezifischen Subklassen (z.B. zur englischen Grammatikprüfung) definieren diese Methoden dann für ihre spezielle Sprache.
- Kein angegebener Modifier: Die Klasse kann nur von Objekten aus dem selben Paket erweitert und benutzt werden.

Die Modifier stehen vor dem Schlüsselwort `class`.

Superklasse

Eine Klasse hat genau eine (direkte) Superklasse (anders als in C++). Die Superklasse, deren Felder und Methoden die Klasse erbt, wird hinter dem Schlüsselwort `extends` angegeben. Wenn keine Superklasse angegeben ist, betrachtet Java die Klasse als Subklasse von `java.lang.Object`.

Schnittstellen

Eine Klasse kann mehrere Schnittstellen haben; die Schnittstellen werden nach dem Schlüsselwort `implements` angegeben, durch Komma getrennt. Schnittstellen dienen als „Ersatz“ für die Mehrfachvererbung.

Methoden einer Klasse

Die komplette Methodendeklaration sieht folgendermaßen aus:

```
Zugriffsrechte Modifier Rückgabetyp nameMethode( Parameter ) throws Exceptions
```

Unbedingt benötigt wird nur der Typ des Rückgabewerts `Rückgabetyp` und der Name der Methode `nameMethode`. Alles andere ist optional. Für den Namen der Methode gelten die selben Regeln wie für Variablennamen, siehe oben; Methodennamen werden gewöhnlich mit kleinem Anfangsbuchstaben geschrieben.

Zugriffsrechte

Damit man die OOP-Idee der Kapselung verwirklichen kann, muss man den Zugriff auf die Methoden folgendermaßen einschränken, damit die Klasse am Ende eine Black Box ist, in die von außen nicht hineingepfuscht werden kann (selbst wenn man das wollte).

- `public`: Die Methode kann von allen anderen Klassen benutzt werden, egal aus welchem Paket. Ein Grund dafür könnte sein, dass dies die Ein- bzw. Ausgabemethoden der „Black Box“ sind. Natürlich sollten nur solche Methoden `public` sein, die es aus irgendeinem Grund sein müssen.
- `protected`: Die Methode ist wie `public`, aber nur für Klassen aus dem selben Paket. Andere Klassen können sie nicht benutzen.
- `private`: Die Methode kann nur von Methoden aus der selben Klasse verwendet werden. Ein Grund dafür könnte sein, dass es sich um einen öfters oder von mehreren Methoden benötigten Programmabschnitt handelt, der in eine „Hilfsmethode“ ausgelagert wurde; diese Hilfsmethode geht aber außerhalb der Klasse niemanden etwas an.
- `private protected`: Die Methode kann nur von Methoden aus der selben Klasse und aus Subklassen verwendet werden, aber nicht von Methoden in anderen Klassen (egal ob aus dem selben Paket oder nicht). Außerdem kann sie nicht von Instanzen der Klasse oder Subklasse verwendet werden. Beispiel:

```
class MeineKlasse
{
     private protected int getX()
     {
     }
}

class MeineSubklasse extends MeineKlasse
{
     void machIrgendwas()
     {
             MeineKlasse instanzMeinerKlasse = new MeineKlasse();

             super.getX();                    // ist korrekt
             instanzMeinerKlasse.getX();      // ist falsch
     }
}
```

Das ist auch ein Beispiel zur Verwendung des Schlüsselworts `super`.

- Keine Zugriffsrechte angegeben: die Methode ist innerhalb der Klasse und innerhalb des Pakets verfügbar.

Modifier

`public`, `static`, `native`, `abstract`, `synchronized`, `final`

Rückgabetyp

Der Variablentyp, den der zurückgelieferte Wert hat. Das kann sein: einer der Basistypen (siehe oben, z.B. `int`), oder ein komplexer Variablentyp wie z.B. ein Array oder eine Klasse, oder einfach der Typ `void` (was bedeutet, dass gar nichts zurückgegeben wird).

Parameter

Die Parameter werden in Klammern angegeben, durch Komma getrennt. Wenn kein Parameter angegeben wird, bleiben die Klammern leer, ansonsten steht bei jeden Parameter zuerst der Variablentyp und dahinter der Variablenname. Beispiele:

```
float machDrittel( int zahl, boolean vorzeichen )
void schreibeHallo()
```

Der Funktionsrumpf

Zwischen den geschweiften Klammern stehen der Programmcode, die Variablendeklarationen usw. Um einen Wert zurückzugeben, verwendet man das Schlüsselwort `return`. Beispiel:

```
public int holDasDoppelte( int zahl )
{
        int ergebnis = zahl * 2;

        return( ergebnis );
}
```

Die runden Klammern bei return kann man weglassen. Handelt es sich um einen Konstruktor oder um den Rückgabetyp `void`, darf nichts zurückgegeben werden: `return;`

Konstruktoren

Konstruktoren sind besondere Methoden:

- sie haben den selben Namen wie die Klasse
- sie haben keinen Rückgabeparameter (weil sie immer die Instanz der Klasse zurückliefern)
- sie werden häufig überladen
- sie werden automatisch aufgerufen, wenn eine Instanz der Klasse erzeugt wird - sie sind also die erste Methode, die aufgerufen wird, wenn die Instanz erzeugt ist.
- sie werden benutzt, um die Felder der Klasse zu initialisieren (sinnvollerweise mit Hilfe der angegebenen Parameter, sonst könnte man auf den Konstruktor gleich verzichten), um irgendwelche Anfangsberechnungen durchzuführen usw.

- sie werden meist als `public` deklariert (Wenn man allerdings möchte, dass andere Klassen keine Instanz der Klasse erzeugen können, wohl aber auf die statischen Funktionen zugreifen können, deklariert man den Konstruktor als `private`).
- sie dürfen nicht als `native`, `abstract`, `static`, `synchronized` oder `final` deklariert sein
- sie werden nicht vererbt; um den Konstruktor der Superklasse aufzurufen, verwendet man die Anweisung `super( Parameter )`, wobei der Parameter ein oder mehrere beliebige Parameter ist, den oder die ein Konstruktor der Superklasse verarbeiten kann. Diese Anweisung muss ganz am Anfang des Konstruktors stehen!

