Programmieren mit

Java

Lars Knipping

April 2003

Java nach Sun

Java ist eine einfache, objektorientierte, verteilte, interpretierte, robuste, sichere, architekturneutral portable, hochleistungsfähige, multithread-fähige und dynamische Sprache.

Sun Microsystems Inc. über Java (frei übersetzt)

Ausgewählte Eigenschaften

- *objektorientiert (OO)*
- *einfach*
  kleiner Sprachumfang, umfangreiche Bibliotheken
- *interpretiert, architekturneutral portabel*
  „*write once, run anywhere*“
  - Compiler erzeugt Bytecode
  - Bytecode wird von Virtueller Maschine (JVM) interpretiert
- *robust und sicher*
  - Ausnahmebehandlung (*exceptions*)
  - Keine Zeigerarithmetik
  - Autom.Speicherverwaltung (*garbage collection*)
  - Sog. Sandkastenprinzip für Applets
  - ...

Werkzeuge

- JRE (Java Runtime Environment)
  Java Bytecode Interpreter

- JDK (Java Development Kit),
  Java SDK (Software Development Kit)

  – *java*, *jre*: Java Bytecode Interpreter
  – *javac*: Java Compiler (Übersetzer)
  – *javadoc*: Generator für API Dokumentationen
  – *appletviewer*
  – ...

Gratis ...

- JDK/SDK von Sun
- JDK von IBM
- Kaffe von TransVirtual (GPL)
- Jikes von IBM, Compiler (Open Source)
- guavac, Compiler (GPL)
- Japhar, Java Bytecode Interpreter (LGPL)

IDEs

- Borland JBuilder
- Forte for Java von NetBeans.com/Sun (Open Source)

  `http://www.sun.com/forte/ffj/`
  `http://www.netbeans.org`
- IBM Visual Age for Java
- BlueJ `http://www.bluej.org`
- Eclipse `http://www.eclipse.org`
- ...

Hello, World!

```
public class HelloWorld {

  public static void main(String[] argv) {

    System.out.println("Hello, world!");

  }
}
```

Hello?

- `public class HelloWorld { ... }`

  Definiert die *Klasse* mit Namen *HelloWorld*.

  In Klammern steht, was zu der Klasse gehört.

- `public static void main(String[] argv) { ... }`

  Einsprungspunkt beim Ausführen der Klasse.

  Im Innern die auszuführenden *Anweisungen* (*statements*).

- `System.out.println("Hello, world!");`

  Eine Anweisung, die *Hello, world!* schreibt.

  Anweisungen werden mit Semikolon abgeschlossen.

No News is Good News

- Mit Texteditor **HelloWorld.java** erstellen

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

- Übersetzen mit **javac HelloWorld.java**

no news is good news

- Ausführen

```
> java HelloWorld
Hello, world!
>
```

Anweisungssequenz

```
public class HelloGoodbye {

  public static void main(String[] argv) {
    System.out.println("You say yes,");
    System.out.println("  I say no");
    System.out.println("You say stop,");
    System.out.println("  I say go, go, go");
    System.out.println("Oh no");
  }
}
```

It was hard to write, so it should be hard to read

- `//` Kommentar bis Zeilenende
- `/*` und `*/` klammern Kommentar, nicht schachtelbar
- `/**` und `*/` wie oben, Inhalt für *javadoc*

```
/**
 * Gibt Anfang von <I>Hello Goodbye</I> aus.
 * @author Lars (Programm, nicht Lied)
 */
public class HelloGoodbye { //heisst wie Lied
  /** Der Einsprungspunkt ins Programm. */
  public static void main(String[] argv) {
    System.out.println("You say yes, I say no");
    System.out.println("You say stop,");
    System.out.println("  I say go, go, go");
    System.out.println("Oh no"); /* Ende... */
  }
}
```

Variablen

Variablen (*variables*) bezeichnen Speicherplätze („Behälter“) für Werte eines Datentyps.

Variablen werden vor Verwendung deklariert, d.h. der Name (Bezeichner, *identifier*) und Datentyp werden festgelegt.

```
// Integer (ganze Zahl) mit Namen x:

int x; // Deklarationen sind Anweisungen
```

Zuweisungen

Variablenwerte können mit der *Zuweisungsanweisung* gesetzt werden.

```
int x;
x = 5;
int y;
y = x;
x = 7;
System.out.println(x); // Ausgabe: 7
System.out.println(y); // Ausgabe: 5
```

Mehr Deklarationen

Bei der Deklaration kann die auch Variable gleich initialisiert werden:

```
int x = 42;
```

Mehrere Variablen gleichen Typs können mit einer Anweisung deklariert werden. Dabei werden die einzelnen Variablen durch Komma getrennt:

```
int i = 23, j = 5, k, l, m = -7;
```

Dreieckstausch

Vertauschen zweier Variablen mittels einer Hilfsvariable:

```
// i, j zu tauschende Integervariablen
int h = i;
i = j;
j = h;
```

Datentypen

- einfache (in Java auch *Primitive Datentypen*)
  - Wert unteilbar
  - kann nur als ganzes manipuliert werden
- zusammengesetzte (in Java immer *Objekte*)
  - mehrere, einzeln manipulierbare Teilwerte
  - rekursives Bauprinzip

Primitive Datentypen

Typ	Bits	Werte
Ganze Zahlen		
`byte`	8*	-128 bis 127
`short`	16*	-32768 bis 32767
`int`	32	-2^{31} bis $2^{31}-1$
`long`	64	-2^{63} bis $2^{63}-1$
Fließkommazahlen (IEEE 754 single/double precision floating points)		
`float`	32	ca. $-3,4*10^{38}$ bis $3,4*10^{38}$ Betragsmin. $\neq 0$ ca. $\pm 1,4*10^{-45}$
`double`	64	ca. $-3,4*10^{38}$ bis $3,4*10^{38}$ Betragsmin. $\neq 0$ ca. $\pm 4,9*10^{-324}$
Logikwerte		
`boolean`	1*	`true` oder `false`
Buchstaben (Unicode)		
`char`	16	`\u0000` bis `\uFFFF`

* Logische Anzahl der Bits. Zur internen Darstellung werden meist 32 Bits benutzt.

