Kapitel 3: Objektorientierung

Von Elementaren zu Abstrakten Typen

Elementare Typen

- Bestandteile:
  - Format (z.B. ganze Zahl)
  - Operationen (z.B. ‚+')
- Von der Sprache fest vorgegeben
- Initialisierung durch direkte Zuweisung eines Wertes
- Instanzen werden als **Variablen** bezeichnet

Abstrakte Typen

- Bestandteile:
  - **Attribute**
  - **Methoden**
- Werden in Java durch **Klassen** definiert
- Initialisierung mittels **Konstruktor**
- Instanzen werden als **Objekte** bezeichnet

Beispiel

Konstruktor

Methoden

Attribute

Bezug zum Objekt

```
public class Rechteck {

   private int breite;
   private int hoehe;

   public Rechteck(int breite, int hoehe) {
      setBreite(breite);
      setHoehe(hoehe);
   }
   public void setBreite(int breite) {
      if (breite > 0) this.breite = breite;
   }
   public int getBreite() {
      return breite;
   }
   public void setHoehe(int hoehe) {
      if (hoehe > 0) this.hoehe = hoehe;
   }
   public int getHoehe() {
      return hoehe;
   }
   public void paint() {
      System.out.println("------");
      System.out.println("|    |");
      System.out.println("------");
   }
}
```

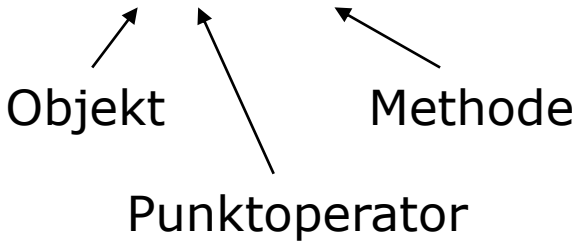
Anwendung

- Instanziierung

```
Rechteck r1 = new Rechteck(10, 50);
```

- Methodenaufruf

```
r1.paint();
```




```
public class Rechteck {
   private int breite;
   private int hoehe;
   public Rechteck(int breite, int hoehe) …
   public void setBreite(int breite) …
   public int getBreite() …
   public void setHoehe(int hoehe) …
   public int getHoehe() …
   public void paint() …
}
```

Klasse

- Klassen bestehen aus Attributen und Methoden.
- Klassen haben einen Namen.
- Klassen definieren einen Datentyp.
- Klassen sind statisch.
- Klassen definieren die statischen Eigenschaften von Objekten.

Objekt

- Objekte sind Instanzen von Klassen.
- Objekte haben Attributwerte bzw. einen Zustand sowie ein Verhalten.
- Objekte sind dynamisch, d.h. ihr Zustand ist zur Laufzeit veränderlich.

Methode

- Methoden bestimmen das Verhalten eines Objekts.
- Methoden dienen zur Manipulation und zum Auslesen von Attributwerten.

Konstruktor

- Konstruktoren sind spezielle Methoden, die bei der Instanziierung aufgerufen werden.
- Ihre Aufgabe ist die Initialisierung der Attribute mit gültigen Werten.
- Konstruktoren können nur bei der Instanziierung aufgerufen werden.

Defaultkonstruktor

- Konstruktoren ohne Parameter werden als Default- oder Standardkonstruktoren bezeichnet.
- Wenn in einer Klasse explizit kein Konstruktor definiert ist, wird implizit eine Defaultkonstruktor definiert.
- Ein impliziter Defaultkonstruktor tut nichts.

Überladen von Methoden

- Wenn in einer Klasse mehrere Methoden mit gleichen Namen existieren, werden diese als überladen bezeichnet.
- Die Methoden werden anhand der verschiedenen Parameter unterschieden.
- Die Methodenbindung diesbezüglich ist statisch.

Beispiel

- Überladen des Konstruktors der Klasse `Rechteck` durch hinzufügen eines Defaultkonstruktors:

```
public class Rechteck {
    …
    public Rechteck(int breite, int hoehe) {
        setBreite(breite);
        setHoehe(hoehe);
    }
    public Rechteck() {
        setBreite(10);
        setHoehe(10);
    }
    …
}
```

this

- Mit `this` im Konstruktor kann ein überladender Konstruktor aufgerufen werden.
- Im Konstruktor muss `this` die erste Anweisung sein.
- Mit `this` können Attribute und Methoden des angewendeten Objekts explizit adressiert werden, insbesondere um Überschneidungen mit Parameternamen zu verhindern.

```
public class Rechteck {

   private int breite;
   …
   public Rechteck(int breite,
                   int hoehe) {
      setBreite(breite);
      setHoehe(hoehe);
   }

   public Rechteck() {
      this(10, 10);
   }

   public void setBreite(
                   int breite) {

      if (breite > 0)
         this.breite = breite;
   }
   …
}
```

Objekte im Speicher

- Vorbereitung: Hinzufügen der `print()`-Methode zur Klasse `Rechteck` zur Ausgabe von Breite und Höhe.

```
public class Rechteck {
    …
    public void print() {
        System.out.println("(" + breite + "/" + hoehe + ")");
    }
}
```

- Beispiel: `r1.print()` → „(10/50)“

Objekte im Speicher (2)

- Was wird bei folgenden `print()`-Aufrufen ausgegeben?

```
Rechteck r1 = new Rechteck(10, 50);
r1.print();
Rechteck r2 = new Rechteck(20, 30);
r2.print();
Rechteck r3 = r1;
r3.print();
r3.setBreite(75);
r3.print();
r1.print();
```

Objekte im Speicher (3)

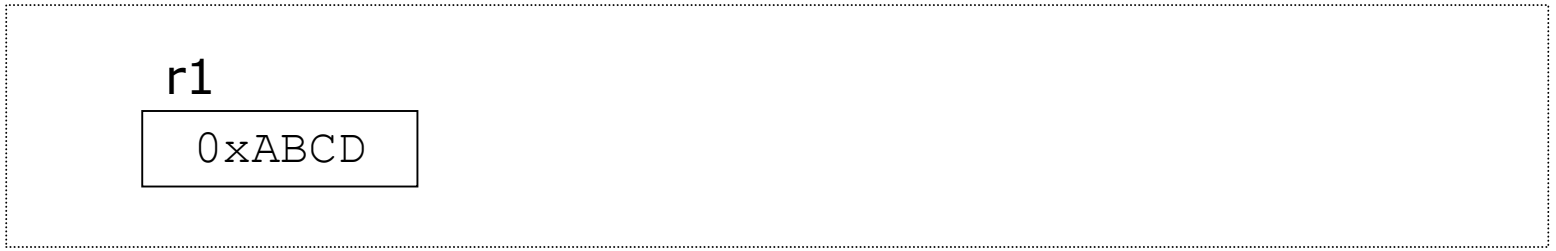
- Was wird bei folgenden `print()`-Aufrufen ausgegeben?

```
Rechteck r1 = new Rechteck(10, 50);
r1.print();              → (10/50)
Rechteck r2 = new Rechteck(20, 30);
r2.print();              → (20/30)
Rechteck r3 = r1;
r3.print();              → (10/50)
r3.setBreite(75);
r3.print();              → (75/50)
r1.print();              → (75/50) !!!
```

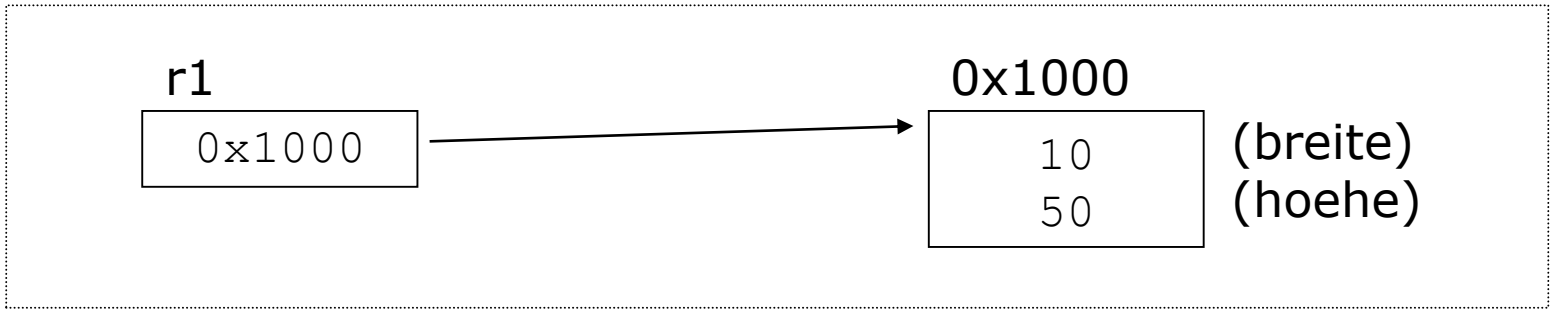
- Warum wird bei einer Änderung von `r3` auch `r1` geändert? Antwort: **Klassen sind Referenztypen!**

Objekte im Speicher (4)