Ein Beispiel:

```
class Spielbrett extends Brett
{
        private String name;
        private int felder;
        private static final STANDARDFELDER = 30;

        public Spielbrett( String spielname )
        {
                // zuerst Konstruktor der Superklasse aufrufen
                super( KLEINES_BRETT, 1, 2, 3.5543 );

                name = new String( spielname );
                felder = STANDARDFELDER;
        }

        public Spielbrett( String spielname, int anzahlFelder )
        {
                name = new String( spielname );
                felder = anzahlFelder;
        }
}

class Schach
{
        private Spielbrett schachbrett;
        /* .... */

        public void neuesSpiel()
        {
                schachbrett = new Spielbrett( "Schach", 64 );
                /* .... */
        }
}
```

Destruktoren?

Im Gegensatz zu C++ gibt es bei Java keine 'richtigen' Destruktoren. Diese Aufgabe erledigt die *Garbage Collection*: wenn ein Objekt nicht mehr referenziert wird, kann die Garbage Collection das Objekt löschen und den Speicher wieder freigeben. Das hängt davon ab, wieviel Speicher vorhanden ist (ist der Speicher knapp, wird das Objekt schnell gelöscht) und wie sehr das System ausgelastet ist (wenn der Computer beschäftigt ist, dann wartet die Garbage Collection) - man kann also nicht vorhersagen, wann ein Objekt letztendlich gelöscht wird, muss sich dafür nicht mehr darum kümmern, die Objekte ordentlich zu löschen (d.h. so, dass der gesamte vom Objekt belegte Speicher freigegeben wird (keine „Speicherleichen“ bleiben übrig), aber kein Speicherbereich, der noch benötigt wird (führt zum Absturz)).

Die Methode *finalize()*

Diese `protected`-Methode gehört zu `java.lang.Object` und wird somit von jeder Klasse geerbt. Sie ist normalerweise leer, und wird aufgerufen, wenn die Garbage Collection das Objekt löscht. (Sie hat also eine Destruktor-ähnliche Funktion.) Man kann sie überschreiben und mit Programmcode füllen, der unbedingt noch ausgeführt werden muss, bevor das Objekt zerstört wird (z.B. Netzwerkverbindungen beenden). Natürlich ist es kein guter Programmierstil, alle Aufräumarbeiten nach `finalize()` zu verlegen, sondern Ressourcen sollten schon vorher freigegeben werden, wenn sie nicht mehr benötigt werden, `finalize()` ist sozusagen der letzte Notnagel.

Schnittstellen

Schnittstellen sind in Java der Ersatz für die Mehrfachvererbung in C++. Entsprechend ist eine Schnittstelle etwas ähnliches wie eine Klasse, allerdings mit Einschränkungen: Schnittstellen können nur abstrakte Methoden (= leere Methoden) und finale Felder (= Konstanten) definieren, die von der sie benutzenden Klasse implementiert werden. Eine Schnittstelle sieht z.B. so aus:

```
interface Product
{
        public static final String MAKER = "my Company";

        public int getPrice( int id );
}
```

Die Deklaration einer Schnittstelle

Die komplette Schnittstellendeklaration sieht so aus:

```
public interface NameDerSchnittstelle extends NameAndererSchnittstellen
```

Unbedingt benötigt wird nur das Schlüsselwort `interface` sowie der Name der Schnittstelle `NameDerSchnittstelle`. Alles andere ist optional. Für den Namen der Schnittstelle gelten die selben Regeln wie für Variablennamen, siehe oben; Schnittstellennamen werden gewöhnlich mit großem Anfangsbuchstaben geschrieben.

Hinter diese Deklaration kommt ein Semikolon, keine geschweiften Klammern mit Programmcode dazwischen!

Zugriffsrechte

Per Voreinstellung können Schnittstellen nur von den Klassen im selben Paket implementiert werden. Damit alle Klassen die Schnittstelle implementieren können, muss sie als `public` deklariert werden, und außerdem in einer Datei stehen, die den Namen *`NameDerSchnittstelle`*`.java` hat - das empfiehlt sich aber auch für alle anderen Schnittstellen.

Die Methoden der Schnittstelle sind dagegen von vornherein `public` und können auch nichts anderes sein, daher kann das Schlüsselwort `public` auch weggelassen werden.

Genauso sind alle Variablen von vornherein `public`, `final` und `static`, diese Schlüsselwörter kann man auch weglassen. Aber natürlich ist es hilfreich, es explizit hinzuschreiben.

Erweiterung anderer Schnittstellen

Mit dem Schlüsselwort `extends` in der Deklaration gibt man eine oder mehrere Superschnittstellen an. Die Subschnittstelle erbt dabei die Methoden und Konstanten genauso wie eine Subklasse die Methoden und Felder ihrer Superklasse erbt. Weil Schnittstellen keine Methoden definieren dürfen, sondern nur abstrakte Methoden deklarieren können, können sie entsprechend keine Methoden ihrer Superschnittstelle erweitern. Sie können nur zusätzliche Konstanten und Methoden deklarieren, und es ist die Aufgabe der Klasse, die die Schnittstelle benutzt, alle Methoden der Schnittstelle sowie alle Methoden ihrer Superschnittstelle zu implementieren. Werden nicht alle Methoden implementiert, dann wird die Klasse abstrakt - weil sie leere Methoden hat.

Implementieren von Schnittstellen

Eine Klasse, die eine Schnittstelle implementiert, muss, wie gesagt, alle Methoden überschreiben (sonst wird die Klasse abstrakt). Das bedeutet, dass auch die Parameter (d.h. die Reihenfolge der Variablentypen - die Namen der Parameter kann man sehr wohl verändern) der Methode exakt gleich sein müssen - weil es sich ansonsten nicht um Überschreiben handelt (d.h. die Methode wird ersetzt), sondern um Überladen (d.h. es wird eine zweite Methode mit dem selben Namen definiert, aber die ursprüngliche Methode bleibt immer noch abstrakt). Das geht natürlich auch, aber nur zusätzlich; auf jeden Fall muss die Methode überschrieben werden. Wenn man möchte, dass die Methode gar nichts macht, kann man trotzdem nicht auf das Überschreiben verzichten, sondern

überschreibt mit einer Methode, die einen leeren Funktionskörper hat (also nur aus zwei geschweiften Klammern besteht) - dann ist allerdings die Frage, ob man die Schnittstelle wirklich braucht....

