Kapitel 10: Typerweiterung und Vererbung

- Wiederverwendung von Klassen
  - Bestehende Klassen sollen für neue Aufgaben verwendet werden.
  - *Typerweiterung*: Erweiterung bestehender Klassen um neue Attribute und Methoden, um zusätzliche Anforderungen zu bewältigen.
  - *Vererbung:* Erweiterte Klassen erben alle Attribute und Methoden der ursprünglichen Klassen.
  - *Redefinition* (*Overriding*): Anstatt das Verhalten der Basisklasse unverändert beizubehalten, können Methoden überschrieben werden.

- Begriffe
  - intensionale Sicht (Beschreibung der Klasse)
    Basisklasse abgeleitete Klasse, erweiterte Klasse
  - extensionale Sicht (Menge der umfassten Objekte):
    Oberklasse Unterklasse

Beispiel ohne Vererbung

Beispiel Unternehmensdatenbank

AngestellteR
name: String gehalt: float
erhöheGehalt (float) toString (): String

```
class AngestellteR {
    String name;
    float gehalt;
    void erhöheGehalt (float f) {gehalt *= f;}
    String toString () {return name + " " + gehalt;}
}
```

ManagerIn
name: String gehalt: float stufe: int
erhöheGehalt (float) toString (): String

```
class ManagerIn {
    String name;
    float gehalt;
    int stufe;
    void erhöheGehalt (float f) {gehalt *= f;}
    String toString () {return name + " " + stufe + " " + gehalt;}
}
```

… Beispielaufgabe

- Aufgabe Gehaltserhöhung
  „Erhöhe alle Gehälter um 3%.“

```
static AngestellteR[] a;        // nur Angestellte, keine Manager
static ManagerIn[] m;           // eigene Liste für Manager nötig

    int i;
    for (i = 0; i < a.length; ++i) a[i].erhöheGehalt (1.03);
    for (i = 0; i < m.length; ++i) m[i].erhöheGehalt (1.03);
```

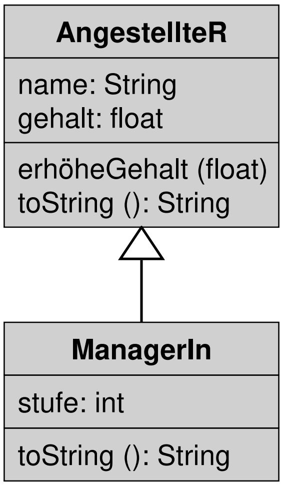

Analyse des Beispiels

- Beobachtungen und Probleme
  - Manager haben (mindestens) dieselben Attribute und Methoden wie Angestellte, da sie selbst auch Angestellte sind.
  - Das spiegelt sich im Klassendiagramm nicht wider:
  - *Redundanz der Implementierung*: Alle Attribute und Methoden von Angestellten sind für Manager noch einmal implementiert.
  - *Inkompatibilität der Typen*: Manager können nicht wie Angestellte verwendet werden.

- Lösungskonzept
  - Erweiterung der Klasse AngestellteR zur Klasse ManagerIn.
  - *Is-a* Beziehung (intensional): „A manager **is-a** employee“.
  - extensionale Sicht: Menge der Angestellten umfasst auch ManagerInnen.

Beispiel mit Vererbung

Beispiel Unternehmensdatenbank




```
class AngestellteR {
    String name;
    float gehalt;
    void erhöheGehalt (float e) {gehalt += e;}
    String toString () {return name + " " + gehalt;}
}
```

```
class ManagerIn extends AngestellteR {
    int stufe;
    String toString () {return name + " " + stufe + " " + gehalt;}
}
```


… Beispielaufgabe (mit Vererbung)

- Aufgabe Gehaltserhöhung
  „Erhöhe alle Gehälter um 3%."

  ```
  static AngestellteR[] a;        // Angestellte inkl. Manager

  for (int i = 0; i < a.length; ++i) a[i].erhöheGehalt (1.03);
  ```

- Beobachtungen
  – Array AngestellteR[] a kann auch ManagerIn-Objekte enthalten.
  – „erhöheGehalt" stimmt für AngestellteR- und ManagerIn-Objekte überein.