- `Rechteck r1;`

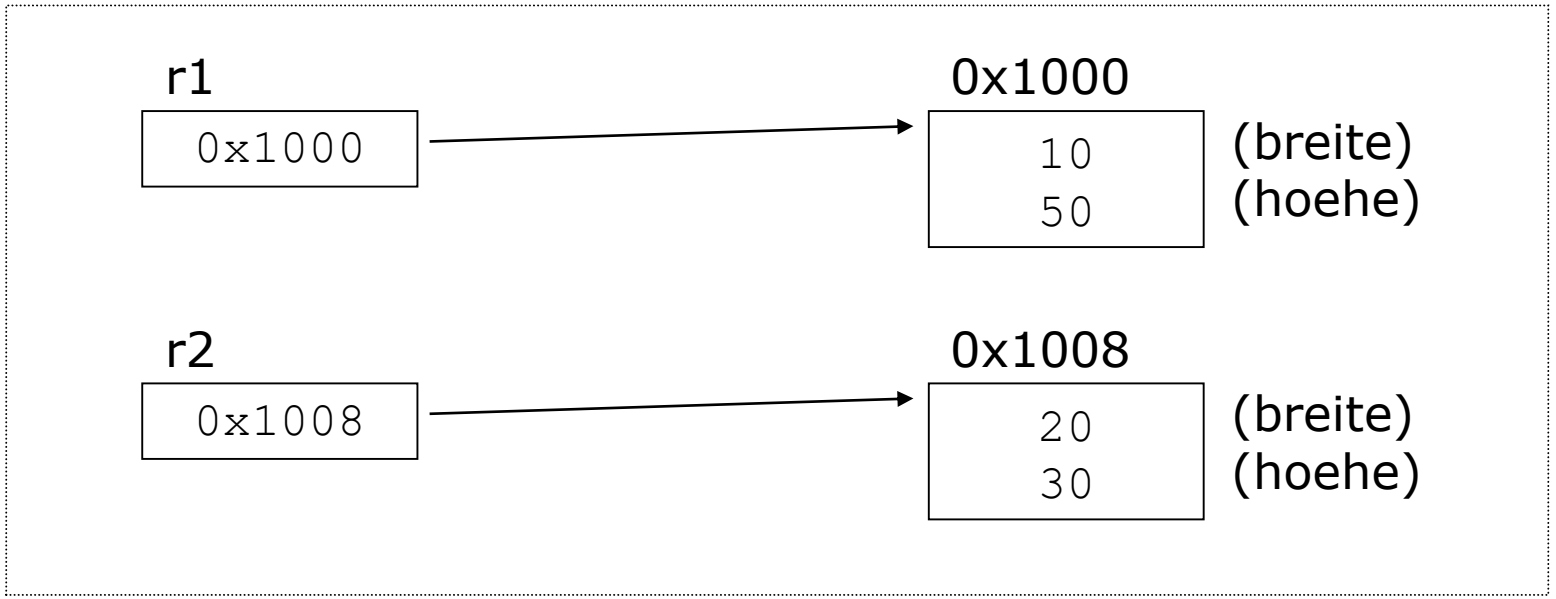


- `r1 = new Rechteck(10, 50);`



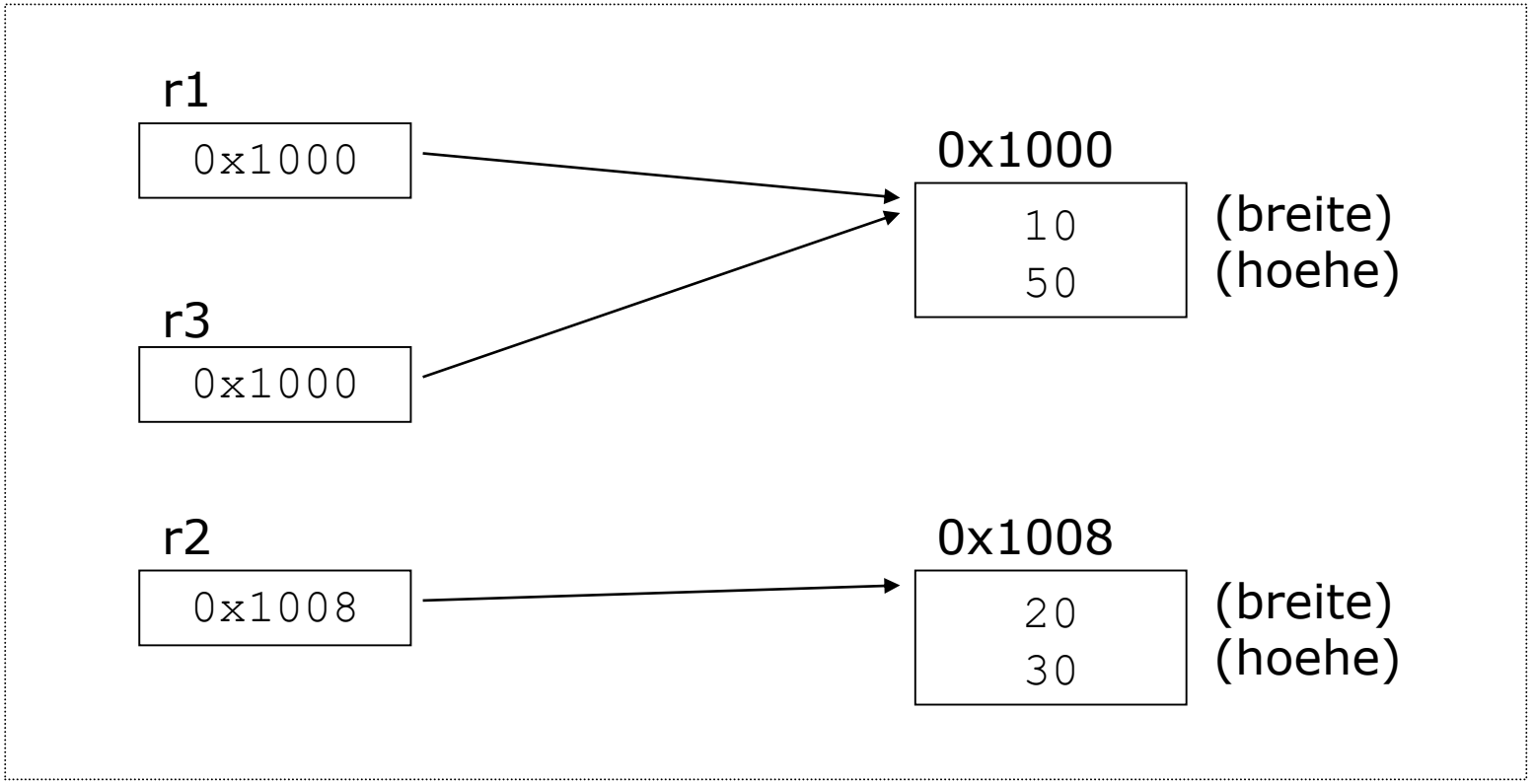
Objekte im Speicher (5)

- `Rechteck r2 = new Rechteck(20, 30);`



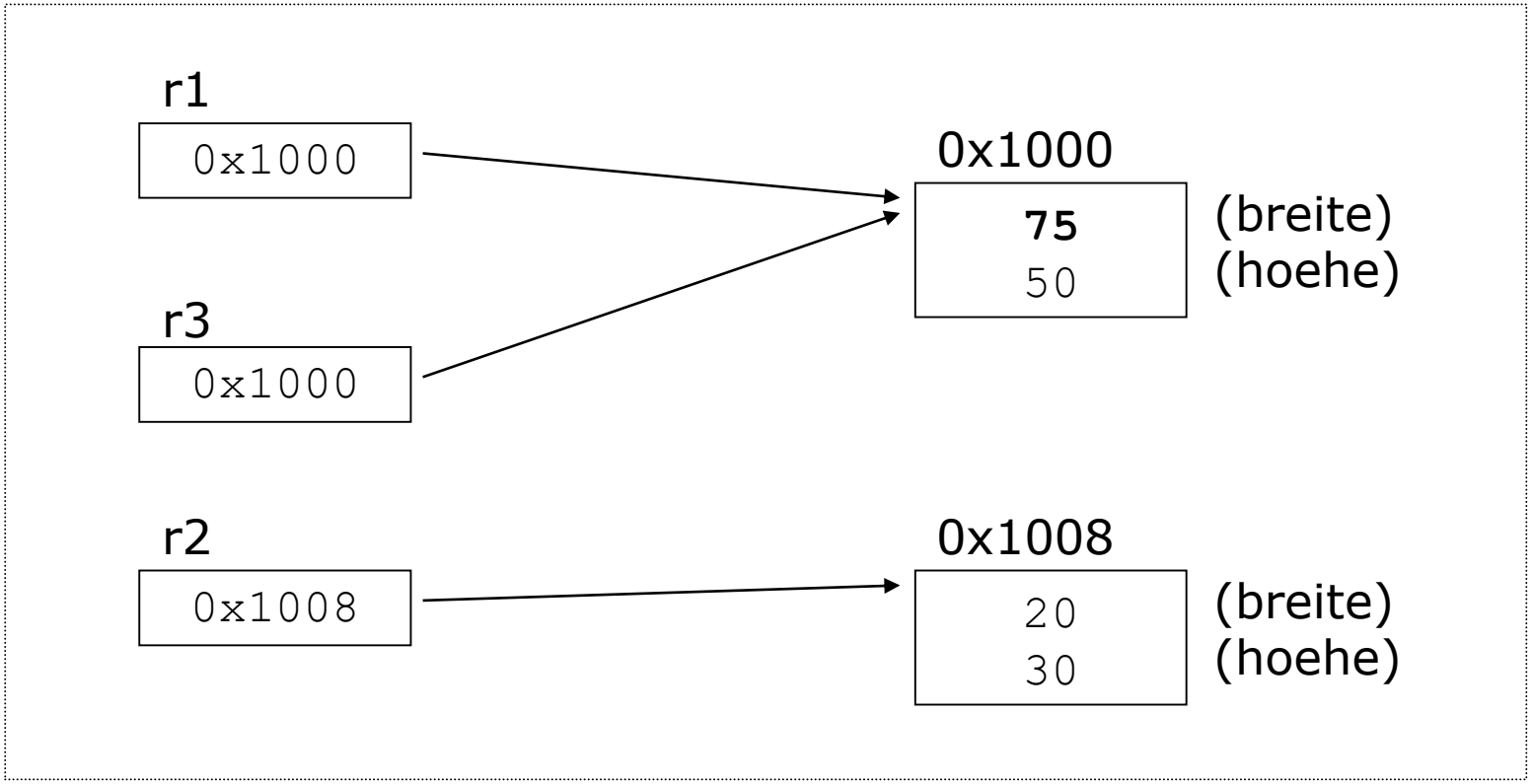
Objekte im Speicher (6)

- `Rechteck r3 = r1;` ← kein `new`, d.h. keine neue Instanz!



Objekte im Speicher (7)

- `r3.setBreite(75);`



Objekte im Speicher (8)

- Was wird bei folgenden `print()`-Aufrufen ausgegeben?

```
Rechteck r1 = new Rechteck(10, 50);
r1.print();
Rechteck r2 = new Rechteck(20, 30);
r2.print();
Rechteck r3 = r1;
r3.print();
r3.setBreite(75);
r3.print();
r1.print();
```

Garbage Collection

- Im **Stack** werden lediglich Objektreferenzen oder Variablen elementarer Typen gespeichert.
- Die Objekte selbst werden mit **`new`** im Speicherbereich namens **Heap** erzeugt.
- Der Speicher im Stack wird freigegeben, wenn die entsprechende Funktionsinstanz abgearbeitet wurde.
- Wodurch wird der Speicher im Heap freigegeben?

➔ Antwort: durch die **Garbage Collection**

Garbage Collection (2)

- Die Garbage Collection („Müllabfuhr“) prüft eigenständig von Zeit zu Zeit, ob die Objekte noch referenziert werden.
- Falls eine Instanz nicht mehr referenziert wird, wird ihr Speicher freigegeben.

`static`

- Klassen können neben normalen Attributen und Methoden auch statische Attribute und Methoden definieren.
- Statische Elemente werden mit dem Schlüsselwort `static` gekennzeichnet.
- Statische Elemente sind an die Klasse gebunden und nicht an ein Objekt.
- Statische Elemente haben auf nicht-statische Elemente keinen Zugriff (das wäre nicht eindeutig…) – umgekehrt ist der Zugriff aber schon möglich.