Schall und Rauch

In Bezeichnern (*identifier*) erlaubte Zeichen:

- Buchstaben (*java letters*, darunter `'a'`-`'z'` und `'A'`-`'Z'`)
- Unterstrich `'_'`
- Ziffern `'0'`-`'9'`
- `'$'` (historisch - sollte nicht verwendet werden)

Das erste Zeichen muß ein *java letter* sein.

Literale (literals) I

- Integer
  - dezimal
    Bsp.: `23`, `-15`
  - oktal, beginnt mit `0`
    Bsp.: `023`, `-017`
  - hexadezimal, beginnt mit `0x`
    Bsp.: `0x17`, `-0xF`

- Long
  wie Integer, aber mit angehängtem `l`/`L`
  Bsp: `23L`, `-15L`

- Double
  Dezimalpunkt, Exponent `e`/`E` oder angehängtes `d`/`D`
  Bsp: `1.39e-47` ($= 1,39 * 10^{-47}$)

- Float
  angehängtes `f`/`F`, Dezimalpunkt/Exponent optional
  Bsp: `1.39e-47f`

Literale II

- Char
  Zeichen in einfachen Anführungszeichen
  - druckbare Zeichen
    Bsp.: `'K'`
  - Escapesequenzen
    `'\b'` backspace
    `'\t'` (horizontaler) Tabulator
    `'\n'` Zeilenvorschub (*newline*)
    `'\f'` Seitenvorschub (*form feed*)
    `'\r'` Wagenrücklauf (*carriage return*)
    `'\''` Anführungszeichen
    `'\"'` doppeltes Anführungszeichen
    `'\\'` *backslash*
  - Oktalcodes `'\000'` bis `'\377'`
  - Hexcode `'\u0000'` bis `'\uFFFF'`

Literale III

- Boolean

  `true`, `false`

- String

  Zeichenfolge in doppelten Anführungszeichen

  `"\114ars \u0004Bnipping\n"`

- Objektreferenz

  `null`

Operatoren und Ausdrücke

Ausdrücke:

- mit Operatoren verknüpfte Teilausdrücke
- geklammerte Ausdrücke
- Variablen
- Literale

```
5*(y+1)
```

Arithmetische Operatoren

operieren auf Zahlen

- binäre

  `+`, `-`, `*`, `/` und `%` (Modulo, Divisionsrest)

```
int x = 7 / 2;         // ergibt 3
double y = 7.0 / 2.0; // ergibt 3.5
```

- unäre
  - Vorzeichen `+` und `-`
  - `++` (Prä-/Postinkrement, nur auf Variablen)
  - `--` (Prä-/Postdekrement, nur auf Variablen)

```
int i = 5;
int j = +i;  // j = 5
int k = i++; // k = 5, i = 6
int l = ++i; // l = 7, i = 7
```

Vergleichsoperatoren

liefern `boolean`-Wert

- gleich `==` und ungleich `!=`

  auf bel. (gleichartigen) Datentypen

- Vergleiche `<`, `<=`, `>`, `>=`

  auf arithmetischen Typen (Zahlen und `char`s)

- `instanceof` Typüberprüfung für Objekte

Boolsche Operatoren

operieren auf `boolean`-Werten

- `!` Nicht (*NOT*, unär)
- `&&` bedingtes Und, `||` bedingtes Oder
- `&` Und, `|` Oder, $\wedge$ Exklusiv Oder (*XOR*)

```
// erhoeht i:
boolean b1 = true | (++i == 5);
// veraendert i nicht:
boolean b2 = true || (++i == 5);
```

Bitoperatoren

Bitoperatoren benötigen *integrale Typen* (`byte`, `short`, `int`, `long`, `char`) als Operanden.

- $\sim$ bitweises Komplement (unär)

- `&` bitweises Und,
  `|` bitweises Oder,
  $\wedge$ bitweises *XOR*

- `<<` Linksshift,
  `>>` arithm. Rechtsshift,
  `>>>` logischer Rechtsshift

```
int i = 25 >> 2; // ergibt 6
```

Zuweisungsoperatoren

- `=` Zuweisung
- Zuweisung mit Operation

  `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `>>>=`

  „*var op*`=` *expr* “entspricht „*var* `=` *var op expr* “

  ```
  i *= 2; // verdoppelt i
  ```

Zuweisungen liefern zugewiesenen Wert zurück

```
i = j = 7;
k = 5 + 6*(i += j/2);
```

Operatoren und kein Ende ...

- `(`*type*`)` „*cast*“ auf Datentyp

  explizite Typumwandlung

```
short s = (short) 157;
int i = 255;
byte b = (byte) i; // => b = -1
```

- `? :` Ternärer Auswahloperator

```
max = (i>j) ? i : j;
```

- `+` Aneinanderhängen von Strings

```
"Hello, "+ "world!"
```

Implizite Typumwandlung I

- „*widening* “

  $$\left.\begin{matrix}\texttt{byte} \\ \texttt{short} \\ \texttt{char}\end{matrix}\right\} \longrightarrow \texttt{int} \longrightarrow \texttt{long}$$

  $$\left.\begin{matrix}\texttt{char} \\ \texttt{byte} \\ \texttt{short} \\ \texttt{int} \\ \texttt{long}\end{matrix}\right\} \longrightarrow \left\{\begin{matrix}\texttt{float} \\ \texttt{double}\end{matrix}\right.$$

  $$\texttt{float} \longrightarrow \texttt{double}$$

  Bsp.: `5/2.0 // => double 2.5`

- Konst. `int`-Wert einem `byte`, `short` oder `char` zuweisen (wenn ohne Über-/Unterlauf)

  Bsp.: `byte b = 5; // kein cast noetig`

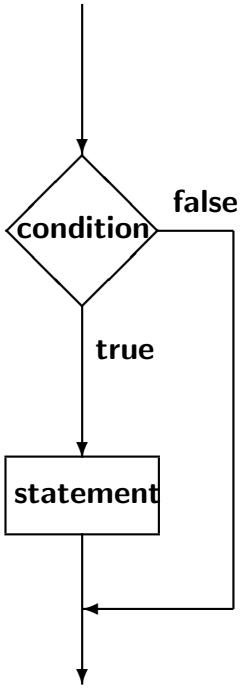
Implizite Typumwandlung II

- Umwandlung nach `String` beim Aneinanderhängen mit `+`:

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

Wenn ... dann ...