Pakete

In Java kann man Klassen in Paketen organisieren, und auch das Standard-Java-API ist in Pakete gegliedert - das ist übersichtlicher bei vielen Klassen. Um eine Klasse, die in einem Paket ist (Beispiel: `Button` in `java.awt`), zu benutzen, kann man:

- entweder den Klassennamen mit vorangestelltem Paketnamen verwenden: `java.awt.Button`
- oder das gesamte Paket importieren - dann kann man alle Klassen des Pakets direkt verwenden. Am Anfang der Datei muss eine `import`-Anweisung stehen. Beispiel:

```
import java.awt;              // importiert das Paket java.awt

// importiert das swing-Paket sowie alle "Unterpakete" wie javax.swing.plaf
import javax.swing.*;

class MeineKlasse
{
     private Button meinButton = null; // geht, weil awt importiert ist
}
```

Um selbst ein Paket zu erstellen, schreibt man in alle Klassendateien, die zum Paket gehören sollen, folgende package-Anweisung:

```
// in der Datei Auto.java:
package MeinPaket.Transport;

// blabla... der Rest der Datei Auto.java



// in der Datei Boot.java:
package MeinPaket.Transport;

// blabla... der Rest der Datei Boot.java
```

Genau genommen ist jede Klasse, auch wenn nicht explizit vereinbart, Bestandteil eines Pakets: nämlich eines namenlosen Pakets, dass alle Klassendateien im selben Verzeichnis umfasst; die Unterpakete sind die Namen der Unterverzeichnisse.

Ein erstes Programm

Schauen wir uns einfach mal ein typisches Java-Programm an:

```
/*
 *      Titel:          Polynom.java
 *      Datum:          24.04.2000
 *      Autor:          Christoph Moder <cmoder@bigfoot.com>
 *      Version:                1.0
 *      Beschreibung:   beschreibt ein Polynom; in polyKoeff[] sind die Koeffizienten,
 *                              in grad der Grad gespeichert
 */

import java.lang.Math;

class Polynom extends Funct
{
        private double polyKoeff[];             // die Koeffizienten des Polynoms
        private int grad = 0;           // der Grad des Polynoms

        public Polynom( double newKoeff[] )     // erster Koeffizient = Leitkoeffizient usw.
        {
                grad = newKoeff.length - 1;     // Ein Polynom 0-ten Grades ist eine Konstante,
                                                // also immer ein Koeffizient mehr als der Grad
                polyKoeff = newKoeff;           // übernimm die Koeffizienten
        }

        // berechnet den Funktionswert des Polynoms an der Stelle x
        public double eval( double x )
        {
                double ergebnis = 0;            // speichert das Ergebnis:
                                                // die Summe der Produkte der Koeffizienten mit
                                                // den entspr. Potenzen von x

                for( int i = grad; i >= 0; i-- )
                {
                        ergebnis += Math.pow( x, (double) i ) * polyKoeff[ grad - i ];
                }

                return ergebnis;
        }
}
```

Diese Klasse `Polynom` ist also eine Subklasse von `Func`, hat zwei private Felder (ein Array und ein Integer), einen zusätzlichen Konstruktor `Polynom( double newKoeff[] )` (zusätzlich zum immer vorhandenen Default-Konstruktor) und eine öffentliche Methode `eval`. Gut, aber wo fängt die Programmausführung eigentlich an, bei welcher Methode wird gestartet? In der Methode `main`, die jede Klasse hat, weil sie sie von `java.lang.Object` erbt (`java.lang.Object` ist die einzige Klasse, die Superklasse von jeder Klasse ist). Ein Beispiel, das obige Klasse nutzt:

```
class TestPolygon
{
        public static void main( String args[] )
        {
                double newKoeff[] = { 1, 0, 0 };                    // Koeffizienten des Polynoms
                Polynom myPolynom = new Polynom( newKoeff );        // der Konstruktor

                Polygon plotParabel = myPolynom.plot( -5, +5, 10 );        //  plot  ist  geerbt  von
Funct

                plotParabel.paintOutput( java.awt.Color.blue, true );
        }
}
```

Wenn man also ein Programm schreibt, muss man irgendwo eine `main`-Methode haben, bei der die Programmausführung beginnt. Das heißt aber nicht, dass man immer Programmcode in die `main` -Methode packen muss. Ein Beispiel: Man möchte ein Fenster haben, in das irgendetwas gezeichnet wird. Jetzt muss man etwas über die Funktionsweise von Fenstern wissen (aus der Java-API-Dokumentation): beim Zeichnen des Fensters wird die Methode `paint` aufgerufen. Also macht man eine Subklasse von z.B. `Frame` (stellt ein Fenster dar), und überschreibt die `paint`-Methode mit dem Code, der die gewünschten Dinge malt. Jedesmal, wenn das Fenster vergrößert oder verkleinert wird oder aus einem anderen Grund neu gezeichnet werden muss, wird die überschriebene `paint` -Methode aufgerufen und das Fenster samt Inhalt neu gezeichnet, genau dann, wenn es sein muss, nicht seltener und nicht öfter. Ähnlich ist es bei Applets - sie erben ebenfalls Funktionen von einer Superklasse (hier `java.applet.Applet`), die bei bestimmten Ereignissen aufgerufen werden, und das kann man nutzen.

Tja, das war's - mehr braucht man nicht für ein Java-Programm. Das Java-API hat eine ganze Menge Klassen zu bieten, dazu stöbert man am Besten in der Java-API-Dokumentation herum, und

bei speziellen Problemen durchsucht man das (viel zu umfangreiche) Java-Tutorial.

Trotzdem ist der Text noch nicht zu Ende, sondern es kommen noch Kapitel zu wichtigen Sprachbestandteilen. Grafik mit AWT und Swing wurde aber weggelassen, weil es erstens eine Wissenschaft für sich ist - man könnte so viel dazu sagen, z.B. Events oder GridBagLayout -, zweitens in diesem Bereich viele Änderungen passieren (das Event-Modell von AWT wurde umgestellt, und überhaupt geht der Trend weg vom AWT und hin zum moderneren Swing), und drittens ist Grafik in praktisch allen Java-Dokumentationen und Anleitungen sehr gut beschrieben. Das Gleiche gilt für andere Themen wie Netzwerkfunktionen; das Java-API hat wie gesagt eine sehr große Menge an Klassen für ein breites Spektrum an Aufgaben, man kann unendlich viel damit machen, deshalb ist das Tutorial auch so umfangreich.

