PiSoftware

www.pisoftware.de

JAVA

Einführung in die objektorientierte Programmierung mit JAVA

Ralf Pichocki

2. Auflage
2003

2. Auflage, 2003

Bahnhofstraße 190
D-45770 Marl-Sinsen

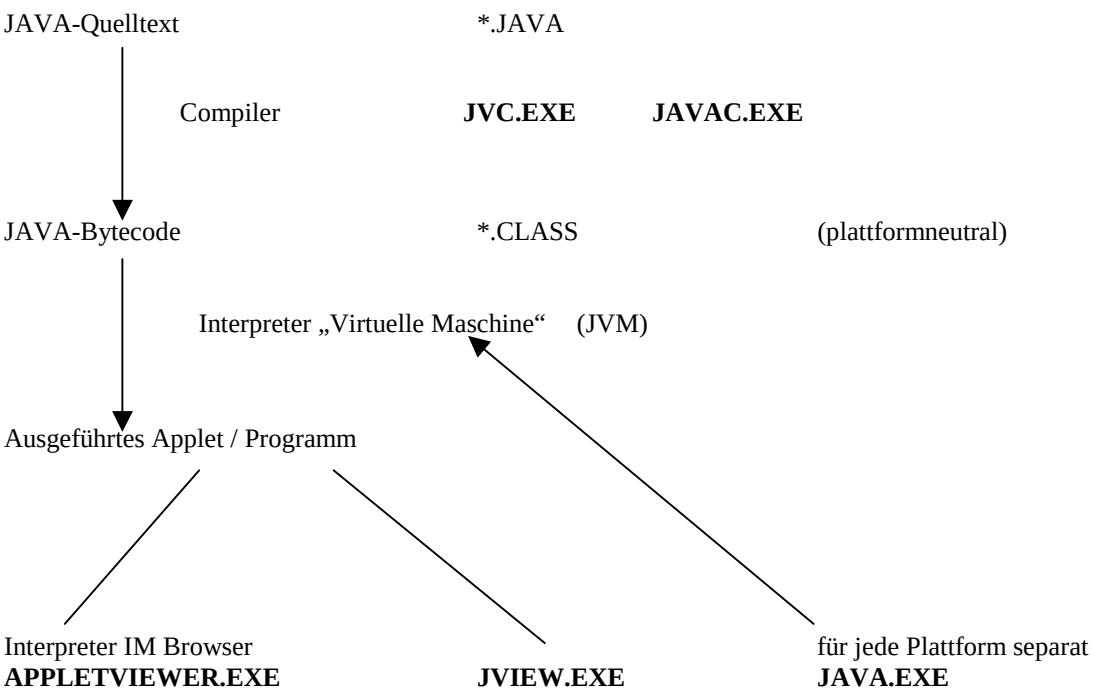
Tel +49 2365 880480
Fax +49 2365 880482
info@pisoftware.de
http://www.pisoftware.de

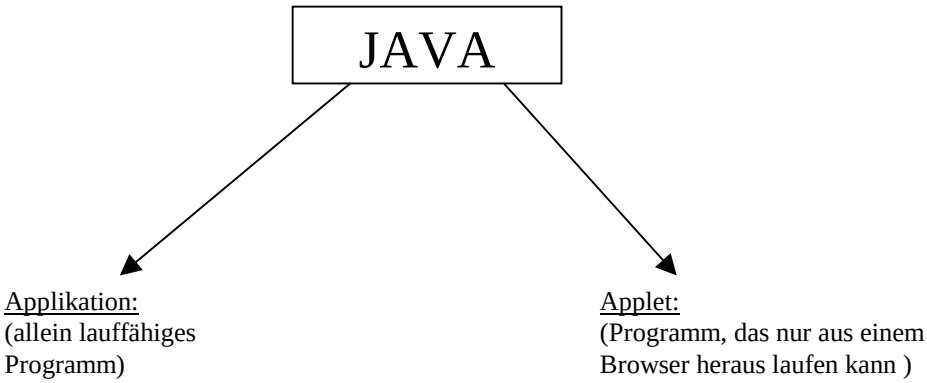
Bearbeitung: Angelika Geßler
Gestaltung: Angelika Geßler
Druck: Angelika Geßler

Inhaltsverzeichnis

Allgemeines über JAVA:

JAVA ist plattformunabhängig.





JAVA ist vollständig objektorientiert.
Die Klassenbibliotheken JFC (Sun) und AFC (MS) sind plattformunabhängig.
Die Klassenbibliothek WFC (MS) ist nicht plattformunabhängig, sie läuft nur unter Win32.
Objektorientierte Sprachen wie JAVA und C++ lassen eigene Klassenbildung und Vererbung zu. In JAVA gibt es jedoch keine Mehrfachvererbung.

Hinter einem Befehl steht ein Semikolon(;) (außer hinter {}). Methoden werden erkannt an den Klammern hinter ihrem Bezeichner. Zwischen den Klammern können sogenannte Argumente/Parameter stehen. Variablen sind Bezeichner für „Speicherplätze“ eines bestimmten Typs. Variablen müssen vor ihrer Benutzung angelegt werden, außerdem müssen ihnen ein Wert zugewiesen werden.

JAVA erlaubt keine Hardwarezugriffe.

Applets erhalten ihr Fenster vom Browser, sie haben eine Init() – Methode und werden immer von der Klasse „Applet“ abgeleitet.

Applikationen sind für ihr Fenster selbst verantwortlich, sie brauchen immer eine Methode main() und sind von keiner speziellen Klasse abgeleitet.
Der „Startpunkt“ einer Applikation ist die Methode:

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

Sie wird beim Programmstart – nach dem Laden der als Argument angegebene Klasse – aufgerufen.

Attribut:

Ein Attribut ist ein Element, das in jedem Objekt einer Klasse gleichermaßen enthalten ist und von jedem Objekt mit einem individuellen Wert repräsentiert wird. Attribute sind vollständig unter der Kontrolle der Objekte, von denen sie Teil sind. Ein Attribut ist eine bestimmte Eigenschaft einer Klasse.

Basisklasse:

Eine Basisklasse is eine Klasse, von der andere Klassen abgeleitet wurden.

Instanz:

Erzeugt man ein Objekt anhand einer bestimmten Klassendefinition, ist dieses Objekt eine Instanz der Klasse. Das Erzeugen eines Objektes anhand einer Klassendefinition wird auch als Instanziieren bezeichnet. Instanziieren bewirkt, dass im Hauptspeicher ein Bereich reserviert wird, in dem die Klasse und alle Member-Variablen aufgenommen werden können. Anschließend erzeugt der Interpreter ein Abbild der Klasse und die Member-Variablen werden initialisiert. Falls ein Default-Konstruktor vorhanden ist, wird dieser aufgerufen.

Klasse:

Eine Klasse beschreibt eine bestimmte Menge von Objekten, die alle über die gleichen Eigenschaften verfügen. Zu den Eigenschaften einer Klasse gehören Attribute. Darüber hinaus verfügt jede Klasse über bestimmte Verhaltensweisen und Nachrichten, auf die sie reagieren kann, diese werden in der Klassendefinition festgelegt.
Eine Klasse ist die Definition der Attribute, Operationen und der Bedeutung für eine Menge von Objekten.

Operation:

Operationen sind Dienstleistungen, die von einem Objekt angefordert werden können.

Nachrichten:

Nachrichten sind ein Mechanismus, mit dem Objekte untereinander kommunizieren können. Eine Nachricht übergibt einem Objekt die Information darüber, welche Aktivität von ihm erwartet wird, d.h. eine Nachricht fordert ein Objekt zur Ausführung einer Operation auf. Eine Nachricht besteht aus einem Namen, einer Liste von Argumenten und geht an genau einen Empfänger.

Methode:

Eine Methode bestimmt, wie eine Klasse auf eine bestimmte Nachricht reagiert. Da eine Klasse auf beliebig viele Nachrichten reagieren kann, ist die Anzahl der Methoden einer Klasse nicht limitiert. Abgeleitete Klassen erben alle Methoden ihrer Basisklasse, können aber durch Überschreiben von Methoden eine unterschiedliche Verhaltensweise implementieren. Der Aufruf einer Methode wird ersetzt durch den Wert, den die Methode zurückgibt, der Aufruf kann also auf der rechten Seite einer Zuweisung stehen. Statische Methoden (static) greifen nur auf statische Variablen zu!

Objekt:

Ein Objekt ist eine im laufenden System konkret vorhandene und agierende Einheit. Jedes Objekt ist ein Exemplar einer Klasse. Ein Objekt enthält, durch Attribute repräsentiert, Informationen, deren Struktur in der Klasse definiert sind. Die Werte der Attribute sind für jedes Objekt individuell. Objekte werden von Klassen produziert. Das Verhalten eines Objektes wird beschrieben durch die möglichen Nachrichten, die es verstehen kann. Ein Objekt ist eine konkrete Instanz einer Klasse. Objekte sind Einheiten aus Diensten (Funktionen) und Eigenschaften (Daten), und zwar untrennbar.

Überschreiben:

Wenn eine Klasse eine Methode ihrer Basisklasse anderes implementiert, sagt man, dass sie diese überschreibt.

Überladen von Methoden

In JAVA kann man mehrere Methoden mit dem gleichen Namen definieren, wenn sich die Methoden durch Art oder Anzahl der Parameter unterscheiden. Das hat den Vorteil, dass man den selben Namen ohne oder mit Parameter verwenden kann.

