Objektorientierte Programmierung mit Java - Einführung

- Hintergrund
- Objektorientierung mit Java
- Programmieren mit Java
- Ein Java-Programm
- Packages

Geschichte

1990 Sun Microsystems

- Programmiersprache für Consumer-Geräte gesucht
- für verteilte Anwendungen in heterogenen Netzen
- C/C++ erscheinen nicht geeignet ⇒ Projektstart Oak 1991

1994 Oak-Projekt

- mangels Anwendung gefährdet
- Markt scheint nicht reif für PDAs, TV-Set-Top-Boxen etc.

1995 Neustart als Programmiersprache für Web-Inhalte

- Entwicklung des Applet-Konzepts
- Integration in verschiedene Web-Browser (z.B. Netscape)

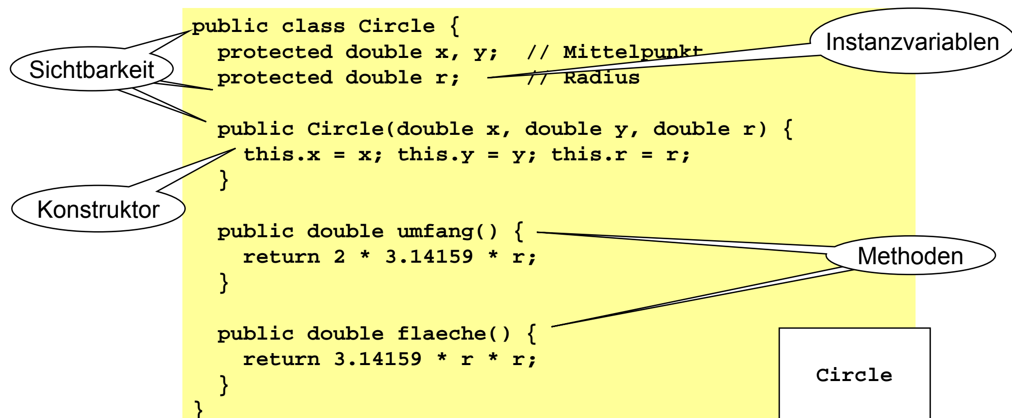
2001 Hohe Popularität

- nicht zuletzt dank aggressivem Marketing → rasantes Wachstum
- aber auch Probleme: Performance, Kompatibilät, ...

Entwurfskriterien

- **Einfach- und Kompaktheit**
  - C++ als syntaktische Grundlage
  - geringerer Sprachumfang und weniger komplexe Konstrukte
- **Zuverlässigkeit**
  - Vermeiden von unsicheren Konstrukten (z.B. Zeiger in C++)
  - Referenzen mit Garbage Collection (GC) statt Zeiger
  - keine Mehrfachvererbung
  - kein Überladen von Operatoren
- **Sicherheit**
  - unbekannte Anwendung darf kein unkalkulierbares Risiko darstellen
  - GC statt Zeiger ⇒ kein Zugriff auf beliebige Speicheradresse!
  - Sandbox-Prinzip ⇒ Anwendung arbeitet nur in zugewiesenem Bereich!
- **Plattform-Unabhängigkeit**
  - durch Interpretation von Bytecode

Klasse



- **Grundlage eines Java Programms**
  - Instanzvariable = Attribut der Klasse
  - Methode = Implementierte Operation der Klasse
  - Konstruktor = Erzeugt ein Objekt der Klasse (spezielle Operation)

Objekt

```
Circle kreis;

kreis = new Circle(8, 4, 2.5);

System.out.println("Radius: " + kreis.r);

System.out.println("Umfang: " + kreis.umfang());
```


- **Klasse ist neuer Typ**
  - von dem Variablen deklariert werden können
  - einer Variablen kann dann ein Objekt der Klasse zugewiesen werden
- **Erzeugen von Objekten**
  - Anweisung `new` erzeugt ein Objekt mit dem angegebenen Konstruktor
  - das Objekt hat alle Operationen/Instanzvariablen der Klasse

Instanzvariablen

```
int g, h;

double zahl = 2.5;

Circle cObject;

static final float PI = 3.1415f;
```


- **Instanzvariablen**
  - werden in einem Block am Anfang einer Klasse deklariert (Konvention!)
  - zusätzlich können an beliebiger Stelle einer Klasse/Anweisungsblocks Hilfsvariablen deklariert werden
- **Java ist Sprache mit starker Typbindung**
  - einer Variablen vom Typ `X` dürfen nur Werte vom Typ `X` zugewiesen werden

Methoden

Rückgabetyp

Name der Methode

Liste der formalen Parameter

Zwingende Rückgabe des Ergebnisses mit `return`

Prozedur ohne Rückgabewert!