Ausnahmen, Fehlerbehandlung

Die Fehlerbehandlung funktioniert in Java etwas anders als in anderen Programmiersprachen. Normalerweise muss nach jeder kritischen Anweisung (z.B. Datei öffnen) untersucht werden, ob die Anweisung erfolgreich war oder ein Fehler aufgetreten ist (z.B. die Datei nicht gefunden wurde), und ggf. den Fehler entsprechend behandeln (Fehlermeldung ausgeben, Wiederholung der Eingabe usw.). Weil Fehler an vielen verschiedenen Stellen auftreten können, gibt es in Programmen, die eine richtige Fehlerbehandlung machen, haufenweise `if...else`-Konstruktionen, was die Lesbarkeit nicht gerade vereinfacht und das Programm ziemlich lang macht. In Java hat man dafür einen anderen Ansatz: die kritischen Anweisungen werden von einem `try`-Block umgeben, dahinter folgt ein oder mehrere `catch`-Blöcke, in dem der oder die möglichen Fehler (= Ausnahmen) behandelt werden; der Programmcode wird also vom Fehlerbehandlungscode getrennt. Optional kann man noch einen `finally`-Block dahinterhängen; dessen Code wird ausgeführt, nachdem eine Ausnahme ausgelöst wurde (und bevor die Kontrolle an einen `catch`-Block, falls vorhanden, weitergereicht wird) oder nachdem das Programm auf die Schlüsselwörter `return`, `break` oder `continue` gestoßen ist. Ein Beispiel:

```
try
{
        irgendeineAnweisung();
        etwasFehlerträchtiges();
        nochEineKritischeAnweisung();
}
catch( NullPointerException n )
{
        // was getan werden muss, wenn dieser Fehler aufgetreten ist
}
catch( IOException i )
{
        // was getan werden muss, wenn dieser Fehler aufgetreten ist
}
catch( Exception e )
{
        // ...
}
catch( Throwable t )
{
        // ...
}
finally
{
        System.out.println(  "Hallo...  diese  Nachricht  kommt  IMMER,  im  Normalfall  und  in  jedem
Fehlerfall" );
}
```

Hier wird also die Meldung im `finally`-Block normalerweise nach den drei Anweisungen im `try` -Block ausgeführt; wenn eine Ausnahme aufgetreten ist, direkt nach der Anweisung, die die Ausnahme verursacht hat.

Was ist eigentlich eine Ausnahme? Es ist eine Klasse, die eine Subklasse von `java.lang.Throwable` ist. Wegen der Polymorphie ist kann jede Ausnahme als Ausnahme vom Typ `Throwable` betrachtet werden. Die `catch`-Blöcke werden von oben nach unten abgearbeitet, also muss man die spezielleren Ausnahmen oben abfangen, bevor die allgemeinen Ausnahmen dran kommen (`IOException` ist eine Subklasse von `Exception`, und `Exception` ist wiederum eine Subklasse von `Throwable`).

Ausnahmen selber erzeugen

Ausnahmen kann man auch selber erzeugen. Man macht dies mit dem Schlüsselwort `throw`, gefolgt vom Ausnahme-Objekt. Dahinter wird die Programmausführung sofort abgebrochen und zum `catch`-Block gesprungen. Beispiel:

```
try
{
        if( 0 == x - 2 )         //  das  darf  auf  irgendeinem  Grund  nicht  vorkommen  =>  Ausnahme
erzeugen
        {
                throw new NullPointerException();

                /* könnte man auch so schreiben:
                NullPointerException meineException = new NullPointerException();
                throw meineException;
                */
        }
```

```
}
catch( NullPointerException e )
{
        // ...
}
```

Ausnahmen, die nicht abgefangen werden

Jede Ausnahme muss abgefangen werden, sonst wird das Programm abgebrochen. Was ist aber, wenn eine Ausnahme zwar aufgetreten kann, es aber aus irgendeinem Grund keinen Sinn ergibt, einen `catch`-Block zu schreiben? Dann muss die die Klasse benutzende Methode die Ausnahme abfangen; um das zu erzwingen, hängt man eine `throws`-Klausel an die Methodendeklaration an. Hinter das Schlüsselwort `throws` kommt eine Liste von unbehandelten Ausnahmen, die diese Methode erzeugt, durch Kommata getrennt. Der Programmierer muss dann bei der Benutzung dieser Methode für die Behandlung der angegebenen Ausnahmen sorgen. Beispiel:

```
void erzeugeAusnahme() throws java.io.IOException
{
        throw new java.io.IOException();  // erzeuge eine Ausnahme, die nicht abgefangen wird
}

// ...

{
        try
        {
                erzeugeAusnahme();

                System.out.println( "Kein Fehler." );
        }
        catch( Throwable t )                    // fange jede Ausnahme ab
        {
                System.out.println( t );
        }
}
```

Es gibt aber auch Ausnahmen, die man nicht mit `throws` angeben muss. Das ist zum einen `java.lang.Error`, weil diese Ausnahmen schwere Fehler darstellen, gegen die das Programm praktisch nichts tun kann (z.B. Speichermangel), und zum anderern `java.lang.RuntimeException`, weil diese so häufig sind, dass sie praktisch in jeder Methode mit `throws` angegeben werden müssten (was das Programm nur unübersichtlicher machen würde).

Natürlich kann man, wenn benötigt, eigene Ausnahmen definieren, indem man Klassen schreibt, die Subklassen von `java.lang.Throwable` sind und die benötigten Fähigkeiten besitzen.

Übel wird es natürlich, wenn im `finally`-Block oder in einem `catch`-Block eine Ausnahme auftritt, weil dadurch die evtl. vorher aufgetretene Ausnahme vergessen wird. Man kann zwar die `try`-`catch`-Blöcke verschachteln, aber das ergibt meist ein unübersichtliches Programm. Lieber bemüht man sich, so zu programmieren, dass in `catch`- und `finally`-Blöcken möglichst wenig Code steht und dort keine Ausnahme auftreten kann.

