Java I – Vorlesung 4
Vererbung und Sichtbarkeit

17.5.2004

Vererbung
Überladen, Überschreiben, Verstecken, Verschatten
Zugriffskontrolle
Statische Members

Wiederholung: OOP

- Programme bestehen aus Klassen.
- Klassen definieren Members:
  - Methoden
  - Felder
- Objekte sind einzelne Instanzen (= Werte) der Klassen. Jedes Objekt hat eigene Kopie der Felder.
- Methoden werden relativ zu Objekten aufgerufen: `kal.istSchaltjahr()`.

Warum OOP?

- Wiederverwendbarkeit von Code
  - heute: Vererbung
- Kapselung: Implementierungsdetails verstecken
  - heute: Zugriffskontrolle

Vererbung

- In oo. Sprachen kann man Klassen von anderen Klassen ableiten.
- Abgeleitete Klasse ("subclass") erbt Felder und Methoden von der Basisklasse ("superclass").
- Objekte der abgeleiteten Klasse können überall verwendet werden, wo Objekte der Basisklasse verwendet werden können.

Wozu Vererbung?

- Modelliert "is-a"-Relationen.
- Wiederverwendbarkeit: Muss Code der Basisklasse nicht neu programmieren.
- Basisklasse definiert Schnittstelle, abgeleitete Klasse implementieren Methoden spezifisch.
- Besonders nützlich für GUI-Programmierung: Erbe von GUI-Klasse, füge eigene Methoden dazu.
- Basisklasse kann Methoden offen lassen, die dann von abgeleiteten Klassen eingesetzt werden.

Vererbung: Ein Beispiel

```
class Kreis {
  int x, y, r;

  int flaecheninhalt() {
    return Math.PI * r * r;
  }
}

class BunterKreis extends Kreis {
  int farbe;
}
```

Vererbung: Ein Beispiel

```
class KreisTest {
  public static void main(String[] args) {
    BunterKreis bk = new BunterKreis();
    bk.r = 3;
    bk.farbe = 6;

    System.out.println(bk.flaechinhalt());
    System.out.println("Radius = " + bk.r);
    System.out.println("Farbe = " + bk.farbe);
  }
}
```


Vererbung: Subtyping

- Sei A Basisklasse und B von A abgeleitet.
- Typkonvertierung von B nach A ist eine erweiternde Typkonvertierung ("up-cast").
- Konsequenz: Variablen von Typ A können Objekte von Typ B enthalten. Methoden mit Parametertyp A können Argumente von Typ B nehmen.
- Variable hat immer noch Typ A, d.h. zur Compilezeit nur Methoden und Felder für Typ A zulässig.
- Zur Laufzeit werden aber Methoden-Implementierungen von Klasse B verwendet ("Überschreiben")!

Subtyping: Ein Beispiel

```
class KreisTest {
  public static void main(String[] args) {
    Kreis k = new BunterKreis();

    k.r = 3;            // das ist ok

    k.farbe = 6;        // Compiler-Fehler

    System.out.println(k.flaechinhalt());
    System.out.println("Radius = " + k.r);
  }
}
```


Überschreiben: Ein Beispiel

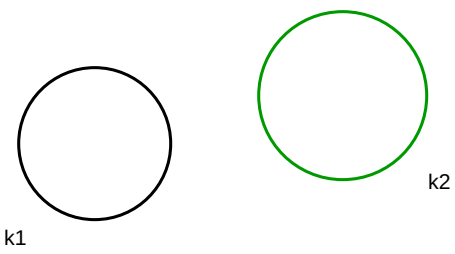
```
class Kreis {
  ...
  void draw() {
    for( .... )  drawPoint( x1, y1 );
  }
}

class BunterKreis extends Kreis {
  ...
  void draw() {
    for( .... )
      drawColorPoint( x1, y1, farbe );
  }
}
```

Überschreiben: Ein Beispiel

```
class KreisTest {
  public static void main(String[] args) {
    Kreis k1 = new Kreis(),
          k2 = new BunterKreis();

    // ... x, y, r, farbe setzen ...

    k1.draw();
    k2.draw();
  }
}
```


Überschreiben: Ein Beispiel

```
class KreisTest {
  public static void main(String[] args) {
    Kreis k1 = new Kreis(),
          k2 = new BunterKreis();

    // ... x, y, r, farbe setzen ...

    k1.draw();
    k2.draw();
  }
}
```



Überschreiben vs. Überladen

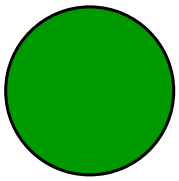
- Überladung: Zwei Methoden haben gleichen Namen, aber verschiedene Parameter. Aufruf wird zur Compilezeit unterschieden.
- Überschreiben: Zwei Methoden mit gleichem Namen und gleichen Parametern, aber eine davon in Basisklasse und eine in abgeleiteter Klasse. Erst zur Laufzeit wird Typ des Objekts geprüft und die richtige Methode ausgewählt.