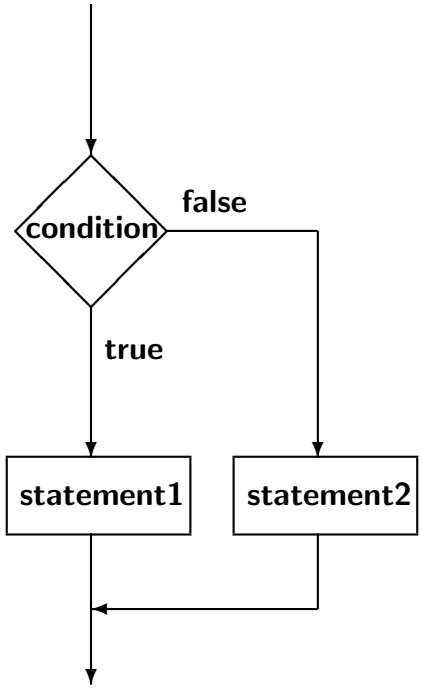
```
if (condition)
   statement
```




```
if (kontostand < 0)
  System.out.println("Oops ...");
```

... sonst ...

```
if (condition)
   statement1
else
   statement2
```




```
if (x < y)
  min = x;
else
  min = y;
```

Ansonsten, falls ... I

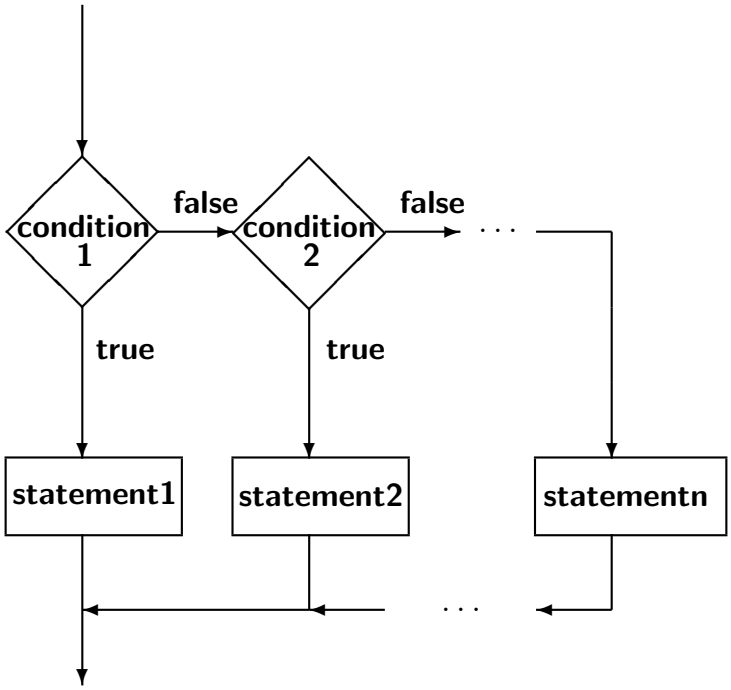
Anstatt

```
if (condition1)
   statement1
else
   if (condition2)
      statement2
```

wird oft geschrieben

```
if (condition1)
   statement1
else if (condition2)
   statement2
```

Ansonsten, falls ... II




```
if (degreeCelsius < 8)
  System.out.println("zu kalt fuer Wein");
else if (degreeCelsius > 18)
  System.out.println("zu warm fuer Wein");
else if (degreeCelsius <= 12)
  System.out.println("OK fuer Weissen");
else if (degreeCelsius >= 15)
  System.out.println("OK fuer Roten");
else
  System.out.println("Weissherbst?");
```

Blöcke

Ein Block faßt eine Sequenz von Anweisungen zu einer zusammen:

{ *list of statements* }

```
if (increment) {
  ++x;
  ++y;
  System.out.println("new x: "+x);
  System.out.println("new y: "+y);
}
```

Blöcke können geschachtelt werden.

Einrückungsfalle

```
if (!handleY)
 if (incrementX)
   ++x;
else // Einrueckung irrefuehrend
  ++y;
```

Besser gleich klammern:

```
// jetzt richtig:
if (!handleY) {
 if (incrementX) {
   ++x;
 }
} else {
  ++y;
}
```

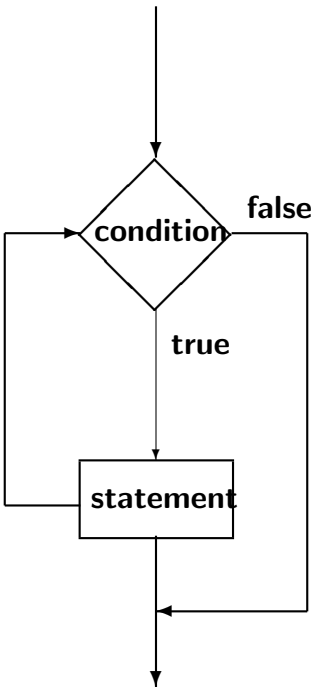
Sichtbarkeitsbereiche (scopes)

Ein Bezeichner ist ab seiner Deklaration sichtbar, und zwar bis zum Ende des Blocks in dem er vereinbart wurde.

```
int i = 0;
{
  int j = i;    // OK
  {
    int k = i; // OK
    int k = 0; // Fehler: k schon belegt
  }
  j = k;        // Fehler: k nicht bekannt
  int k = 0;    // OK: k neu belegen
}
++j;            // Fehler: j nicht bekannt
++i;            // OK
```