  – Mit dem Operator **instanceof** kann der Typ von Objekten geprüft werden:
    if (a[i] **instanceof** ManagerIn) …

Zugriff auf die Basisklasse

- Zugriffsrecht für Unterklassen
  - bisherige Kennzeichen: **public** (öffentlich) und **private** (nicht-öffentlich).
  - **protected**: Attribute und Methoden, die für Unterklassen zugreifbar sind.
- Referenz super
  - Wird eine Methode redefiniert, so ist die Implementierung in der Oberklasse für die Unterklasse nicht mehr unmittelbar zugänglich (*overriding*, vgl. Verschattung von Bezeichnern).
  - Zugriff wird über die spezielle Objektreferenz **super** ermöglicht.
  - Bsp. ((class ManagerIn)):

    String toString () {return **super**.toString() + " " + stufe;}
  - **super** bezeichnet **this** in seiner Rolle als Exemplar der Oberklasse.

Konstruktoren in Unterklassen

- Initialisierung geerbter Attribute
  - Um geerbte Attribute korrekt zu initialisieren, muss ggf. ein Konstruktor der Basisklasse aufgerufen werden.
  - Nützlich bei komplexen Abläufen in den Konstruktoren.
  - Unumgänglich für private Attribute der Basisklasse.
  - Syntax: Aufruf mit **super** und entsprechenden Parametern.
  - Aufruf muss am Anfang des Konstruktors einer Unterklasse stehen.
- Beispiel

```
public AngestellteR (String name, float gehalt) {
    this.name = name; this.gehalt = gehalt;
}
public ManagerIn (String name, float gehalt, int stufe) {
    super (name, gehalt); this.stufe = stufe;
}
```

Polymorphismus

- Begriff
  - Polymorphismus = „Vielgestaltigkeit“
  - Anpassung des strengen Typkonzepts auf Typerweiterungen.
- Polymorphismus von Objekten
  - Ein Objekt ist Exemplar seiner Klasse sowie aller seiner Oberklassen.
  - D.h. Objekt einer Unterklasse darf als Objekt der Oberklasse auftreten.
  - Bsp: Übergabe eines ManagerIn-Objekts als AngestellteR-Parameter.
- Polymorphismus von Objektvariablen
  - Jede Objektvariable (Attribut, formaler Parameter) hat eine Klasse als Typ.
  - Eine Objektvariable kann auch Objekte von Unterklassen referenzieren.
  - Bsp: AngestellteR ang = new ManagerIn (…);
  - Array AngestellteR[] *a* kann auch ManagerIn-Objekte enthalten.

Polymorphismus und Methoden

- Auswirkung des Polymorphismus
  - Sei *p* eine Variable (formaler Parameter, Arrayelement, …) der Klasse *k*, für die die Methode *m* aufgerufen wird: p.m(…)
  - Ein aktuelles Objekt *o* kann Exemplar einer Unterklasse *k‘* von *k* sein.
  - Falls Methode *m* in *k‘* redefiniert wurde, soll für *o* die entsprechende Implementierung aufgerufen werden.
- Bisherige Lösung: Fallunterscheidung

  Fallunterscheidung an Aufrufstellen hat viele Nachteile:
  - Beim Ableiten neuer Unterklassen muss man alle Aufrufstellen erweitern.
  - Erweiterung des Gesamtsystems sehr aufwändig und fehleranfällig.
  - Automatische Zuordnung wäre hilfreich *dynamisches Binden*.

Dynamisches Binden

- Binden zur Laufzeit
  - Automatische Zuordnung der aktuellen Implementierung einer Methode zur Laufzeit (wie bei abstrakten Datentypen, **interface**).

```
void printEmployees (AngestellteR[] a) {          // inkl. Manager
  for (int i = 0; i < a.length; ++i) System.out.println (a[i].toString());
}
```

- Vorteile des dynamischen Bindens
  - Tatsächlicher Typ von aktuellen Objekten ist erst zur Laufzeit bekannt.
  - In der Regel ist nicht einmal bekannt, ob eine Klasse erweitert wurde.
  - Wird eine Methode redefiniert, so funktionieren die verwendenden Programmteile unverändert weiter.
  - Systemerweiterungen lassen sich auf lokale Änderungen beschränken.
    Großer Fortschritt für die Programmierung komplexer Systeme.