Beispiel

```
void  spieleSound( URL eineInternetAderesse)
{
      // holt eine Sounddatei über das Internet
}

void spieleSound( String dateiName )
{
      // spielt eine lokale Sounddatei ab
}
```

Bei Verwendung einer dieser Methoden im späteren Programmtext entscheidet der Compiler anhand der Parameter, welche Version gemeint ist.
Methoden in verschiedenen Klassen, die die selbe Aufgabe haben, können denselben Namen bekommen. Dadurch, dass sie verschiedenen Objekten zugeordnet sind, kann der Compiler die jeweils passende Methode selbstständig herausfinden.

Unterklasse:

Eine Unterklasse, wird von einer anderen Klasse abgeleitet, sie ist quasi das Gegenstück zu einer Basisklasse.

Schnittstelle:

Schnittstellen beschreiben einen ausgewählten Teil des extern sichtbaren Verhalten von Modellelementen.

Vererbung/ Ableitung

Kennzeichnend ist, dass alle Objekte einer Klasse über identische Eigenschaften verfügen und dass Unterklassen eine Spezialisierung darstellen. Die von einer Klasse abgeleitete Unterklasse verfügt automatisch über alle Eigenschaften der Oberklasse. Objekte abgeleiteter Klassen sollen jederzeit anstelle von Objekten ihrer Basisklasse eingesetzt werden können. Vererbung ist ein Umsetzungsmechanismus für die Beziehung zwischen Ober- und Unterklasse, wodurch Attribute und Operationen auch Unterklassen zugänglich werden.

Beispiel

```
//-----------------------------Hauptprogramm-----------------------------//
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;
//
// ErstesApplet
//
public class ErstesApplet extends Applet
{
     Ball ballStdCtor;
      Ball ballWertCtor;

      Buntball buntballStdCtor;
      Buntball buntballWertCtor;

      Punktball punktballStdCtor;
      Punktball punktballWertCtor;

      public void init()
      {
                  ballStdCtor = new Ball();
                  ballWertCtor = new Ball( 200, 200, 60 );

                  buntballStdCtor = new Buntball();
                  buntballStdCtor.setX( 250 );
                  buntballWertCtor = new Buntball( 300, 100, 50,
                  Color.magenta );

                  punktballStdCtor = new Punktball();
                  punktballStdCtor.setY( 250 );
                  punktballWertCtor = new Punktball( 200, 200, 80,
                  Color.blue, Color.red );
      }
```

```
	public void paint( Graphics g )
	{
			ballStdCtor.zeichne( g );
			ballWertCtor.zeichne( g );

			buntballStdCtor.zeichne( g );
			buntballWertCtor.zeichne( g );

			punktballStdCtor.zeichne( g );
			punktballWertCtor.zeichne( g );

			g.drawString( "Ich bin noch da!", 20, 20 );
	}
}
//---------------------------KlasseBall( Basisklasse )--------------------//

import java.awt.Graphics;
//
// Ball
//
public class Ball
{
	//Attribute
	private int x;
	private int y;
	private int r;

	//Std-ctor
	public Ball()
	{
		//ruft Wert-ctor mit Std-Werten auf
		this( 50, 50, 20 );
	}

	//Wert-ctor
	public Ball( int neuX, int neuY, int neuR )
	{
		setX( neuX );
		setY( neuY );
		setR( neuR );
	}
```

```
	public int getX() { return x; }
	public void setX( int neuX ) { x = neuX; }

	public int getY() { return y; }
	public void setY( int neuY ) { y = neuY; }

	public int getR() { return r; }
	public void setR( int neuR ) { if ( neuR >= 0 ) r = neuR; }

	public void zeichne( Graphics g )
	{
		g.fillOval( x-r, y-r, 2*r, 2*r );
	}
}

//----------------------KlasseBuntBall( von Ball abgeleitet )-------------//
import java.awt.Graphics;
import java.awt.Color;
import Ball;
//
// Buntball
//
//
public class Buntball extends Ball
{
	//Attribute
	private Color farbe;

	//Std-ctor
	public Buntball()
	{
		super();
		setFarbe ( Color.red );
	}

	//Wert-ctor
	public Buntball( int neuX, int neuY, int neuR, Color c )
	{
		super( neuX, neuY, neuR );
		setFarbe ( c );
	}

	public Color getFarbe() { return farbe; }
	public void setFarbe( Color neuC ) { farbe = neuC; }
```

```
	public void zeichne( Graphics g )
	{
		//merke mir die ursprüngliche Farbe
		//des "geborgten" Graphics Objektes
		Color oldColor = g.getColor();
		//wenn ich das nicht tue, weiß ich sie später nicht :-(

		//setze jetzt MEINE Farbe zum Zeichnen
		g.setColor( getFarbe() );
		//VERÄNDERE also das Graphics Objekt

		//rufe jetzt das geerbte Zeichnen auf
		super.zeichne( g );

		//stelle die ursprüngliche Farbe wieder her
		g.setColor( oldColor );
		//behandele also das "geborgte" Graphics Objekt pfleglich

	}
}
```

Polymorphismus:

Polymorphismus (Vielgestaltigkeit) heißt, dass gleichlautende Nachrichten an kompatible Objekte unterschiedlicher Klassen ein unterschiedliches Verhalten bewirken können. Beim dynamischen Polymorphismus wird eine Nachricht nicht zur Kompilierzeit, sondern zur Programmlaufzeit einer konkreten Operation zugeordnet. Voraussetzung hierfür ist das dynamische Binden.

Assoziation:

Eine Assoziation beschreibt die gemeinsame Bedeutung und Struktur einer Menge von Objektverbindungen. Eine Assoziation beschreibt eine Verbindung zwischen Klassen. Gewöhnlich ist eine Assoziation eine Beziehung zwischen zwei verschiedenen Klassen. Assoziationen werden auch Benutz-Beziehung genannt. Eine Assoziation ist eine Beziehung zwischen verschiedenen Objekten einer oder mehrer Klassen.

Aggregation:

Eine Aggregation ist eine besondere Variante der Assoziation. Kennzeichnend für alle Aggregationen ist, dass das Ganze Aufgaben stellvertretend für seine Teile wahrnimmt. Eine Aggregation ist eine Beziehung zwischen zwei Klassen, die untereinander in Beziehung stehen, wie ein Ganzes zu seinen Teilen. Eine Aggregation ist die Zusammensetzung von Einzelteilen. Aggregationen sind Hat – Beziehungen.

Komposition

Eine Komposition ist eine strenge Form der Aggregation, bei der die Teile vom Ganzen existenzabhängig sind. Jedes Teil ist nur Teil eines Kompositionsobjektes.

Übersetzung von JAVA – Programmen

Um JAVA zu übersetzen, benutzt man eine IDE (integrierte Entwicklungsumgebung) oder den `javac` Compiler. Eine JAVA–Source sollte mit der Erweiterung „`.java`“ enden. Der `javac` Compiler übersetzt JAVA–Sourcen in Bytecodedateien mit der Erweiterung „`.class`“.
Um ein Programm zu modularisieren, sollten die Klassen in verschiedenen Sourcen definiert werden. Der `javac` erwartet in jeder Source eine Klasse mit dem Attribut `public`. Dies bedeutet, dass diese Klasse von beliebigen anderen JAVA–Sourcen benutzt werden kann. Der Name der `public` Klasse und der Name der JAVA - Source, in der sie enthalten ist, müssen (bis auf die Erweiterung) gleich sein.
Beim Übersetzen muss der JAVA – Compiler auf eventuell schon bestehende Klassen (z.B. die JAVA- Basis - klassen, wie beispielsweise die Klasse `System`) zugreifen.
Um diese im Dateisystem zu lokalisieren, kann man die Environmentvariable `CLASSPATH` auf mehrere, durch ‘‘`;`‘‘ (Win32) oder “`:`“ (UNIX) separierte Directories setzen, in denen dann nach `*.class` Dateien gesucht werden. Ein solcher Klassenpfad kann auch via Schalter `–classpath` beim Übersetzen angegeben werden.

Eine JAVA Applikation (kein Applet) kann mittels des Bytecode Interpreters `java` ausgeführt werden. Als Argumente übergibt man den Namen einer öffentlichen (`public`) Klasse und Applikationsargumente. Der Bytecodeinterpreter läd dann diese Klasse und startet die darin enthaltene `public static void main` Funktion. Werden weitere Klassen benötigt, lädt java diese (ebenfalls unter Zugriff auf die `CLATHPATH` Environment Variable bzw. gemäß – `classpath` beim `java` Aufruf) nach.
Um beim Übersetzen oder Ausführen mit zu verfolgen, welche Klassen gerade nachgeladen werden, kennen `javac` und `java` die Option `–verbose`.

Erstes Beispielprogramm

```
//Hier wird die Klasse namens "HalloApp" definiert
public class HalloApp
{
      //Hier beginnt die öffentliche Methode namens "main"
      public static void main( String[] args )
      {
            //Dieser Befehl gibt eine Meldung aus
            System.out.println( "Hallo, Welt!" );
      }     //Hier endet "main"
}     //hier endet die Klasse "HalloApp"
```

Applets

Applets sind JAVA–Programme für das World Wide Web. Um sie ansehen zu können, müssen sie in einen HTML–Quelltext eingebettet werden. Mit HTML beschreibt der Programmierer das Layout und die Komponenten einer WWW–Seite.