Threads und synchronisierte Blöcke

Was ist ein Thread? Normalerweise wird ein Java-Programm sequenziell abgearbeitet, ein Befehl nach dem anderen. Das ist manchmal nicht die beste Lösung; angenommen, ein Programm macht eine zeitaufwändige Aufgabe (z.B. die Berechnung des Mandelbrot-Apfelmännchens ist eine sehr rechenintensive Sache, oder das Formatieren eines umfangreichen Dokuments für den Ausdruck, oder das Kompilieren großer Projekte oder...), auf die der Benutzer warten muss. Statt dass der Benutzer die ganze Zeit auf den Sanduhr-Mauszeiger starrt, wäre es besser, wenn diese Aufgaben im Hintergrund nebenher laufen, und der Benutzer weiterarbeiten kann. Ein weiteres Beispiel wäre ein Server (sei es nun z.B. http, ftp oder telnet): es geht nicht, dass der Server wartet, bis der Benutzer fertig ist (und der kann sich nicht entscheiden, welches Programm er downloaden will), bevor der nächste an die Reihe kommt, stattdessen müssen alle Benutzer gleichzeitig bedient werden (genügend Rechenleistung ist ja schließlich da, ein Computer ist hier immer um Größenordnungen schneller als der menschliche Benutzer). Das sind die Anwendungsgebiete von Threads.

Ein Thread ist eine Klasse, die von `java.lang.Thread` abgeleitet ist; man überschreibt die `run` -Methode und fügt dort den Code ein, der parallel abgearbeitet werden soll. Mit der Methode `start` wird der Thread gestartet, mit `stop` beendet, und mit `suspend` und `resume` kann er angehalten und wieder gestartet werden. Eine zweite Möglichkeit, Threads zu erzeugen, ist, in einer Klasse das Interface `Runnable` zu implementieren; der Rest ist identisch: der Code kommt in die überschriebene `run`-Methode.

Synchronisation

Probleme treten mit Threads dort auf, wo mehrere Threads versuchen, auf die selbe Variable schreibend und lesend zuzugreifen. Beispiel: ein threadbasierter Datenbankserver. Mehrere Threads lesen Daten, bearbeiten sie und schreiben sie zurück. Damit die Datenbank nicht inkonsistent wird, darf kein Thread etwas lesen oder schreiben, solange ein anderer Thread dabei ist, die Daten zu ändern. Dazu gibt es das Schlüsselwort `synchronized`, das ein ganzes Objekt sperrt. Dieses Objekt kann eine Methode sein (dann schreibt man es in die Methodendeklaration), oder man erzeugt einen Block, in dem ein Objekt gesperrt wird... der Thread wird dabei solange angehalten, bis das gesperrte Objekt wieder frei ist. In der Praxis so:

```
public synchronized void meineFunktion()
{
        if( this.meineVariable < 10 )     // this.meineVariable kann von mehreren Threads geändert
werden
        {                                           // wenn die Änderung jetzt geschieht, ist der Wert
dann >= 10...
                this.meineVariable += 2;          // ...wenn ihn diese Funktion erhöhen will
        }
}
```

Das könnte man auch so schreiben, dann wird nicht die gesamte Funktion gesperrt, sondern nur der kritische Teil innerhalb des `synchronized`-Blocks - etwas eleganter.

```
public void meineFunktion()
{
        // irgendetwas anderes geschieht...

        synchronized( this )
        {
                if( this.meineVariable < 10 )
                {
                        this.meineVariable += 2;
                }
        }

        // irgenetwas anderes geschieht...
}
```

Eigentlich müsste man in dem Fall gar nicht die gesamte Funktion sperren, sondern nur die Variable. Also:

```
public void meineFunktion()
{
        synchronized( this.meineVariable )
        {
                if( this.meineVariable < 10 )
```

```
                {
                        this.meineVariable += 2;
                }
        }
}
```

Sinnvollerweise gewöhnt man sich an (zumindest für Klassen, die man öfters verwenden wird), dass immer, wenn auf nicht-lokale Variablen zugegriffen wird, diese Abschnitte als `synchronized` markiert werden. Dann gibt es keine Probleme, wenn auf den Code einmal Threads losgelassen werden.
Wenn es sich um statische Felder handelt, kann es sein, dass eine andere Instanz der Klasse diese einzige Variablenkopie verändert, da hilft es nichts, wenn die Funktion gesperrt ist. In diesem Fall muss man mit `synchronized( getClass() )` die gesamte Klasse sperren.

Applets

Applets sind Java-Programme, die in Webseiten eingebaut werden, ihre Ausgabe in das Browserfenster schreiben und auch mit dem Browser (eingeschränkt) kommunizieren können. Im Gegensatz zu normalen Java-Programmen (auch Applications genannt) sind sie in ihren Fähigkeiten stark eingeschränkt - aus Sicherheitsgründen: weil jeder Applets erstellen und in jede Webseite integrieren kann, wäre es riskant, wenn Java-Applets Dateien lesen oder schreiben könnten (das Applet könnte unbemerkt Daten löschen oder Passwortdateien übers Internet verschicken). Konkret haben Applets folgende Beschränkungen (falls sie über das Internet geladen werden; sind sie auf dem Rechner lokal vorhanden, gelten die Beschränkungen z.T. nicht):

- Applets können nicht auf lokale Dateien zugreifen oder Dateien erstellen.
- Applets können Netzwerkverbindungen nur zu dem Rechner aufbauen, von dem sie geladen wurden.
- Applets können keine Bibliotheken laden. Aber sie können auf das normale Java-API (`java.*`) zugreifen.
- Applets können keine nativen Methoden aufrufen.
- Applets können nicht mit `System.exit()` die Ausführung des Interpreters unterbrechen.

Wenn ein Applet versucht (und scheitert), so etwas zu tun, wird vom Objekt `SyecurityManager` eine Exception vom Typ `SecurityException` erzeugt, die das Applet auswerten kann.