super

- Methodendefinition in der abgeleiteten Klasse kann auf Definition in der Basisklasse zugreifen.
- Methodenaufruf `super.meth(...)` geht die Klassenhierarchie durch und sucht zuletzt überschriebene Definition von `meth(...)`.
- Vorsicht: `super` allein ist kein gültiger Ausdruck!

super: Ein Beispiel

```
class BunterKreis extends Kreis {
  ...
  void draw() {
    super.draw();

    fillIn(x, y, farbe);
  }
}
```



Vererbung und Konstruktoren

- Konstruktoren werden nicht vererbt.
- Konstruktoren der Basisklasse können mit `super(...)` aufgerufen werden.
- Falls Basisklasse mehrere Konstruktoren hat, ruft `super(...)` den mit den richtigen Parametertypen auf.
- Ein solcher Aufruf von `super(...)` muss die erste Zeile der Konstruktordefinition sein.

Konstruktoren: Ein Beispiel

```
class Kreis {
  Kreis(int px, int py, int pr) {
    x = px;
    // usw.
  }
}

class BunterKreis extends Kreis {
  BunterKreis(int px, int py, int pr,
              int pfarbe) {
    super(px,py,pr);
    farbe = pfarbe;
  }
}
```


Default-Konstruktoren

- Wenn in einer Klasse kein Konstruktor angegeben wird, fügt der Compiler automatisch einen Default-Konstruktor ein:
  `public Klasse() { super(); }`
- Wenn ein Konstruktor keinen Aufruf von `super(...)` enthält, wird automatisch als erstes "`super();`" eingesetzt.
- Das kann zu überraschenden Compile-Fehlern führen, wenn die Basisklasse keinen argumentlosen Konstruktor definiert!

Default-Konstruktoren

```
class Kreis {
  Kreis(int px, int py, int pr) {
    x = px;
    // usw.
  }
}

class BunterKreis extends Kreis {
  BunterKreis(int px, int py, int pr,
              int pfarbe) {
                  // Compiler-Fehler!
    farbe = pfarbe;
  }
}
```


Die Klasse Object

- Wenn bei einer Klassendefinition nicht explizit "extends ..." dabeisteht, wird sie implizit von `Object` (in java.lang) abgeleitet.
- Wichtigste Methoden (zum Überschreiben):
  - `boolean equals(Object o)`: Bin ich gleich zu o?
  - `String toString()`: erzeuge eine String-Repräsentation von mir. Wird automatisch bei +-Operation für Strings aufgerufen.

Verstecken

- Abgeleitete Klasse kann auch Felder mit Namen definieren, die es in Basisklasse schon gibt (z.B. mit anderem Datentyp).
- Das alte Feld wird durch das neue versteckt. Es ist aber immer noch da: `super.feldname`.
- Wenn x ein Ausdruck von Typ A ist, wird immer das Feld der Klasse A verwendet -- selbst wenn x zur Laufzeit Klasse B hat!
- Verstecken ist also etwas anders als Überschreiben.

Verstecken: Ein Beispiel

```
class DoubleKreis extends BunterKreis {
  double x, y, r;

  DoubleKreis(double px, double py,
              double pr, int pfarbe) {
    x = px; y = py; r = pr;
    farbe = pfarbe;
  }
}

...
Kreis k = new DoubleKreis(1,2,3,4);
System.out.println(k.x);
System.out.println(k.farbe);
...
```

Verstecken: Ein Beispiel

```
class DoubleKreis extends BunterKreis {
  double x, y, r;

  DoubleKreis(double px, double py,
              double pr, int pfarbe) {
    x = px; y = py; r = pr;
    farbe = pfarbe;
  }
}

...
BunterKreis k = new DoubleKreis(1,2,3,4);
System.out.println(k.x);       // 0 (int)!
System.out.println(k.farbe);   // 4
...
```


Sichtbarkeit

- In einem Programm kann es Felder und Variablen mit gleichem Namen geben.
- Wie entscheidet der Compiler, welche Definition verwendet wird?
- An jeder Stelle darf nur eine Definition sichtbar sein; die anderen sind verschattet.
- Faktoren:
  - Verstecken
  - Zugänglichkeit
  - lokale Variablen

Sichtbarkeit

- Sichtbarkeit für Methoden funktioniert etwas anders:
  - keine "lokalen Methoden"
  - Überschreiben statt Verschatten
  - Überladung
- Felder und Methoden haben getrennte Namensräume, d.h. kommen sich nicht ins Gehege.

Zugriffskontrolle

- Kapselung: Benutzer der Klasse sollen Implementierungsdetails nicht sehen können.
- Zugriffskontrolle: Verstecke Members so, dass Zugriff aus anderen Klassen ein syntaktischer Fehler ist.
- Schon Compiler verhindert also, dass fremde Programme auf Implementierung zugreifen.