Beispiel

```
<HTML>
<HEAD>
<TITLE>

Beispiel fuer die Einbindung von Applets

</TITLE>
</HEAD>
<BODY>
<H3 align = center>

So werden Applets in HTML – Seiten beschrieben:

</H3>

<APPLET code = Name_der_Quelldatei width = 80 heigth = 10 >

</APPLET>
</BODY>
</HTML>
```

Code beschreibt den Namen der Quelldatei. Ist das Wurzelverzeichnis nicht genauer angegeben, so muss sich die Datei im selben Verzeichnis wie die HTML – Seite befinden.

Import von Klassen

In JAVA gibt es keinen Präprozessor und keine include-Dateien. Am Anfang eines Programms kann jedoch angegeben werden, welche bereits kompilierten Klassen verwendet werden sollen; der Compiler liest die benötigten Informationen aus den bereits übersetzten Klassendateien.

```
import Verzeichnis.Paketname.Klasse;
import Verzeichnis.Paketname.*;
```

Wenn nicht nur die einzelne Klasse importiert werden soll, wird das Sternchen `*` als Platzhalter verwendet. Der Compiler durchsucht das angegebene Paket nach benötigten Referenzen.
Die grundlegenden Klassen, die sich in dem Paket `java.lang.*` befinden, werden immer automatisch importiert. Alle bereits von Sun vergefertigten Klassen befinden sich in der Datei `CLASSES.ZIP`. Die einzelnen kompilierten Klassendateien sind allerdings nicht komprimiert, wie die .ZIP – Endung vermuten lassen könnte. Die wichtigsten Pakete, die über import angesprochen werden können, sind:

`java.lang.*`	JAVA – Basisklassen
`java.awt.*`	Abstract Windows Toolkit
`java.awt.event.*`	Ereignisbehandlung
`java.awt.image.*`	Bilddarstellung
`java.net.*`	Internetzugriff
`java.io.*`	Dateizugriff
`java.util.*`	Verschiedene Utilities, z.B. Datum
`java.applet.*`	Zur Darstellung von Applets

Der `import` – Befehl vereinfacht die Schreibweise bei Verwendung von Objekten aus anderen Klassen. Im Prinzip ist jede Klasse durch Kombination von Paketen, Verzeichnis und Klassenname erreichbar. Anstatt jedesmal `java.awt.Button` zu schreiben, kann auch am Programmbeginn `java.awt.Button` importiert werden, und auf die benötigte Klasse mit `Button` zurückgegriffen werden.

Beispiel

```
import java.applet.Applet;
import java. awt.Graphics;

 public class Hello extends Applet
 {
   public void paint( Graphics g )
   {
      g.drawString( "Java!!", 50,50 );
   }
}   //kein Semikolon
```

Datentypen

boolean	Wahrheitswert	true, false
byte	Byte	0...255
short	kleine Ganzzahl	-32768..32767
int	Ganzzahl	$-2^{31}...2^{31}$
long	große ganze Zahl	$-2^{63}...2^{63}$
float	Fließkommazahl	normaleGenauigkeit
double	Fließkommazahl	doppelte Genauigkeit
char	ein(Text-) Zeichen	a..z, A...Z, 0...9

Es ist möglich Datentypen in andere umzuwandeln, dies geschieht durch Casting (cast).

Ein weiterer „Datentyp“ ist die Klasse String. Diese stellt genaugenommen ein „Array von char“ dar, d.h. ein Feld aus Einzelzeichen. Die Besonderheit ist, dass die Größe/Länge des Feldes auch nach der Deklaration veränderbar ist. Insbesondere kann ein String die „Länge“ 0 haben! Er kann aber auch die Länge 1 haben, also GENAU EIN Zeichen enthalten. Damit ist er aber trotzdem ein String und wird nicht zum char.

Ein Beispiel

```
public class Datentypen
{
	// Hauptfunktion main()
	public static void main (String [] args)
	{
		//Deklaration der Variablen
		double d;
		float f;
		byte b;
		long l;

		//Initialisierung der Variablen
		d = (double) 1/3;
		f = (float) 1/3;
		b= (byte) 216;
		l = 4096;

		// Ausgabe des Wertes d( double )
		System.out.print ("Wert Double = (cast) d = ");
		System.out.println(d);

		//Neue Zuweisung d = f
		d = f;

		//Ausgabe des Wertes d als Fließkommazahl
		System.out.print ("Wert Double = Float d = ");
		System.out.println(d);

		//Neue Zuweisung d = b
		d = b;
```

```
            //Ausgabe des Wertes d als Byte
            System.out.print("Wert Double = Byte d = ");
            System.out.println(d);

            //Neue Zuweisung d = l
            d = l;

            //Ausgabe des Wertes d als Long
            System.out.print("Wert Double = Long d = ");
            System.out.println(d);
      }
}
```

Typumwandlung

In JAVA sind alle Variablen streng typisiert. Zur Umwandlung eines Typs in einen anderen ist eine ausdrückliche Aufforderung an den Compiler notwendig. Wenn zwei Zahlenwerte ineinander umgewandelt werden sollen, reicht die Angabe des neuen Typs in Klammern aus. Dieses Vorgehen nennt man Casting

Beispiel

```
int i;
float f;
// weiterer Code
f = (float) i;              // Wandelt i in den Typ float um
i = (int) f;                // Wandelt f in den Typ int um
```

Bei Verwendung der AWT ist es oft erforderlich, aus einem Textfeld einen String in eine Zahl und umgekehrt zu wandeln.

Beispiel

```
Textfield t;
int i;

// Erstens: String in eine Zahl wandeln

String s = t.getText();     // Inhalt des Textfeldes in s speichern
i = Integer.parseInt( s );  // Umwandlungsfunktion für Integer aufrufen

// Zweitens: Zahl in einen String wandeln

s = String.valueOf( i );    // Integer in einen String wandeln
t.setText( s );             // und dem Textfeld zuweisen
```

Konstanten

Konstanten werden in JAVA im Prinzip wie Variablen deklariert, allerdings erhalten sie den Modifizierer `final`. Dadurch wird festgelegt, das der Wert nur ein Mal initialisiert wird und der Wert sich nicht mehr ändert.

Zuweisungen

Das Gleichheitszeichen (=) weist einer Variablen einen Wert zu, der auch berechnet sein kann.
Bei Zuweisungen wird zunächst alles, was rechts vom Gleichheitszeichen steht, ausgewertet.

Beispiele

```
int nZahl;
nZahl = 3;              // nZahl hat hiernach den Wert 3

String strName;
StrName = "Alf";        // strName ist nun „gleich“ ‘‘Alf“
```

Variablen müssen nach der Deklaration, um genauer zu sein, vor der ersten Benutzung mit einem Wert belegt werden, dies kann per Zuweisung geschehen, oder direkt bei der Deklaration als sogenannte Initialisierung.

Beispiel

int nZahl = 3;

Bemerkung

JAVA kennt keine Trennung zwischen Deklaration und Definition wie C und C++. Man redet daher in der Regel von der Deklaration einer Variablen, Methode oder Klasse. Diese ist aber immer gleichzeitig auch Definition.

Kommentare

Kommentieren kann man auf zwei verschiedene Arten.
Mit // wird ein einzeiliger Kommentar eingeleitet, er beginnt direkt hinter den Zeichen und endet am Ende der Zeile.
Mit **/*** wird ein mehrzeiliger Kommentar eingeleitet, der mit ***/** beendet wird.
Es ist sinnvoll, so viel wie nur möglich zu kommentieren, damit man sich in seinem Quelltext auch noch nach einem oder mehreren Jahren zurechtfindet und der Quelltext „wartungsfreundlich“ ist.

Konventionen

- Klassennamen beginnen mit Großbuchstaben: `String`
- Alle Konstanten werden ganz groß geschrieben: `CENTER`
- Andere Bezeichner beginnen klein: `zahl`
- Bezeichner aus mehreren Worten werden aneinander geschrieben; jedes weitere Wort beginnt gross: `einMehrwort`
- Die „oberste“ Klasse muss public sein und die Startklasse muss eine Methode:
  `public static void main( String[] args]`
  haben.

Methoden:

Syntax:

```
<Zugriffspezifizierer><static> <Rückgabetyp> <Name der Methode>(<Parameter>)
```

Beispiel

```
public static void Name(  String ich )
```

Für jeden formalen Parameter einer Funktion wird bei ihrem Aufruf eine lokale Variable angelegt und mit dem Wert des Aufrufparameters initialisiert. Einfache Datentypen werden immer als Wertkopie übergeben.
Eine Funktion kann entweder KEINEN Wert zurückgeben (void) oder genau EINEN.
Rückgabe mit:

```
return<wert>;
```

Aus Funktionen ohne Rückgabewert kann mit return; zurückgesprungen werden. Funktionen können mehr als einen Parameter haben. Sie müssen durch Kommata voneinander getrennt werden. Der Befehl return<wert> kann mehrmals in einem Methodenrumpf auftauchen.

Bemerkung

In JAVA ist - anders als in C/C++ - die Typangabe als Präfix von Bezeichnern unüblich.

Zugriffspezifizierer für Methoden

public	jeder darf zugreifen
(default)	jeder aus dem selben Package darf zugreifen
protected	jeder aus einer abgeleiteten Klasse oder dem selben Package darf zugreifen
private	niemand außer die eigene Klasse darf zugreifen

Die Standardsicherheit ist strikter als protected. Wenn eine Variable oder Methode ohne einen Zugriffsspezifizierer deklariert wird, so ist sie nur innerhalb der Klasse, in der sie definiert ist und in den Klassen des gleichen Pakets sichtbar. Sie ist in den Subklassen (außer es sind Subklassen des gleichen Pakets) nicht sichtbar.