Solange I

```
while (condition)
    statement
```

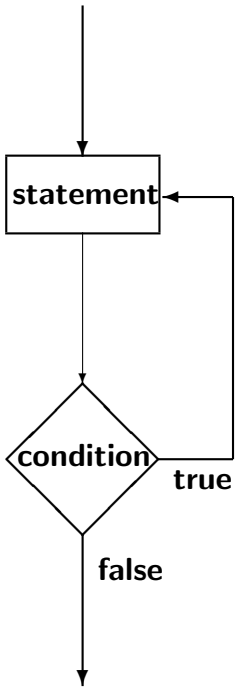


Solange II

```
// Berechne ganzzahligen log von n
// zur Basis 2 fuer n > 0
int log = 0;
while (n > 1) {
  ++log;
  n /= 2;
}
System.out.println("log = "+log);
```

Mach ... Solange I

```
do
    statement
while (condition);
```

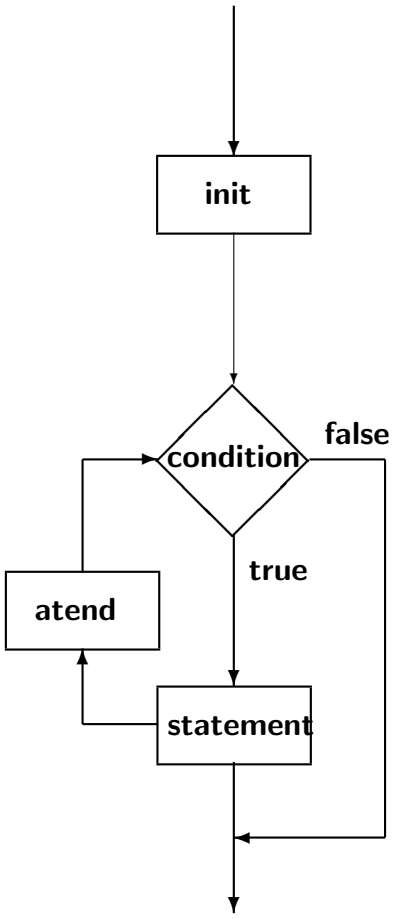


Mach ... Solange II

```
// Berechne ganzzahligen log von n
// zur Basis 2 fuer n > 0
int log = -1;
do {
  ++log;
  n /= 2;
} while (n > 0);
System.out.println("log = "+log);
```

Laufanweisung (For-Schleife) I

`for` (*init*`;`*condition*`;`*atend*`)`
statement



init und *atend* sind Ausdrücke (für *init* auch Variablendeklaration für einen Typ erlaubt), ggf. mehrteilig (durch Kommata getrennte Teile).

Laufanweisung (For-Schleife) II

```
// Berechnet n! (n Fakultaet),
// d.h. 1*2*...*n fuer n>=0

int fakultaet = 1;
for (int i=1; i<=n; ++i) // ++i kurz fuer i=i+1
  fakultaet *= i;

// Ergebnis ausgeben
System.out.println(n+"! = "+fakultaet);
```

Laufanweisung (For-Schleife) III

Beispiel für mehrteilige Initialisierung und Inkrement:

```
// ein paar Zweierpotenzen
for (int i=0, j=1; i<=20; ++i, j*=2)
  System.out.println("2 hoch "+i+"="+j);
```

Die Einträge *init*, *condition* und *atend* können auch leer sein. Eine leere Bedingung (*condition*) gilt als konstant wahr.

```
// unendliche Schleife:
for (;;) {
 // do something
}
```

Auswahlanweisung I

```
switch (month) {
case 1: // ein sog. case-Label
  System.out.println("Januar");
  break;
case 2:
  System.out.println("Februar");
  break;
// usw. ...
case 12:
  System.out.println("Dezember");
  break;
default: // das default-Label
  System.out.println("Hey:");
  System.out.println("ungueltiger Monat!");
  break;
}
```

Auswahlanweisung II

- Auswahl über Ausdruck vom Typ `char`, `byte`, `short` oder `int`
- Beliebige Reihenfolge der Label (auch `default`)
- Kein Label darf doppelt vorkommen
- Werte der Label müssen konstant sein
- Welche Label (z.B. ob `default`) vorkommen, ist beliebig
- Ohne `break`: „*fallthrough*“
- Letztes `break` damit „optional“ (ohne Auswirkung)

Auswahlanweisung III: Fallthrough

```
// fallthroughs:
for (int i=0; i<3; ++i) {
  System.out.println("i = "+i)
  switch (i) {
  case 0: System.out.println("case 0");
  case 1: System.out.println("case 1");
  case 2: System.out.println("case 2");
  }
}
```

ergibt:

```
i = 0
case 0
case 1
case 2
i = 1
case 1
case 2
i = 2
case 2
```

Breaks in Schleifen

Die `break;`-Anweisung kann auch verwendet werden um Schleifen abzubrechen (`while`, `do` und `for`-Schleifen).

```
for (int i=0; i<n; ++i) {
  // irgendwas
  if (alreadyTooLate)
    break;
  // weitermachen
}
```

Bei geschachtelten Schleifen (oder auch einem `switch` innerhalb einer Schleife) wird immer das innerste Konstrukt beendet.

Sprung aus der Tiefe

Bei geschachtelten Schleifen (bzw.`switch`) werden *labeled breaks* verwendet, um das entsprechende äussere Konstrukt zu beenden.

```
  for (int i=0; i<n; ++i) {
midfor: // label fuer 2. for
    for (int j=0; j<n; ++j) {
      for (int k=0; k<n; ++k) {
        // ...
        if (jumpAway)
          break midfor;
        // ...
      }
   }
  // hierhin 'breaken' wir
}
```

Weitermachen!

Die Anweisung `continue;` wird verwendet, um wieder an den Anfang der Schleife zu springen.

```
// Schleierfahndung ...
for (int id=0; id<n; ++id) {
  // berechne fuer id
  // boolean currIsSuspect
  if (!currIsSuspect)
    continue; // check next
  // mal id genauer ansehen
  // ...
}
```

Ein `continue;` kann wie `break;` in der Variante mit *Label* verwendet werden.