Kein `return` nötig!

```
public float gewicht(float d, int b)
{
  ...
  return something;
}

public void print()
{
  ...
}
```

- **Methoden können nicht geschachtelt werden**
  - wie z.B. in Modula-3
- **Aufruf der Methoden:**

Name des Objekts

Speichert Rückgabewert!

Aktuelle Parameter

```
y = pyramid.gewicht(1.8, 54)

printer.print()
```


Überladen von Methoden

Verschiebt Figur relativ um `dx`/`dy`

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

public void verschiebe(Point p)
{  x = p.x; y = p.y; }
```

Verschiebt Figur absolut an Position von `p`

- **Überladen von Methoden**
  - Methoden haben den gleichen Namen
  - unterscheiden sich nur durch ihre Signatur
    - Anzahl, Position und Typ der formalen Parameter
    - des Rückgabetyps
- **Sinnvoller Einsatz**
  - wenn Operationen ähnliche Semantik haben
  - aber für verschiedene Eingabe-/Ausgabedaten realisiert werden sollen

Konstruktoren

```
public Circle()
{
  this.x = 0; this.y = 0; r = 1;
}

public Circle(double x, double y, double r)
{
  this();
  this.x = x; this.y = y; this.r = r;
}
```


- **Spezielle Methode**
  - um ein Objekt zu erzeugen
- **Eine Klasse kann mehrere Konstruktoren haben**
  - unterscheiden sich durch ihre Signatur (Anzahl, Position und Typ der Parameter)
  - entspricht dem Überladen von Konstruktoren

Klassenvariable/-methode

```
public class Circle
{
  private static int numOfCircles = 0;

  public Circle(double x, double y, double r)
  { ... numOfCircles++; }

  public double flaeche()
  { return 3.14159 * r * r; }

  public static double flaeche(double radius)
  {  return 3.14159 * radius * radius; }      }
```


- **Klassenvariable**
  - existiert nur einmal für alle Objekte der Klasse
  - kann als (über Klassenname eindeutige) "globale" Variable betrachtet werden
- **Klassenmethode**
  - kein Objekt (Instanz) für den Aufruf erforderlich:


```
System.out.println("Kreisflaeche: " + Circle.flaeche(3.5));
```

Datenkapselung

```
public class Circle
{
  private   double x, y;  // Mittelpunkt
  protected double r;     // Radius

  ... // Konstruktor etc.

  public void setRadius(double r)
  {  this.r = r;  }

  public double getRadius()
  {  return this.r;  }                }
```


- **Datenkapselung**
  - die Daten (Variablen) eines Objekts sollen verborgen bleiben
  - und nur über entsprechende konsistenz-wahrende Methoden zugänglich sein
  - Schutz vor wissentlichem und unwissentlichem Mißbrauch
  - spätere Änderung der internen Realisierung bei gleicher Schnittstelle möglich
- **Java erlaubt gezielte Steuerung der Sichtbarkeit**
  - von Variablen, Methoden und sogar Klassen
  - dazu dienen sog. Sichtbarkeitsmodifikatoren (`public, protected, private`)

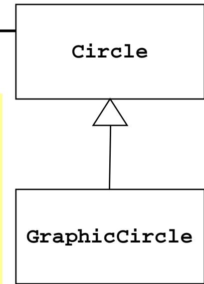
Sichtbarkeit

Modifikator	Angewendet auf	
	Klasse	Element
Kein Modifikator	Nur innerhalb des Packages sichtbar	Nur innerhalb des Packages sichtbar
`public`	Sichtbar, überall wo es ihr Package ist	Sichtbar, überall wo es seine Klasse ist
`private`	Nicht anwendbar	Nur innerhalb seiner eigenen Klasse sichtbar
`protected`	Nicht anwendbar	Innerhalb seines Packages und seiner Unterklassen sichtbar

ACHTUNG: Package-Zugehörigkeit beeinflußt Sichtbarkeit

- Klassen entsprechend ihren Abhängigkeiten in Packages zusammenfassen
- "Verwendung" des namenlosen Standard-Packages vermeiden!

Vererbung




```
public class GraphicCircle extends Circle
{
  private Color outline, fill;

  public GraphicCircle(double x, double y, double r,
                       Color ol, Color f)
  { ... }

  public void draw(DrawWindow dw)
  { dw.drawCircle(x, y, r, outline, fill); }
}
```


Einfachvererbung

- neue Klasse durch Erweiterung einer existierenden Klasse
- neue Klasse hat alle Instanzvariablen und Methoden der Superklasse
- als `private` deklarierte Elemente werden jedoch nicht vererbt

Polymorphe Verwendung

```
GraphicCircle gc;
double flaeche;

gc = new GraphicCircle(3, 4, 3.2, rot, blau);

gc.draw(window);
flaeche = gc.flaeche();

