

Java: Sprachmerkmale

Johannes Rössel

2006-11-14

Dinge von Wichtigkeit

1. Variablen
2. Datentypen
3. Operatoren
4. Kontrollstrukturen
5. Objektorientierung
6. Klassen, Interfaces
7. Packages
8. Fehlerbehandlung

Java

- Im Grunde ausschließlich objektorientiert („Alles ist ein Objekt“)
- In dem Sinne nicht wirklich plattformunabhängig, sondern eher eine eigene Plattform
- Große Standardbibliothek
- Syntax im Wesentlichen an C++ angelehnt
- Objektmodell entstammt zum Teil Smalltalk
- Keine Operatorüberladungen, daher recht konsistentes Operatormodell

Hello World!

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

1. Variablen

- Variablen können zweierlei Dinge enthalten: Primitive Datentypen und Objektreferenzen
- Deklaration in der Form

Typ **Bezeichner**[, **Bezeichner**[, ...]]:

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

- Variablennamen berücksichtigen Groß-Klein-Schreibung:

```
int x = 5;  
double X = 3.14;
```

1. Konstanten

- **final** *Typ Bezeichner*:

```
// private Klassenkonstante,  
// außen nicht sichtbar  
private static final int blubb = 5;  
// öffentlich für alle  
// hängt am Objekt  
public final String foo = "bar";  
// normal sichtbar  
static final double pi = 3.14;
```

2. Datentypen

- Primitive Typen sind **keine** Objekte!
- Zu den primitiven Datentypen existiert jeweils eine **Klasse**, die eben jenen Typ beinhaltet
- Strings und Arrays werden hier ebenfalls behandelt, obwohl sie Objekte sind, aber von der Sprache einiges an Ausnahmen eingeräumt bekommen

2.1. Integer-Typen

- **byte** 8 Bit
- **short** 16 Bit
- **int** 32 Bit
- **long** 64 Bit

Sämtliche Integer-Typen werden als Zweierkomplement mit Vorzeichen dargestellt. Java kennt keine vorzeichenlosen Integer.

2.1. Integer-Typen ...

Integer können auf verschiedene Art und Weise im Quelltext geschrieben werden:

- Dezimalzahlen:

```
|| int decVal = 26; // 26, decimal
```

- Oktalzahlen:

```
|| int octVal = 032; // 26, oktal
```

- Hexadezimalzahlen:

```
|| int hexVal = 0x1a; // 26, hexadezimal
```

2.1. Char

Etwas abseits davon, aber im Grunde auch ein Integertyp:

- **char** 16 Bit: `'\u0000'`–`'\uffff'` (Unicode-Zeichen)

char bezeichnet Zeichen in der Unicode-BMP und kennt daher weder negative Werte noch Zeichen außerhalb der BMP.

Literale vom Typ **char** werden in Hochkommata geschrieben, es gibt verschiedene Escapevarianten:

```
char capitalC    = 'C'; // einfache Zeichen
char tab         = '\t'; // benannte Escapes
char lowercaseA = '\141' // oktal
char omega       = '\u03c9'; // Unicode
```

2.1. Boolean

- Einfacher Wahr/Falsch-Wert (ein Bit an Information)
- Wird in einigen Kontrollstrukturen genutzt, um den Kontrollfluß zu bestimmen.
- Boolean-Literale sind:

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

2.2. Fließkomma-Typen

- **float** 32 Bit
- **double** 64 Bit

Die Fließkommatypen entsprechen der in **IEEE 754** festgelegten Spezifikation, d. h. ∞ , $-\infty$, $+0$, -0 und **NaN** existieren als gesonderte „Werte“:

```
double a = 1.0 / 0.0 // ergibt +Inf
double b = 1.0 / -0.0 // ergibt -Inf
double c = 0.0 / 0.0 // ergibt NaN
boolean x = 0.0 > -0.0 // ergibt false
boolean y = (c != c) // ergibt true
```

2.2. Kleinigkeiten zu `float` und `double` ...

Die positive und negative Null sind tatsächlich identisch und nicht etwa verschieden.

Vergleiche mit **NaN** sind grundsätzlich **false**.

Jegliche arithmetische Operation mit **NaN** ergibt als Ergebnis wiederum **NaN**.

$\infty - \infty$ ergibt **NaN**.