Beispiel

- Jede Rechteck-Instanz vergibt sich selbst eine eindeutige Nummer:

```
public class Rechteck {

    private static int maxNr = 0;

    private int nr;
    private int breite;
    private int hoehe;

    public Rechteck(int breite, int hoehe) {
        nr = maxNr++;
        setBreite(breite);
        setHoehe(hoehe);
    }
    …
}
```

Beispiel (2)

- Die `main()`-Funktion:

```
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello World!");
    }
}
```

final

- `INTERVALL` ist eine Konstante, d.h. ihr Wert ist nicht veränderbar.
- Konstantennamen werden üblicherweise groß geschrieben.

```
public class Rechteck {

   private static final int INTERVALL = 10;
   private static int maxNr = 0;

   private int nr;
   …
   public Rechteck(int breite, int hoehe) {
      nr = maxNr++;
      if (nr % INTERVALL == 0)
         System.out.println("Rechteck Nr. " + nr + " angelegt.");
      …
   }
   …
}
```

Ausgabe:
Rechteck Nr. 0 angelegt.
Rechteck Nr. 10 angelegt.
Rechteck Nr. 20 angelegt.
Rechteck Nr. 30 angelegt.
Rechteck Nr. 40 angelegt.
…

Die Klasse `String` (Zeichenketten)

- Instanziierung von Strings:

```
String str = new String("Hallo");
```

oder kürzer:

```
String str = "Hallo";
```

- Stringliterale werden in doppelten Anführungszeichen geschrieben (, während Zeichenliterale in einfachen Anführungszeichen stehen).
- Strings können durch den +-Operator miteinander verknüpft werden. Auch z.B. Zahlen können so mit Strings verknüpft werden.

```
String str = "Hallo" + " Wolke " + 7;
```

Wichtige Methoden

`char charAt(int index)`	Gibt das Zeichen an einer bestimmten Position zurück.
`int compareTo(` `  String anotherString)`	Vergleicht zwei Strings lexikographisch. Sind beide Strings gleich, wird 0 zurückgegeben, ist der String kleiner als `anotherString`, so wird ein Wert kleiner als 0 zurückgegeben, ansonsten ein Wert größer als 0.
`boolean equals(` `  Object anObject)`	Prüft zwei Strings auf Gleichheit.
`int length()`	Gibt die Anzahl der Zeichen zurück.

`String substring(`
`  int beginIndex,`
`  int endIndex)`
	Gibt einen Teilstring zurück, der alle Zeichen von `beginIndex` bis einschließlich `endIndex` - 1 beinhaltet.

`String toLowerCase()`	Wandelt alle Buchstaben in Kleinbuchstaben um.
`String toUpperCase()`	Wandelt alle Buchstaben in Großbuchstaben um.
`String trim()`	Schneidet führende und abschließende Whitespaces (Leerzeichen, Tab-Zeichen, Zeilenumbrüche) ab.

Beispiel

```
public static boolean hatSubstring(String orig, String suche) {
   orig = orig.toLowerCase();
   suche = suche.trim();
   suche = suche.toLowerCase();
   if (suche.length() > orig.length()) return false;

   for (int i = 0; i < orig.length() - suche.length(); i++) {
      String comp = orig.substring(i, i + suche.length());
      if (suche.equals(comp)) return true;
   }
   return false;
}
public static void main(String[] args) {
   String orig = "Dampfschifffahrtsgesellschaft";
   String suche = " Fahrt ";
   System.out.println(hatSubstring(orig, suche));
   // ergibt "true"
}
```

Die Klasse `StringBuffer`

- Die Zeichenkette eines `String`-Objekts wird intern als nicht veränderbarer `char`-Array gespeichert, d.h. sie wird lediglich durch den Konstruktor geschrieben.
- Alle weiteren schreibenden Methoden erzeugen neue `String`-Instanzen.
- Zur Manipulation von Zeichenketten empfiehlt sich daher die Klasse `StringBuffer`.

Wichtige Methoden

`StringBuffer(String s)`	Initialisiert einen StringBuffer durch den Inhalt des Strings `s`
`StringBuffer append(String s)`	Hängt am Ende eine Zeichenkette an
`StringBuffer delete(` `  int start, int end)`	Löscht eine Teil der Zeichenkette von Index `start` bis `end - 1`
`StringBuffer insert(` `  int offset, String s)`	Fügt eine Zeichenkette `s` an der Position `offset` ein
`StringBuffer replace(` `  int start, int end, String s)`	Ersetzt die Zeichenkette von Index `start` bis `end - 1` durch den String `s`
`String toString()`	Gibt die Zeichenkette als `String` zurück

Über nichtinitialisierte Variablen

- Was passiert, wenn versucht wird, Variablen auszulesen, die bisher nicht initialisiert wurden?
- Was erwarten Sie diesbezüglich von Java?

Nichtinitialisierte lokale Variablen

```
public class Test {
   public Test(){}          // Der Konstruktor macht nichts
   public void f() {
      int x;                // x wird KEIN Wert zugewiesen!
      Rechteck r;           // r wird KEIN Wert zugewiesen!
      int y = x + 1;                    // Compiler-Fehler!
      int z = r.getBreite() + 1;        // Compiler-Fehler!
   }
   public static void main(String[] args) {
      Test t = new Test();
      t.f();
   }
}
```

„Nichtinitialisierte“ Attribute

```
public class Test {
   private int x;
   private Rechteck r;
   public Test() { }
   public void f() {
      int y = x + 1;                // Funktioniert!
      System.out.println(y);        // Gibt 1 aus
      int z = r.getBreite() + 1;    // NullPointerException
      System.out.println(r);
   }
   public static void main(String[] args) {
      Test t = new Test();
      t.f();
   }
}
```

Der Wert `null`

```
public class Test {

    private Test() { }

    private void f() {
        Rechteck r = null;
        g(r);
    }

    private void g(Rechteck rechteck) {
        ...
    }

    ...
}
```

Über Nichtinitialisierte Variablen (2)

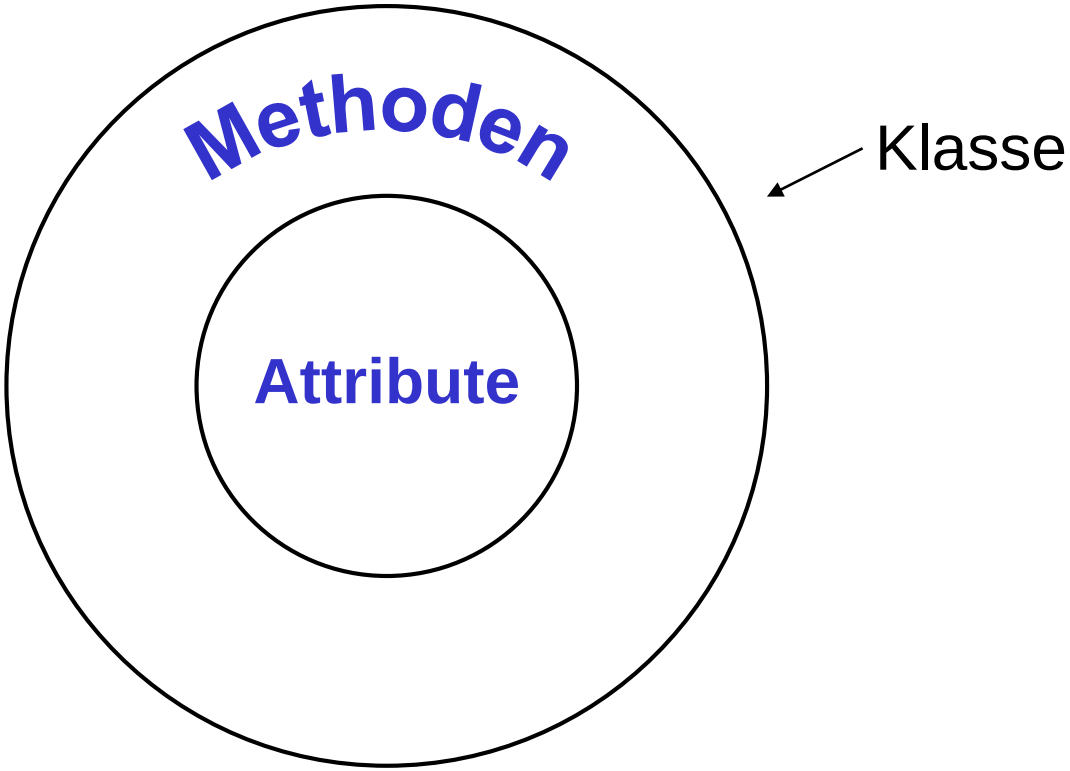
- Was passiert, wenn versucht wird, Variablen auszulesen, die bisher nicht initialisiert wurden?
  - Nichtinitialisierte lokale Variablen verursachen einen Compilerfehler.
  - Attribute werden implizit mit einem Defaultwert initialisiert.
- Tipp: Attribute vor Verwendung immer initialisieren, um Klarheit zu schaffen.

Über Nichtinitialisierte Variablen (3)

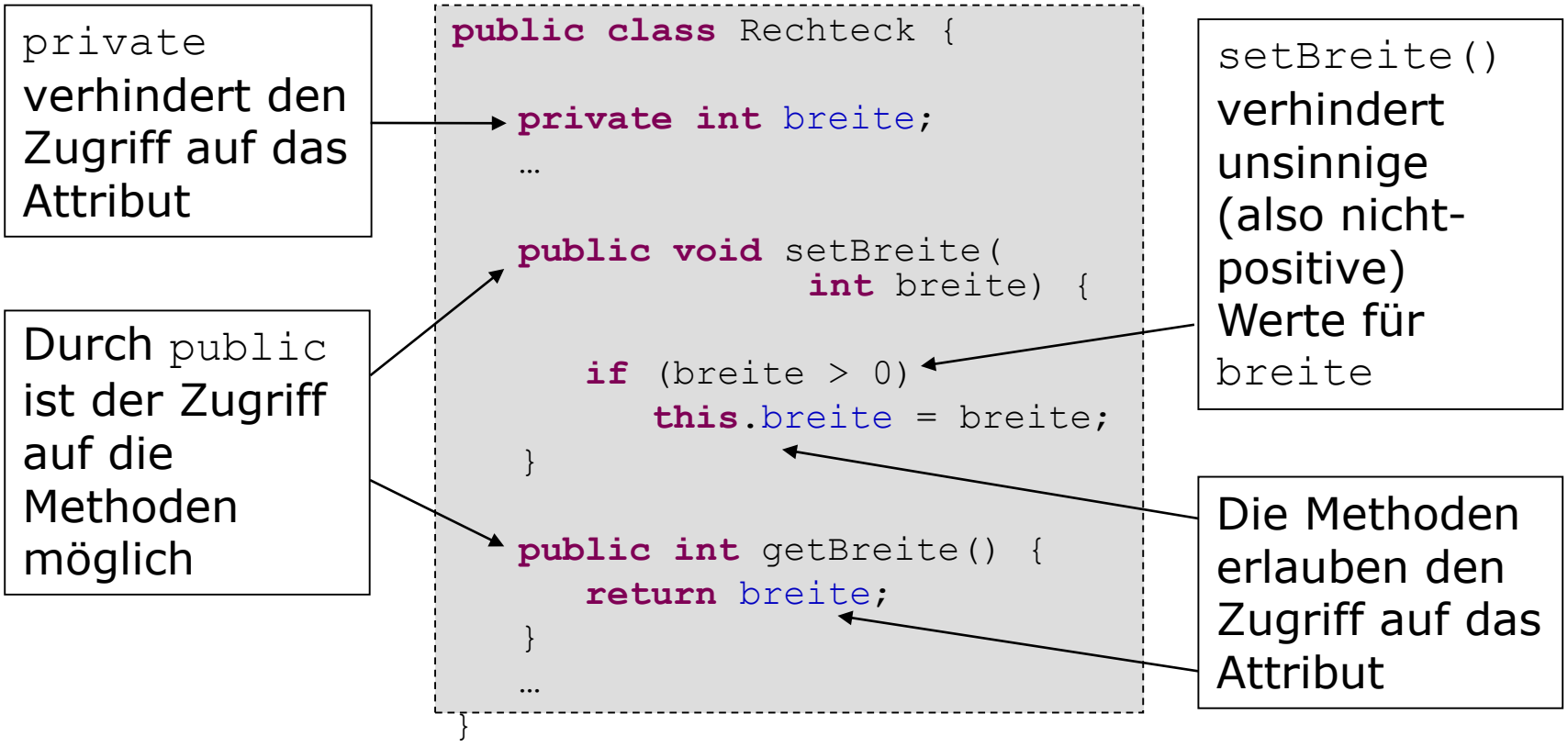
- Java erlaubt keinen Zugriff auf nichtinitialisierte Variablen (im Gegensatz zu C).
- Was könnte passieren, wenn man auf eine nichtinitialisierte Variable eines Referenztyps zugreifen könnte?