Situation	public	default	protected	private
Erreichbar für nicht-Subklasse aus dem gleichen Paket	ja	ja	ja	nein
Erreichbar für Subklasse aus dem gleichen Paket	ja	ja	ja	nein
Erreichbar für nicht-Subklasse aus anderem Paket	ja	nein	nein	nein
Erreichbar für Subklasse aus anderem Paket	ja	nein	nein	nein
Geerbt von Subklasse in gleichem Paket	ja	ja	ja	nein
Erreichbar für Subklasse in anderem Paket	ja	nein	ja	nein

Bezeichner:

In JAVA beginnt ein Bezeichner mit einem Buchstaben, dem beliebig viele weitere Bauchstaben, Zahlen und Unterstriche folgen können.
Der erste Buchstabe eines Bezeichner darf kein Umlaut sein. Für alle anderen Zeichen eines Bezeichners sind alle Zeichen des 16 – Bit – Unicodes erlaubt. Es ist jedoch davon abzuraten Umlaute zu verwenden, da manche Betriebssysteme Fehler melden können, oder da es möglich ist, dass Klassen nicht richtig verarbeitet werden.
Die Namen von Variablen und Methoden beginnen mit kleinen Buchstaben, die von Schnittstellen und Klassen mit großen Buchstaben. Falls Bezeichner aus mehreren Wörtern besteht, wird das folgende Wort mit einem Großbuchstaben begonnen.
Methoden, die den Wert einer Variablen liefern, werden immer mit get und solche zum Verändern einer Variable mit set gekennzeichnet.
Methodennamen unterscheiden sich von Variablennamen dadurch, dass immer runde Klammern folgen. Dies ist selbst dann der Fall, wenn keine Parameter übergeben werden.
Konstanten werden ganz in Großbuchstaben geschrieben. Sollte hier der Name einer Konstanten aus mehreren Wörtern bestehen, ist ein Unterstrich einzufügen.
Die Klassenbibliotheken von JAVA enthalten nur englische Bezeichner; für eigene Bezeichner können aber natürlich auch deutsche Bezeichner verwendet werden.

Sichtbarkeit von Bezeichnern

Klassen:

(default)	Der Name dieser Klasse ist nur innerhalb des eigenen Paketes sichtbar; d.h. nur in Klassen dieses Paketes darf der Name angesprochen werden.
public	Die Klasse ist auch von außerhalb des Paketes ansprechbar.

Methoden und Variablen auf Klassenebene:

private	Der Bezeichner ist nur in der eigenen Klasse bekannt, kann also nicht von außerhalb angesprochen werden.
(default)	Der Bezeichner ist im gesamten Paket und in abgeleiteten Klassen bekannt.
protected	Der Bezeichner ist im gesamten Paket bekannt.
public	Der Bezeichner ist überall dort bekannt, wo seine Klasse bekannt ist.

Variablen in Blöcken („lokal“):

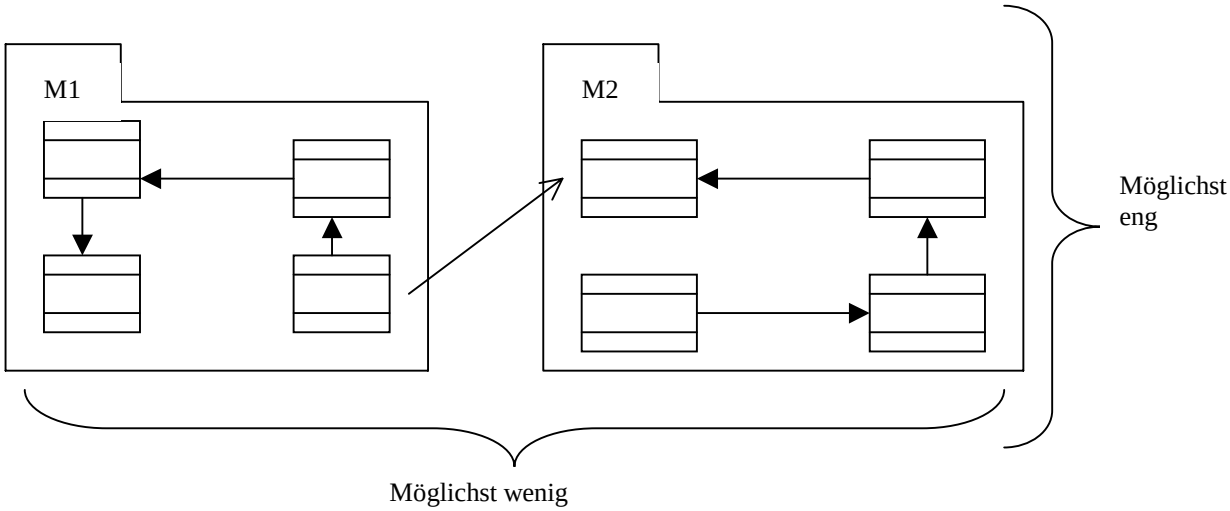
Die Variablen sind innerhalb der geschweiften Klammern ({}) bekannt. Bei Namenskonflikten mit einem Bezeichner auf Klassenebene dominiert die lokale Definition.

Packages

Klassen werden organisiert in Packages. Dabei startet die Hirachie in einem Verzeichnis, das auf der Kommandozeile mit der Umgebungsvariable CLASSPATH = eingestellt wird.
Packages entsprechen den Verzeichnissen im JAVA Verzeichnis. Klassen sind zurnächst nur innerhalb ihres Paketes bekannt (protected). Um sie außerhalb sichtbar zu machen, müssen sie mit dem Modifier „public“ versehen werden.
Logische Zusammenfassungen (Subsysteme, Module, Pakete, Telsysteme) mehrerer Klassen und Schnittstellen werden mit den Zielen:

- Großer Zusammenhalt (hohe Kohäsion) innerhalb des Moduls
- Lose Koppelung zischen den Modulen

zusammengefasst.



Konvention:

Packages werden beginnend mit dem "umgekehrten Domainnamen" benannt, beispielsweise

- `de.pisoftware.bicklassen`
- `com.sun.specialpackage`

Threads

Threads sind quasi parallel ablaufende Steuerflüsse innerhalb eines Programms. Das „quasi“ gilt insbesondere für Rechner mit nur einem Prozessor. Echte Parallelität ist dann nicht möglich. In solchen Fällen simuliert ein Scheduler die Parallelität, indem er den verschiedenen parallelen Steuerflüssen in gewissen Abständen den Prozessor zuteilt und gegebenenfalls wieder entzieht. Es gibt dabei sehr unterschiedliche Scheduler – Varianten. Der JAVA – Scheduler lässt einen aktiven Thread so lange laufen, bis er entweder selber die Kontrolle abgibt, bis er bei einer Ein – oder Ausgabe blockiert, bis er auf die Freigabe einer Ressource wartet oder bis ein Thread mit höheren Priorität abgearbeitet werden soll.

Beispiel:

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

 public class UhrApplet extends Applet implements Runnable
 {
   Date  dateJetzt;
   Tread threadUhr;

   public void start()
   {
     if (threadUhr == null)
     {
        threadUhr = new Thread(this);
        threadUhr.start()
     }
   }

   public void stop()
   {
     if (threadUhr != null)
     {
        threadUhr.stop();
        threadUhr = null;
     }
   }

   void pause(int intervall)
   {
      Thread myThread;

      try {
         myThread.sleep(intervall);
      }
      catch (InterruptedException i){}
   }
```

```
   public void paint ( Graphics g )
   {
       g.drawString( dateJetzt.toString(),10,10);
   }
   public void run()
   {
     while(true)
     {
        dateJetzt=new Date();
        repaint();
        pause(2000);
     }
   }
}
```

Rückgabewerte

Methoden mit Rückgabewert (also nicht void) werden aufgerufen und „liefern“ bei Rückkehr einen Wert, d.h. „ der Aufruf wird ersetzt durch den Rückgabewert“. In der Methode selbst wird die Rückgabe durch den Befehl return <wert> ausgelöst.

Vergleichsoperatoren

== gleich	!= ungleich
< kleiner	<= kleiner - gleich
< größer	>= größer – gleich

Ein Vergleichsausdruck wird ausgewertet und durch einen Wahrheitswert ersetzt.

(3<4) ⟶ true

(4!=4) ⟶ false

Bemerkung:

Der JAVA-Typ `bool` kann *nicht* in eine Ganzzahl konvertiert werden und auch nicht durch Konvertierung (Cast) aus einer Ganzzahl ermittelt werden. Es ist *immer* eine explizite Zuweisung notwendig:

```
bool bErgebnis = ( 3 == 4 );  //weist bErgebnis den Wert false zu
```

oder

```
int nWahrheitBeiEins = ( 3 == 4 ? 1 : 0 ); //weist nWahrheitBeiEins
den Wert 0 zu
```

Insbesondere ist also das "Abtesten einer Zuweisung auf Null" nicht möglich

```
int n = 0;
bool bUngleichNull = ( n = 4 );     //liefert einen Compiler-Fehler
```

sondern muss explizit gemacht werden