Viele dieser Beschränkungen kann man umgehen, wenn das Applet ein Java-Programm auf dem Server, von dem es heruntergeladen wurde, kontaktieren kann, das anstelle des Applets Dateien lesen und schreiben (aber auf dem Server) und Netzwerkverbindungen zu anderen Servern aufbauen kann - wenn, dann entstehen die Sicherheitsprobleme auf dem Server, der entsprechend abgesichert wird.

Aufbau von Applets

Applets sind immer Subklassen von `java.applet.Applet`, die folgende interessante Methoden zur Verfügung stellt:

- `init()` wird aufgerufen, nachdem der Browser das Applet geladen und den Bytecode verifiziert hat.
- `start()` wird aufgerufen, wenn die Ausführung des Applets beginnt.
- `stop()` wird aufgerufen, wenn die Ausführung des Applets endet (z.B. wenn man mit dem Browser auf eine andere Seite wechselt, oder das Browserfenster minimiert). Solange das Applet nicht gelöscht wurde, kann es mehrmals gestartet und wieder gestoppt werden!
- `destroy()` wird aufgerufen, wenn das Applet aus dem Speicher gelöscht wird.
- `paint()` wird beim Zeichnen des Applets aufgerufen.
- `update()` wird aufgerufen, wenn das Applet neu gezeichnet werden muss.

Diese Methoden (sie sind alle vom Typ `public void`) überschreibt man je nach Bedarf; eine sinnvolle Einteilung wäre z.B.: in `init()` bringt man Code unter, der sich um Initialisierungen kümmert (es ist also sozusagen der Konstruktor des Applets), der Parameter analysiert und Ressourcen lädt (z.B. Bilder und Sounddateien), in `start()` werden Threads gestartet und in `stop()` wieder beendet. `destroy()` braucht man in den meisten Fällen nicht.

Weil `java.applet.Applet` eine Subklasse von `java.AWT.Panel` ist, kann man die ganzen AWT-Grafikfunktionen verwenden.

Einbettung von Applets in HTML-Seiten

Ein Applet wird mit dem `<APPLET>`-Tag eingebunden, dessen Syntax so aussieht:

```
<APPLET codebase="pfad" code="MeineKlasse.class" height="Höhe" width="Breite">
</APPLET>
```

Natürlich ist, wie üblich bei HTML, die Groß- und Kleinschreibung egal, außer bei dem, was in den Anführungszeichen steht (in der Praxis ist es oft auch egal, und es funktioniert sogar, wenn die Anführungszeichen weggelassen werden - obwohl das eigentlich nicht korrekt ist). Der optionale Parameter `codebase` gibt den Pfad an, an dem das Applet und dazugehörige Dateien und Klassen zu finden sind, und kann als relative oder absolute URL angegeben werden. Die Parameter `height` und `width` geben die Höhe und Breite des Applets in Pixel an. Und `code` gibt natürlich den Dateinamen des Applets selber an. Was zwischen den Applet-Tags steht, wird nicht angezeigt - außer der Browser versteht kein Java, d.h. hier lassen sich wunderbar entsprechende Hinweise unterbringen.

Mit dem `<PARAM>`-Tag können an das Applet Parameter übergeben werden:

```
<PARAM name="Parametername" value="Wert">
```

Ein Beispiel:

```
<APPLET    codebase="http://beispiel.applet.net/classes"    code="MeineKlasse.class"    height="100"
width="300">  <PARAM  name="Lieblingsfarbe"  value="blau">  Leider  versteht  Ihr  Browser  kein  Java!
</APPLET>
```

Das `<APPLET>`-Tag hat noch einige Parameter mehr, die aber seltener gebraucht werden:

- `name` gibt der Instanz des Applets einen Namen (so dass Applets untereinander kommunizieren können)
- `alt` gibt einen Alternativtext ab, der angezeigt wird, falls das Applet nicht angezeigt werden kann (genauso wie beim `<IMG>`-Tag)
- `align` gibt die Ausrichtung an (`left`, `right`, `top`, `texttop`, `middle`, `absmiddle`, `baseline`, `bottom`)
- `hspace` und `vspace` geben den freien Platz in Pixel an, der links und rechts bzw. oben und unten um das Applet herum frei bleiben muss
- `archive` gibt ein oder mehrere JAR-Archivdateien an (durch Komma getrennt), die vom Applet benötigte Dateien enthalten. Das macht vor allem dann Sinn, wenn das Applet auf viele Dateien (z.B. Bilder, Sounds) zugreift; im JAR-Archiv werden die Daten erstens komprimiert, und zweitens muss nur einmal eine Datei übertragen werden.

Interessante Methoden

Das Java-Applet kann die in der HTML-Seite mit dem `<PARAM>`-Tag angegebenen Parameter mit der Methode `getParam("Parametername")` auslesen, die den Parameterwert als String zurückliefert.

Mit `showStatus("Statusmeldung")` kann das Applet eine Nachricht in die Statuszeile schreiben.

Ein Bild lädt man so: `Image image = getImage(getCodeBase(), "imgDir/a.gif")`.

Um eine HTML-Seite anzuzeigen, gibt es `public void showDocument(java.net.URL url)` und `public void showDocument(java.net.URL url, String targetWindow)` (mit `targetWindow` ist der HTML-Frame gemeint).

Ansonsten: siehe Java-API-Dokumentation von `java.applet.Applet`.

Programmier-Konventionen

Warum Konventionen? Weil ein Großteil der Zeit beim Programmieren auf die Fehlersuche verwendet wird und weil oft mehrere Personen mit einem Quelltext arbeiten oder sich ihn zumindest durchlesen, sollte er so übersichtlich und einfach zu verstehen wie möglich zu sein. Die folgenden Konventionen stammen von http://java.sun.com/docs/codeconv/ .

Dateinamen

Java-Quelltextdateien haben die Endung .java, Dateien mit dem kompilierten Java-Bytecode heißen .class.
Die Datei, in der der Inhalt eines Unterverzeichnisses erklärt und zusammengefasst ist, heißt README.

Aufbau einer Datei

Quelltext-Dateien sollten nicht länger als 2000 Zeilen sein, sonst werden sie unübersichtlich.

