Praxis der Programmierung

Vererbung, Kapselung, Klassenglobale Member, Arrays

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

Grundelemente objektorientierter Sprachen (1)

- **Objekt:** Repräsentation eines Objektes der realen Welt
  in der Terminologie objektorientierter Programmiersprachen
  - Zustand eines Objekts ist bestimmt durch Werte seiner **Attribute**
  - reagiert auf **Botschaften** durch ein gewisses Verhalten
- **Klasse:** Objektorientierte Repräsentation eines Konzepts/Begriffs;
  Gesamtheit von Objekten desselben Typs;
  Objekte sind *Exemplare*/*Instanzen* einer Klasse
  - **Datenelemente (Instanz-Variablen)**
    Variablen für die Werte der *Attribute* der Objekte
  - **Methoden** (Funktionen)
    Botschaften, auf die reagiert wird;
    Implementierung bestimmt das Verhalten der Objekte auf eine *Botschaft*

Grundelemente objektorientierter Sprachen (2)

- **Kapselung**
  Interna von Objekten sind nach außen unsichtbar und können von außen nicht manipuliert werden

- **Vererbung**
  Weitergabe von Merkmalen und Fähigkeiten (Datenelementen und Methoden), wenn neue Klassen aus vorhandenen abgeleitet werden

  $\longrightarrow$ hierarchisches Klassensystem

- **Polymorphismus**
  verschiedene Reaktionen von Instanzen verschiedener Unterklassen auf eine gemeinsam verstandene Botschaft

Vererbung und Polymorphismus

Achsenparallele Quadrate in der Ebene (naiv)

```
class Square {
   int a;     // Kantenlaenge
   int x, y;  // Koordinaten

   Square(int x, int y, int a) { ... }
   void moveTo(int x, int y) {
      this.x = x;
      this.y = y;
   }
   void moveRel(int dx, int dy) {
      x += dx;
      y += dy;
   }
   void reSize(int a) {this.a = a;}
   double area() { return a*a; }
}
```

Kreise in der Ebene (naiv)

```
class Circle {
   int a;     // Radius
   int x, y;  // Koordinaten

   Circle(int x, int y, int a) { ... }
   void moveTo(int x, int y) {
      this.x = x;
      this.y = y;
   }
   void moveRel(int dx, int dy) {
      x += dx;
      y += dy;
   }
   void reSize(int a) {this.a = a;}
   double area() { return 3.141*a*a; }
}
```

WRITE ONCE!

Prinzipien bei der Vererbung

- Definition von Klassen (*Unterklassen*), die von einer anderen Klasse (*Oberklasse*) abgeleitet sind
- Vererbung aller Datenelemente und Methoden von der Oberklasse an jede ihrer Unterklassen
- Erweiterung der Oberklasse durch neue Datenelemente und Methoden möglich
  $\rightsquigarrow$ *verdecken gleichnamige Elemente der Oberklasse*
- Re-Definition (Überschreiben) geerbter Methoden (mit derselben Signatur!)
- hierarchische Vererbung (Kinder, Enkel, Urenkel, ...)
  Wurzel: alle Klassen erben implizit von `java.lang.Object`
- *keine* Mehrfachvererbung (kein Erben von mehreren Oberklassen gleichzeitig)

Definition einer Unterklasse

```
class Unterklasse extends Oberklasse {

    // neue Datenelemente

    // ggf. Konstruktoren
    Unterklasse(XXX) {   // ruft als erstes den Standardkonstruktor
                         // der Oberklasse auf
        // weitere Anweisungen
    }

    // neue Methoden
}
```

Vererbung am Beispiel

```
class Auto {
    void hupen() {
        System.out.println("huup");
    }
}

class Pkw extends Auto {
    void airCondition() {
        System.out.println("fauch");
    }

}
```

```
// in main:

   Auto karre = new Auto();
   Pkw pkw = new Pkw();
   karre.hupen()  // "huup"
   pkw.hupen();   // "huup"
   pkw.airCondition();   // "fauch"
   karre.airCondition(); // Fehler
```

Überschreiben einer Methode

```
class Auto {
    void hupen() {
        System.out.println("huup");
    }
}

class Pkw extends Auto {
    void hupen() {
        System.out.println("tuut");
    }
}
```

```
// in main:

   Auto karre = new Auto();
   Pkw pkw = new Pkw();
   karre.hupen()  // "huup"
   pkw.hupen();   // "tuut"
```