Beispiel

- Neue Erweiterung der Klasse **AngestellteR**

```
class WerkstudentIn extends AngestellteR {
   String Studienfach;
   String toString ()     {return super.toString() + ” ” + Studienfach;}
}
```

- Mit Fallunterscheidung

```
void printEmployee (AngestellteR ang) {
   if (ang instanceof ManagerIn)
               System.out.println ((ManagerIn) ang.toString());
   else if (ang instanceof WerkstudentIn)
               System.out.println ((WerkstudentIn) ang.toString());
   else /*ang instanceof AngestelleR*/
               System.out.println (ang.toString());
}
```

- Mit dynamischem Binden

// Methode *printEmployee* funktioniert korrekt ohne Änderung.

Kontrolle der Vererbung

- Beobachtung
  - Für eine Klasse ist im allgem. nicht bekannt, ob es Unterklassen gibt.
  - Die Unabhängigkeit der Oberklassen von möglichen Unterklassen ist ein großer Vorteil für Aufbau und Erweiterbarkeit großer Softwaresysteme.
  - Redefinition könnte aber z.B. Systemsicherheit gefährden.

    Bsp: boolean validatePassword (…) {**return** true;} // in Unterklasse

    Möglichkeit zur Unterbindung von Redefinitionen ist nützlich.

- Vererbungskontrolle in Java
  - Unterbinden der Redefinition einer Methode: **final**
  - Unterbinden der Erweiterung einer Klasse: **final class**
  - Erzwingen der (Re-)Definition einer Methode: **abstract**
  - Erzwingen der Erweiterung einer Klasse: **abstract class**

Unterbinden der Redefinition (**final**)

- Nicht-redefinierbare Methoden
  - Bsp. **final** boolean validatePassword (…) {…}
  - diese Methode darf in erweiterten Klassen nicht überschrieben werden.
  - dadurch Optimierungsmöglichkeiten (auch bei **static** und **private**).

    z.B. Einkopieren (inline) von nicht-rekursiven Methoden

    final int getSpeed () {**return** speed;}

    … 2 * getSpeed() … wird dann zu … 2 * speed …

- Nicht-erweiterbare Klassen
  - Bsp. **final class** SecurityManager {…}
  - Keine Erweiterung möglich, d.h. insbesondere keine Redefinitionen.
  - nützlich zur Erhöhung der Systemsicherheit.
  - ermöglicht weitere statische Prüfungen und Optimierungen.

Erzwingen der Implementierung (**abstract**)

- Abstrakte Methoden
  - Methoden, die nur einen Kopf (= Signatur), aber keinen Rumpf haben.
  - (Syntaktisch) korrekte Verwendung der Methode kann geprüft werden.
  - Bsp: Methodendeklarationen in abstrakten Datentypen (**interface**).
  - Auch Klassen können abstrakte Methoden haben: **abstract** int xyz (…);

- Abstrakte Klassen
  - Klassen mit abstrakten Methoden können nicht direkt instantiiert werden, nur deren Unterklassen, wenn diese alle abstrakten Methoden definieren.
  - Explizite Markierung von abstrakten Klassen: **abstract class** … {…}
  - Abstrakte Klassen sind Zwischenform von Klasse und abstraktem Typ (d.h. können Zustand halten, sind aber nicht direkt instantiierbar).
  - andere Möglichkeit, um unmittelbare Instantiierung zu verhindern: Markierung aller Konstruktoren als **protected**.

Beispiel: Software-Rahmen zur Zeitmessung

- Abstrakte Klasse als Software-Framework

```
abstract class Benchmark {
  abstract void aSingleCall();    // to be benchmarked

  public void doTheBenchmark (String[] arg) {
    int number = Integer.parseInt (arg[0]);
    long start = System.currentTimeMillis();
    for (int i = 0; i < number; ++i) aSingleCall();
    long msecs = System.currentTimeMillis() – start;
    System.out.println (“calls: “ + number);
    System.out.println (“avg duration: “ + float(msecs)/number);
  }
}
```