Der einleitende Kommentar

Zuerst kommt ein Kommentar, der die Klasse beschreibt, und den Programmierer, die Version und das Datum nennt:

```
/*
 * Name der Klasse
 *
 * kurze Beschreibung
 *
 * Programmierer
 *
 * Datum
 *
 * Versionsinfo
 *
 * Copyright
 */
```

Pakete und Import

Danach kommen die Pakete, und dahinter die `import`-Anweisungen:

```
package java.awt;
import java.awt.peer.CanvasPeer;
```

Deklaration der Klassen und der Schnittstellen

- Zuerst ein JavaDoc-Kommentar, der die Klasse oder Schnittstelle beschreibt.
- Dann kommt die eigentliche Klassen- oder Schnittstellendeklaration; innerhalb der Deklaration kann ein Kommentar über die Klasse stehen, falls er nicht zum obigen JavaDoc-Kommentar dazupasst.
- Dann kommen die `static`-Variablen, und zwar in der Reihenfolge `public`, `protected`, `private`.
- Dann kommen die Instanzvariablen, ebenfalls in der Reihenfolge `public`, `protected`, `private`.
- Dann die Konstruktoren,
- und schließlich die Methoden.

Einrückung und Aussehen der Zeilen

Tabulatoren sind 8 Zeichen breit; eingerückt wird in 4-Zeichen-Schritten, und die Zeilen sollten nicht länger als 80 Zeichen sein (evtl. sogar noch kürzer, je nach Bedarf).

Bei if-Verzweigungen generell 8 Zeichen einrücken.

Zeilenumbruch:

Am Besten nach einem Komma oder vor einem Operator umbrechen, möglichst so, dass inhaltlich Zusammengehöriges auch zusammen stehen bleibt. Die neue Zeile wird mindestens genauso weit eingerückt wie die vorige, wenn es unübersichtlich aussieht, dann noch weiter.

Kommentare

Kommentare bieten einen Überblick und erläutern Dinge, die nicht direkt aus dem Code ersichtlich sind, und wichtig für das Verständnis des Programms sind. Daher: nichts Unwichtiges, nichts, was sowieso aus dem Code ersichtlich ist, als Kommentar schreiben. Trotzdem mit Kommentaren nicht sparen, wenn es nötig ist.
Bei einem komplizierten Stück Code sollte man sich überlegen, ob man nicht besser den Code selbst verständlicher schreibt, anstatt einen langen Kommentar dazuzuschreiben.

Kommentare sollten keine Sonderzeichen enthalten, und 'Kunstwerke' wie Kästen aus Sternchen oder anderen Zeichen außenherum sind auch nicht nötig.

Es gibt ein Tool namens `indent`, das den Code formatiert. Dieses erkennt sog. „*block comments*“ daran, dass die erste Zeile nur aus `/*` besteht, und jede Zeile mit einem Sternchen beginnt (siehe oben: der einleitende Kommentar ist ein *block comment*), und formatiert diese nicht um.

Deklarationen

Jede Deklaration einer Variable, einer Methode usw. sollte auf einer eigenen Zeile stehen. Keinesfalls dürfen in einer Zeile z.B. verschiedene Variablentypen oder Variablen und Methoden gemischt deklariert werden. Beispiele:

```
int a, b;               // ist o.k., bei längeren Namen lieber auf zwei Zeilen verteilen
int c, d[];      // NIEMALS
float e, getValue();     // NIEMALS
```

Deklarationen stehen immer am Anfang des Blocks. Auch wenn z.B. Variablen erst später im Block benötigt werden - es verwirrt, wenn sie nicht am Anfang deklariert werden. Ausnahme: `for` -Schleifen:

```
for( int i=10; i<20; i++ )
```

Als Variablennamen sollten gleiche Namen vermieden werden (bei denen in einem inneren Block die lokale Variable die andere außerhalb des Blocks verdeckt).

Variablen sollten sofort bei der Deklaration initialisiert werden, oder zumindest so schnell wie möglich, damit sie einen definierten Wert haben.

Bei Methoden steht zwischen dem Methodennamen und den runden Klammern mit den Parametern kein Leerzeichen.

Ausdrücke

Möglichst nur ein Ausdruck pro Zeile, und (außer in Ausnahmefällen) nie durch Komma getrennt:

```
argv++; argc--; // Lieber nicht!
format.print(System.out, "error"), exit(1);       // NIE!!!
```

Bei Schleifen, Verzweigungen usw., überall dort, wo mehrere Anweisungen in geschweiften Klammern geschrieben werden, sollte man immer geschweifte Klammern verwenden, auch wenn es nur ein einzelner Ausdruck ist, der ohne Klammern stehen könnte - das macht den Code lesbarer und verhindert Fehler.
Der Inhalt von geschweiften Klammern wird eine Stufe tiefer eingerückt als der Code außenherum.

Bei `for`-Schleifen sollte man nicht mehr als drei durch Komma getrennte Variablen verwenden.

Bei `switch-case`-Ausdrücken sollte überall, wo das `break` weggelassen wird, stattdessen ein Kommentar stehen, der auf diese Tatsache hinweist. Außerdem sollte in jedem `switch-case`

-Ausdruck einen `default`-Abschnitt besitzen, der durch `break` abgeschlossen ist.

Leerzeilen und Leerzeichen

Zwischen Methoden, zwischen lokalen Variablen einer Methode und der ersten Anweisung der Methode, vor einem Kommentar, der eine oder mehrere ganze Zeilen belegt und zwischen logischen Abschnitten einer Methode sollte eine Leerzeile stehen.

Zwischen Klassen- und Schnittstellendefinition und generell zwischen verschiedenen Abschnitten eines Quelltextes sollten zwei Leerzeilen stehen.