2.2. Fließkommaliterale

Fließkommazahlen können auf verschiedene Weise repräsentiert werden:

- Float:

```
float a = 1e1f;  
float b = 2.f;  
float c = 6.022137e+23f;
```

- Double:

```
double a = .3;  
double b = 1e-9d;  
double c = 1e137;
```

```
double d = Double.longBitsToDouble(0x400921FB54442D18L);
```

2.3. Strings

- **java.lang.String** ist eine Klasse, kein primitiver Typ, soll hier dennoch behandelt werden, da Java dieser Klasse einige Besonderheiten einräumt
- "Anführungszeichen" erzeugen automatisch neue String-Objekte:

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

2.3. Strings ...

- Strings sind unveränderlich
- Jedes Objekt hat eine Methode `toString()`, die überschrieben werden kann und in aller Regel eine Stringrepräsentation des Objektes liefert:

```
|| String myString = myObj.toString();
```

- Strings sind **keine** Arrays von Zeichen (`char`) und werden **nicht** mit einem Nullzeichen (`'\u0000'`) beendet

2.3. Strings ...

- Strings können auf verschiedene Weise aneinandergehängt werden:

```
|| "Hello, ".concat("World!");
```

oder

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

oder

```
|| String fs =  
||     String.format("Hallo, %s!", "Welt");
```

2.4. Konvertierungen

Es gibt in Java verschiedene Arten von Konvertierungen, die manchmal interessante Überraschungen bereithalten:

- Explizite Konvertierung
- String-Konvertierung
- Widening conversion
- Promotion
- Konvertierung beim Methodenaufruf

2.4.1. Explizite Konvertierung (*Casting*)

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

Dies wäre ohne den Cast-Operator ein Compilerfehler, da nicht jeder `float`-Wert in einen `int` paßt.

2.4.2. String-Konvertierung

```
int i = (int)12.5f;  
System.out.println("(int)12.5f==" + i);
```

Bei der Verwendung von + zum Verknüpfen zweier Objekte von denen eines ein String ist, wird das andere Objekt automatisch in einen String umgewandelt; bei primitiven Datentypen über die `toString()`-Methode des zugehörigen Wrapper-Objekts.

2.4.3. Widening conversion

```
int i = (int)12.5f;  
System.out.println("(int)12.5f==" + i);  
float f = i;
```

Hier ist die Konvertierung von **int** zu **float** problemlos möglich, da **int** einen (subjektiv) kleineren Wertebereich beschreibt als **float**.

2.4.3. Widening conversion ...

Die verschiedenen Konvertierungen, die unter *Widening conversion* fallen, sind folgende:

- **byte** zu **short**, **int**, **long**, **float** oder **double**
- **short** zu **int**, **long**, **float** oder **double**
- **char** zu **int**, **long**, **float** oder **double**
- **int** zu **long**, **float** oder **double**
- **long** zu **float** oder **double**
- **float** zu **double**.

2.4.3. Widening conversion ...

Bei der Konvertierung zwischen Integer-Typen geht keinerlei Information verloren, die Zahl hinterher ist exakt die selbe. Bei der Konvertierung eines Integer-Typen zu `float` oder `double` hingegen **kann** Genauigkeit in den letzten Bits verloren gehen:

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

Dieses Beispiel gibt `-46` aus, da `float` hier nicht die Genauigkeit von `int` erreicht.

2.4.4. Numeric Promotion

```
int i = (int)12.5f;
System.out.println("(int)12.5f==" + i);
float f = i;
f = f * i;
```

Beim letzten Ausdruck wird `i` zunächst in einen `float` umgewandelt, da der zweite Operand der Multiplikation hier (`f`) vom Typ `float` ist. Nach der Konvertierung, die *Promotion* genannt wird, ist die Operation eine Multiplikation zweier `floats`.

2.4.5. Konvertierung beim Methodenaufruf

```
int i = (int)12.5f;
System.out.println("(int)12.5f==" + i);
float f = i;
f = f * i;
double d = Math.sin(f);
```

Hierbei wird `f` automatisch von `float` nach `double` umgewandelt, da `Math.sin()` nur `double` als Argument akzeptiert.