```
bool bUngleichNull = ( (n = 4) != 0 );    //weist einen bool-Wert zu
```

Garbage Collection

Wenn eine JAVA Applikation keinen Handle mehr auf ein Objekt besitzt, vergeudet dieses nun nur noch Speicherplatz. Deshalb läuft bei einer JAVA – Applikation ein Garbage Collector, der automatisch solche Objekte wegräumt.
Ein explizites Anstoßen des Garbage Collectors ist zwar nicht nötig aber über den Aufruf `System.gc()` möglich.

Konstruktoren – Destruktoren

Jedes Objekt benötigt einen Konstruktor. Ein Objekt wird durch new erzeugt, das Aufräumen wird vom System erledigt, und zwar durch die sogenannte „Garbage Collection“. Es gibt in JAVA keine Destruktor.
Falls doch eine Methode für Aufräumarbeiten benötigt wird, kann die Methode `finalize()` den Destruktor ersetzen. `finalize()` wird während der Garbage Collection aufgerufen
Jede Klasse hat (mindestens) einen Konstruktor. Dieser Konstruktor „baut“ das angeforderte Objekt. Wenn (und nur wenn) kein Konstruktor explizit definiert wird, legt das System einen Standard-Konstruktor an. Dieser hat also die Signatur:

```
public <Klassenname>()
```

Konstruktoren haben keine Rückgabetyp (auch nicht void), sie heißen so wie ihre Klasse. Ein Konstruktor dient in der Regel zur Initialisierung der Attribute. Als erste Aktion muss ein Konstruktor den Konstruktor der Basisklasse aufrufen. Geschieht das nicht, so macht JAVA das automatisch. Sobald (mindestens) ein Konstruktor explizit definiert wird, erstellt JAVA keinen Konstruktor (auch keinen Standard-Konstruktor) mehr.

Beispiel

```
Class Hallo
{
      // statische Variable anzahl vom Typ int
      static int anzahl = 0;

      // Konstruktor
      public Hallo()
      {
            // die Variable wird bei jedem Aufruf der Methode um 1 erhöht
            anzahl++;
      }

      // der Deskruktor wird durch die Methode finalize() ersetzt
      public void finalize()
      {
            // bei jedem Aufruf der Methode wird von der Variablen 1
            // abegzogen
            anzahl--;
      }
}

// Die Garbage Collection wird von Hand aufgerufen
System.gc();
```

Bemerkung:

In JAVA darf eine Methode so heißen wie ihre Klasse, d.h.

```
class Verwirrend {
      void Verwirrend( int i ){};
}
```
ist gültiger Code! Diese Klasse hat allerdings nur den Standard-Ctor als Konstruktor.

Logische Operatoren

1) Negation, ist dann nur wahr (true), wenn der Wert falsch (false) ist.
   `!Wert`
2) Logisches UND, ist dann wahr, wenn Wert1 und Wert2 wahr sind.
   `Wert1 && Wert2`
3) Logisches ODER, ist genau dann waht, wenn Wert1 oder Wert2 (oder Beide) wahr sind
   `Wert1 || Wert2`
4) Logisches XOR (Exclusive OR), ist genau dann wahr, wenn Wert1 oder Wert2 (aber nicht Beide) wahr sind
   `Wert1 ^^ Wert2`

&&	true	false
true	true	false
false	false	false

||	true	false
true	true	true
false	true	false

^^	true	false
true	false	true
false	true	false

!	true	false
	false	true

Kurzschreibweisen

```
int i;
i++;        bedeutet: i = i + 1;
i--;        bedeutet: i = i - 1;
i+=3;       bedeutet: i = i + 3;
i-=32;      bedeutet: i = i – 32;
i*=10;      bedeutet: i = i * 10;
i/=4;       bedeutet: i = i / 4;
```

Das Plus Zeichen

Das Plus Zeichen kann Zahlen addieren, z.B. 3+4; das Ergebnis wäre hier 7.
Es kann auch Strings verketten, z.B. “3“+“4“; das Ergebnis wäre hier “34“.
Es wandelt bei der Verkettung von Zahlen mit Strings die Zahlen ebenfalls in Strings um, z.B. “3“ + 4; das Ergebnis wäre “34“.

Beispiel

```
public class Rechnen
{

      public static void main( String[] args )
      {

            //verkette zwei Strings
            System.out.println("Das ist ein "+"zusammengesetzter"+" Satz!");

            //verrechne zwei Zahlen
            System.out.println( 17 + 4 );

            //hänge eine Zahl an einen String
            System.out.println( "17" + 4 );

            //hänge einen String an eine Zahl
            System.out.println( 17 + "4" );

            //mache meine FALSCHE Ausgabe
            System.out.println(3 + 4 + "ist NICHT das Gleiche wie" + 3 + 4);

            //mache die RICHTIGE Ausgabe
            System.out.println(3 + 4 + "ist DOCH das Gleiche wie"+ (3 + 4));

      }

}
```

Schlüsselwort super

Das Schlüsselwort super bietet Zugriff auf Funktionen der Elternklasse.

Beispiel

Keine eigene Funktion definiert:

```
setBackground()
```
→ ruft die Funktion der Elternklasse auf

Eigene Funktion definiert:

```
setBackgroung()
```
→ ruft die eigene Funktion auf

```
super.setBackground()
```
→ ruft die Funktion der Elternklasse auf

Bemerkung

In JAVA kann somit nur auf die geerbte Definition einer Methode zugegriffen werden, nicht aber auf *deren* eventuelle Veorfahren, wie dies in C++ durch Angabe des Klassennamens möglich ist.

Schlüsselwort this

Mit this wird auf die eigenen Elemente der Klasse zugegriffen.

```
Freund( String name, int seit )
{
      this.name = name;
      this.seit = seit;
}
Freund()
{
      this( "Noname" , 0 );   //ruft den eigenen Wert-Ctor auf
}
```

Der Lebenszyklus eines Applets

Eine Instanz der entsprechenden Klasse wird erzeugt. Das Applet wird initialisiert, gestartet und wieder gestoppt. Ein Applet hat eine Option, sich selbst zu stoppen, z.B. wenn Sie die HTML-Seite verlassen. Das Applet kann wieder starten, z.B. wenn sie wieder in die HTML-Seite zurückzukehren. Das Applet räumt endgültig auf, z.B. gibt Ressourcen wieder frei, bevor es endgültig entladen wird.

Kontrollstrukturen

if-Anweisung:

Bedingte Anweisungen sind Anweisungen, die in Abhängigkeit einer Bedingung ausgeführt werden oder nicht. JAVA stellt für die Programmierung bedingter Anweisungen die if-Anweisung zur Verfügung.

Die allgemeine Syntax der if-Anweisung:

```
if(<Bedingung>)
      <Anweisung>;
```

Die allgemeine Syntax der if-else Anweisung:

```
if(<Bedingung>)
      <Anweisung>;
else
      <Anweisung>;
```

Jede Anweisung kann wiederum ein ganzer „Block“ sein, d.h. zwischen { und } stehen, und gilt dann als eine Anweisung.

Beispiel

```
import java.applet.Applet;
import java.awt.Graphics;
//
//
// HalloApplet
//
//
public class HalloApplet extends Applet
{
      //hier zähle ich die paint()-Aufrufe
      private int anzahlVerbleibendeAufrufe;

      //hier wird der Zähler initialisiert
      public void init()
      {
            anzahlVerbleibendeAufrufe = 15;
      }
```

```
	public void paint( Graphics g )
	{
		//wenn den Zähler noch über null steht
		if ( anzahlVerbleibendeAufrufe > 0 )
		{
			//mache ihn kleiner
			anzahlVerbleibendeAufrufe--;
			//und tätige die Ausgabe
			g.drawString(
				"Hiernach noch " +
				anzahlVerbleibendeAufrufe +
				" paint()-Aufrufe.",
				10, 20 );
		}
	}
}
```

switch-Anweisung

Eine switch-Anweiung kommt immer dann zum Einsatz, wenn man anhand eines Ausdrucks verschiedene Anweisungen ausführen möchte, und dabei unterschiedlich viele Fälle zu berücksichtigen sind.

Die allgemeine Syntax der switch-Anweisung:

```
switch( <Ausdruck> )
{
	case<Literal5>:
	case<Literal3>:
		<Anweisung>;
	case<Literal2>:
		default:
			<Anweisung>;
}
```

Der Ausdruck muss eine Ganzzahl sein.

Beispiel

```
//Definition der öffentlichen Klasse
public class switch_Anweisung
{
	// Hauptfunktion main()
	public static void main (String [] args)
	{
		//Deklaration der Variablen
		int zahl;

		//Fetslegung des Bereichs für den die Anweisung gilt
		for(zahl=1; zahl<=9; zahl++)
		{
			//switch-Anweisung
			//Ausdruck
			switch(zahl)
			{
			//Wert der berücksichtigt werden soll
			case 8:
			case 4:
				//Anweisung, die dann ausgeführt werden soll
				System.out.println("Die Zahl " +zahl+ " ist durch
				vier teilbar !!");

			//Wert der berücksichtigt werden soll
			case 6:
			case 2:
				//Anweisung, die dann ausgeführt werden soll
				System.out.println("Die Zahl " +zahl+ " ist durch
				zwei teilbar !!");

			// break beendet die Anweisung/Schleife.
			// Das Programm wird dann nach der
			// Schleife/Anweisung weiter ausgeführt.

			break;