Circle c = gc;

flaeche = c.flaeche();

c.draw(window);
```


Objekt der Klasse `GraphicCircle`

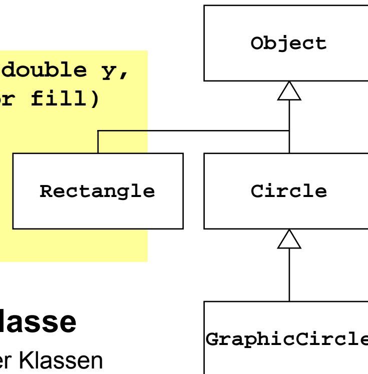
- ist auch ein Objekt der Klasse `Circle`
- kann daher genauso wie ein Objekt dieser Klasse verwendet werden

Superklasse

Muß als erste Anweisung stehen!

Aufruf des Konstruktors der Superklasse

```
public GraphicCircle(double x, double y,
  double r, Color outline, Color fill)
{
  super(x, y, r);
  this.outline = outline;
  this.fill = fill;
}
```



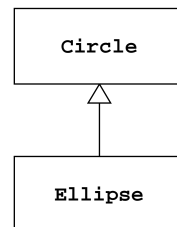
- **Jede Klasse in Java hat eine Superklasse**
  - Ausnahme: Klasse `Object` = Superklasse aller Klassen
  - Schlüsselwort `super` erlaubt Zugriff auf Superklasse
  - `this` und `super` werden analog verwendet
- **Konstruktor der Superklasse wird immer aufgerufen**
  - entweder explizit durch Programmierer wie im obigen Beispiel
  - oder implizit durch Aufruf des Standard-Konstruktors: `super()`

Redefinieren von Methoden

Redefiniert äquivalente Methode aus `Circle`

Zugriff auf ursprüngliche Definition aus `Circle`

```
public class Ellipse extends Circle {
  ...
  public double flaeche() {
    super.flaeche() + ...
    return ... // Flaeche einer Ellipse
  }
}
```



- **Redefinition einer Methode**
  - Subklasse kann neue Implementierung für Methode der Superklasse anbieten
  - Name, formale Parameter und Rückgabetyp (= Signatur) müssen gleich sein
  - Methode kann komplett neu definiert oder erweitert (d.h. ursprüngliche Methode wird mittels `super` aufgerufen) werden

Dynamische Bindung

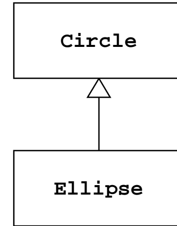
```
Circle  c;
Ellipse e;
Double  area;

c = new Circle(15, 15, 3);
e = new Ellipse(10, 10, 2, 3);

area = c.flaeche();

area = e.flaeche();

c = e;
area = c.flaeche();
```

Kreisfläche

Ellipsenfläche

Ellipsenfläche!

Zuweisung an Variable der Superklasse!



Methodenaufruf

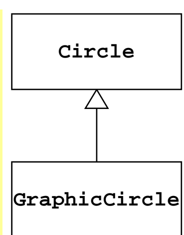
- die aufzurufende Methode wird erst zur Laufzeit ermittelt (gebunden)
- Laufzeittyp (dynamischer Typ) liefert die gültige Implementierung
- statische Zuweisung auf eine Variable der Superklasse hat keinen Einfluss

Verdeckte Instanzvariable

```
public class GraphicCircle extends Circle {
  float r; // Aufloesung der Darstellung

  ... // Deklaration Konstruktor etc.

  public void setResolution(float res) {
    r = res;
  }

}
```

Verdeckt Radius `r` aus Klasse `Circle`

Setzt lokale Variable `r`



Instanzvariablen

- Instanzvariablen werden in Java nicht dynamisch gebunden!
- statischer Typ entscheidet über geltende Definition

final

```
public class Circle {

  static final double PI = 3.1415;

  public final double umfang() {
    return 2 * PI * r;
  }

  public final double flaeche() {
    return PI * r * r;
  }
}
```

`static final` signalisiert dem Übersetzer eine Konstante

`final` erlaubt genau eine Wertzuweisung!

Übersetzer kann Ausdruck `2 * PI` bereits berechnen

- **Fixierung mit `final`**
  - keine weitere Redefinition der Variable/der Methode in einer Subklasse
- **Gründe**
  - Dynamische Bindung ist teuer (Ausführungszeit/Lookup zur Laufzeit)
  - Übersetzer kann Berechnungen optimieren
  - Manchmal ist es sinnvoll eine Klasse/Methode unveränderbar zu machen
- **Klassenmethoden und `private`-Methoden**
  - alle mit **`static`/`private`** gekennzeichnete Methoden sind implizit `final`

Abstrakte Klasse

Shape