2.5. Arrays

- Feld von Variablen genau einen Typs
- Die Länge gehört nicht zum Typ:

```
|| int [] intArr;
```

- Array-Instanzen haben allerdings immer eine konstante Länge, die man beispielsweise folgendermaßen festlegen kann:

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

- Die folgenden Deklarationen sind äquivalent:

```
||| int [] intArr;  
||| int intArr [];  
||| int [] intArr [];
```

2.5. Arrays ...

- Bei vorgegebenem Inhalt ist die Größe des Arrays implizit festgelegt:

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

- Zugriff auf Arrays erfolgt über `intArr[i]`, wobei `i` der Index des jeweiligen Elementes ist
- Arrays sind grundsätzlich nullbasiert, d. h. die Indizes eines Arrays der Länge n laufen von 0 bis $n - 1$

3. Operatoren

Die 37 Operatoren lassen sich grob in Kategorien einteilen:

- Vergleich: `>`, `<`, `<=`, `>=`, `!=`, `==`
- Boole'sche: `!`, `&&`, `||`
- Bitweise: `~`, `&`, `|`, `^`, `>>`, `<<`, `>>>`
- Arithmetisch: `+`, `-`, `*`, `/`, `%`, `++`, `--`
- Bedingung: `?` und `:`
- Zuweisung: `=`, `+=`, `-=`, `*=`, `/=`, `&=`, `|=`, `^=`, `%=`, `<<=`, `>>=`,
`>>>=`

4. Kontrollstrukturen und Anweisungen

- Größtenteils aus C/C++ entlehnt
- **Leere Anweisung:** `;`
- **Bedingte Anweisungen:** `if`, `switch`
- **Iterations-Anweisungen:** `while`, `do`, `for`
- **Anweisungen für abrupten Ausstieg aus dem Kontrollfluß:** `break`, `continue`, `return`
- **Fehlerbehandlung:** `throw`, `try` (später)

4.1. Die leere Anweisung

|| ;

- Tut nichts, es gibt sie trotzdem

4.2. Die `if`-Anweisung

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

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

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

4.3. Die switch-Anweisung

```
switch (i) {  
    case 1: doSomething();  
           break;  
    case 2: doSomethingElse();  
           break;  
    case 3: doSomethingDifferent();  
           break;  
    default: doAnything();  
}
```

4.4. Die while-Anweisung

```
while (true) {  
    // Endlosschleife  
}
```

```
while (i < 5) {  
    doSomething();  
    i++;  
    // nicht ganz endlos :)  
}
```

4.5. Die do-Anweisung

```
public static String toHexString(int i) {
    StringBuffer buf = new StringBuffer(8);
    do {
        buf.append(Character.forDigit(i & 0xF, 16));
        i >>= 4;
    } while (i != 0);
    return buf.reverse().toString();
}
```

4.6. Die for-Anweisung

- „Normale“ for-Anweisung:

```
for (int i = 0; i < 100; i++) {  
    doSomething();  
}
```

- „Erweiterte“ for-Anweisung:

```
int sum(int[] a) {  
    int sum = 0;  
    for (int i : a)  
        sum += i;  
    return sum;  
}
```

4.7. Die break-Anweisung

- Springt sofort aus dem aktuellen Block (**switch**, **while**, **do** oder **for**)

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

- Optional kann ein Label angegeben werden, welches bestimmt, aus welchem Block genau **break** herausspringt:

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

4.8. Die `continue`-Anweisung

- Darf nur in `while`, `do` oder `for` auftauchen
- Der Programmablauf setzt bei der nächsten Iteration fort

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

- `continue` kann wie `break` ein Label erhalten, welches bestimmt, wo weitergemacht wird

4.9. Die `return`-Anweisung

- Springt aus der aktuellen Methode oder dem Konstruktor
- Muß beim Aussprung einen Wert zurückgeben (nicht bei `void`-Methoden oder Konstruktoren)

```
void test() {  
    // void braucht keinen Rückgabewert,  
    return;           // damit geht das  
}
```

```
int test2() {  
    return 2; // muß int zurückgeben  
}
```

5. Objektorientierung

Erinnern wir uns an Programmierungstechnik I, ADT Stack:

```
unit stacka;  
  
interface  
  
const maxs      = 5;  
      errorel   = -99  
  
type STACK =  
      record  
          elts : array [1..maxs] of integer;  
          ptr  : integer;  
      end;
```

5. Objektorientierung ...

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

```
procedure pop(var s : STACK);
```

```
function top(s : STACK) : integer;
```

```
procedure push(var s : STACK; e : integer);
```

```
implementation
```

```
{ ... }
```

Eine naive Umsetzung dieses Codes in Java wäre beispielsweise folgende ...