Datenkapselung

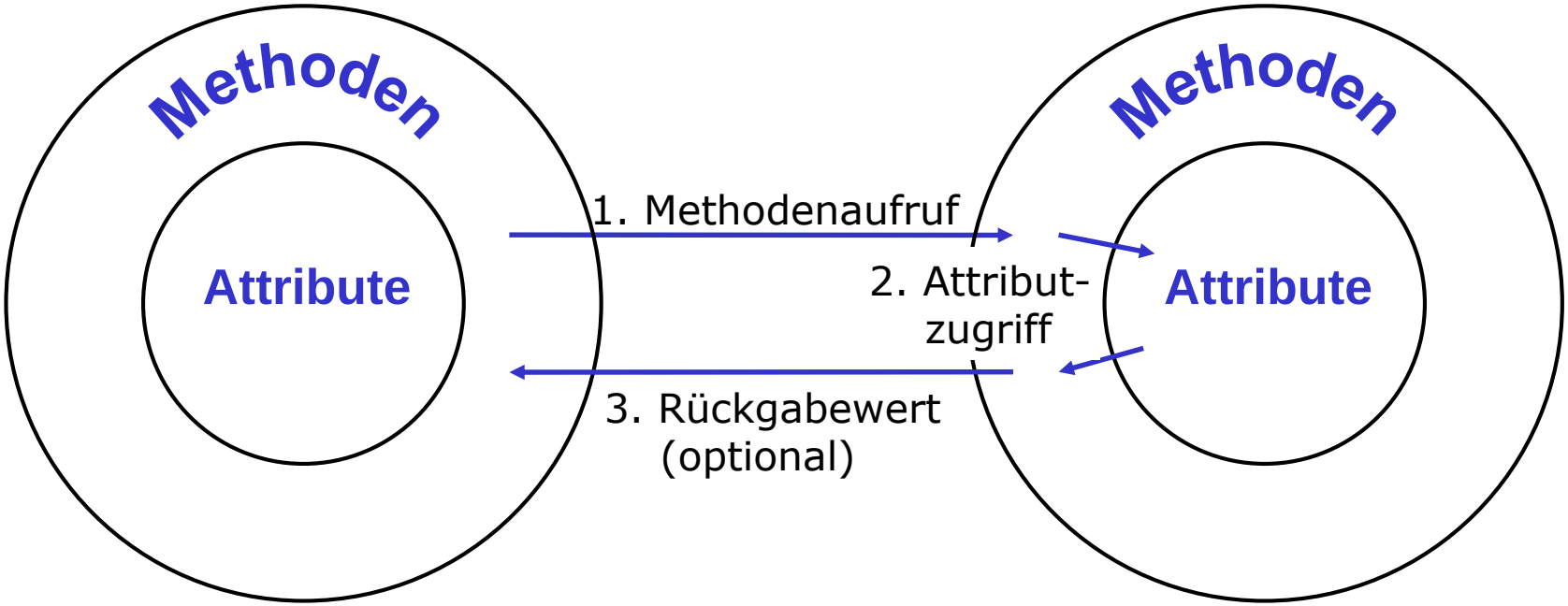
- Methoden kapseln ihre Attribute, um deren Integrität zu wahren.



Beispiel


private verhindert den Zugriff auf das Attribut
Durch public ist der Zugriff auf die Methoden möglich
public class Rechteck {
private int breite;
…
public void setBreite(
int breite) {
if (breite > 0)
this.breite = breite;
}
public int getBreite() {
return breite;
}
…
}
setBreite() verhindert unsinnige (also nicht-positive) Werte für breite
Die Methoden erlauben den Zugriff auf das Attribut

Kommunikation zwischen Objekten



- Beispiele:

```
r1.setBreite(30);
int b = r1.getBreite();
```

Sichtbarkeit

	In derselben Klasse	**In Unterklasse**	**Im selben Package**	**Überall**
`private`	Sichtbar			
`protected`	Sichtbar	Sichtbar	Sichtbar	
`public`	Sichtbar	Sichtbar	Sichtbar	Sichtbar
Nichts angegeben	Sichtbar		Sichtbar	

- Methoden sind typischerweise `public`
- Attribute sind typischerweise `private`
- *Unterklassen* und *Packages* werden später noch behandelt

Sichtbarkeit (2)

- Die Sichtbarkeit ist klassenspezifisch, nicht objektspezifisch.
- Die Prüfung der Sichtbarkeit erfolgt zur Übersetzungszeit (statisch) und nicht zur Laufzeit (dynamisch).
- Beispiel eines zulässigen Zugriffs:

```
public class O {
    …
    private int m1() {
        …
    }
    public int m2(O o) {
        return o.m1();
    }
}
```

Packages

- Packages gruppieren zusammengehörige Klassen.
- Packages erlauben, die Klassen eines Projektes zu strukturieren.
- Packages wahren die Übersichtlichkeit von Projekten und machen leichter wartbar.

Syntax

- Deklaration: `package packagename;`
- Das Schlüsselwort `package` muss die erste Anweisung einer Java-Datei sein.
- Die entsprechende Java-Datei muss ausgehend vom Projektpfad in einem Verzeichnis liegen, das genauso heißt wie das Package.
- Standardmäßig befinden sich Java-Dateien im *Default-Package*. Im Dateisystem müssen die Dateien dann direkt im Projektverzeichnis liegen.

Packagehierarchie

- Packages können ineinander verschachtelt werden.
- Die einzelnen Hierarchieebenen werden durch den Punktoperator voneinander getrennt.
- Syntax:

```
package oberpackage.zwischenpackage.unterpackage;
```

Namensräume

- Packages trennen *Namensräume*, d.h. Klassen müssen nur innerhalb eines Packages einen eindeutigen Namen besitzen.
- Demnach müssen Klassen grundsätzlich mit vollqualifizierten Namen adressiert werden, z.B.:

```
public class Abc {
    private op.zp.up.Xyz test;
    …
}
```

Namensräume (2)

- Alternativ können Klassen zunächst importiert werden. Dann kann auf den „Pfad" der Klasse verzichtet werden, z.B.:

```
import op.zp.up.Xyz;

public class Abc {
    private Xyz test;
    …
}
```

- Eine vollqualifizierte Adressierung ist dann zwingend, wenn auf mehrere Klassen mit gleichem Namen zugegriffen werden soll.

Sichtbarkeit von Klassen

	Im selben Package	**Überall**
`public`	Sichtbar	Sichtbar
Nichts angegeben	Sichtbar	

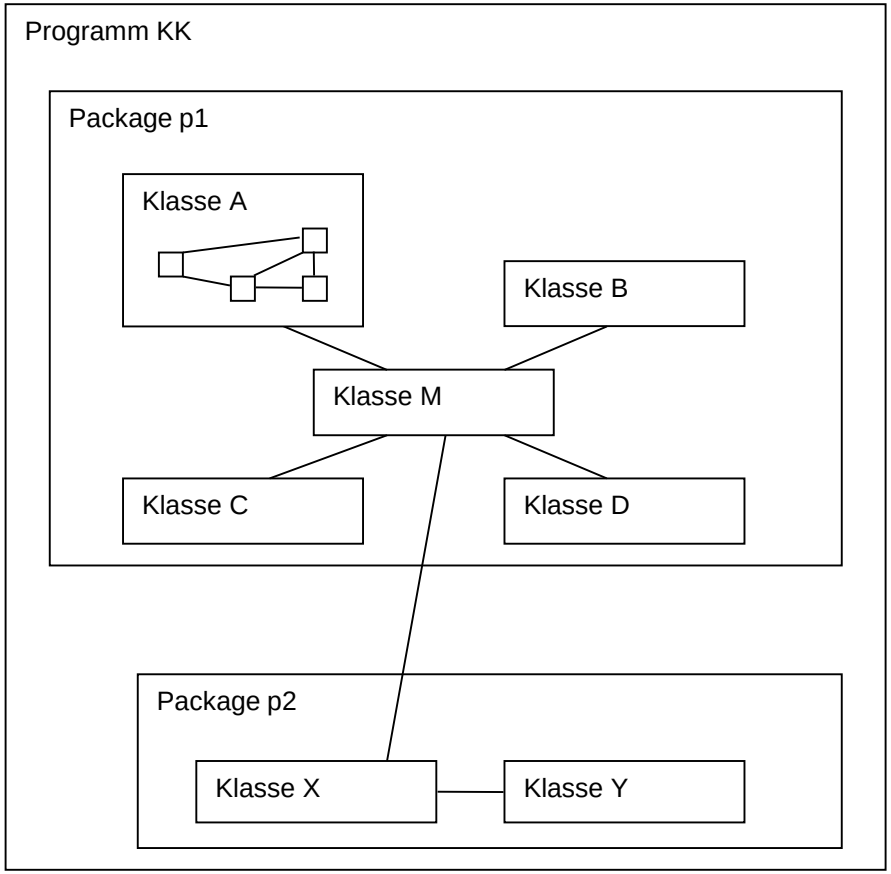
Das Package `java.lang`

- Das die Klassen des Packages `java.lang` werden standardmäßig importiert.
- Es beinhaltet fundamentale Klassen wie `System`, `Object`, `String`, `Math` und `Exception`.

Kohäsion und Kopplung

- Kohäsion bezeichnet die Anzahl von Abhängigkeiten innerhalb eines Systems.
- Kopplung ist durch die Anzahl von Abhängigkeiten zwischen Systemen bestimmt.
- Ein gutes Design ist geprägt von hoher Kohäsion und geringer bzw. loser Kopplung.

Beispiel


Programm KK
Package p1
Klasse A
Klasse B
Klasse M
Klasse C
Klasse D
Package p2
Klasse X
Klasse Y

Vererbung

Aufgabe:

- Analog zur Klasse `Rechteck` die Klasse `Kreis` definieren.
- Beide Klassen um eine Position (x- und y-Koordinate) ergänzen, um die Objekte schließlich zeichnen zu können.