```
public abstract class Shape {

  public abstract double umfang();

  public abstract double flaeche();
}

...
Shape c = new Circle(4, 6, 3.7);
Shape r = new Rectangle(3, 5);
```

Superklasse aller Figuren (Kreis, Rechteck, ...)

Hier keine sinnvolle Definition möglich

Erlaubt Definition einer gemeinsamen Schnittstelle!

Müssen beide `umfang()` und `flaeche()` implementieren

- **Abstrakte Methode**
  - definiert nur die Signatur einer Methode
  - bietet keine Implementierung an
  - muss in einer Subklasse implementiert werden
- **Abstrakte Klasse**
  - eine Klasse mit einer abstrakten Methode ist automatisch auch abstrakt
  - es können keine Objekte einer abstrakten Klasse erzeugt werden

Interface

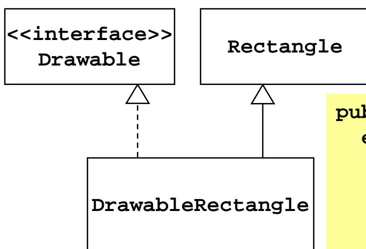
Objekte des Typs **Drawable** müssen das Interface implementieren

```
public interface Drawable {
  public static final int WHITE = 255;

  public void setColor(Color c);

  public void draw(DrawWindow dw); }
```

Konstanten-deklaration

- **Java kennt keine Mehrfachvererbung**
  - Implementierung einer Methode kann nur von einer Superklasse geerbt werden
  - viele semantische Probleme der Mehrfachvererbung werden so vermieden
- **Interface**
  - definiert nur die abstrakte Schnittstelle einer Klasse
  - sämtliche Methoden sind abstrakt, sonst nur Deklaration von Konstanten zulässig
- **Interface-Vererbung ersetzt Mehrfachvererbung**
  - Klasse kann beliebig viele Interfaces implementieren
  - ein Interface kann von einem oder mehreren anderen Interfaces erben

Interface - Beispiel




```
public class DrawableRectangle
  extends Rectangle implements Drawable {
    public Color c;       // Farbe des Rechtecks
    public double x, y;   // Position

    public DrawableRectangle(double w, double h) {
      super(w, h);
    }

  public void setColor(Color c) { this.c = c; }

  public void draw(DrawWindow dw) { ... }
}
```

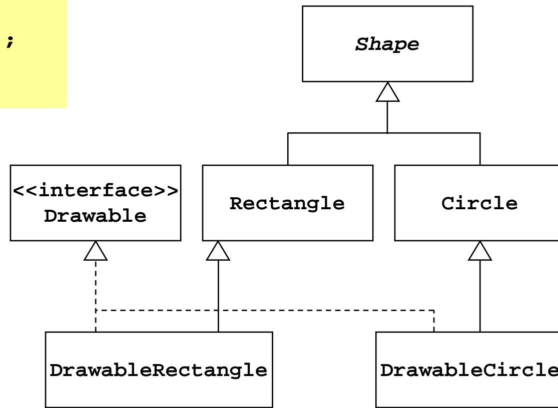
Konstruktor der Superklasse **Rectangle**

- **Konkrete Klasse `DrawableRectangle`**
  - erbt Attribute und implementierte Methoden aus der Superklasse **`Rectangle`**
  - muss alle Methoden des Interface **`Drawable`** implementieren
  - kann theoretisch beliebig viele Interfaces implementieren

Interface - Polymorphe Verwendung

```
Drawable d;

d = new DrawableCircle(3, 4, 5.2);
d.setColor(red);
d.draw(window);

d = new DrawableRectangle(3, 5);
d.setColor(blue);
d.draw(window);
```

Können als Objekte vom Typ `Drawable` verwendet werden



- **Variablen**
  - können auch den Typ eines Interfaces haben
  - dieser können alle Objekte zugewiesen werden, die das Interface implementieren

Programmieren in Java

```
public class BerechneGGT {
  public int g;

  public static void main(String[] args) {
    BerechneGGT ggtObject;
    ggtObject   = new BerechneGGT();
    ggtObject.g = ggtObject.ggt(90, 54);
    System.out.println(ggtObject.g);
  }

  public int ggt(int a, int b) {
    while (a != b) {
      if (a > b)
       a = a - b;
      else
        b = b - a;
    }
    return a;
  }
}
```

Bezeichner dürfen Buchstaben, Unterstrich und Ziffern enthalten, müssen aber mit einem Buchstaben beginnen.

Java-Compiler

```
>javac BerechneGGT.java

>java BerechneGGT
18

>_
```

Java-Interpreter

Ausführung von der Kommandozeile

{ ... } definieren Anfang und Ende eines Anweisungsblocks.

Konventionen für Bezeichner

- **Innere Wortteile werden zwecks Lesbarkeit groß geschrieben**
- **Klassennamen**
  - Auch das erste Wort wird groß geschrieben, z.B. `LargeTemperatureTable`
- **Variablennamen und Methodennamen**
  - Das erste Wort wird klein geschrieben, z.B. `oldWeight`, `noOfEnquiries`
  - Methodennamen sollen Aktivität anzeigen, z.B. `addEntry(), changeSize()`
- **Konstantennamen**
  - Werden komplett groß geschrieben, z.B. `PI, ENTRIES`
- **Package-Namen**
  - Werden komplett klein geschrieben (und sollten nur aus einem Wort bestehen)
- **WICHTIG: Java unterscheidet Groß-/Kleinschreibung!**

Kommentare