5. Objektorientierung ...

Unsere Datenstruktur (die Typdefinition aus Pascal):

```
public class Stack {  
    public static final int maxs = 5;  
  
    public int[] elts = new int[maxs];  
  
    public int ptr;  
}
```

5. Objektorientierung ...

Weiterhin unsere Umsetzung der eigentlichen Unit.
Zunächst die Konstanten:

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

5. Objektorientierung ...

Umsetzung der einzelnen Funktionen und Prozeduren:

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

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

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

5. Objektorientierung ...

Umsetzung der einzelnen Funktionen und Prozeduren:

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

5. Objektorientierung ...

Umsetzung der einzelnen Funktionen und Prozeduren:

```
public static int top(Stack s) {
    if (isEmpty(s)) {
        System.out.println("Stack empty");
        return errore1;
    } else if (isError(s)) {
        System.out.println("Stack broken");
        return errore1;
    } else
        return s.elts[s.ptr];
}
```

5. Objektorientierung ...

Umsetzung der einzelnen Funktionen und Prozeduren:

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

5. Objektorientierung ...

Umsetzung der einzelnen Funktionen und Prozeduren:

```
public static void push(Stack s, int i) {  
    if (i == errorel) {  
        System.out.println("Bad Element");  
        // do nothing  
    } else if (isError(s)) {  
        System.out.println("Stack broken");  
        // do nothing  
    }  
}
```

5. Objektorientierung ...

Umsetzung der einzelnen Funktionen und Prozeduren:

```
        else if (s.ptr == maxs) {
            System.out.println("Stack Overflow");
            mkErrorStack(s);
        } else {
            s.ptr++;
            s.elts[s.ptr] = i;
        }
    }
}
```

5. Objektorientierung ...

Ist das nun eigentlich objektorientiert?

5. Objektorientierung ...

Ist das nun eigentlich objektorientiert?

Nicht wirklich.

5. Objektorientierung ...

Schauen wir uns doch einmal an, wo wir hier genau mit Objekten gearbeitet haben:

- **Gar nicht**
- Unser „Objekt“ ist lediglich eine Datenstruktur
- Das sollte doch besser gehen

6. Klassen

- Eine Klasse in Java kapselt Datenstruktur und zugehörige Funktionen (**Methoden**)
- Klassen, sowie ihre Daten und Methoden haben einen Sichtbarkeitsbereich:
 - Standardmäßig sind Klassen und ihre Member nur innerhalb des selben Packages sichtbar
 - **public**: Überall sichtbar
 - **private**: (nur für Member) Sichtbar in der Klasse, in der er deklariert wurde, außerhalb jedoch nicht
 - **protected**: (nur für Member) Sichtbar in der Klasse selbst und in davon abgeleiteten Klassen
- **Alles** in Java ist eine Klasse
- „Klassen in Massen“

6. Klassen ...

- Klassen können zweierlei Arten von Mitgliedern enthalten:
 - **Statische** (**static**) Mitglieder gehören zur Klasse
 - **Nicht-statische** Mitglieder gehören zu einer Instanz der Klasse, d. h. einem Objekt
- Weiterhin gibt es verschiedene Typen von Mitgliedern:
 - Konstruktoren
 - `finalize()`
 - Methoden
 - Felder
 - weitere Klassen
- ein Konstruktor ist, auch wenn er eine Instanz erzeugt, **nicht** statisch

6. Methoden und Konstruktoren

- **Methoden** werden folgendermaßen deklariert:
Typ Bezeichner(Parameter)
- Es kann mehrere Methoden mit gleichem Namen aber unterschiedlicher Signatur geben
- Wir werden gleich ein paar Beispiele zu Methoden sehen
- **Konstruktoren** sind besondere Methoden, die dazu verwendet werden, ein Objekt zu initialisieren
- Konstruktoren heißen **immer** so wie die Klasse
- Es kann auch mehrere Konstruktoren mit unterschiedlicher Signatur geben

6. Unser Beispiel

Fangen wir nun einmal an, das Beispiel von vorhin umzuarbeiten:

```
public class Stack {  
    public static final int maxs = 5;  
  
    public int[] elts = new int[maxs];  
  
    public int ptr;  
}
```

6. Unser Beispiel

Die Klasse Stack können wir schon fast so behalten:

```
public class Stack {  
    public static final int maxs = 5;  
  
    public int[] elts = new int[maxs];  
  
    public int ptr;
```

allerdings ist sie hier noch nicht zu Ende.