… Konkretisierung durch Typerweiterung

- Konkretes Beispiel: Zeitmessung von Methodenaufrufen

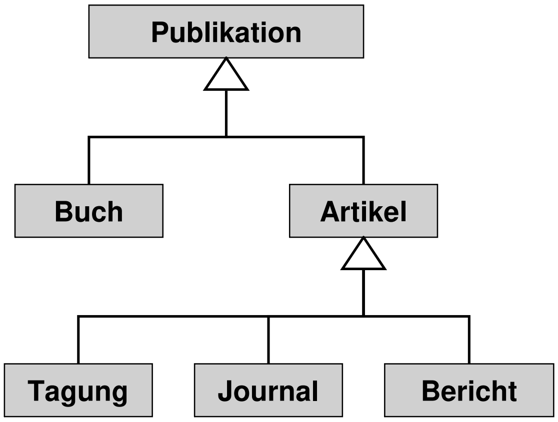
```
class CallBenchmark extends Benchmark {
  void aSingleCall()  { }        // here: just an empty method

  public static void main (String[] arg) {
    Benchmark b = new CallBenchmark();
    b.doTheBenchmark (arg);
  }
}
```

- Organisationsprinzip
  - Abstrakte Klasse bietet allgemeinen Rahmen für viele Aufgaben.
  - Einzelne Erweiterungen können konkrete Aufgaben modellieren.

Mehrfachvererbung

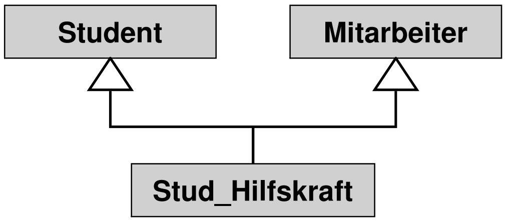
- Einfachvererbung

  Jede Klasse hat höchstens eine direkte Oberklasse.



- Mehrfachvererbung

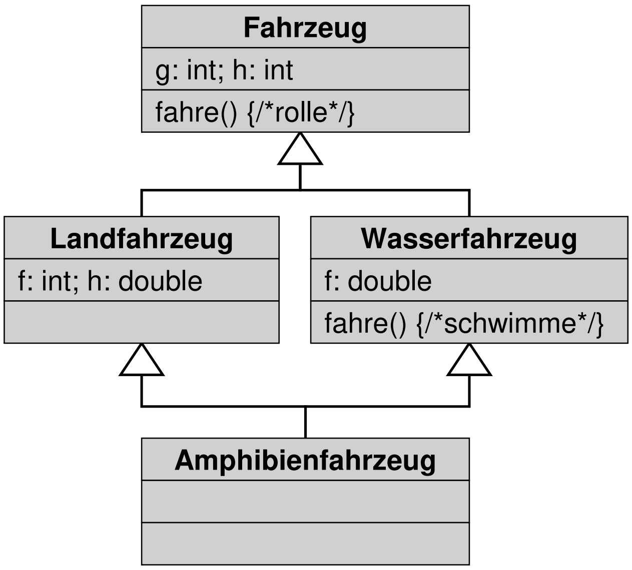
  Klassen können mehrere direkte Oberklassen haben.



Probleme bei Mehrfachvererbung

- Namenskonflikte
  Verschiedene ererbte Komponenten können denselben Namen tragen.
  Bsp. „f", „h"
- Methodenimplementierungen
  Unterschiedliche Versionen für dieselbe geerbte Methode.
  Bsp. „fahre()"
- Attribute
  Bei rautenförmiger Vererbung kann dasselbe Attribut mehrfach geerbt werden.
  Bsp. „g", „h" (verschattet)

- Beispiel



Mehrfachvererbung in Java

- Analyse der Probleme
  - Probleme werden insbesondere durch Implementierungskomponenten hervorgerufen (Attribute, Methodenrümpfe).
  - Keine Probleme durch reine Schnittstellenelemente (Methodenköpfe).

- Lösung in Java
  - Mehrfachvererbung von Klassen wird nicht unterstützt.
  - Abstrakte Datentypen (**interface**) können Mehrfachvererbung bilden, d.h. ein interface kann mehrere andere interfaces erweitern.

    interface StudentTyp {…} **interface** MitarbeiterTyp {…}

    interface StudMitarbTyp **extends** StudentTyp, MitarbeiterTyp {…}
  - Mischformen möglich: Eine Klasse kann eine (einzige) andere Klasse erweitern und gleichzeitig mehrere interfaces implementieren.