Klasse `Kreis`

```
public class Kreis {

   private int xPos;
   private int yPos;
   private int radius;

   public Kreis(int x, int y, int radius) {
      setX(x);
      setY(y);
      setRadius(radius);
   }
   public void setX(int x) {
      if (x >= 0) xPos = x;
   }
   public int getX() { return xPos; }
   public void setY(int y) {
      if (y >= 0) yPos = y;
   }
   public int getY() { return yPos; }
   public void setRadius(int radius) {
      if (radius > 0) this.radius = radius;
   }
   public int getRadius() { return radius; }

   public void paint() {
      System.out.println("/---\\");
      System.out.println("|   |");
      System.out.println("\\---/");
   }
}
```

Exkurs: Escape-Sequenzen

- Diese Methode…:

```
public void paint() {
    System.out.println("/---\\");
    System.out.println("|   |");
    System.out.println("\\---/");
}
```

…führt zu dieser Ausgabe:

```
/---\
|   |
\---/
```

- Escape-Sequenzen in Java:

\n	Zeilenumbruch
\t	Tabulator
\"	" (Anführungszeichen)
\'	' (Hochkomma)
\\	\ (Backslash)

```java
public class Rechteck {

   private int xPos;
   private int yPos;
   private int breite;
   private int hoehe;

   public Rechteck(int x, int y,
           int breite, int hoehe) {
      setX(x);
      setY(y);
      setBreite(breite);
      setHoehe(hoehe);
   }
   public void setX(int x) {
      if (x >= 0) xPos = x;
   }
   public int getX() { return xPos; }
   public void setY(int y) {
      if (y >= 0) yPos = y;
   }
   public int getY() { return yPos; }
   public void setBreite(int breite) {
      if (breite > 0) this.breite = breite;
   }
   public int getBreite() { return breite; }
   public void setHoehe(int hoehe) {
      if (hoehe > 0) this.hoehe = hoehe;
   }
   public int getHoehe() { return hoehe; }
   public void paint() {
      System.out.println("------");
      System.out.println("|    |");
      System.out.println("------");
   }
}
```

```java
public class Kreis {

   private int xPos;
   private int yPos;
   private int radius;


   public Kreis(int x, int y, int radius) {

      setX(x);
      setY(y);
      setRadius(radius);

   }
   public void setX(int x) {
      if (x >= 0) xPos = x;
   }
   public int getX() { return xPos; }
   public void setY(int y) {
      if (y >= 0) yPos = y;
   }
   public int getY() { return yPos; }
   public void setRadius(int radius) {
      if (radius > 0) this.radius = radius;
   }
   public int getRadius() { return radius; }



   public void paint() {
      System.out.println("/---\\");
      System.out.println("|   |");
      System.out.println("\\---/");
   }
}
```

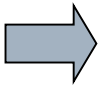
```java
public class Rechteck {

   private int xPos;
   private int yPos;
   private int breite;
   private int hoehe;

   public Rechteck(int x, int y,
                   int breite, int hoehe) {
      setX(x);
      setY(y);
      setBreite(breite);
      setHoehe(hoehe);
   }
   public void setX(int x) {
      if (x >= 0) xPos = x;
   }
   public int getX() { return xPos; }
   public void setY(int y) {
      if (y >= 0) yPos = y;
   }
   public int getY() { return yPos; }
   public void setBreite(int breite) {
      if (breite > 0) this.breite = breite;
   }
   public int getBreite() { return breite; }
   public void setHoehe(int hoehe) {
      if (hoehe > 0) this.hoehe = hoehe;
   }
   public int getHoehe() { return hoehe; }
   public void paint() {
      System.out.println("------");
      System.out.println("|    |");
      System.out.println("------");
   }
}
```

```java
public class Kreis {

   private int xPos;
   private int yPos;
   private int radius;

   public Kreis(int x, int y, int radius) {
      setX(x);
      setY(y);
      setRadius(radius);
   }
   public void setX(int x) {
      if (x >= 0) xPos = x;
   }
   public int getX() { return xPos; }
   public void setY(int y) {
      if (y >= 0) yPos = y;
   }
   public int getY() { return yPos; }
   public void setRadius(int radius) {
      if (radius > 0) this.radius = radius;
   }
   public int getRadius() { return radius; }
   public void paint() {
      System.out.println("/---\\");
      System.out.println("|   |");
      System.out.println("\\---/");
   }
}
```

Problem: Redundanz

- Ein beträchtlicher Teil des Codes ist redundant.
- Redundanz erfordert bei Änderungen eine Synchronisierung. Beispiel:

```
public void setX(int x) {
   if (x >= 0) xPos = x;
}
```



```
public void setX(int x) {
   if (x > 0) xPos = x;
}
```

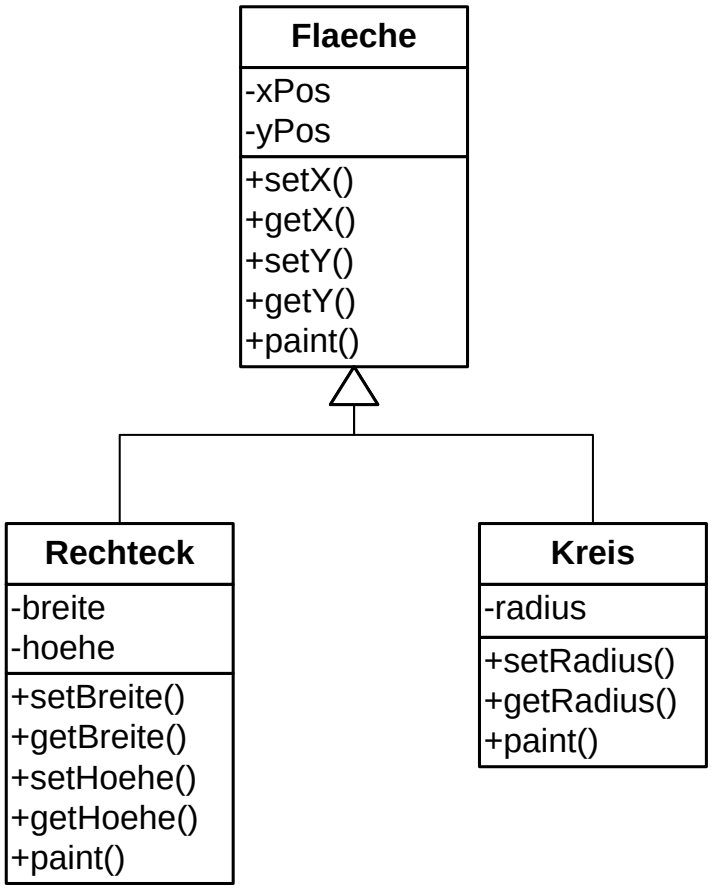
Das müsste in beiden Klassen angepasst werden!

Merksatz: Redundanz ist böse!

Lösung: Vererbung

- OOP erlaubt das Vererben von Klassen
- Die gemeinsamen Bestandteile aus `Rechteck` und `Kreis` lassen sich in eine gemeinsame **Oberklasse** auslagern.
- `Rechteck` und `Kreis` werden dann jeweils als deren **Unterklasse** definiert.

UML-Klassendiagramm



Die Oberklasse `Flaeche`

```
public class Flaeche {

   private int xPos;
   private int yPos;

   public Flaeche(int x, int y) {
      setX(x);
      setY(y);
   }
   public void setX(int x) { if (x >= 0) xPos = x; }
   public int getX()       { return xPos; }
   public void setY(int y) { if (y >= 0) yPos = y; }
   public int getY()       { return yPos; }

   public void paint() {
      System.out.println("XXX");
   }
}
```

Sieht nicht sehr sinnvoll aus. Lösung folgt…

Die Unterklassen

```
public class Rechteck extends Flaeche {
   private int breite;
   private int hoehe;

   public Rechteck(int x, int y,
           int breite, int hoehe) {
      super(x, y);
      setBreite(breite);
      setHoehe(hoehe);
   }
   public void setBreite(int breite) {
      if (breite > 0) this.breite = breite;
   }
   public int getBreite() { return breite; }
   public void setHoehe(int hoehe) {
      if (hoehe > 0) this.hoehe = hoehe;
   }
   public int getHoehe() { return hoehe; }
   public void paint() {
      System.out.println("------");
      System.out.println("|    |");
      System.out.println("------");
   }
}
```

Bestimmt Oberklasse

Konstruktor der Oberklasse

```
public class Kreis extends Flaeche {
   private int radius;

   public Kreis(int x, int y, int radius) {
      super(x, y);
      setRadius(radius);
   }
   public void setRadius(int radius) {
      if (radius > 0) this.radius = radius;
   }
   public int getRadius() { return radius; }

   public void paint() {
      System.out.println("/---\\");
      System.out.println("|   |");
      System.out.println("\\---/");
   }
}
```

super

- Im Konstruktor muss `super` die erste Anweisung sein.
- Wird `super` nicht explizit angegeben, so wird implizit der Defaultkonstruktor aufgerufen. Ist kein solcher vorhanden, führt das Weglassen von `super` zu einem Compilerfehler.

```
public class Rechteck extends Flaeche {
   private int breite;
   private int hoehe;

   public Rechteck(int x, int y,
           int breite, int hoehe) {
      super(x, y);
      setBreite(breite);
      setHoehe(hoehe);
   }
   …
}
```

Reihenfolge der Konstruktorenaufrufe

- Die Konstruktoren werden der Vererbungshierarchie entlang von *oben* nach *unten* aufgerufen (`super` ist immer die erste Anweisung…).
- Da die Unterklasse grundsätzlich auf die Attribute der Oberklasse zugreifen kann und nicht umgekehrt, müssen die Attribute der Oberklasse zuerst initialisiert werden.

toString()

- ```
  Rechteck r = new Rechteck(10, 10, 50, 80);
  System.out.println(r.toString());
  ```