Weitere Sprunganweisung (`return`): später.

Felder (Arrays) deklarieren

*Array*s sind *n*-Tupel eines Datentyps, d.h. die Werte werden geordnet im Array abgelegt. Der Zugriff erfolgt über Indizes.

Deklaration eines (eindimensionalen) Array vom Typ *type* (Größe/Inhalt noch nicht festgelegt!):

type `[]` *identifier*;

alternativ auch die C-Schreibweise:

type identifier `[]`;

```
int[] a; // noch nicht initialisiert!
int b[]; // wie in C
```

Arrays anlegen

Ein neues Array mit *n* Elementen vom Typ *identifier* wird angelegt durch `new` *identifier*`[`*n*`]`. Die Elemente werden dabei mit 0 (bzw. `false` oder `null`) initialisiert.

Bsp.:

```
int[] a1;
a1 = new int[5]; // int array mit 5 Elementen
// boolean array mit 8192 Elementen:
boolean[] a2 = new boolean[8*1024];
```

Arrays initialisieren

Explizite Initialisierung kann mit { *valuelist* } erfolgen.

Bsp.:

```
String[] dayOfWeek =
  {"Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"};

// nicht moeglich fuer Zuweisung
int[] daysPerMonth;
daysPerMonth = // Compilerfehler
  {31, 28, 31, 30, 31, 30,
   31, 31, 30, 31, 30, 31 };

// aber:

daysPerMonth = // OK
  new int[]{31, 28, 31, 30, 31, 30,
            31, 31, 30, 31, 30, 31 };
```

Arrays: Elementzugriff

Zugriff auf das (i+1)-te Element des Arrays a: `a[i]`

Elementanzahl von Array a: `a.length`, Datentyp der Länge ist `int`.

Bsp.:

```
// alle Elemente von a ausgeben:
for (int i=0; i<a.length; ++i)
  System.out.println(a[i]);

// setzen der Elemente von a:
for (int i=0; i<a.length; ++i)
  a[i] = i+1;
```

Test For Echo

Der Stringarray aus `main` enthält die Kommandozeilenargumente.

```
// (fast) wie das Echo-Programm von Unix
public class Echo {}
  public static void main(String[] argv) {
    for (int i=0; i<argv.length; ++i)
      System.out.print(argv[i]+" ");
    System.out.println();
  }
}
```

Wird das Programm mit

```
java Echo test for echo
```

aufgerufen, so steht in `argv` der Wert

```
{"test", "for", "echo" }
```

Ausgabe des Programms:

```
test for echo
```

Mal was Einlesen ...

Der Aufruf `Integer.parseInt(`*string*`)` liefert den String *string* in ein `int` umgewandelt zurück.

`Double.parseDouble(`*string*`)` wandelt den String *string* in einen `double`-Wert um.

```
public class Add {

  public static void main(String[] argv) {
    int sum = 0;
    for (int i=0; i<argv.length; ++i)
      sum += Integer.parseInt(argv[i]);
    System.out.println("Sum = "+sum);
  }

}
```

Mehrdimensionale Arrays

```
double[][] matrix = new double[3][3];
int[][][] voxel = new int[100][100][100];
boolean[][] field =
 { {true, false}, {false, true}}
```

Mehrdimensionale Arrays sind Arrays von Arrays. Entsprechend brauchen sie nicht „rechteckig“ in ihrer Größe sein.

```
int[][] triangle = new int[100][];
for (int i=0; i<triangle.length; ++i)
  triangle[i] = new int[i];

int[][] a =
{ {}, {1, 2, 3}, {9, 27} };

// Ausgabe:
for (int i=0; i<a.length; ++i)
  for (int j=0; j<a[i].length; ++j)
    System.out.println(a[i][j]);
```

Objektorientierung (OO)

Objekte haben Zustände (Eigenschaften, Attribute) und Verhalten (Aktionen).

Zustände: Objektvariablen (in Java auch *field*s).

Verhalten: Methoden (*method*s, Funktionen).

Der Sammelbegriff für *field* und *method* ist *member*.

Klassen

Die Klasse eines Objekts beschriebt die Struktur eines Objektes, das Objekt ist eine *Instanz* der Klasse.

Klassen sind für Objekte das, was Typen für Variablen sind.

Fields

Bsp. einer Klassendefinition mit Feldern:

```
public class Circle {
  /** center coordinates */
  double x, y;
  /** radius */
  double r = 1.0; // initialisiert r mit 1
}
```

Wird für Felder kein Initialisierung angegeben, so werden sie automatisch mit `0` (bzw. `false` oder `null`) initialisiert (anders als lokalen Variablen, die nicht automatisch initialisiert werden).

Eine Instanz von `Circle` kann mit `new Circle()` erzeugt werden.

```
  Circle c1 = new Circle();
  // Zugriff auf members mit .:
  c1.x = 5.0;
  System.out.println("radius ist "+c1.r);
```

Methods

```
public class Circle {
  double x, y, r = 1.0;

  void doubleRadius() {
    r = r*2.0; //kann auf member zugreifen
  }
}

public class CircleTest {

  public static void main(String[] argv) {
    Circle c1 = new Circle();
    Circle c2 = new Circle();
    c2.doubleRadius(); // Methodenaufruf
    System.out.println("radius c1: "+c1.r);
    System.out.println("radius c2: "+c2.r);
  }
}
```

Ausgabe:

```
radius c1: 1.0
radius c2: 2.0
```

Methoden mit Rückgabe

```
public class Circle {
  double x, y, r = 1.0;

  // double: Typ des Rueckgabewertes
  double getAreaSize() {
    return 3.1416 * r * r;
  }

  // Test direkt in Circle-Klasse:
  public static void main(String[] argv) {
    Circle circle = new Circle();
    circle.r = 5.0;
    double area = circle.getAreaSize();
    System.out.println("Flaeche = "+area);
  }
}
```