Wiederverwendung von Code der Oberklasse

- **Konstruktoren:**
  - automatischer Aufruf des Standardkonstruktors der Oberklasse (muss vorhanden sein; sonst Compilerfehler)
  - gezielter Aufruf eines anderen Konstruktors mit anderer Parameterliste:

```
Unterklasse(XXX) {
    super(YYY);
    // weitere Anweisungen
}
```

  - wenn `super`, dann immer als erste Anweisung in Konstruktor-Implementierungen

- **Methoden:**
  - Aufruf einer Methode der Oberklasse mit `super.methode();`
  - Parameterlisten und Position des Aufrufs beliebig wählbar

Aufruf überladener Konstruktoren

- Mittels `this(`*`Parameterliste`*`)` als erste Anweisung im Konstruktor: Aufruf anderer, überladener Konstruktoren (mit passender Parameterliste)
- Vermeidung von Code-Verdopplung

```
Class Cls {
    // 400 Datenelemente
    int number;

    Cls() { /* Initialisierung der 400 Datenelemente */ }

    Cls(int number) {
       this();   // Aufruf von Cls()
       this.number = number;
    }
}
```

`super`-`this`-**Konflikt**

- Sowohl `super` als auch `this` müssen die erste Anweisung in Konstruktoren sein.
- Ist kein Standardkonstruktor in der Oberklasse, ist `super` unverzichtbar.
- Statt `this`:
  - Auslagern der gemeinsamen Implementierung überladener Konstruktoren in eine Methode
  - Konstruktoren rufen diese Methode auf

super-this-Konflikt – Beispiel

```
class Cls extends Foo {
    // String title wird von Foo geerbt
    // 400 neue Datenelemente

    Cls() {
        this.initialize();
    }

    Cls(String title) {
       super(title);
       initialize();
    }

    void initialize() { /* Initialisierung der 400 Datenelemente */ }
}
```

Zuweisungskompatibilität

- Unterklassenobjekte sind spezialisierte Oberklassenobjekte
- Unterklasse steht für eine Teilmenge der Oberklasse
- Unterklassenobjekte können an Variablen vom Typ der Oberklasse zugewiesen werden:

```
Auto a = new Pkw();
```

Dynamische Bindung

- In Java sind alle Methoden implizit *virtuell*:
- Wird eine überschriebene Methode aufgerufen, entscheidet der Typ des Objekts, für das der Aufruf erfolgt, welche Implementierung abgearbeitet wird.
  $\rightsquigarrow$ Entscheidung erst zur Laufzeit $\rightsquigarrow$ dynamische Bindung
- Der (statische) Typ der Variablen, an den das Objekt gebunden ist, ist *nicht* entscheidend.

Dynamische Bindung – Beispiel

```
class Auto {
    void hupen() {
        System.out.println("huup");
    }
}

class Pkw extends Auto {
    void hupen() {
        System.out.println("tuut");
    }
}
```

```
// in main:

   Auto a = new Auto();
   a.hupen();      // "huup"
   a = new Pkw();
   a.hupen();      // "tuut"
```

Kapselung

Zugriffsmodifikation

- Kapselung: Interna von Objekten von außen nicht sichtbar
- Festlegung für jeden Member (Datenelemente, Methoden, Konstruktoren), welche Objekte (lesend und schreibend) zugreifen können
- Datenelemente müssen nicht immer sichtbar/bekannt sein
  (*Beispiel:* Klassen für bestimmte graphische Objekte (z.B. Fenster))
- Berechtigung, Methoden aufzurufen, kann auf Objekte bestimmter Klassen eingeschränkt werden
- ebenso die Berechtigung, Objekte einer Klasse zu erzeugen
  (Aufruf von Konstruktoren)

Schlüsselwörter zur Zugriffsmodifikation

- Datenelemente/Methoden/Konstruktoren mit dem Modifier

`public`	sind für alle Klassen sichtbar;
`private`	sind nur für die Klasse sichtbar, in der sie vereinbart sind;
ohne Modifier	sind für Klassen aus demselben Paket (Verzeichnis) sichtbar
`protected`	für alle Unterklassen und Klassen aus demselben Paket sichtbar

- Der Modifier ist das *erste* Schlüsselwort in der Definition/ Signatur.
- Klassen dürfen auch Modifier haben. (`public class ...`)
- Schlüsselwort `final` zur Vereinbarung von Konstanten