6. Unser Beispiel

Die Konstanten aus der alten Hilfsklasse können wir mit übernehmen:

```
public class Stack {  
    public static final int maxs = 5;  
  
    public static final int errorel = -99;  
  
    public int[] elts = new int[maxs];  
  
    public int ptr;
```

6. Unser Beispiel ...

Weiterhin müssen wir an den Methoden noch etwas ändern:

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

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

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

6. Unser Beispiel ...

Weiterhin müssen wir an den Methoden noch etwas ändern:

```
public boolean isEmpty() {  
    return ptr == 0;  
}  
  
public void mkErrorStack() {  
    ptr = maxs + 1;  
}  
  
public boolean isError() {  
    return ptr > maxs;  
}
```

6. Unser Beispiel ...

Weiterhin müssen wir an den Methoden noch etwas ändern:

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

6. Unser Beispiel ...

Weiterhin müssen wir an den Methoden noch etwas ändern:

```
public void empty() {  
    ptr = 0;  
}
```

6. Unser Beispiel ...

Weiterhin müssen wir an den Methoden noch etwas ändern:

```
public static int top(Stack s) {
    if (isEmpty(s)) {
        System.out.println("Stack empty");
        return errore1;
    } else if (isError(s)) {
        System.out.println("Stack broken");
        return errore1;
    } else
        return s.elts[s.ptr];
}
```

6. Unser Beispiel ...

Weiterhin müssen wir an den Methoden noch etwas ändern:

```
public int top() {
    if (isEmpty()) {
        System.out.println("Stack empty");
        return errore1;
    } else if (isError()) {
        System.out.println("Stack broken");
        return errore1;
    } else
        return elts[ptr];
}
```

6. Unser Beispiel ...

Weiterhin müssen wir an den Methoden noch etwas ändern:

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

6. Unser Beispiel ...

Weiterhin müssen wir an den Methoden noch etwas ändern:

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

6. Unser Beispiel ...

Weiterhin müssen wir an den Methoden noch etwas ändern:

```
public static void push(Stack s, int i) {
    if (i == errorel) {
        System.out.println("Bad Element");
        // do nothing
    } else if (isError(s)) {
        System.out.println("Stack broken");
        // do nothing
    }
}
```

6. Unser Beispiel ...

Weiterhin müssen wir an den Methoden noch etwas ändern:

```
public void push(int i) {
    if (i == errorel) {
        System.out.println("Bad Element");
        // do nothing
    } else if (isError()) {
        System.out.println("Stack broken");
        // do nothing
    }
}
```

6. Unser Beispiel ...

Weiterhin müssen wir an den Methoden noch etwas ändern:

```
        else if (s.ptr == maxs) {
            System.out.println("Stack Overflow");
            mkErrorStack(s);
        } else {
            s.ptr++;
            s.elts[s.ptr] = i;
        }
    }
}
```

6. Unser Beispiel ...

Weiterhin müssen wir an den Methoden noch etwas ändern:

```
        else if (ptr == maxs) {
            System.out.println("Stack Overflow");
            mkErrorStack();
        } else {
            ptr++;
            elts[ptr] = i;
        }
    }
}
```

6. Vererbung

- Klassen können von anderen Klassen **abgeleitet** werden (*Vererbung*). Dabei werden alle Member und Methoden der „Elternklasse“ auf das „Kind“ vererbt; dieses kann sie dann auch teilweise überschreiben oder um neue ergänzen
- Vererbung ist nur von **genau einer Klasse** möglich
- Und das ganze sieht dann so aus:

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

6. Weitere Feinheiten von Klassen

- **Abstrakte Klassen** (über **abstract**) sind Klassen, die unvollständig implementiert sind und können abstrakte **Methoden** beinhalten, die zwar deklariert, aber nicht implementiert sind
- **Finale Klassen** (über **final**) sind Klassen, von denen keine abgeleiteten Klassen erstellt werden können
- **Innere Klassen** sind Klassen, die innerhalb einer anderen Klasse deklariert und normalerweise an eine Instanz dieser Klasse gebunden sind
- **Lokale innere Klassen** sind Klassen, die innerhalb einer Methode deklariert werden
- **Anonyme Klassen** sind lokale innere Klassen, die nicht benannt wurden

6. Interfaces

- Interfaces sind im Grunde Klassen, die keinerlei Implementation beinhalten
- Als Member sind lediglich Konstanten (**static final**) erlaubt und Methodensignaturen
- Deklaration:

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

6. Interfaces ...

- Interfaces können von mehreren anderen Interfaces **erben**
- Klassen können mehrere Interfaces **implementieren**:

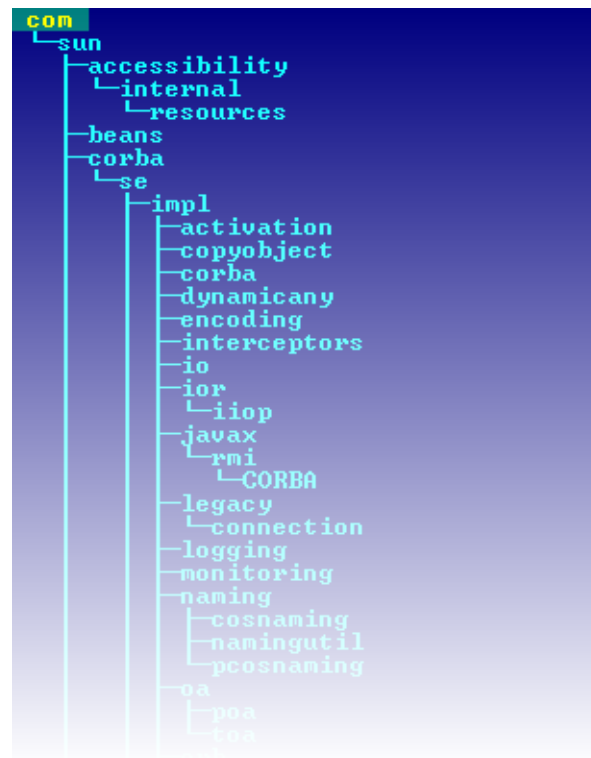
```
class MyClass implements MyInterface {  
    // vom Interface definierte Methode  
    // muß implementiert werden  
    void doSomething() {  
        // ...  
    }  
  
    // ...  
}
```

7. Packages

- Organisieren Klassen hierarchisch in Namespaces
- Entsprechen einer Ordnerstruktur im Dateisystem
- **Benennungsschema für öffentliche Packages:**
Domain, komponentenweise rückwärts, danach eigene Ordnung:
 - de.hypftier.vortrag.java.test
 - com.sun.sunsoft.DOE
- Die von Java selbst bereitgestellten Packages folgen hingegen einem nicht ganz so expliziten, wenngleich dennoch logischem Benennungsschema:
 - java.util
 - java.lang.reflect
 - ...

7. Packages ...

Ein kleiner Ausschnitt aus dem JRE-1.5-Packagebaum:



7. Packages ...

- Packages werden im Quelltext einer Klassendatei durch

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

gekennzeichnet

- **Importieren** kann man Packages über die **import**-Anweisung, entweder ganze Packages:

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

oder aber einzelne Klassen:

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

8. Fehlerbehandlung

- Wird über sog. **Exceptions** (*Ausnahmen*) realisiert
- Ausnahmen werden über **throw** ausgelöst und unterbrechen den Kontrollfluß des Programms
- Die meisten Ausnahmen **müssen** behandelt oder explizit weitergereicht werden
- Weiterreichen einer Ausnahme erfolgt implizit, wenn sie nicht behandelt wird
- Die meisten Ausnahmen erfordern allerdings eine explizite Deklaration des Weiterreichens:

```
|| void doSomething() throws Exception {  
||     // ...  
|| }
```

8. Fehlerbehandlung ...

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

```
try {  
    // Code, der eine Exception wirft  
} catch (SpecialException) {  
    // Fehlerbehandlung  
} catch (Exception) {  
    // mehr Fehlerbehandlung  
} finally {  
    // Aufräumarbeiten  
}
```

8. Fehlerbehandlung ...

- **Errors** sind ähnlich wie Ausnahmen, werden allerdings im Normalfall von der JVM ausgelöst und signalisieren einen schwerwiegenden Fehler, der nicht zu beheben ist: z. B. **OutOfMemoryError** oder **StackOverflowError**
- Wenn Anweisungen wie **return** oder **break** innerhalb eines **try**-Blocks aufgerufen werden, so werden zunächst alle ausstehenden **finally**-Blöcke ausgeführt, bevor der Sprung ausgeführt wird

Literatur und Quellen

- The Java Language Specification, Third Edition

(http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html)

- The Java Tutorials: Learning the Java Language

(<http://java.sun.com/docs/books/tutorial/java/index.html>)

Dank an: Dr. Elmar Ludwig, Andreas Kohn und Andreas Tschritter für hilfreichen Rat zu Java, sowie Andreas Dähn für den Rohbau dieser Folien und beständige Hilfe bei \LaTeX .

Danke für die Aufmerksamkeit