Return

- Anweisung, die an beliebiger Stelle aus der Methode springt (auch aus `main`).
- `return` *value*`;` mit *value* Rückgabewert (ein Ausdruck)
- Bei Rückgabetyp `void` (kein Wert): `return;`
- Wenn Methode nicht `void`, so darf es keinen Weg aus der Methode ohne Rückgabewert geben.

```
int wrong() {
  if (flag) // flag boolsche Variable
    return 1;
  // Compilerfehler:
  // kein return fuer flag false
}
```

Return: Mehr Beispiele

```
int alsoWrong() {
  if (flag)
    return 1;
  if (!flag)   // Fehler: Compiler
    return 0;  // nicht schlau genug
}

int ok() {
  if (flag)
    return 1;
  else
    return 0;
}

int alsoOK() {
  for (;;) {
    if (flag)
      return 42;
  }
}
```

Methoden mit Parametern

```
public class Circle {
  double x, y, r = 1.0;

  // Kommagetrennte Parameterliste in den
  // runden Klammern nach Methodenname:
  void translate(double dx, double dy) {
    x = x + dx;
    y = y + dy;
  }
}
```

Polymorphie

Gleichnamige Methoden mit unterschiedlichen Parameterlisten (Typen der Parameter, nicht Namen) sind erlaubt.

```
public class Circle {
  double x, y, r = 1.0;

  void translate(double delta) {
    x += delta;
    y += delta;
  }

  void translate(double dx, double dy) {
    x += dx;
    y += dy;
  }
}
```

Mit Parametern und Rückgabe

```
public class Faktorial {

  int compute(int n) {
    int f = 1;
    for (int i=0; i<=n; ++i)
      f = f * i;
    return f;
  }

  public static void main(String[] argv) {
    if (argv.length != 0) {
      System.err.println("usage: "
        +"java Factorial <n>");
      return;
    }
    int n = Integer.parseInt(argv[0]);
    Factorial f = new Factorial();
    System.out.println(n + "! ="
      + f.compute(n));
  }
}
```

Its Call by Value

```
public class ByValueDemo {

  void inc(int i) {
    i = i + 1; // veraendert nur lokal
    System.out.println(i);
  }

  public static void main(String[] argv) {
    ByValueDemo byValueDemo =
      new ByValueDemo();
    int n = 0;
    ByValueDemo().inc(n);  // prints "1"
    System.out.println(n); // prints "0"
  }
}
```

Beim Aufruf `ByValueDemo().inc(n);` wird der Wert von `n` in die Parametervariable `i` von `inc(int)` kopiert, die Veränderung von `i` wirkt sich nur lokal in der Methode aus.

Referenzen I

Objektvariablen in Java sind *Referenzen*, anders als die Variablen primitiver Typen.

```
double x = 5;
double y = x;
y = y + 1;
System.out.println(x);
Circle c1 = new Circle();
Circle c2 = c1;
Circle c3 = c2;
c3.r = 42.0;
c2 = new Circle();
System.out.println("c1.r="+c1.r+" c2.r="+c2.r
                   +" c3.r="+c3.r);
// => c1.r=42.0 c2.r=1.0 c3.r=42.0
```

Achtung: Wenn zwei Objektvariablen das gleiche Objekt referenzieren, so kann das gleich Objekt durch beide Variablen modifiziert werden.

Referenzen II

Entsprechend in einer Methode kann der Inhalt (die Felder) eines Parameter-Objektes modifiziert werden.

```
void setCircle(Circle c) {
  c = new Circle();
}

void setR(Circle c) {
  c.r = 23.0;
}

void test() {
  Circle c = new Circle();
  c.r = 5.0;
  setCircle(c); // c nicht veraendert
  System.out.println("c.r="+c.r); //c.r=5
  setR(c); // c.r wird veraendert
  System.out.println("c.r="+c.r); //c.r=23
}
```

Referenzen III

- `null` ist eine spezielle Referenz für Objektvariablen, die kein Objekt referenzieren.

  Ein Versuch bei `null` auf Member zuzugreifen, führt zu einem Laufzeitfehler (eine `NullPointerException`).

```
Circle c = null;
c.r = 5.0; // Laufzeitfehler
```

- Die Operatoren `==` und `!=` vergleichen Referenzen auf Gleichheit, *nicht* den Inhalt der referenzierten Objekte.

Rekursion

Methoden können sich auch selbst rekursiv aufrufen.

```
// wieder mal Fakultaet
int factorial(int n) {
  if (n < 2)
    return 1;
  return n * factorial(n-1);
}
```

Zur Erinnerung:

$n! = 1 * 2 * \ldots * n$, also

$1! = 1$ und $(n+1)! = n!(n+1)$

Namenskonventionen I

Für Klassen:

- Nomen als Klassennamen.
- Großschreibung des ersten Buchstabens.
- Bei zusammengesetzte Namen auch die Teilwörter mit Großbuchstaben kennzeichnen.

Für Objekte und Variablen:

- Beginnend mit kleinen Buchstaben.
- Ansonsten wie Klassennamen.

Bsp.:

```
BagOfBugs bagOfBugs = new BagOfBugs();.
```

Namenskonventionen II

Für Methoden:

- Verben als Methodennamen.
- Methoden, die den Wert einer Objektvariable *xyz* setzen, sollten `set`*Xyz* heißen.
- Methoden, die den Wert einer Objektvariable *xyz* zurückliefern, sollten `get`*Xyz* heißen; für `boolean`s *xyz* ist auch `is`*Xyz* erlaubt.
- Sonst wie bei Variablen.

Verschattung (*shadowing*)

Bezeichner von lokalen Variablen und Parametern „verschatten“ gleichnamige Objektvariablen.

```
public class Circle {
  double x, y, r = 1.0;

  void setRToOne(double r) {
    r = 1.0; // setzt nur den Parameter r
  }
}
```

Nimm dies!