Beispiel: Klasse gebrochener Zahlen

```
public class Fraction {
   private int numerator, denominator;

   public Fraction(int numerator, int denominator) {...}

   public double getvalue() {
       return numerator / denominator;
   }
```

- interne Darstellung als gemeiner Bruch bleibt verborgen
- Konstruktor stellt sicher, dass nur erlaubte Werte übergeben werden (z.B. Nenner nicht 0)

Getter und Setter

```
public class Date {
   private int day, month, yaer;           // gekapselt

   public int getDay()                     /* Getter-Methoden,
      {return day;}                         *
   public int getMonth() {...}              * damit die Datenelemente
   public int getYear() {...}               * gelesen werden koennen   */

   public void setDay(int day)             /* Setter-Methoden,
      {this.day = day;}                     *
   public void setMonth(int month) {...}    * damit die Datenelemente
   public void setYear(int year) {...}      * veraendert werden koennen */
}
```

Ausnutzung der Kapselung

- Unterscheiden von Lese- und Schreibzugriffen durch gezielte Definition von Gettern und Settern

- Kontrolle über die Art der Zugriffe:

```
public int getDay() {
   return day;
}

protected void setDay(int day) {
   if (0 < day && day < 32)
      this.day = day;
}
```

Konstanten

- werden mit dem Schlüsselwort `final` definiert
- der Wert konstanter Variablen kann nicht verändert werden
- von Klassen, die `final` sind, können keine Unterklassen abgeleitet werden
- Methoden, die `final` sind, können nicht überschrieben werden
- Klassenmethoden gelten implizit als `final`

Klassenglobale Datenelemente und Methoden

Klassenvariablen und -methoden

- werden mit dem Schlüsselwort `static` vereinbart
- sind der Klasse, nicht den Objekten zugeordnet; d.h.
  - **Klassenmethoden** können auch aufgrufen werden, ohne dass ein Objekt der Klasse erzeugt wurde (*Beispiele:* `main`, `Math.sin(double a)`)
  - **Klassenvariablen** sind (mit ihren Werten) allen Exemplaren der Klasse gemeinsam, sind also *klassenglobal* (*Beispiel:* `Math.PI`)
  - Klassenvariablen werden vor dem ersten Exemplar der Klasse, also wie globale Variablen definiert
- werden mit dem Klassennamen (statt mit dem Objektnamen) qualifiziert
- Klassenmethoden dürfen direkt nur auf Datenelemente und Methoden zugreifen, die ebenfalls `static` sind!

Beispiel Klassenvariablen

```
class Static_Beispiel {
   int n;
   static int exCounter = 0;     // wird im Speicher angelegt, bevor
                                 // das erste Objekt der Klasse erzeugt wird
   Static_Beispiel(int n) {
       this.n = n;
       exCounter++;
   }

   int getInstanceVariable() {
       return n;
   }
   static int getExNumber() {
       return exCounter;         // exCounter muss static sein
   }
}
```

Arrays

Arrays (1)

- Arrays sind Verweisdatentypen.
- Arrays stehen alle Methoden zur Verfügung, die von allen Objekten (Instanzen beliebiger Java-Klassen) benutzt werden können.
  (Arrays erben von der Klasse `java.lang.Object`.)
- Array-Elemente werden (wie die Datenelemente von Objekten) automatisch initialisiert.

Aber:

- Es gibt keine Klasse, von der Arrays Instanzen sind.
- Arrays haben keine Konstruktoren. Stattdessen gibt es eine spezielle Syntax des `new`-Operators.

Arrays (2)

- Arrays sind immer **eindimensional**, können aber geschachtelt werden (d.h. Arrays als Elemente enthalten).

- Deklaration:

```
int[] bsp;
int bsp[];
String[][] aStr;
String aStr[][];
```

  Die Anzahl der Elemente wird erst bei der Initialisierung angegeben.

Initialisierung von Arrays

- direkte Initialisierung (gleichzeitig mit der Deklaration):

```
int[] bsp = {5, 4, 3, 2, 1};
String[][] aStr = {{"ja", "nein"}, {"yes", "no"}, {"?"}};
```

- nachträgliche Initialisierung nach Verwendung des new-Operators:

```
bsp = new int[5];   // Standardinitialisierung der Elemente
aStr = new String[3][2];
```

Danach ist z.B. möglich:

```
bsp[0] = 5; bsp[1] = 4; bsp[2] = 3; bsp[3] = 2; bsp[4] = 1;
int laenge = bsp.length;    // ergibt 5
```