```
/* HelloWorld.java
   Erstellt: 13.08.99
   Geändert: 22.10.99 */

...
System.out.println("Hallo!);    // Gibt das Wort Hallo aus!
...
```

Alles zwischen `/*` und `*/` ist Kommentar!

Alles in der Zeile nach `//` ist Kommentar!

- **Java kennt zwei Arten des Kommentars**
  - den Kommentarblock und den Kommentar am Zeilenende
  - deren Inhalt wird vom Übersetzer ignoriert.
- **Diese Möglichkeiten sollten auch genutzt werden!**
- **Javadoc-Dokumentation**
  - wird aus annotierten Kommentaren erzeugt
  - diese werden per Konvention durch `/**` eingeleitet!

Primitive Datentypen

- **Java bietet eine Reihe von vordefinierten primitiven Typen**
  - Ganze Zahlen (alle mit Vorzeichen):

byte	8 bit	-128 ... 127
short	16 bit	-32768 ... 32767
int	32 bit	-2147483648 ... 2147483647
long	64 bit	etwa ± 10^{18}

  - Gleitkommazahlen (IEEE 754):

float	32 bit	etwa ±3.4E+38 ... ±1.4E-45
double	64 bit	etwa ±1.79E+308 ... ±4.94E-324

  - Zeichen

char	**16 bit**	Unicode-Zeichen: \u0000 ... \uFFFF

  - Logische Werte

boolean	1 bit	true, false

- **Variable benötigt genau den angegebenen Speicher**

© M. Schnizler, Th. von der Maßen 2002, RWTH Aachen

Primitive Datentypen im Detail

- **boolean**
  - Besitzt die vordefinierten Werte `true` oder `false`
  - Kann keiner Variablen eines anderen Typs zugewiesen werden
  - Ergebnis aller Vergleichsoperationen (<, >, `==`, `!=`, `>=`, `<=`)
  - Operatoren: **!, ^, &&, ||, &, |**

Bei `&` / | werden immer beide Terme ausgewertet!

- **char**
  - Kein Vorzeichen ⇒ Bei Konversion in `byte` oder `short` negativer Wert möglich!
  - Speichert zwei Byte Unicode-Wert (transparent für Programmierer)
  - Zulässige Literale: `'a'`, `'2'`, `'\n'`, `'\t'`, `'\u1234'`

Beliebiges Unicode-Zeichen als Hex-Code!

- **float, double**
  - float-Literale : `1.3f`, `1.4E-5F`
  - double-Literale : `2.5d`, `3.14E+308`
  - Es gibt auch spezielle NaN-Werte ("Not a number")
- **byte, short, int, long**
  - long-Literal : `4561`, `1067L`

© M. Schnizler, Th. von der Maßen 2002, RWTH Aachen

Envelope-Klassen

```
Integer iObj1 = new Integer(i);
s = iObj1.toString();
Integer iObj2 = new Integer("35");
i = iObj2.intValue();
```

Erzeugt ein Objekt für `int`-Variable `i`!

Gibt String-Darstellung zurück!

Erzeugt ein Objekt für die im String enthaltene Zahl!

Gibt `int`-Wert zurück!

- **Java definiert Envelope-Klassen für alle primitiven Typen**
  - dienen als Wrapper für primitive Typen
  - Teil des Packages `java.lang`
  - Definiert: `Boolean`, `Character`, `Double`, `Float`, `Byte`, `Short`, `Integer`, `Long`
- **Implementieren ein vollwertiges Objekt des primitiven Typs**
  - vermeidet Unterscheidung zwischen Objekten und primitiven Daten
  - bieten Reihe von Konversions(klassen)methoden (notwendig für Ein-/Ausgabe)

Anweisungen

- **Methodenaufruf**

```
ggtObject.ggt(x, 54);
```

Kann je nach Sichtbarkeit auch entfallen!

- **Zuweisung**

```
a = a - b;
```

- **Iteration**

```
while (a != b) { ... }
```

- **Bedingte Anweisung**

```
if (a > b) ... else ... ;
```

- **Ausnahme**

```
try { ... } catch (Exception e) { ... }
```

Zuweisung

```
ggtObject.g = ggtObject.ggt(90, 54);

a = a - b;

int zahl = zahl + 1;

zahl = (int) 2.53;
```


Numerische Typen

- Bei Zuweisungen in folgender Hierarchie keine explizite Typkonversion nötig:
  byte ⇒ **short** ⇒ **int** ⇒ **long** ⇒ **float** ⇒ **double**
- Explizite Typkonversion erfordert ausreichend Speicherplatz ⇒ sonst Fehler!
- Exp. Typkonversion von `float`/`double` nach Ganzzahl ⇒ ganzzahliger Anteil
- Typ eines Ausdrucks hängt von den Beteiligten ab:
  - 1 / 2 ergibt 0 (`int`-Wert)
  - 1.0 / 2 ergibt 0.5 (`double`-Wert)

Iterationen

FOR-Schleife