  Das funktioniert und führt zu folgender Ausgabe:

  ```
  Rechteck@82ba41
  ```

- Aber woher stammt die Methode `toString()`?
- Antwort: Sie ist in `Object` definiert, der gemeinsamen Oberklasse aller Klassen.
- Folgendes führt zum gleichen Ergebnis, da `println()` standardmäßig die `toString()`-Methode eines beliebigen Objekts aufruft:

  ```
  System.out.println(r);
  ```
  ```

Die Klasse `Object`

- `Object` ist implizit die direkte Oberklasse aller Klassen, die explizit keine Oberklasse bestimmen.
- Dies gilt mit Ausnahme der Klasse `Object` selbst.
- Damit ist `Object` direkt oder indirekt Oberklasse aller Java-Klassen.
- `Object` besitzt einen Defaultkonstruktor (daher ist kein `super` notwendig).

Methoden der Klasse `Object`

`Object()`	`Object` hat einen Defaultkonstruktor…
`Object clone()`	Gibt eine (echte) Kopie des Objekts zurück
`boolean equals( Object obj)`	Gibt an, ob das Objekt einem anderen Objekt `obj` „gleicht“.
`void finalize()`	Wird vom Gargabe Collector vor dem Zerstören des Objekts aufgerufen, um Ressourcen freizugeben
`Class<?> getClass()`	Gibt Informationen über die Klasse zurück (Reflection)
`int hashCode()`	Gibt einen Hashcode zurück
`String toString()`	Gibt eine String-Repräsentation des Objekts zurück

Vererbung allgemein

- Eine Oberklasse wird mit dem Schlüsselwort `extends` definiert.
- Durch Vererbung wird eine Klassenhierarchie gebildet; mit der Klasse `Object` als Wurzel.
- Die Vererbung verläuft kaskadierend, d.h. eine Klasse erbt nicht nur die Attribute und Methoden ihrer unmittelbaren Oberklasse, sondern auch solche, die diese wiederum von ihrer Oberklasse geerbt hat usw.
- Vererbung bezieht sich auf Klassen (nicht auf Objekte). Es ist also ein statisches Konzept.
- In Java gibt es nur Einfachvererbung und keine Mehrfachvererbung, d.h. eine Klasse kann nur *eine* direkte Oberklasse haben.

Typumwandlung

- Typumwandlung allgemein
- Typumwandlung elementarer Typen (Wiederholung)
- Typumwandlung abstrakter Typen
- Vergleich elementar und abstrakt

Typumwandlung allgemein

- Bei der Typumwandlung wird eine Variable eines Typs in jene eines anderen Typs konvertiert.
- In Java ist keine beliebige Typumwandlung möglich, sondern Quell- und Zieltyp müssen zueinander kompatibel sein.
- Typumwandlung wird auch als **Casten** bezeichnet.
- Ein statisch typsicheres Casten ist in Java implizit möglich.
- Ein statisch nicht-typsicheres Casten ist in Java explizit (also mit Cast-Operator) erlaubt, wenn die beteiligten Typen zueinander kompatibel sind. In diesem Fall ist der Programmierer verantwortlich, dass zur Laufzeit kein Fehler auftritt.

Typumw. elementarer Typen (Wdh.)

- ```
  int i = 5;
  short s = i;
  ```
- ➔ Compilerfehler! Korrekte Umwandlung kann nicht in jedem Fall sichergestellt werden und ist daher implizit nicht erlaubt.

- ```
  int i = 5;
  short s = (short)i;
  ```
- ➔ OK! Wir als Programmierer übernehmen die Verantwortung für ein korrektes Casten.

- ```
  short s = 5;
  int i = s;
  ```
- ➔ Kein Problem! `int` bietet mehr Platz als `short`. Die Typumwandlung ist immer sicher.
  ```

Typumw. elementarer Typen (Wdh.)

Umwandlung von Typ der Spalte nach Typ der Zeile:

	byte	short	int	long	float	double	boolean	char
byte		e	e	e	e	e	x	e
short	i		e	e	e	e	x	e
int	i	i		e	e	e	x	i
long	i	i	i		e	e	x	i
float	i	i	i	i		e	x	i
double	i	i	i	i	i		x	i
boolean	x	x	x	x	x	x		x
char	e	e	e	e	e	e	x	

- i: implizite Typumwandlung möglich
- e: nur explizite Typumwandlung möglich (Cast-Operator notwendig)
- x: keine Typumwandlung möglich

Typumwandlung abstrakter Typen

- Typen entlang der Vererbungshierarchie sind zueinander kompatibel.
- Ein (typsicheres) Casten zu einer (direkten oder indirekten) Oberklasse ist implizit möglich.
- Ein (nicht-statisch-typsicheres) Casten zu einer (direkten oder indirekten) Unterklasse ist explizit möglich.

Beispiele

- `Rechteck r1 = new Rechteck(0, 0, 30, 50);`
  `Kreis k = (Kreis)r1;`

➔ Compilerfehler! `Rechteck` und `Kreis` sind nicht zueinander kompatibel.

- `Flaeche f = r1;`

➔ OK! Hin zur Oberklasse kann man implizit casten.

- `Rechteck r2 = (Rechteck)f;`

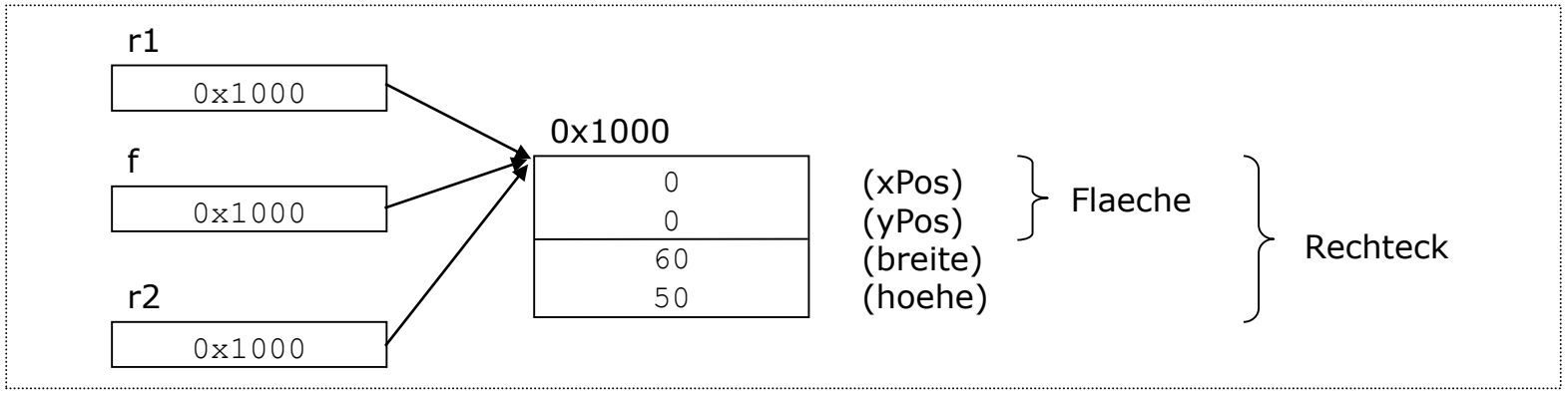
➔ OK, da `Flaeche` zu `Rechteck` kompatibel ist.

➔ Es führt zur Laufzeit zu keinem Fehler, weil sich hinter `f` tatsächlich eine `Rechteck`-Instanz verbirgt.

Vergleich elementar u. abstrakt

```
int i = 5;
short s = (short)i;
```


```
Rechteck r1 = new Rechteck(0, 0, 30, 50);
Flaeche f = r1;
Rechteck r2 = (Rechteck)f;
```



Polymorphie

- `Flaeche f = new Rechteck(0, 0, 30, 50);`
  `f.paint();`
- Welche `paint()`-Methode wird aufgerufen?

Alternative 1

```
public class Flaeche {

   …

   public void paint() {
      System.out.println("XXX");
   }
}
```

Alternative 2

```
public class Rechteck extends Flaeche {

   …

   public void paint() {
      System.out.println("------");
      System.out.println("|    |");
      System.out.println("------");
   }
}
```

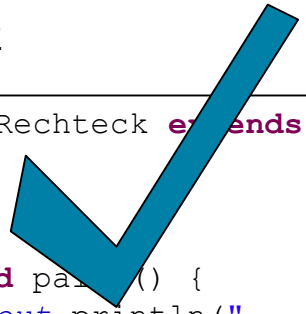
Polymorphie

- `Flaeche f = new Rechteck(0, 0, 30, 50);`
  `f.paint();`
- Welche `paint()`-Methode wird aufgerufen?

○ Alternative 1

```
public class Flaeche {

   …

   public void paint() {
      System.out.println("XXX");
   }
}
```

◉ Alternative 2



```
public class Rechteck extends Flaeche {

   …

   public void paint() {
      System.out.println("------");
      System.out.println("|    |");
      System.out.println("------");
   }
}
```

Erläuterung

- `f` ist als `Flaeche` deklariert und nicht als `Rechteck`.
- I.d.R. kann zur Übersetzungszeit der tatsächliche Typ hinter `f` nicht bekannt sein. Üblicherweise variiert er sogar.
- Also muss zur Laufzeit bzw. dynamisch entschieden werden, welche Methode aufgerufen wird.
- Man bezeichnet das entsprechende Konzept als Polymorphie.

Späte Bindung

- Welche Methode aufgerufen wird, wird zur Laufzeit entschieden. Das bezeichnet man als **dynamische bzw. späte Bindung** der Methode.
- Im Gegensatz dazu passiert eine **statische bzw. frühe Bindung** zur Übersetzungszeit.

Begriffserklärung

- Polymorphie leitet sich aus dem Griechischen ab und bedeutet soviel wie *Vielgestaltigkeit*.
- Die als `Flaeche` deklarierte Variable kann zur Laufzeit vom Typ einer beliebigen Unterklasse sein. Entsprechend unterschiedlich kann ihr Verhalten sein (ihre Gestalt…).

Polymorphie in Java

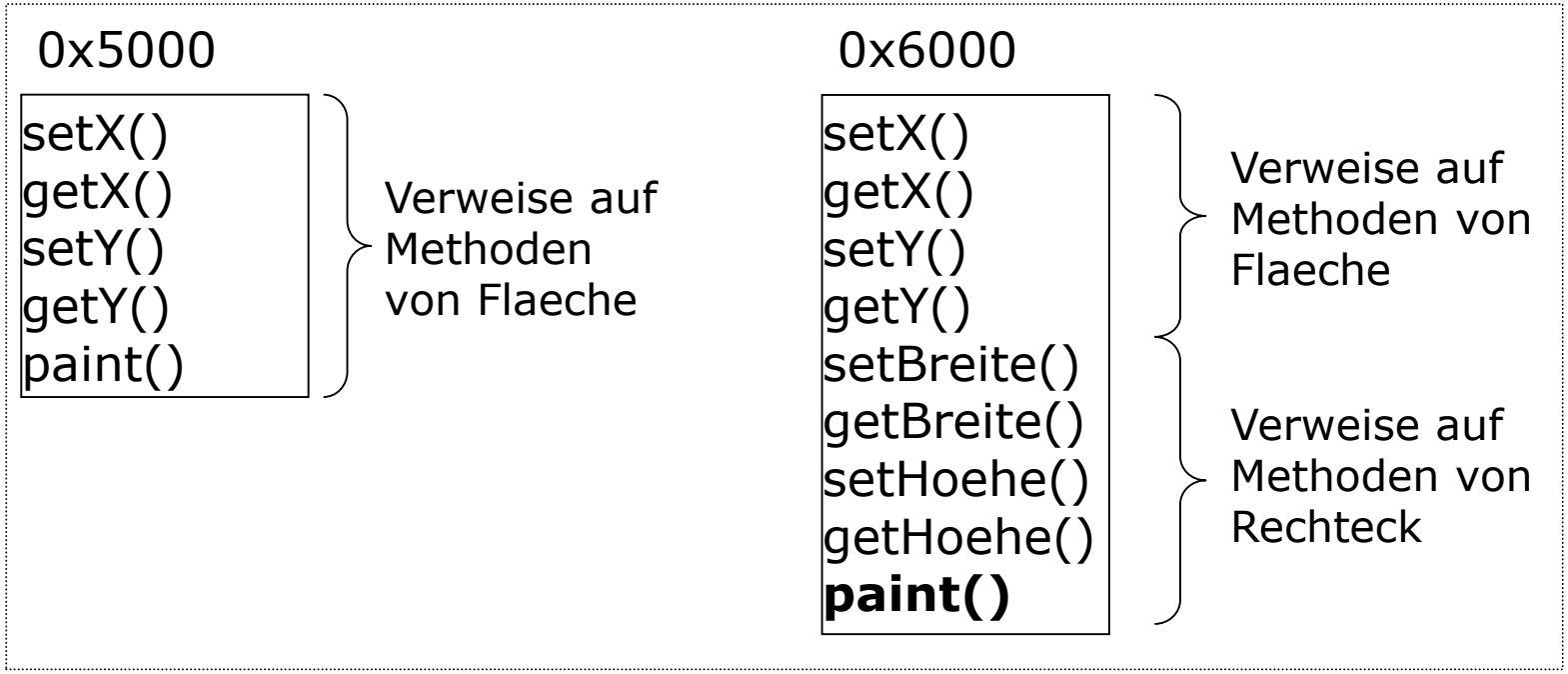
- Späte Bindung bzw. Polymorphie wird in Java grundsätzlich unterstützt.
- Kein spezielles Schlüsselwort (wie `virtual` in C#/C++) ist zur Aktivierung notwendig.