			// Die default-Anweisung wird für die nicht zu
			// brücksichtigten Werte der switch-Anweisung
			// ausgeführt

			default:
				// Die Anweisung die aufgrund von default
				// ausgeführt wird
				System.out.println("Die Zahl " +zahl+ "  ist nicht
				durch vier oder zwei teilbar !!");
			}
		}
	}
}
```

for-Schleife

Die for-Schleife ist eine Zählschleife, die eine Folge von Anweisungen eine feste Anzahl an Malen wiederholt.

Die allgemeine Syntax der for-Schleife:

```
for( <Ausdruck1>; <Bedingung>; <Ausdruck2> )
      <Anweisung>;
```

Ausdruck1 = Initialisierung
Bedingung = Abbruchbedingung
Ausdruck2 = Veränderung der Schleifenvariablen

Beispiel

```
//Definition der Klasse
public class Bibliothek {

      //Definition der main-Methode => Application
      public static void main( String[] args ) {

            //meine Variablen
            int monat;
            int tag;
            //diese Schleife zählt die Monate
            for ( monat = 1; monat <= 12; monat++ )
            {
                  //und für jeden davon zähle die Tage
                  for ( tag = 7; tag <= 28; tag = tag + 7 )
                  {
                        //das ist EIN Rückgabetermin
                        System.out.print( tag + "." + monat + ".2000 " );
                  } //Ende der Tagesschleife
                  //jetzt mache eine neue Zeile
                  System.out.println( "" );
            } //Ende der Monatsschleife
      } //Ende der main()
}// Ende der Klasse
```

while-Schleife

Eine while-Schleife wird verwendet, wenn eine Anweisung mehrfach ausgeführt werden muss. Die Anweisung wird solange wiederholt, bis die Bedingung den Wert false einnimmt. Eine while-Schleife prüft erst die Bedingung, und führt dann gegebenenfalls die Anweisung aus.

Die allgemeine Syntax der while-Schleife:

```
while(<Bedingung>)
      <Anweisung>;
```

Beispiel

```
//Definition der Klasse
public class Millionenspiel
{

      //Definition der main-Methode => Application
      public static void main( String[] args )
      {

            //speicher die aktuelle Potenz
            int potenz;
            potenz =1;

            //solange wir UNTER einer Million sind
            while ( potenz < 1000000 )
            {
                  //erhöhe die Potenz (Kurzschreibweise!)
                  potenz *= 2;
            }

            //jetzt ist potenz >= 1000000

            //muss also einen "Schritt" zurück (Kurzschreibweise!)
            potenz /= 2;

            //and the winner is...
            System.out.println(
                  "Die groesste Zweierpotenz kleiner als 1 Mio ist " +
                  potenz + ".");
      }

}
```

do-while Schleife

Die Anweisung wird mindestens einmal ausgeführt.

Die allgemeine Syntax der do-while Schleife:

```
do<Anweisung>
while(<Bedingung>);
```

Abbruchbefehl

Der Befehl break beendet eine Schleife. Das Programm wird dann nach der Schleife weiter ausgeführt.

Ausnahmebehandlung

JAVA bietet ein modernes und einfaches Konzept zum Abfangen von Fehlern während der Programmausführung an. Immer wenn ein Laufzeitfehler auftritt, wird ein Objekt der Klasse Throwable erzeugt. Man kann auch bildlich sagen, dass eine Ausnahme geworfen wird.
Der Fehler kann nun innerhalb des Programms aufgefangen werden, und zwar innerhalb einer catch – Anweisung.

Ausnahmetypen

Der Programmteil, in dem eventuelle Ausnahmen auftreten können, wird von einer `try` – Anweisung eingeschlossen. Anschließend wird für jeden Fehlertyp ein eigener catch – Abschnitt definiert.
Nach Beenden des kritischen Abschnittes kann ein `finally` – Abschnitt definiert werden. Dieser dient dazu, eventuelle Aufräumarbeiten vorzunehmen. Die Anweisungen nach `finally` werden in jedem Fall ausgeführt, unabhängig davon, ob ein Fehler aufgetreten und abgefangen worden ist. Die wichtigsten Ausnahmetypen sind:

`IOException`	Ein – Ausgabefehler
`FileNotFoundException`	Datei nicht gefunden
`BOFException`	Dateiende überschritten
`ArrayIndexOutOfBoundsException`	Arraygrenzen überschritten
`ArithmeticException`	arithmetischer Fehler
`OutOfMemoryError`	zu wenig Speicher
`MalformedURLException`	falsche Internetadresse
`Exception`	Ausnahme, nicht näher gekennzeichnet
`Error`	schwerer Laufzeitfehler, nicht näher gekennzeichnet

Der kritische Programmabschnitt, in dem Ausnahmen und Fehler abgefangen werden sollen, befindet sich im ersten Block hinter dem `try`.
Jedes `try` muss mit mindetstens einem `catch` oder `finally` kombiniert werden; es ist aber auch möglich, alle drei Elemente zusammen zu verwenden.
Es gibt in JAVA Methoden die immer von einem `try – catch – finally` – Block umschlossen sein müssen. Dazu gehören zum Beispiel alle Ein – und Ausgaben von Dateien. Wenn eine Dateioperation ohne Ausnahmebehandlung programmiert wird, gibt der Compiler eine Fehlermeldung aus.

throw und throws

Eine Ausnahme kann, wenn erforderlich auch durch die `throw` – Anweisung ausgelöst werden. Es ist dann anzugeben, welche Ausnahme erzeugt werden soll; es kann dabei auf die standardmäßige definierten Ausnahmen zurückgegriffen werden oder auch neue Ausnahmen erzeugt werden.
Eine Methode, die Ausnamhmen erzeugt, muss unmittelbar bei ihrer Deklaration angeben, um welche Ausnahme es sich handelt. Dazu dient das Schlüsselwort `throws`.
Mehrere Ausnahmetypen, die von dieser Methode erzeugt werden, können durch Kommata getrennt hinter `throws` aufgelistet werden.
Ausnahmen, die das Laufzeitsystem auslöst oder schwere Fehler können an beliebiger Stelle im Programm auftreten. Daher müssen die beiden Ausnahmetypen `RunTimeException` und `Error` nicht explizit hinter `throws` angeführt werden.

Beispiel

```
public void meine Funktion() throws IllegalArgumentException
{
      // meine Funktion tut was
      if(<Bedingung für Ausnahme>)
      throw new IllegalArgumentException( “Textmeldung“);
      // wenn kein Fehler auftritt, mache weiter
}

try
{
      meineFunktion();
}

catch( IllegalArgumentException e )
{
      String strMeldung = e.getMessage();
      // z.B. mit
      g.showStatus( strMeldung );
      // ausgeben
}
```

Nachrichten

Ein Objekt muss ein anderes Objekt veranlassen, seine Dienste auszuführen. Dies geschieht durch das Versenden von Nachrichten. Das Senden einer Nachricht entspricht mindestens einen Methodenaufruf. Es ist notwendig, dass der SENDER den EMPFÄNGER sieht, aber nicht anders herum. Es ist notwendig, dass eine Nachrichtenverbindung (Link) vorhanden ist, entweder eine Assoziation, oder eine Aggregation, diese müssen gerichtet sein.

Variablen

Speicherplatz/Behälter für (einen) Wert. Vor der Benutzung muss die Variable deklariert werden. Eine Variable hat immer einen bestimmten Datentyp. Einfache Datentypen sind: `bool`, `short`, `int`, `long`, `float`, `double`, `char`.
Vor dem ersten Lesen muss einer Variablen ein Wert zugewiesen werden.

Arrays/Felder

Ein Array ist eine Zusammenfassung mehrerer Variablen gleichen Datentyps, die unter einen Namen angesprochen werden können. Die Anzahl der Elemente (Variablen) und der Datentyp der Elemente werden beim Erzeugen des Arrays festgelegt und können nachträglich nicht mehr geändert werden. Die Größe eines Arrays ergibt sich aus der Anzahl der Elemente im Array. Ein Array ist eine „Liste“ von Einträgen, die Liste hat eine feste Anzahl von Einträgen. Ein Array ist damit EINE Variable mit mehreren Komponenten. Arrays sind Objekte, d.h. sie müssen mit `new` erzeugt werden.

Arrays = Referenzdatentypen.

Beispiel

```
String namensFeld[5];
```

Legt ein Feld mit FÜNF Elementen an; diese werden „nummeriert“ von **0** bis **4** !!!

`NamensFeld[2]` ist der dritte Eintrag.

Index

In JAVA kann die Zahl der Elemente eines Arrays mit Hilfe des nur lesbaren "Pseudo-Attributes" length abgefragt werden

Beispiel

args.length liefert die Anzahl der Kommandozeilenparameter

Die Class Klasse:

Die Klasse `Class` ist direkt von `Object` abgeleitet. Ein Objekt der Klasse `Class` beinhaltet Informationen über eine Klasse der JAVA- Applikation. Zu jeder Klasse gibt es genau ein zugehöriges `Class` Objekt.

- `String getName()`
  liefert den Namen der Klasse
- `Class getSuperclass()`
  liefert ein Class Objekt mit den Informationen über die Oberklasse zurück
- `Object newInstance() throws ...`
  liefert ein neues Objekt der Klasse zurück. (Parameterloser Konstruktor)
- `static Class forName(String) throws ClassNotFoundException`
  liefert zu einem Klassennamen das zugehörige Class Objekt zurück.

Beispiel

Ein Web-Browser oder Applet-Viewer enthält Code, der sinngemäß das HTML-Tag