```
for (int i = 0; i < 10; i++) {
  ... wird genau 10 mal ausgeführt ...
}
```


WHILE-Schleife

```
while (a != b) {
  ... wird ausgeführt bis a gleich b ist ...
}
```


DO-WHILE-Schleife

```
do {
  ... wird ausgeführt bis a gleich b ist ...
  ... jedoch mindestens einmal ...
}
while (a != b);
```

Bedingte Anweisungen

Verzweigung

```
if (a > b)
  a = a - b;
else
  { ... };
```

Ausdruck muß vom Typ **boolean** sein!

else-Zweig kann auch entfallen!

Auswahl

Ganzzahl- oder **char**-Typ!

```
switch (month) {
  case  1:
  case  2:
  case 12: System.out.println("Winter");
           break;
  case  3:
  case  4:
  case  5: System.out.println("Fruehling");
           break;
  case  6:
  case  7:
  case  8: System.out.println("Sommer");
           break;
  case  9:
  case 10:
  case 11: System.out.println("Herbst");
           break;
  default: System.out.println("Ungueltig!");
}
```

Bricht Ausführung der **switch**-Anweisung ab!

Optionaler **default**-Fall!

Ausnahmebehandlung

```
String s;
try
{
  while (true)
  {
    s = liesZeile();
    System.out.println(s);
  }
}
catch (EOFException e)
  {  System.out.println("Schluss!"); }
```

Leitet den überwachten Block ein!

Behandle alle Ausnahmen der angegebenen Klasse/Super-klasse

liesZeile wirft nach Eingabe eines **Ctrl-D** eine **EOFException**!

Nach erfolgreicher Behandlung wird Ausführung nach dem **catch**-Block fortgesetzt!

Ausnahme

- zeigt an, dass etwas Außerplanmäßiges passiert ist (z.B. Fehler)
- eine Ausnahme kann
  - einerseits ausgelöst werden (throw-Operation)
  - andererseits behandelt werden (catch-Block)
- eine Ausnahme wird entlang der Aufrufhierarchie propagiert
  - bis eine passende Behandlung (catch-Block) gefunden wird
  - andernfalls bricht die Runtime das Programm mit einem Laufzeitfehler ab
- wichtiger Mechanismus für die Behandlung von Fehlern und Problemen

Ausnahmen

Programmierer

- kann bei Bedarf selber Ausnahmen auslösen:

```
if (a == null) throw new NullPointerException("Nicht definiert!");
```

- kann eigene Ausnahmen als Unterklasse von **`java.lang.Throwable`** bzw. **`java.lang.Exception`** implementieren
- muß eventuell unbehandelte Ausnahmen für eine Methode deklarieren:

```
public String liesZeile() throws IOException { ... }
```

Nicht nötig für Unterklassen von **`Error/Runtime-Exception`**!

FINALLY-Klausel

- wird immer ausgeführt (z.B. zum Aufräumen)

```
try { ... }
catch (FirstException e)  { ... }
catch (SecondException e) { ... }
finally
{
  ...
}
```

Wird garantiert immer ausgeführt: Mit und ohne Ausnahme!

Null und mehr catch-Blöcke sind erlaubt!

Referenzen

Primitive Datentypen

- wie z.B. **`int`** oder **`boolean`** werden direkt in einer Variablen gespeichert
- sie werden als Werte (by value) behandelt

Komplexe Datentypen

- Objekte, Arrays und Strings in Java
- für diese wird nur die Speicheradresse in einer Variablen gespeichert
- sie werden als Referenzen (by reference) verwaltet

Referenzen

- Referenz = Speicheradresse der eigentlichen Datenstruktur
- Referenzierung und Dereferenzierung der Objekte erfolgt vollautomatisch
- die tatsächliche Speicheradresse kann nicht ermittelt oder manipuliert werden
- eine undefinierte Referenzvariable hat den Wert **`null`**
- Objekte können nicht explizit gelöscht werden
- nicht mehr benötigte Datenstrukturen (keine Referenz im Programm mehr vorhanden!) werden vom Garbage Collector eingesammelt

Umgang mit Referenzen