Überschreiben von Methoden

- Wird in einer Unterklasse eine Methode der entsprechenden Oberklasse wiederholt definiert, so *überschreibt* die Methode der Unterklasse die Methode der Oberklasse.

Virtuelle Methodentabellen (VMT)

- Späte Methodenbindung erfolgt zur Laufzeit mithilfe von VMTs.
- Im Arbeitsspeicher befindet sich für jede Klasse eine VMT, die Einsprungadressen zu den Methodenimpl. beinhaltet, z.B.:



Virtuelle Methodentabellen (2)

- `Rechteck r1 = new Rechteck(0, 0, 60, 50);`
- r1 im Speicher:

- `Flaeche f = r1;`
  `f.paint();`

➔ `paint()`-Methode von `Rechteck` wird aufgerufen.

Nachteile der Polymorphie

- Performanceeinbußen durch die indirekte Methodenadressierung.
- Zusätzlicher Speicherplatz für VMTs ist notwendig.

Polymorphie allgemein

- Polymorphie ist ein dynamisches Konzept.
- Polymorphie wird mittels VMTs realisiert.
- Polymorphie erlaubt Aufrufe von Methoden in Unterklassen („von oben nach unten“).

Weitere OO-Konzepte in Java

- Abstrakte Methoden und abstrakte Klassen
- Innere Klassen
- Interfaces

Abstrakte Methoden und -Klassen

```
public class Flaeche {

   private int xPos;
   private int yPos;

   public Flaeche(int x, int y) {
      setX(x);
      setY(y);
   }
   public void setX(int x) { if (x >= 0) xPos = x; }
   public int getX()       { return xPos; }
   public void setY(int y) { if (y >= 0) yPos = y; }
   public int getY()       { return yPos; }

   public void paint() {
      System.out.println("XXX");
   }
}
```

Wir haben `paint()` nur, um Polymorphie anzuwenden.

Abstrakte Methoden und –Klassen (2)

```
public abstract class Flaeche {

   private int xPos;
   private int yPos;

   public Flaeche(int x, int y) {
      setX(x);
      setY(y);
   }
   public void setX(int x) { if (x >= 0) xPos = x; }
   public int getX()       { return xPos; }
   public void setY(int y) { if (y >= 0) yPos = y; }
   public int getY()       { return yPos; }

   public abstract void paint();
}
```

AKs sind nicht instanziierbar.

AMs müssen in Unterklassen implementiert werden.

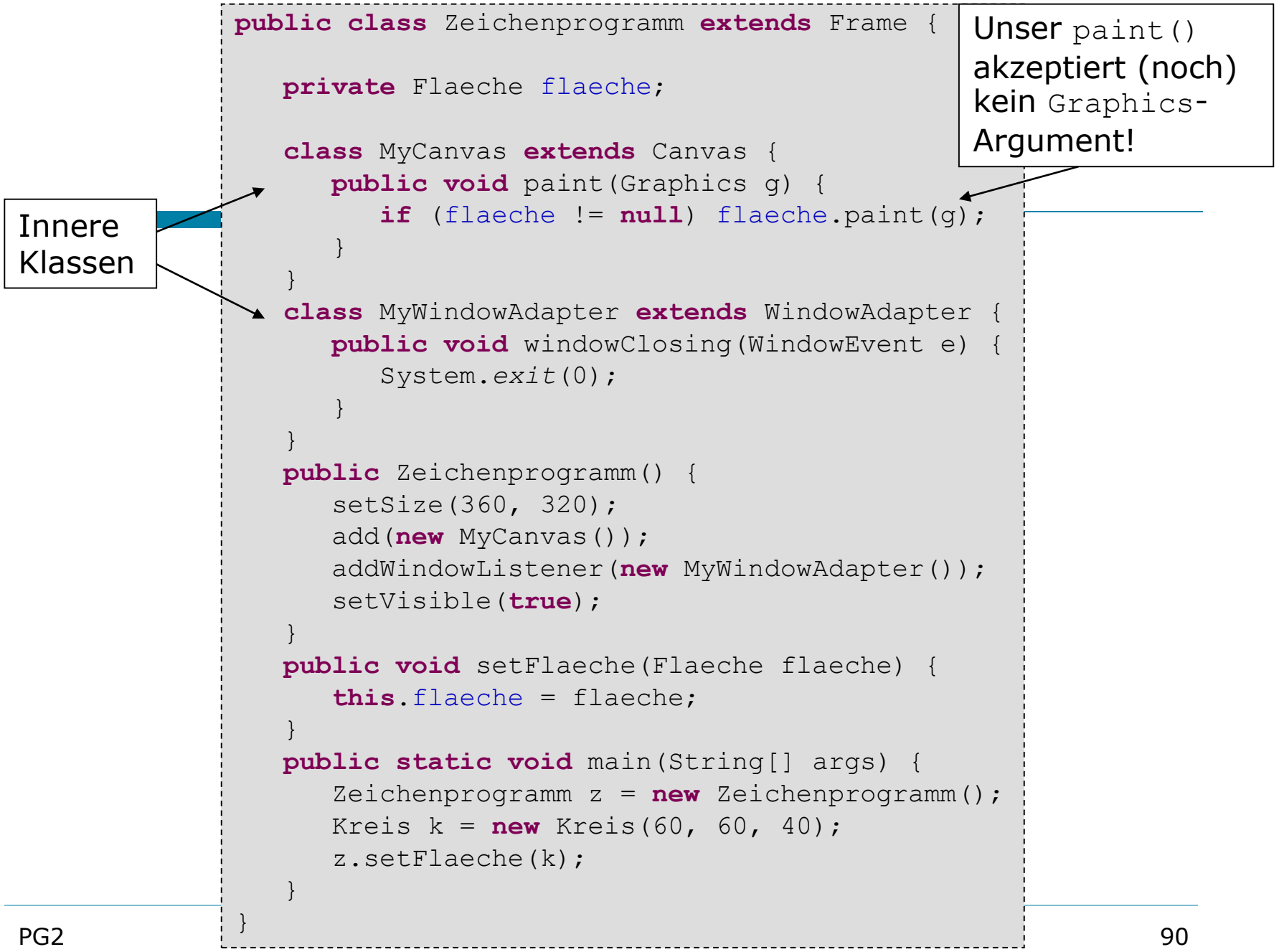
AMs können nur in AKs definiert werden.

AMs besitzen keine Implementierung.

Innere Klassen (u.a.)

Ziele:

- Erstellen eines *Zeichenprogramms*
- Rechtecke und Kreise sollen auf einer grafischen Oberfläche ausgegeben werden.




```
public class Zeichenprogramm extends Frame {

   private Flaeche flaeche;

   class MyCanvas extends Canvas {
      public void paint(Graphics g) {
         if (flaeche != null) flaeche.paint(g);
      }
   }
   class MyWindowAdapter extends WindowAdapter {
      public void windowClosing(WindowEvent e) {
         System.exit(0);
      }
   }
   public Zeichenprogramm() {
      setSize(360, 320);
      add(new MyCanvas());
      addWindowListener(new MyWindowAdapter());
      setVisible(true);
   }
   public void setFlaeche(Flaeche flaeche) {
      this.flaeche = flaeche;
   }
   public static void main(String[] args) {
      Zeichenprogramm z = new Zeichenprogramm();
      Kreis k = new Kreis(60, 60, 40);
      z.setFlaeche(k);
   }
}
```