```
<APPLET code = MeinApplet>
```

wie folgt verwendet (ohne Fehlerkontrolle mit try/catch):

```
Class appletKlasse = Class.forName( "MeinApplet" );
Applet appletObjekt = appletKlasse.newInstance();
appletObjekt.init();
appletObjekt.start();
```

Die Thread Klasse

Methoden:

1) `public void start()`
   meldet den Thread an und ruft `run()` auf.
2) `public void run()`
   tut nichts, muss überschrieben werden.
3) `public void stop()`
   beendet den Thread.

Ein mit `stop()` beendeter Thread läßt sich NICHT wieder starten. Zum Anhalten und Weiterlaufen gibt es stattdessen die Methoden `suspend()` und `resume()`
Ein Thread ist ein separater Kontrollfluß unabhängig vom Haupt-Thread, dem Applet. Das Applet ist grundsätzlich ein separater Thread.

Bemerkung

Wegen des oben angegebenen Codes muss ein eigenes Applet immer von der Klasse `java.applet.Applet` abgeleitet sein, kann also nicht gleichzeitig von `java.lang.Thread` abgeleitet sein.
Daher kennt JAVA zusätzlich die Schnittstelle `java.lang.Runnable`, die nur aus der parameterlosen Methode `public abstract void run()` besteht. Zusammen mit der Tatsache, dass ein Thread einen weiteren Konstruktor besitzt, der ein `Runnable`-Objekt übernimmt, ist es möglich, einem Thread ein Applet zu übergeben, dessen `run()` -Methode dann als `run()` -Methode des Threads genutzt wird.

Definition einer Klasse

```
<spezifizierer>class<Klassenname>extends<Basisklasse>
(implenents<Schnittstellenklasse>)
```

Beispiel

```
public class Hallo extends Applet
{
      // hier steht die Implementierung der Klasse
}
```

Rezept für die Erstellung einer Klasse

1) NeueKlasse extends AlteKlasse
2) Hat die Klasse neue Attribute?
   Wenn ja: definiere sie.
3) Hat die Klasse neue Methoden()?
   Wenn ja: definiere sie.
4) Hat die Klasse irgendwo ein anderes Verhalten?
   Wenn ja: überschreibe die geerbte Methode

Klassifizierung

1) Objekte beschreiben
2) Gemeinsamkeiten finden und benennen
3) Klasse definieren
4) Methoden, Attribute rückübertragen bzw. streichen
5) Klassenzugehörigkeit eintragen

Abstrakte Klassen

In manchen Programmentwürfen tauchen Klassen auf, die Methoden vorsehen, die aber (mangels Information) nicht implementiert werden wollen oder können.
Wenn die Signatur einer solchen Methode bekannt ist kann man diese Methode in der Oberklasse aufnehmen, ohne jedoch diese Methode zu implementieren.
Eine solche Klasse wird mit dem Schlüsselwort `abstract` gekennzeichnet und die nicht implementierten Methoden werden ebenfalls mit dem Schlüsselwort `abstract` gekennzeichnet.
Eine solche Klasse ist nicht direkt instanziierbar, sondern der Zweck dieser Klasse ist der , zum Ableiten zur Verfügung zu stehen.
Die abgeleiteten Klassen sind erst dann instanziierbar, wenn alle abstrakten Methoden implementiert sind.
Aufgrund der Polymorphie kann natürlich jede Instanz einer abgeleiteten Klasse wiederum als eine Instanz der abstrakten Klasse aufgefasst werden.

Container und Components

- class Container extends Component
- Component hat viele Eigenschaften und Methoden, die bisher für Applets benutzt wurden.
- Container ruft paint() für alle seine Komponenten auf.
- Components empfangen unter anderem Maus – Nachrichten, `mouseDown()` etc.

Jeder Container hat einen Layoutmanager der mit setLayout() gesetzt wird. z.B.

- `FlowLayout` (Standard)
- `GridLayout` (Raster)
- `BorderLayout` (Rand)
- `CardLayout` (Seiten)

Komponenten werden ihrem Container mit `add()` hinzugefügt.

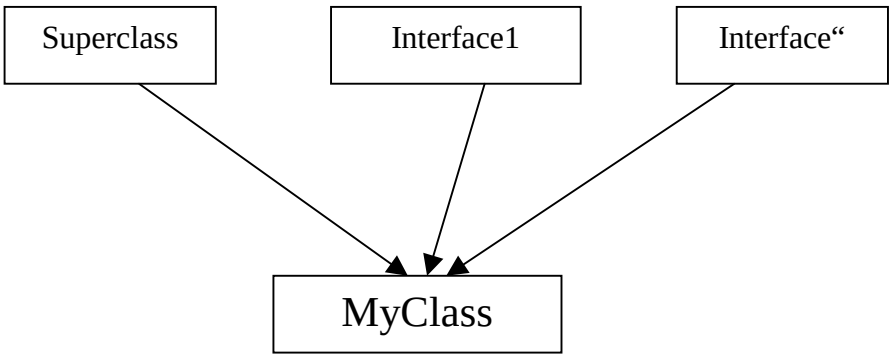
Die Klasse Component hat folgende Methoden:

- `void disable()`
- `void enable()`
- `void enable( boolean einschalten )`

Schnittstellen

Jede Klasse in JAVA besitzt genau eine Oberklasse. Manchmal wäre es aber wünschenswert, wenn eine Klasse mehrere Oberklassen besitzen würde.
Um Mehrfachvererbung, die es in JAVA eigentlich nicht gibt, wenigstens teilweise zur Verfügung zu haben, gibt es Interfaces (Schnittstellen). Eine Schnittstelle ist im Prinzip eine abstrakte Klasse, die nur abstrakte Methoden besitzt. Eine Schnittstelle wird über das Schlüsselwort `interface` definiert. Eine Klasse erbt Schnittstellen wie folgt:

```
Class <MyClass> extends<Superclass> implements Interface1, Interface2
{
      ...
}
```



Bis auf die spezielle Syntax kann man ein Interface als abstrakte Klasse auffassen. Auch gibt es zu jedem Interface ein Class Objekt. Die Class Methode `getInterfaces()` liefert ein Feld mit allen Klassenobjekten für Schnittstellen zurück, die diese Klasse unterstützt. `Class.isInterface()` gibt zurück, ob es sich um eine Shnittstelle handelt.

- `public Class[] getInterface();`
- `public boolean isInterface();`

Sicherheit

Ein Applet ist naturgemäß ein Programm, das von einem Rechner kommt (Server) und auf einem anderen Rechner abläuft (Client). Dies stellt in vielerlei Hinsicht ein großes Sicherheitsrisiko dar. Deshalb unterliegen Applets strengen Sicherheits-Restriktionen:

- Ein Applet kann zwar Nachrichten an andere Applets schicken, Sicherheitsrestriktionen schränken das ein:
  - ➔ Nur wenn das andere Applet auf der selben Seite, im selben Frame und im selben Browserfenster ist
  - ➔ Und die Applets vom selben Server kommen
- Applets dürfen nur mit dem Rechner kommunizieren, von dem sie geladen wurden. Applets müssen also herausfinden woher sie kommen.
- Danach kann eine Verbindung nach dort aufgebaut werden. (Mit einer der JAVA-Netzwerk-Klassen)
- Applets können keine Netzwerkverbindung aufbauen außer zum Host, von dem sie geladen wurden.
- Wenn der Host, auf dem das Applet läuft, dem Quell-Host die entsprechenden Rechte einräumt, kann von da auf das lokale Dateisystem geschrieben werden
- Applets können keine Bibliotheken laden und keine C-Funktionen aufrufen.
- Applets können keine Dateien auf dem lokalen Host lesen oder schreiben.
- Applets können keine anderen Programme auf dem lokalen Host starten
- Applets können nicht alle Systemressourcen lesen.
- Applets dürfen nicht auf dem Drucker ausgeben.
- Fenster, die von Applets aufgemacht werden, sind deutlich als „unsecure“ markiert, um das Vortäuschen von login-Fenstern o.ä. zu verhindern.
- Applets mit Sicherheitszertifikat können mehr Rechte eingeräumt bekommen.

Architektur eines Window Systems:

Betriebssysteme stellen grafische Benutzeroberflächen oft mit einem Fenstersystem bereit. Auf einem Rootwindow liegen weitere Fenster, in denen weitere Fenster enthalten sind. Fenster besitzen eine Eltern – Kind Beziehung. Eine Applikation, die Fenster erzeugt, bekommt vom Betriebssystem für dieses Fenster Nachrichten zugestellt, sogenannte Events:

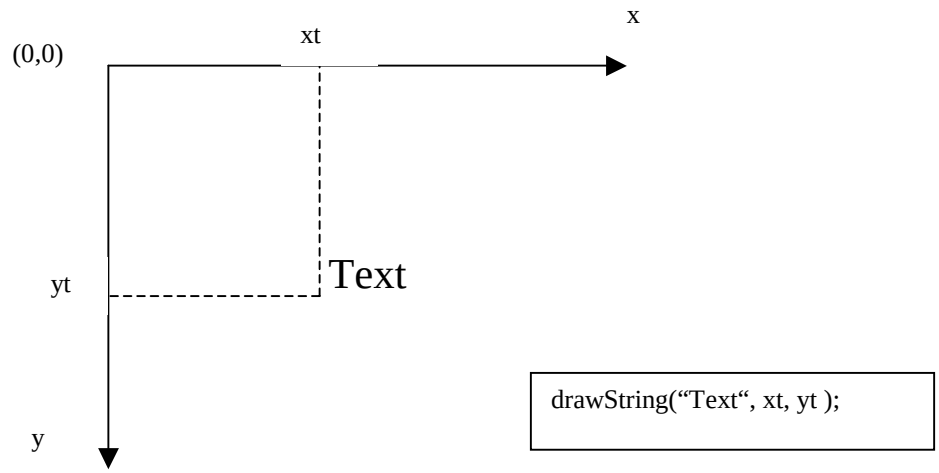
- java.awt.event.MouseEvent:
  `MOUSE_CLICKED, MOUSE_DRAGGED, MOUSE_, MOUSE_EXITED, MOUSE_MOUVED, MOUSE PRESSED, MOUSE_RELEASED`
- java.awt.event.KeyEvent:
  `KEY_PRESSED, KEY_RELEASED, KEY_TYPED`
- jave.awt.event.WindowEvent:
  `WINDOW_ACTIVATED, WINDOW_CLOSED, WINDOW_CLOSING, WINDOW_DEACTIVATED, WINDOW_DEICONIFIED, WINDOW_ICONIFIED, WINDOW_LAST, WINDOW_OPENED`
- java.awt.event.FocusEvent:
  `FOCUS_GAINED, FOCUS_LOST`

- java.awt.event.ActionEvent:
  `ACTION_PERFORMED`
- java.awt.event.ItemEvent:
  `DESELECTED, ITEM_STATE_CHANGED, SELECTED`
- java.awt.event.AdjustmentEvent:
  `ADJUSTMENT_VALUE_CHANGED, BLOCK_DECREMENT, BLOCK_INCREMENT, TRACK, UINT_DECREMENT, UINT_INCREMENT`
- java.awt.event.TextEvent:
  `TEXT_VALUE_CHANGED`
- java.awt.event.ComponentEvent:
  `COMPONENT_HIDDEN, COMPONENT_, COMPONENT_REZIED, COMPONENT_SHOWN`
- java.awt.event.ContainerEvent
  `COMPONENET_ADDED, COMPONENT_REMOVED`

Die paint Methode

Wenn ein Fenster gezeichnet wird, geht die Zeichnung verloren, wenn sie durch ein anderes Fenster überdeckt, oder wenn die Anwendung ikonisiert wurde. Wenn der Inhalt eines Frames oder ein Teil davon verloren gegangen ist, wird automatisch `update(Graphics)` bzw. `paint(Graphics)` aufgerufen.
`update()` stellt den Hintergrund des Fensters her und ruft dann `paint()` auf. Die `paint()` Methode muss vom Software Entwickler selbst programmiert werden. Um ein Fenster neu zu zeichnen gibt es die Methode `repaint()`. Diese ruft einfach die Methode `update()` auf. Es gibt keinen Weg in JAVA, die Fensterinhalte persistent zu machen. Eine Instanz der Klasse Graphics entspricht einem Device Context bzw. einem Graphis Context. Ein Graphics Objekt ist immer an ein bestimmtes JAVA AWT Objekt gekoppelt. Alle Zeichenfunktionen (auch Texte zu schreiben ist Zeichnen) sind Methoden der Graphics Klasse.

Koordinaten



Methoden der Graphics Klasse