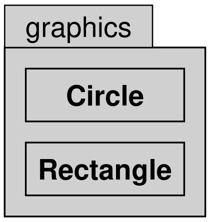
    class HiWi **extends** Person **implements** StudentTyp, MitarbeiterTyp {…}

Pakete

- Motivation
  - Strukturierung großer Mengen von Klassen (Bibliotheken).
  - Organisation von Zugriffsrechten (Sichtbarkeit).
  - Vermeidung von Namenskonflikten.
- Definition von Paketen
  - Deklaration am Dateianfang: **package** *name*;
  - Ohne **package**-Zeile: Standardpaket (nur für Testzwecke verwenden).
  - Einfache Erweiterung von Paketen um neue Klassen.
- Beispiel (Java, UML)

  Datei *Circle.java:*
  package graphics;
  class Circle {…}

  Datei *Rectangle.java:*
  package graphics;
  class Rectangle {…}



Verwendung von Paketen

- Paket als abgeschlossener Namensraum
  - Was zu einem Paket gehört, ist außerhalb des Pakets nicht sichtbar.
  - Innerhalb eines Paketes sind alle Komponenten sichtbar (als Standard; Einschränkungen durch **private** und **protected** möglich).
- Export von Namen

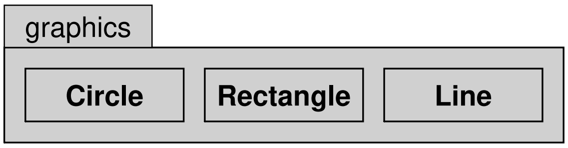
  Zur Verwendung in anderen Paketen können Klassen, Attribute und Methoden mit **public** nach außen sichtbar gemacht werden.
- Import aus Paketen
  - Expliziter Import am Dateianfang: **import** graphics.Circle; … Circle c;
  - Import *aller* Klassen eines Paketes: **import** graphics.*; … Circle c;
  - Punktnotation ermöglicht impliziten Import einer Klasse (d.h. ohne **import**-Zeile am Dateianfang): graphics.Circle c;

Klassen und Pakete im Dateisystem

- Abbildung von Klassen auf Dateien
  Datei Circle.java enthält **class** Circle {…}

- Abbildung von Paketen auf Verzeichnisse
  Verzeichnis /…/graphics/ enthält Paket graphics

  /…/graphics/Circle.java
  /…/graphics/Rectangle.java
  /…/graphics/Line.java



- Abbildung hierarchischer Pakete auf die Verzeichnishierarchie
  package java.io im Verzeichnis /…/java/io/
  package java.util im Verzeichnis /…/java/util/

Vordefinierte Pakete in Java (Beispiele)

- Basisklassen
  - java.lang Basisklassen der Sprache (System, Math, Integer, …)
  - java.util nützliche Bausteine zur Programmierung

- Kommunikation mit dem Rechner
  - java.io Ein- und Ausgabeströme, z.B. für Dateien
  - java.net Zugriff auf Ressourcen im Netz (URLs, Sockets, …)

- Graphische Benutzeroberflächen
  - java.awt Elemente für graphische Benutzeroberflächen
  - javax.swing umfangreichere Bibliothek für graphische Oberflächen
  - java.applet Einbettung von Java-Programmen in HTML-Seiten

Beispiel: *IntList* mit *java.util.Vector*

```
class IntVector implements IntList {
  java.util.Vector elems = new java.util.Vector();

  public void insert (int v) {
    elems.addElement (new Integer (v));
  }

  public boolean contains (int v) {
    return elems.contains (new Integer (v));
  }

  public void delete (int v) {
    elems.removeElement (new Integer (v));
  }
}
```


Basisklasse *java.lang.Object*

- Eigenschaften
  - Klasse *Object* ist implizite Basisklasse jeder anderen Klasse.
  - eignet sich als generischer Platzhalter für beliebige Klasse.

- Methoden (Auswahl)

  public Object clone ();

  Duplizieren eines Objekts

  Typ muss ggf. umgewandelt werden: Circle = (Circle) c.clone();

  public String toString ();

  Textausgabe für Objekte

  vorgegebene Implementierung gibt Adresse des Objekts aus.

  public boolean equals (Object);

  Prüfung zweier Objekte auf Gleichheit

  als Standardimplementierung werden Objektreferenzen verglichen.

  für echten Wertevergleich muss equals redefiniert werden.