Public und Private

- Ein als `public` deklariertes Member ist aus allen Klassen auf der Welt sichtbar.
- Ein als `private` deklariertes Member ist nur aus Methoden der Klasse selbst sichtbar.
- Ignoriere fürs Erste `protected` und package-weite Zugänglichkeit.
- Gewöhnt euch an, zwischen public und private zu unterscheiden!
- Normalerweise sind Felder private, Methoden public oder private.

Zugriffskontrolle: Ein Beispiel

```
class Kreis {
  private int x, y, r;

  public int flaecheninhalt() {
    return Math.PI * r * r;
  }

  // "Zugriffsfunktion", um x zu lesen
  public int getX() {
    return x;
  }
}
```

Zugriffskontrolle: Ein Beispiel

```
class KreisTest {
  Kreis k = new Kreis(1,2,3);

  System.out.println(k.flaecheninhalt());
  System.out.println(k.x); // Compiler-Fehler
  System.out.println(k.getX());
}
```


Zugänglichkeit

- Eine Feld- oder Methodendefinition heißt zugänglich von einer Stelle im Quelltext, wenn
  - sie in der gleichen Klasse definiert ist, oder
  - die Definition public ist.
- Es gibt noch mehr Fälle, die wir jetzt noch nicht brauchen.

Die Wahrheit über Methodenaufrufe

- Zur Auswertung eines Methodenaufruf-Ausdrucks `o.meth(`a_1`,...,`a_n`)`:
  - Bestimme Laufzeit-Typ von o
  - Bestimme Compilezeit-Typen der Ausdrücke a_i
  - Berechne alle zugänglichen Methoden-definitionen, die zu diesen Typen passen.
  - Bestimme unter diesen Definitionen die spezifischste und rufe sie auf.
- Wenn es keine eindeutige spezifischste Definition gibt, wirft JVM einen Laufzeitfehler.

Lokale Variablen

- Lokale Variablen sind Variablen, die innerhalb einer Methodendeklaration definiert werden.
- Variable ist sichtbar von ihrer Deklaration bis zum Ende des Blocks, in dem sie definiert wird.
- Wo eine lokale Variable (oder Parameter) x sichtbar ist, darf keine weitere lokale Variable dieses Namens definiert werden.
- Lokale Variablen verschatten Felder der Klasse (d.h. diese sind hier nicht sichtbar).

Lokale Variablen: Ein Beispiel

```
class KreisTest {
  public static void main(String[] args) {
    Kreis k1 = new Kreis(),
          k2 = new BunterKreis();

    for( int i = 0; i < 10; i++ ) {
      Kreis k = new Kreis(i,3,1);
      k.draw();
    }
  }
}
```


Lebensdauer von Variablen

- Lebensdauer eines Feldes ist die Lebensdauer des Objekts, zu dem es gehört.
  - wird erzeugt, wenn das Objekt erzeugt wird
  - ein Exemplar pro Objekt der Klasse
- Lebensdauer einer lokalen Variable ist ein einziger Aufruf der Methode.
  - wird erzeugt, wenn die Methode aufgerufen wird
  - ein Exemplar pro Methodenaufruf
  - jeder rekursive Aufruf hat sein eigenes Exemplar

Statische Members

- Statische Felder und Methoden gehören nicht zu Objekten, sondern zur Klasse als Ganzes.
- Statisches Feld wird erzeugt, sobald die Klasse geladen wird; ein Exemplar, Punkt.
- Statische Methoden dürfen nur statische Members (und lokale Variablen) verwenden: Die normalen haben ja keinen globalen Wert.

Statische Members

- Deklaration von statischen Members: Schlüsselwort `static`.
- Verwendung von statischen Members:
  - `KlassenName.membername`
  - `objekt.membername`: Verwendung des Members `membername` in Compilezeit-Klasse von `objekt`.
  - `membername`: Verwendung des (evtl. ererbten) Members `membername` in aktueller Klasse.

Statische Members: Ein Beispiel

```
class Zahlengenerator {
  private static int next = 0;

  public static int getNext() {
    return next++;
  }
}

class Test {
  public static void main(String[] args) {
    System.out.println(Zahlengenerator.getNext());
    System.out.println(Zahlengenerator.getNext());
    System.out.println(Zahlengenerator.getNext());
  }
}
```


Beispiele für statische Members

- `main()`: Aufruf "java Klasse" ruft die öffentliche statische Methode main auf, die in Klasse definiert ist.
- `Math.sin`
- `Math.PI`

Statische Members und Verstecken

- Statische Members (auch Methoden!) der abgeleiteten Klasse verstecken statische Members der Basisklasse.
- Funktioniert genauso wie Verstecken von normalen Feldern.
- Subtile Unterschiede, wenn man nichtstatische Members durch statische versteckt. Regel: Nicht machen!

Zusammenfassung

- Vererbung erlaubt uns, Members einer Klasse an eine andere Klasse weiterzugeben.
- Abgeleitete Klasse kann Definitionen überschreiben und verschatten.
- Konstruktoren werden nicht vererbt, aber es gibt Defaultkonstruktoren.
- Zugriffskontrolle dient der Kapselung.
- Statische Members gehören zur Klasse als Ganzes.