Im Gegensatz zu Methoden sollte bei Schlüsselwörtern vor der Klammer mit den Parametern (z.B. bei for, while) ein Leerzeichen stehen. Außerdem sollte bei binären Operatoren links und rechts des Operators ein Leerzeichen stehen, sowie nach jedem Komma oder Semikolon, hinter dem in der gleichen Zeile noch etwas steht, und hinter jedem Cast. Beispiel:

```
while (x != 5 )
{
    myMethod((byte) aNum, (Object) x); x++;
}
```

Namen

Namen von Klassen und Schnittstellen sollten Nomen sein mit großem Anfangsbuchstaben (auch im Englischen). Wenn es sich um einen aus mehreren Wörtern zusammengesetzten Begriff handelt, so wird alles zusammengeschrieben, und der Anfangsbuchstabe jedes Teilwortes wird auch groß geschrieben:

```
class ImageSprite;
```

Namen von Methoden sollten Verben sein mit kleinem Anfangsbuchstaben. Wenn es sich um einen aus mehreren Wörtern zusammengesetzten Begriff handelt, so wird alles zusammengeschrieben, und der Anfangsbuchstabe jedes folgenden Teilwortes wird groß geschrieben:

```
getBackground();
```

Für die Namen von Variablen gilt im Prinzip das Gleiche; sie sollten möglichst kurz und aussagekräftig sein. Eine Ausnahme sind Variablen, die nur kurz gebraucht werden, z.B. als Schleifenzähler: dafür hat sich `i, j, k, m, n` eingebürgert, wenn es sich um eine Variable vom Typ `int` handelt, und `c, d, e` für `char`-Variablen.

Konstanten werden komplett groß geschrieben, die Teilwörter werden durch Underscore getrennt:

```
final int MIN_WIDTH = 4;
```

Was sonst noch einen guten Programmierstil ausmacht

- Zahlenwerte sollten möglichst nicht direkt im Code stehen, sondern durch Konstanten ersetzt werden (verbessert die Lesbarkeit).
- Nicht mehrere Zuweisungen in einer Zeile oder verschachtelte Zuweisungen, wie z.B.:

  ```
  FooBar.fChar = barFoo.lchar = 'c';
  d = (a = b + c) + r;
  ```

- Bei `if`-Konstruktionen den Vergleichsoperator nicht weglassen.
- Mit Klammern nicht sparen, auch wenn man wegen der Rangfolge der Operatoren die Klammern weglassen könnte:

  ```
  if (a == b && c == d)     // nicht so gut lesbar
  if ((a == b) && (c == d)) // besser lesbar, weniger fehleranfällig
  (x >= 0) ? x : (-x)
  ```

- Bei einem Vergleich zuerst die Konstante und dann die Variable schreiben:

  ```
  3 == a
  ```

  Dann wird nämlich eine Fehlermeldung erzeugt, wenn ein Gleichheitszeichen vergessen wurde, während

```
a = 3
```

korrekt ist und nicht bemerkt wird, auch wenn man etwas ganz anderes schreiben wollte.

- Nicht ohne guten Grund eine Variable `public` machen. Stattdessen Methoden zur Verfügung stellen, mit denen die Variable geändert werden kann.
  Einer der wenigen Gründe wäre, eine Klasse ohne Methoden wie einen `struct` in C zu verwenden.

- Um auf eine Klassenvariable oder –Methode zuzugreifen, sollte man den Klassennamen statt dem Namen der Instanzvariablen verwenden:

```
Aclass.classMethod();         // o.k.
anObject.classMethod();       // lieber nicht
```

- Bei Stellen, bei denen man sich nicht sicher ist, ob der Code korrekt arbeitet, sollte man das mit Kommentaren kennzeichnen:

```
// XXX   bedeutet: es scheint zu funktionieren, aber irgendwas ist seltsam
// FIXME bedeutet: da ist auf jeden Fall irgendwo ein Fehler
```

Debugging von Java-Code mit jdb

Um Java-Bytecode debuggen zu können, muss er mit der Option `-g` kompiliert sein (damit wird nichtoptimierter Bytecode erzeugt):

```
javac -g MeineKlasse.java
```

Danach kann man debuggen; wenn die Klassen in einem oder mehreren anderen Verzeichnissen sind, muss man dem Debugger den gesamten Pfad zu den Klassen mitteilen. Der Aufruf des Debuggers sieht dann so aus:

```
jdb -classpath C:/Meinzeug/Projekt;C:/java/java1_0/lib/classes.zip MeineKlasse
```

Man kann das Programm auch im normalen Interpreter mit der Option `-debug` starten. Dann wird ein Passwort ausgegeben, der Debugger kann auf einem anderen Rechner laufen, ihm muss das Passwort mitgeteilt werden:

```
java -debug MeineKlasse
jdb [-host <HostName>] -password <Passwort>
```

Um ein Applet zu debuggen, startet man den Appletviewer mit der Option `-debug`:

```
appletviewer -debug MeineAppletSeite.html
```

Befehle für jdb

- `stop at MeineKlasse:181` setzt einen Breakpoint in Zeile 181
- `stop in MeineKlasse.meineMethode` setzt einen Breakpoint am Anfang der angegebenen Methode
- `run` startet die Programmausführung
- `list` zeigt den Programmcode in der Umgebung des Breakpoints an, `list 150` den Code in Zeile 150
- `locals` zeigt die lokalen Variablen an
- `print meineInstanz.meineVariable` gibt den Wert der Variablen `meineVariable` aus
- `dump meineInstanz` gibt alle Mitgliedsvariablen des Objekts `meineInstanz` aus
- `next` führt die nächste Codezeile aus
- `where` zeigt den Stack und den Stack-Kontext an
- `up` führt im Stack eine Ebene nach oben
- `cont` setzt die Ausführung fort
- `gc` startet die Garbage Collection und gibt ungenutzten Speicher frei
- `help` oder `?` zeigt eine kurze Hilfe an
- `!!` wiederholt den letzten Befehl
- `exit` oder `quit` beenden den Debugger

Literaturverzeichnis

- Paul Schulz: Java 1.1, Markt & Technik-Verlag, München, 1997
- Friendly, Campione, Walrath, Huml: The Java Tutorial, http://java.sun.com/docs/books/tutorial/index.html
- Brandl, Göbel, Harlfinger, v. Oheimb: Skript zum Java-Kurs, http://www.informatik.tu-muenchen.de/~javakurs/skript.html
- Michael Morrison u.a.: Java Unleashed, Sams.net Publishing
- Java Coding Conventions: http://java.sun.com/docs/codeconv/
- Java API-Dokumentation: http://java.sun.com/j2se/1.3/docs/api/index.html, http://java.sun.com/j2se/1.3/docs.html (Download)
- Java-Tutorial: http://java.sun.com/docs/books/tutorial/index.html