```
Circle a = new Circle(12, 10, 3);
Circle b = new Circle(12, 10, 3);

if (a == b) { ... }
if (a.equals(b)) { ... }

b = a;

if (a == b) { ... }
```


- **Zuweisung**
  - zwischen zwei Referenzen kopiert nur die Speicheradresse
- **Vergleich**
  - zwischen zwei Referenzen vergleicht nur die Speicheradresse
- **Kopieren/Vergleichen der Objekte**
  - jedes Objekt sollte deshalb die Methoden `clone()`/`equals()` implementieren

Arrays

```
byte octetBuffer[] = new byte[1024];

Button[] buttons = new Button[10];

int lookupTable[] = {1, 2, 4, 8, 16, 32};

byte matrix[][] = new byte[256][16];
```


- **Arrays von beliebiger Dimension möglich**
  - werden als Array von Arrays behandelt
  - müssen nicht "rechteckig" sein! z.B. Dreiecksmatrix
  - Folgende Zuweisung ist legal:

```
String lotsOfStrings[][][][] = new String[5][3][][];
```

  - Statische Initialisierung mit geschachtelten `{...}` möglich.

Zugriff auf Array

0 ist immer erstes Element!

Übliche Indizierung mit Ganzzahl-Index!

Standardmäßig definiert! Read-only!

```
int[] a = new int[100];

a[0] = 5;

for(int i; i < a.length; i++)
  a[i] = i + a[i-1];
```

- **Indexgrenzen werden vor Zugriff überprüft!**
- **Ein Array ist einem Objekt vergleichbar**
  - Verwaltet über Referenz-Variable, dynamisch erzeugt mit `new`
  - Garbage Collector sammelt nicht benötigten Speicherplatz
  - Kann einer Variable vom Typ `Object` zugewiesen werden
- **Dennoch spezieller Charakter, da zusätzliche Syntax**
  - `[ ]` - Operator
  - Statische Initialisierung mit `{...}`

HelloWorld mit Java

```
public class HelloWorld {
    /* Unser erstes Programm in Java */

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

Einstiegs-punkt in Java-Anwendung

- **Jede Klasse steht in gleichnamiger `.java` Datei**
  - in obigem Beispiel in der Datei `Helloworld.java`
- `main()`-Methode
  - Klassenmethode die standardmäßig als Programmeinstieg dient
  - Java-Interpreter startet Programmausführung mit ihrem Aufruf:

```
HelloWorld.main(arguments)
```

Parameter auf Kommandozeile

Übersetzen und Ausführen

Übersetzer

Quelldatei

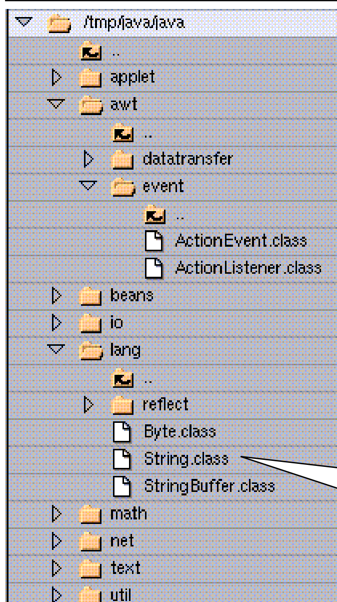
Erzeugter Bytecode

java-Interpreter führt Bytecode aus

```
>javac HelloWorld.java

>dir
10/22/01   11:18a   171 HelloWorld.java
10/22/01   11:18a   472 HelloWorld.class

>java HelloWorld
Hello World!

>
```

- **Übersetzer erzeugt plattform-unabhängigen Bytecode**
- **java-Interpreter**
  - führt den Bytecode der `.class` - Datei mit der Java Virtual Machine (JVM) aus
- **`javac` und `java` sind Teil des frei erhältlichen JDK**

Packages



- **Internet-Anwendungen mit Java**
  - Java vermeidet Namensraumkonflikte
  - Keine globalen Variablen!
  - Jede Variable/Methode ist Teil einer Klasse
- **Jede Klasse ist Teil eines Package**
  - jede Variable/Methode kann mit ihrem vollständigen Namen referenziert werden:

```
java.lang.String.valueOf(zeichenfeld)
```

Package-Name `java.lang`

Klassenname

Methodenname (Klassenmethode)

Package-Name `java.lang` entspricht Verzeichnisstruktur `/java/lang/String.class`

Umgang mit Packages

```
package shapes;

import basics.Point;
import java.util.*;
import java.awt.image;

public class Circle {
  ...  // Variablen, Konstruktoren etc.

  public void verschiebe(Point p) {
    ... }                           }
```


Umgebungsvariable CLASSPATH

- gibt an in welchen Verzeichnissen Klassen/Package-Hierarchien beginnen:

  `setenv CLASSPATH "/tmp/java:/home/fred/myclasses"`
- muß vom Programmierer entsprechend gesetzt werden
- `public`-Klassen können dann mit ihrem vollständigen Namen referenziert werden
- `import` erlaubt Abkürzung des Namens
- fehlt die `package` Klausel, ist die Klasse Teil des namenlosen Standard-Packages