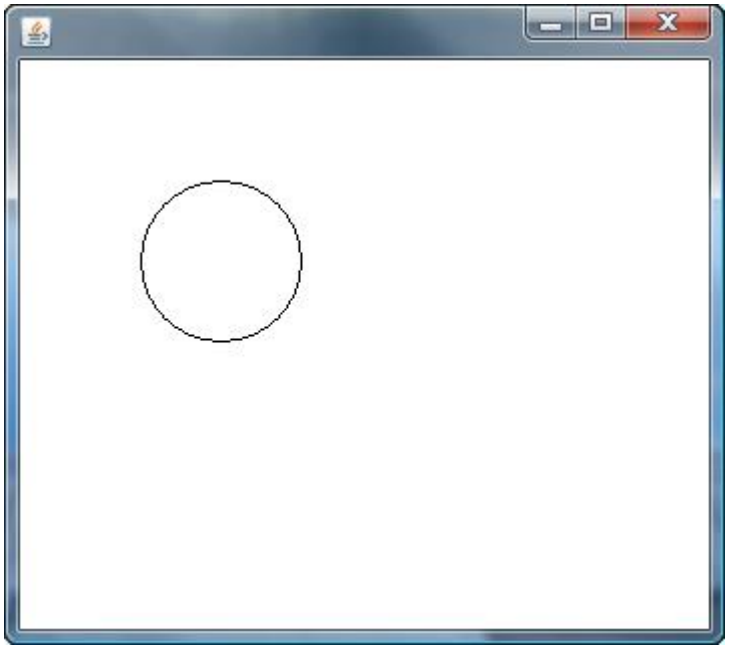
Anpassen der `paint()`-Methoden

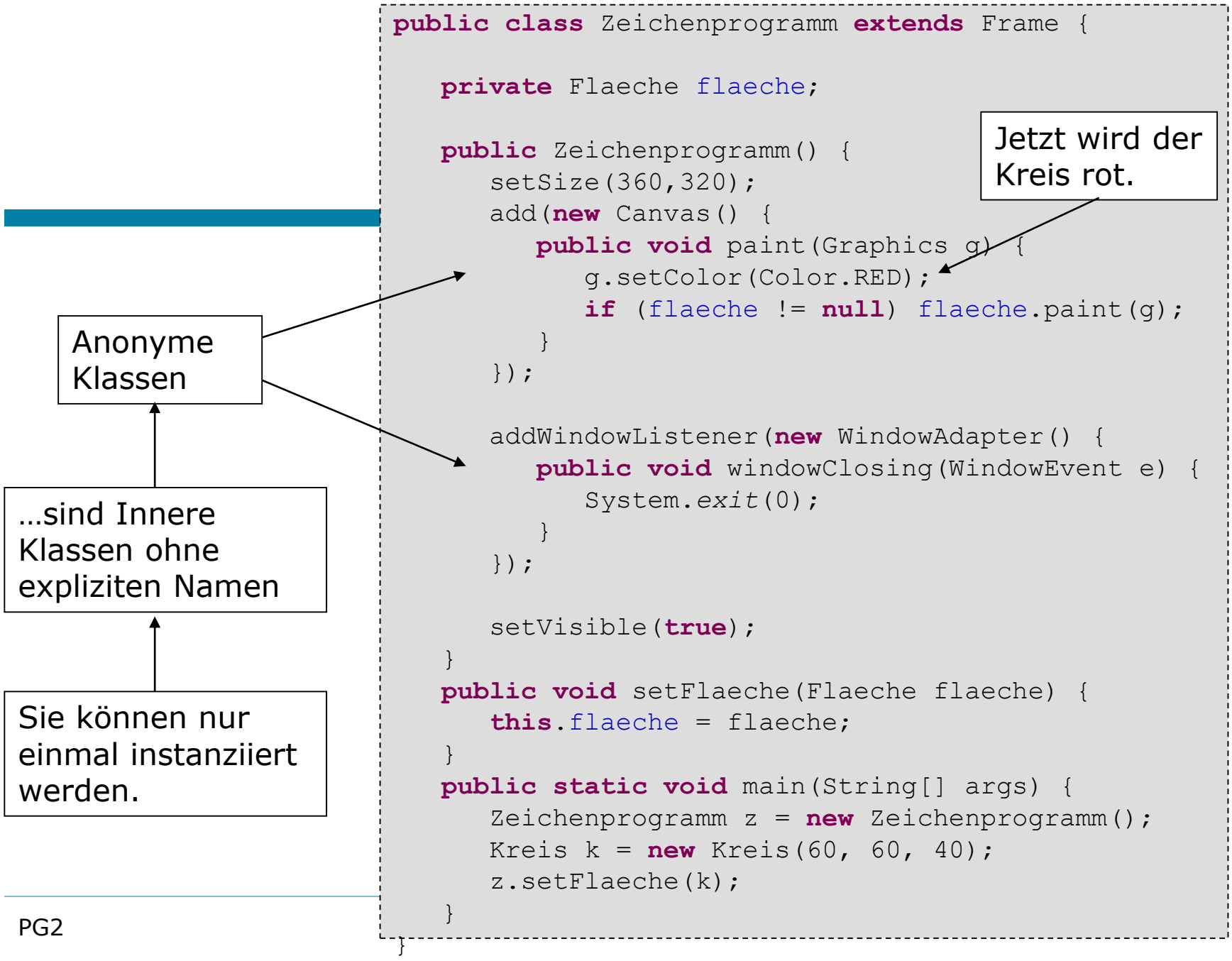
```
public abstract class Flaeche {
    …
    public abstract void paint(Graphics g);
}
```

```
public class Kreis extends Flaeche {
    …
    public void paint(Graphics g) {
        g.drawOval(getX(),getY(),radius*2,radius*2);
    }
}
```

```
public class Rechteck extends Flaeche {
    …
    public void paint(Graphics g) {
        g.drawRect(getX(), getY(), breite, hoehe);
    }
}
```

Das Ergebnis




public class Zeichenprogramm extends Frame {
private Flaeche flaeche;
public Zeichenprogramm() {
setSize(360,320);
add(new Canvas() {
public void paint(Graphics g) {
g.setColor(Color.RED);
if (flaeche != null) flaeche.paint(g);
}
});
addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {
System.exit(0);
}
});
setVisible(true);
}
public void setFlaeche(Flaeche flaeche) {
this.flaeche = flaeche;
}
public static void main(String[] args) {
Zeichenprogramm z = new Zeichenprogramm();
Kreis k = new Kreis(60, 60, 40);
z.setFlaeche(k);
}
}
Jetzt wird der Kreis rot.
Anonyme Klassen
…sind Innere Klassen ohne expliziten Namen
Sie können nur einmal instanziiert werden.

Besonderheiten von Inneren Klassen

- Instanzen Innerer Klassen haben Zugriff auf sämtliche (auch private) Attribute und Methoden der umschließenden Klasse.
- Sie haben Zugriff auf jene umschließende Instanz von/über der sie erzeugt wurden.
- Ggf. muss die umschließende Instanz für den Zugriff mit dem Klassennamen qualifiziert werden, z.B.:
  - `Zeichenprogramm.this`
  - `Zeichenprogramm.this.flaeche`

Interfaces

- Interfaces besitzen keine Methoden und keine Attribute.
- Sie besitzen Methodendeklarationen.
- Sie sind im Wesentlichen wie abstrakte Klassen mit ausschließlich abstrakten Methoden.
- Ein Klasse kann nur eine Oberklasse haben, aber beliebig viele Interfaces implementieren.

Beispiel

```
public interface I {
    void a();
    void b();
}
```

```
public class C implements I {

    public void a() {
        // Implementierung von a()…
    }
    public void b() {
        // Implementierung von b()…
    }
}
```

`C` verpflichtet sich damit `a()` und `b()` zu implementieren.

Warum Interfaces?

- Erweitert die Restriktion der Einfachvererbung.
- Mindert dabei die Probleme der Mehrfachvererbung.
- Erlaubt eine erweiterte Nutzung der Polymorphie.

[bs] Dependency Injection

- Beispiel einer engen Kopplung:

```
class A {
   private B x;

   public A() {
      x = new B();
   }
}
```

```
class B {
   public void f() { }
}
```

- A ist statisch an B gebunden.

[bs] Dependency Injection

- Beispiel einer losen Kopplung mittels DI:

```
class A {
   private I x;

   public void setDep(I x) {
      this.x = x;
   }
}
```

```
interface I {
   public void f();
}

class B implements I {
   public void f() { }
}
```

- A kann dynamisch an B gebunden werden.

[bs] Dependency Injection

- Die Abhängigkeit zu B wird zur Laufzeit „injiziert".
- B könnte zur Entwicklungszeit von A noch gar nicht programmiert sein…
- DI schafft eine typsichere, dynamische Adaptierbarkeit.
- Technische Grundlage für DI bilden Vererbung und Polymorphie
- Für eine ausführliche Beschreibung zu Dependency Injection siehe folgenden Artikel von Martin Fowler: http://martinfowler.com/articles/injection.html

Von der Prozeduralen zur Objektorientierten Programmierung

- Programmierparadigmen definieren grundlegend eine Art und Weise der Programmierung.
- Programmiersprachen unterstützen Programmierparadigmen durch ihre Syntax.
- Programmiersprachen können mehrere Programmierparadigmen unterstützen.

Von der Prozeduralen zur Objektorientierten Programmierung (2)

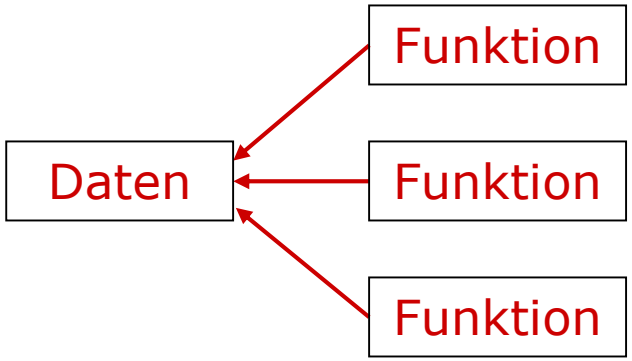
- Programmierung im Kleinen
  - Fokussiert die Entwicklung einzelner Algorithmen.
  - Java unterstützt die Strukturierte Programmierung, d.h. Programmierung mit Schleifen und Verzweigungen und keine beliebigen Sprünge (GOTO-Befehl)
  - Strukturierte Programmierung schafft Abstraktion, d.h. es schränkt die Flexibilität ein und macht die Programme leichter verständlich.

Von der Prozeduralen zur Objektorientierten Programmierung (3)

- Programmierung im Großen
  - Fokussiert i.W. die Anordnung von Funktionen und globalen Daten.
  - Java unterstützt die Objektorientierte Programmierung, d.h. insb. die Datenkapselung
  - Objektorientierte Programmierung schafft Abstraktion, d.h. es schränkt die Flexibilität ein und macht die Programme leichter verständlich.

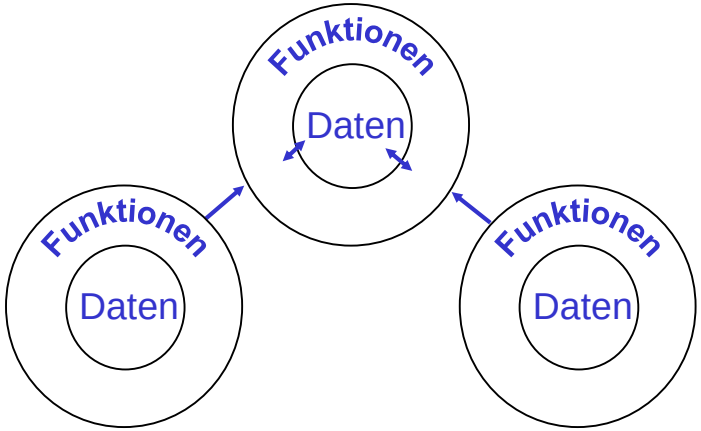
Von der Prozeduralen zur Objektorientierten Programmierung (4)

Prozedural



- Beliebiger Datenzugriff
- Gefahr von Inkonsistenzen

Objektorientiert



- Funktionen der Klasse kapseln ihre Daten
- Gefahr von Inkonsistenzen ist geringer, da weniger Funktionen untereinander abgestimmt sein müssen.

Zusammenfassung OOP

- Klassen beschreiben die statischen Eigenschaften ähnlicher Objekte.
- Klassen bestehen aus Attributen und Methoden.
- Objekte haben einen Zustand (definiert durch Attributwerte) und ein Verhalten.
- Objekte werden mit `new` erzeugt.
- Der Speicher von Objekten wird durch die Garbage Collection freigegeben.
- Methoden kapseln den Zugriff ihrer Attribute und wahren deren Integrität.
- Ein Konstruktor ist eine spezielle Methode zur Initialisierung eines Objekts.

Zusammenfassung OOP (2)

- Basiskonzepte der OOP:
  - Datenkapselung
  - Vererbung
  - Polymorphie
- Während Datenkapselung und Vererbung statische Konzepte sind, handelt es sich bei der Polymorphie um ein dynamisches Konzept.
- Polymorphie wird mittels VMTs realisiert.