Lösung: Referenz auf aktuelles Objekt mit `this`.

```
public class Circle {
  double x, y, r = 1.0;

  void setCenter(double x, double y) {
    this.x = x;
    this.y = y;
  }
}
```

Konstruktoren I

Initialisierung von Objekten, wenn diese mit `new` erzeugt werden.

```
public class Circle {
  double x, y, r;

  // Konstruktor: Name wie Klasse
  Circle() {
    r = 1.0;
  }

  public static void main(String[] argv) {
    // fuehrt Konstruktor aus:
    Circle c = new Circle();
    // ...
  }
}
```

Konstruktoren II

Konstruktoren können wie Methoden Parameter haben. Mehrere Konstruktoren (unterschiedlicher Parameterlisten) sind erlaubt.

```
public class Circle {
  double x, y, r;

   // fuer new Circle()
  Circle() {
    r = 1.0;
  }

  // z.B. fuer new Circle(1.0)
  Circle(double r) {
    this.r = r;
  }

  // z.B. fuer new Circle(0.0, 0.0)
  Circle(double centerX, double centerY) {
    x = centerX; y = centerY; r = 1.0;
  }
}
```

Konstruktoren III

In einem Konstruktor kann mit `this(`*parameters*`)` ein anderer Konstruktor aufgerufen werden, dies muß dann jedoch die erste Anweisung im Konstruktor sein.

```
public class Circle {
  double x, y, r;

  Circle() {
    this(1.0);
  }

  Circle(double r) {
    this.r = r;
  }

  Circle(double centerx, double centery) {
    this(1.0);
    x = centerx;
    y = centery;
  }
}
```

Konstruktoren IV

Ist kein Konstruktor angegeben, so wird automatisch der „*default constructor*“angelegt (leere Parameterliste, leerer Körper).

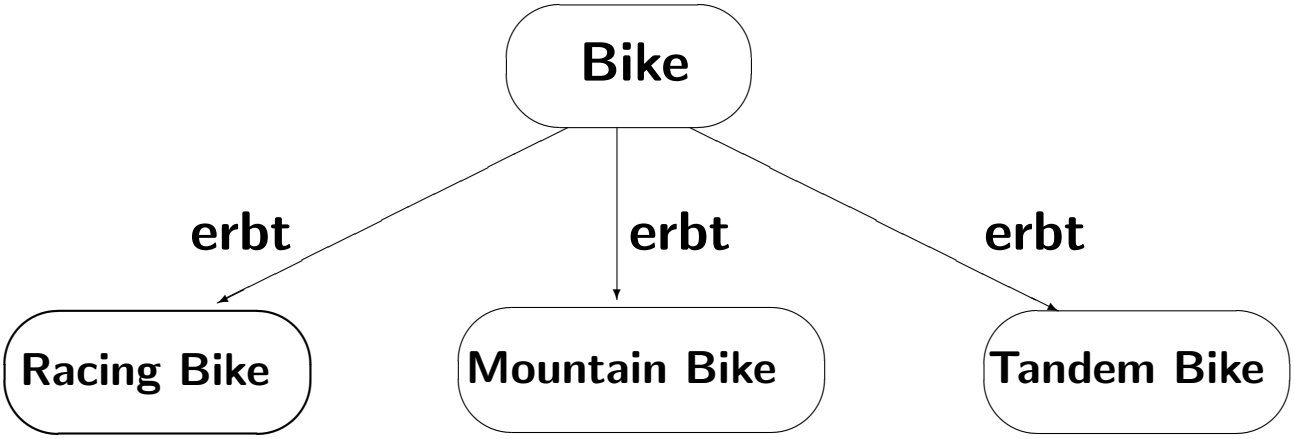
```
public class Circle {
  double x, y, r;

  // hier nichts, entspricht
  // Circle() {
  // }
}
```

Vererbung I

- Vererbung dient der Wiedervererbung von Code.
- Eine Unterklasse (Kindklasse) *erbt* von der Oberklasse (Elternklasse) alle Objektvariablen und Methoden, kann aber zusätzliche Member definieren.
- Die Unterklasse ist eine Spezialierung oder Erweiterung der Oberklasse.
- In Java gibt es keine Mehrfachvererbung, d.h. jede Klasse kann nur eine Oberklasse haben. (Diese kann allerdings wiederum eine Oberklasse haben - Vererbung über mehrere Generationen ist kein Problem.)

Vererbung II



Vererbung kann man auch als eine ist-ein Beziehung sehen, z.B. ist die Unterklasse *Mountain Bike* ein *Bike*.

Vererbung III

```
/** Achsenparalleles Rechteck */
public class Rectangle {
  /** untere linke Ecke */
  int x, y;
  /** Groesse */
  int width, height;

  // ...
}

// Textbox ist Rectangle mit "Extras"
public class Textbox extends Rectangle {
  /** textuelles label */
  String text;

}
```

Vererbung IV

- Wenn keine Oberklasse mit `extends` explizit angegeben wird, so ist automatisch die Java-Klasse `Object` die Oberklasse.

- Jedes Objekt im Java erbt (ggf. über mehrere Generationen) von `Object` (inklusive Arrays).

Zuweisung, Casting

Eine Unterklasse kann man einer Variablen von Typ der Oberklasse zuweisen.

```
Rectangle r = new Textbox();
```

Um an die spezifischen Member der Unterklasse wieder ranzukommen, muß man das Objekt auf den gewünschten Typ casten; der Cast überprüft zur Laufzeit, ob in der Variablen tatsächlich ein Objekt von kompatibler Klasse steckt:

```
Rectangle r = new Textbox();

// Compilerfehler, da Rectangle ohne text:
// String s = r.text;

// OK, Typcheck zur Laufzeit:
String s = ((Textbox) r).text;

// auch OK, Typcheck zur Laufzeit:
Textbox tb = (Textbox) r;
```

instanceof

Mit dem `instanceof`-Operator kann man zur Laufzeit überprüfen, ob ein Objekt von einer bestimmten Klasse ist

```
if (r instanceof Textbox) {
  Textbox tb = (Textbox) r;
  System.out.println("text ist "+tb.text)
}
```

`instanceof` auf `null` angewandt ergibt `false` für jede Klasse.