Java Standard API

Folgende Packages sind wichtiger Bestandteil des JDK:

java.applet	**Umfaßt Klassen für die Applet-Programmierung**
java.awt	**Klassen für GUI-Programmierung (Abstract Window Toolkit)**
java.beans	**Klassen für das Java Komponentenmodell**
java.io	**Klassen für die Ein-/Ausgabe (System, Dateien)**
java.lang	**Grundlegende Klassen der Sprache Java**
java.math	**Klassen für beliebig genaue Arithmetik**
java.net	**Klassen für Netzwerkanwendungen**
java.rmi	**Klassen für *R*emote *M*ethod *I*nvocation**
java.security	**Umfaßt das Krypto-Framework in Java**
java.sql	**Klassen für Datenbankzugriff**
java.text	**Klassen für die Arbeit mit Texten usw.**
java.util	**Klassen von allgemeinem Nutzen (u.a. Collections Framework)**

java.lang - Strings

- **Strings**
  - dienen der Speicherung beliebiger Zeichenketten
  - Instanzen der vordefinierten Klasse `java.lang.String`
- **Dennoch spezieller Charakter**
  - Übersetzer behandelt Literale in `"..."` automatisch als String-Objekte:

```
String s = "Das ist ein String.";
```

  - `+` - Operator zur Verknüpfung von Strings:

```
String s = "Das ist " + "ein String.";
```

- **Strings dürfen nicht über mehrere Zeilen gehen**
  - stattdessen Verknüpfung mit `+` - Operator
- **Inhalt eines String-Objekts ist nicht veränderbar!**
  - hierfür gibt es eine eigene Klasse `java.lang.StringBuffer`

© M. Schnizler, Th. von der Maßen 2002, RWTH Aachen

IO.java

```
public class IO {

    public static int Eingabe() { ... }

    public static double DoubleEingabe() { ... }

    public static String TextEingabe() { ... }
}
```

- **IO.java**
  - Liest Integer-, Double-Werte und Strings von der Konsole ein
  - Verwendung:

```
int a = IO.Eingabe();
double b = IO.DoubleEingabe();
String c = IO.TextEingabe();
```

© M. Schnizler, Th. von der Maßen 2002, RWTH Aachen

java.util - Vector

```
Vector objects = new Vector();

objects.addElement(new Integer(1));
objects.addElement(new Integer(2));
objects.addElement(new Integer(3));

Object o = new Integer(2);

if (!objects.isEmpty() && objects.contains(o))
  o = objects.elementAt(objects.indexOf(o));

Enumeration e = objects.elements();
```

Akzeptiert nur Objekte!

Vergleich mit `equals()`!

Leer?

Alle Elemente als Aufzählung !

Vector

- implementiert ein dynamisch wachsendes Array von Objekten
- gut geeignet um Aggregation beliebig vieler Objekte zu implementieren
- flexibler, wenn auch weniger effizient als Standard-Array

java.util - Enumeration

```
Enumeration e = objects.elements();

while (e.hasMoreElements())
{
   doSomething(e.nextElement());
}
```

Wird von Collection-Klasse zurück-gegeben!

Weitere Elemente?

Nächstes Element!

Enumeration

- wird nur als Interface in `java.util` definiert und von Collection-Klassen implementiert
- repräsentiert eine Aufzählung der Objekte einer Collection-Klasse
- hat nur die zwei Methoden `hasMoreElements()` und `nextElement()`

Java Collection Framework (ab JDK 1.2)

- überarbeitete und erweiterte Version der ursprünglichen Collection-Klassen
- z.B. `Vector` wird nach wie vor unterstützt
- `Enumeration` soll ab JDK 1.3.1 durch `Iterator` ersetzt werden

JavaDoc


```
/**
 * An object that implements the Enumeration interface …
 * @see      java.util.Iterator
 * @author  Lee Boynton
 * @version 1.18, 02/02/00
 * @since   JDK1.0
 */

 /**
  * Reads the next byte of data from the input stream. …
  * @return      the next byte of data, or …
  * @exception  IOException  if an I/O error occurs.
  */
 public abstract int read() throws IOException;

javadoc -d doc mypack mypack.gui mypack.model
```

- **Werkzeug JavaDoc**
  - erzeugt aus annotierten Kommentaren eine HTML-Dokumentation des Codes
  - die Code-Dokumentation ist so auch Teil des Programmcodes

Literatur

Judy M. Bishop, **Java Gently - Programming Principles Explained,** 2. Auflage, Addison-Wesley, 1998.

Judith Bishop, **Java lernen**, Deutsche Übersetzung, 2. Auflage, Addison-Wesley, 2001.

David Flanagan, **Java in a Nutshell, 4th Edition**, O'Reilly, (März) 2002.