```
public abstract void drawLine( int x1, int y1, int x2, int y2 );
public abstract void drawString( String str, int x, int y );
public abstract void setFont( Font font );
public abstract FontMetrics getFontMetrics( Font f);
```

Methoden der Klassen Font und FontMetrics

```
public Font( String name, int style, int size );
      // style kann BOLD, ITALIC, PLAIN sein
      // name sind die Schriftarten wie „TimesRoman“, „Courier“, ...
protected FontMetrics( Font font );
      public int getLeading();
      public int getMaxAsccent();
      public int getMaxDescent();
      public int stringWidth( String str );
```

Die getSize() Mothode von Component:

```
public Dimension getSize()
public class java.awt.Demension
{
      public int height;
      public int width;
      ...
}
```

Farben und Zeichnen

Farben sind in JAVA Objekte der Klasse Color. Ein Konstruktor von Color akzeptiert eine RGB Kombination, nämlich ein Tripel von Zahlen im Bereich von 0 – 255. Für die Systemfarben gibt es Konstanten `Color.black, Color.red, ... .`

```
public Color( int red, int green, int blue );
```

Beispiel

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;
//
// ErstesApplet
//
public class ErstesApplet extends Applet
{
      public void paint( Graphics g )
      {
            g.setColor( Color.red );
            g.drawString( "Ach so!", 20, 40 );

            g.setColor( Color.green );
            g.drawLine( 0, 270, 400, 270 );
            g.drawRect( 70, 50, 50, 190 );
            g.drawRect( 50, 80, 40, 170 );
            g.drawRect( 105, 90, 40, 175 );

            g.setColor( Color.blue );
            g.drawLine( 0, 200, 50, 200 );
            g.drawLine( 145, 200, 400, 200 );

            g.drawLine( 270, 270, 310, 200 );
            g.drawLine( 310, 200, 290, 270 );

            Color c = new Color( 145, 116, 78 );
            g.setColor( c );
            g.drawRect( 250, 230, 20, 20 );
            g.drawLine( 245, 235, 260, 220 );
            g.drawLine( 260, 220, 275, 235 );

            g.setColor( new Color( 45, 16, 178 ) );
            g.drawOval( 320, 40, 40, 40 );
            g.drawLine( 310, 60, 280, 60 );
            g.drawLine( 370, 60, 400, 60 );
            g.drawLine( 340, 30, 340, 0 );
            g.drawLine( 340, 90, 340, 120 );
      }
}
```

Methoden der Klasse Componet

```
public void setBackground( Color );
public Color getBackground( Color );
public Graphics getGraphics();
```

Methoden der Klasse Graphics

```
public void dspose();
public void setColor();
public Color getColor();
public void setXORMode( Color );
public void setPaintMode();
public abstract void drawArc( int x, int y, int width, int height,
                              int startAngle, int arcAngle);
public abstract void fillArc(int x, int y, int width, int height,
                             int startAngle, int arcAngle);
public abstract void drawOval( int x, int y, int width, int height );
public abstract void fillOval(int x, int y, int width, int height );
```

Beispiel

```
import java.applet.Applet;
import java.awt. Graphics;
import java.awt. Color ;
//
// Ein Beispiel für das Zeichnen
//
//
public class Zeichnen extends Applet
{
      public void init ()
      {
            setBackground ( Color.black );
            setForeground ( Color.blue );
      }

      public void paint (Graphics g )
      {
            // Farbmethoden des Applets

            g.drawLine ( 0 ,0 , 400 , 400);
            g.setColor (Color.red);
            g.fillRect ( 10 , 100 , 110 , 200 );
            g.setColor (Color.blue);
            g.fillRect ( 100 ,200 , 110 , 100 );
            Color b ;
            b = new Color ( 150 , 50, 90 );
            g.setColor (b);
            g.drawRect ( 210 ,250 , 90 , 50);
            g.setColor (Color.yellow);
```

```
            g.drawOval ( 300 , 80 , 70 , 70 );

            Color neueFarbe;
            neueFarbe = new Color(0, 255, 0);
            g.setColor(neueFarbe);
            g.fillRect(350,200,48,100);
            g.fillOval ( 300 , 80 , 50 , 50 );

            // Warteschleife
            try
            {
                  Thread.sleep(3000);
            }
            catch(Exception e) {}
      }
}
```

Graphicfiles und Bilder

Grafiken (z.B. GIF, JPEG, ...) können in JAVA benutzt werden, um sie in Fenstern darzustellen.
Das Laden geschieht über ein Toolkit – Objekt. Ein Toolkit – Objekt kapselt die Betriebssystem – Plattformabhängigkeiten des Fenstersystems. Toolkit ist eine abstrakte Klasse, die durch plattformspezifische Ableitungen erst instanziierbar wird. Für ein das AWT benutzende Programm wird automatisch ein Toolkit – Objekt erzeugt.
Durch `getImage()` wird das Bild noch nicht geladen, sondern nur ein Image – Objekt erzeugt.
Das Laden des Bildes wird durch einen `drawImage()` – Aufruf initiiert, oder durch einen `prepareImage()` – Aufruf.
Das Laden geschieht asynchron. Während der `drawImage()` – Aufruf sofort zurückkehrt (und eventuell nur ein leeres Rechteck zeichnet) läd ein Thread das Bild.
Bei jedem Paket geladener Bildinformation – oder auch wenn Fehler auftreten – wird der beim `drawImage()` bzw. `prepareImage()` angegebene `ImageObserver` durch den Aufruf von `imageUpdate()` benachrichtigt.

Literaturverzeichnis

[1] David Flanagan, JAVA IN A NUTSHELL, O´Reilly, 1999

[2] Mark Grand, JAVA Language Reference, O´Reilly, 1997

[3] Ralf Kühnel, Die Java-Fibel, ADDISON-WESLEY, Bonn, 1996

[4] Laura Lemay, Rogers Cadenhead, SAMS Teach Yourself Java in 21 Days, SAMS, 1998

[5] O´Niel Annand Valetin Som, Java- plattformunabhängig programmieren, bhv, 1997

[6] John Zukowski, JAVA AWT Reference, O´Reilly, 1997

Stichwortverzeichnis