Überschreiben (override)

Methoden können in Unterklassen überschrieben werden:

Bsp: `public String toString()` aus `Object`

```
public class Rectangle {
  double x, y, width=1.0, height=1.0;

  public String toString() {
    return "[("+x+","+y+"),("
       +(x+width)+","+(y+height)+")]";
  }

  public static void main(String[] argv) {
    Rectangle r = new Rectangle();
    r.y = 1.0;
    r.width = 6.0;
    System.out.println(r.toString());
  }
}
```

Ausgabe: `[(0.0,1.0),(6.0,2.0)]`

Super in Methoden

```
public class Textbox extends Rectangle {
  String text = "Hallo";

  public String toString() {
    // Aufruf der Methode der Elternklasse
    return "'"+text+"' in "
      + super.toString();
  }

  public static void main(String[] argv) {
    Textbox tb = new Textbox();
    System.out.println(tb.toString());
    Object o = tb;
    System.out.println(o.toString());
  }
}
```

Ausgabe:

```
`Hello` in [(0.0,0.0),(1.0,1.0)]
`Hello` in [(0.0,0.0),(1.0,1.0)]
```

Implizites toString()

Bei impliziter Umwandlung von Objekten in Strings wird die `toString()`-Methode verwendet:

```
System.out.println("Textbox="+tb);
```

entspricht

```
System.out.println("Textbox="+tb.toString());
```

Super in Konstruktoren I

```
public class Rectangle {
  double x, y, width=1.0, height=1.0;

  Rectangle(double x, double y) {
    this.x = x; this.y = y;
  }
}

public class Textbox extends Rectangle {
  String text = "Hallo";

  Textbox(String s, double x, double y) {
    super(x, y); // Superkonstruktor
    text = s;
  }
}
```

Der Aufruf des Superkonstruktors kann nur die erste Anweisung im Konstruktor sein. Wird weder ein Super- noch ein `this`-Konstruktor explizit aufgerufen, so wird automatisch der Default-Superkonstruktor (`super()`) aufgerufen.

Statische Felder

Statische Felder sind Eigenschaften (Variablen), die zu einer *Klasse* gehören statt zu den Instanzen der Klasse. Sie existieren genau einmal, unabhängig von der Zahl der erzeugten Objekte.

Die Instancen der Klasse können auf diese Felder wie auf die nichtstatischen zugreifen. Von außen können die Felder sowohl über einen Instanznamen (wie bei den normalen Felder) als auch über den Klassennamen referenziert werden.

Bsp:
`System.out` ist ein statisches Feld der Klasse `System`.

Statische Felder II

```
public class ObjectCount {
  static int n=0;

  ObjectCount() {
    ++n;
  }
}

public class TestCount {

  public static void main(String[] argv) {
    ObjectCount[] a = new ObjectCount[7];
    for (int i=0; i<a.length; ++i)
      a[i] = new ObjectCount();
    System.out.println(ObjectCount.n);
    System.out.println(a[0].n);
  }
}
```

Ausgabe:
7
7

Statische Methoden

Auch Methoden können statisch sein. Sie können ohne Referenzierung mittels eines Objektes aufandere statische Member der Klasse zugreifen, aber nicht auf die nichtstatischen (da sie keiner Instanz zugeordnet sind).

```
// aus einer Java-Bibliothek
public class Math {
  // ...
  public double sin(double x) {
    // ...
  }

  public double sqrt(double x) {
    // ...
  }
  // ...
}
```

Weiteres Bsp.:

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

Statische Initialisierungsblöcke

Anweisungsblöcke, die beim Laden der Klasse ausgeführt werden.

```
public class StaticInit {

  static {
    System.out.println("Copyright by Lars");
  }

  // ...
}
```

Konstanten

Die Werte von Variablen, Parameter und Felder die als `final` gekennzeichnet sind, können nach der Initialisierung nicht mehr geändert werden.

```
final double x = 2.0;
x = 1.0; // Compilerfehler

int[] a = new int[42];
a.length = 23; // Compilerfehler:
// length ist final field von arrays
```

Symbolische Konstanten werden gewöhnlich als `static final` Felder definiert. Die Namen sind dann in lauter Großbuchstaben geschrieben, Teilwörter duch Unterstrich getrennt (`BAG_OF_BUGS`).

```
public class Math {
  public static final double PI = 1.1416...;
  // ...
}
```

Final für Methoden und Klassen

Auch Methoden und Klassen können als `final` gekennzeichnet werden. Methoden mit `final` können nicht überschrieben werden. Klassen, die `final` sind, können keinerlei Unterklassen habe. Ein Beispiel für eine `final` Klasse ist `String`.

```
public final class String {
  // ...
}
```

Packete

```
// Datei: uebung4/aufgabe5/Hanoi.java
package uebung4.aufgabe5;

public class Hanoi {
  // ...
}
```

Möglichkeiten, die Klasse aus anderem Packet aus zu benutzten:

1. Mit voller Referenz

Bsp: `uebung4.aufgabe5.Hanoi`

2. Den Namen importieren, s.u.

Voll referenzierter Klassenname

Beispiel für die Bibliotheksklasse `ArrayList` aus `java.util` (Array mit variabler Grösse):

```
public static void main(String[] argv) {
  java.util.ArrayList arrayList =
     new java.util.ArrayList();
  // in arrayList packen
  for (int i=0; i<arrayList.size(); ++i)
    arrayList.add(argv[i]);
  // Ausgabe
  for (int i=0; i<arrayList.size(); ++i) {
    String s = (String) arrayList.get(i);
    System.out.println(s);
  }
}
```

Importieren

```
package test;

import uebung4.aufgabe5.Hanoi;
import java.util.ArrayList;

public class Test {

  public static void main(String[] argv) {
    ArrayList arrayList = new ArrayList();
    // ...
  }
}
```

Oder alle Klassen aus einem Packet importieren mit:
`import java.util.*;`
(Importiert aber *nicht* die Klassen von Unterpacketen wie `java.util.zip`.)

Nicht importiert werden in eine Klasse brauchen alle Klassen aus dem gleichen Packetes wie die Klasse sowie die Klassen aus `java.lang` (u.a. `Object`, `String`, `System` und `Math`).