Objektorientierung (OO)

Objekte haben Zustände (oder Eigenschaften, Attribute) und Verhalten

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

Verhalten (oder Aktionen): Methoden (*method*s, Funktionen)

*member*s ist der Sammelbegriff für *field*s und *method*s

Klassen

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

Klassen sind für Objekte das, was Typen für Variablen sind

fields

Bsp. einer Klassendefinition mit Feldern:

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

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

eine Instanz von `Circle` kann man jetzt mit `new Circle()` erzeugen

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

methods

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

  void doubleRadius() {
    r = r * 2.0;
  }
}

public class CircleTest1 {

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

Ausgabe:

```
Radius von c1: 1.0
Radius von c2: 2.0
```

Methoden können was zurückgeben

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

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

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

Return

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

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

Return: Mehr Beispiele

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

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

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

Methoden mit Parametern

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

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

Polymorphie

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

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

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

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

Mit Parametern und Rückgabe

```
public class Faktorial {

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

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

Its Call by Value

```
public class ByValueDemo {

  void inc(int i) {
    // keine Ver"anderung bei aufrufenden Wert:
    i = i + 1;
  }

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

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

Referenzen I

Objektvariablen in Java sind *Referenzen* (anders als die Variablen primitiver Typen)

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

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

Referenzen II

entsprechend in einer Methode kann der Inhalt eines Parameter-Objektes modifiziert werden

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

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

void test() {
  Circle c = new Circle();
  c.r = 5.0;
  setCircle(c); // keine Modifikation von c
  System.out.println("c.r="+c.r); // c.r=5.0
  setR(c);      // radius von c wird veraendert
  System.out.println("c.r="+c.r); // c.r=23.0
}
```

Referenzen III

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

  ein Versuch bei `null` auf Member zuzugreifen, führt zu einem Laufzeitfehler (`NullPointerException`)

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

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

Rekursion

Methoden können sich selbst rekursiv aufrufen

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

Zur Erinnerung:

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

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

Namenskonventionen für Methoden

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

Geh mir aus der Sonne ...

Verschattung (*shadowing*): Bezeichner von Lokalen Variablen und Parametern „verschatten“ gleichnamige Objektvariablen.

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

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

Nimm dies!

Lösung: Referenz auf aktuelles Objekt mit `this`

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

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

Konstruktoren I

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

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

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

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

Konstruktoren II

Konstruktoren können wie Methoden Parameter haben; mehrere Konstruktoren (unterschiedlicher Parameterliste) sind erlaubt

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

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

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

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

Konstruktoren III

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

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

  Circle() {
    this(1.0);
  }

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

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

Konstruktoren IV

Ist kein Konstruktor angegeben, so wird automatisch der „*default constructor*“ angelegt (leere Parameterliste, macht nichts)

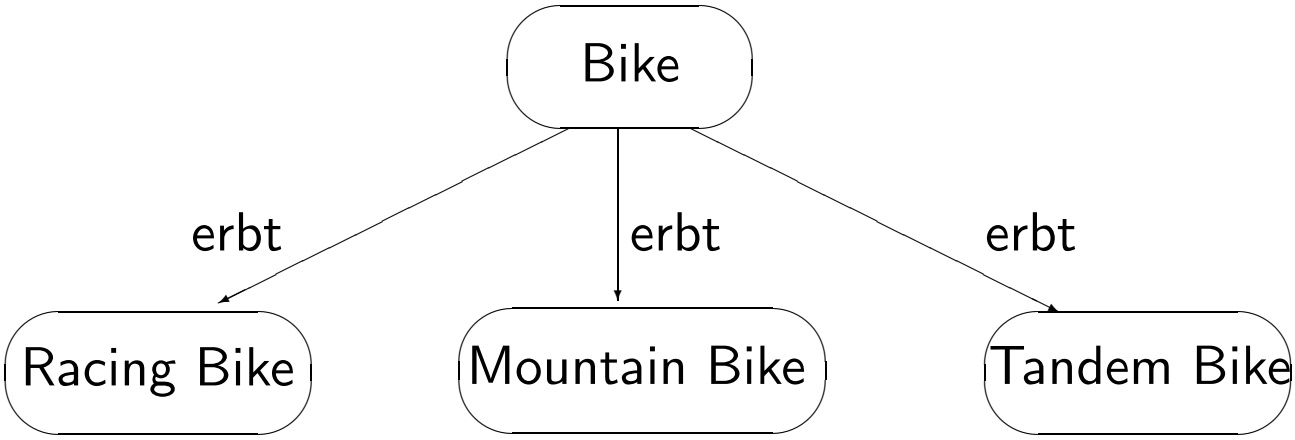
```
public class Circle {
  double x, y, r;

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

Vererbung I

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

Vererbung II



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

Vererbung IV

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

  // ...
}

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

}
```

Vererbung IV

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

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

Zuweisung, Casting

eine Unterklasse kann man einer Variablen von Typ der Oberklasse zuweisen

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

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

```
Rectangle r = new Textbox();

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

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

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

instanceof

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

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

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