Einführung in die Informatik

Extending Class Behavior

Ableitung von Klassen, Vererbung, Polymorphie, Abstraktion

Wolfram Burgard
Cyrill Stachniss

Einleitung

- Bisher haben wir Klassen immer vollständig implementiert, d.h. wir haben alle Instanzvariablen und Methoden der Klassen selbst definiert.
- Ziel dieses Kapitels ist die Vorstellung von Techniken, um neue Klassen ausgehend von bereits bestehenden Klassen zu definieren.
- Im Gegensatz zum bisherigen Vorgehen, bei dem wir Methoden anderer Klassen verwendet haben, um Klassen zu realisieren, werden wir jetzt neue Klassen durch Ableitung bestehender Klassen definieren.

Beispiel: Ein besserer `BufferedReader`

- Die Klasse `BufferedReader` stellt Methoden für das Einlesen von `String`-Objekten zur Verfügung.
- Wann immer wir Objekte von der Tastatur oder aus einer Datei lesen wollen, müssen wir ein `BufferedReader`-Objekt verwenden.
- Um jedoch eine Zahl einzulesen, müssen wir immer zuerst einen String einlesen und danach die Zahl aus diesem String mittels `Integer.parseInt` herauslesen.
- In Situationen, in denen häufig Zahlen eingelesen werden müssen, ist man jedoch an `BufferedReader`-Objekten interessiert, die Zahlen direkt einlesen können.
- Im Folgenden werden wir daher untersuchen, wie wir von der Klasse `BufferedReader` eine neue Klasse ableiten können, die direkt Zahlen einlesen kann.

Zwei Möglichkeiten zur Erweiterung von Klassen

Zur Erweiterung einer Klasse um weitere Funktionalität bestehen im Prinzip zwei Möglichkeiten:

1. Wir ändern die Definition der Originalklasse.

2. Wir definieren eine neue Klasse, die alle Methoden und Instanzvariablen der Originalklasse übernimmt, und fügen lediglich die fehlenden Methoden und Instanzvariablen hinzu. Diesen Prozess nennen wir Ableitung.

Vorteile der Ableitung von Klassen

Die Ableitung von Klassen mit Übernahme der bestehenden Eigenschaften der Originalklasse hat gegenüber einer Erweiterung einer bestehenden Klasse verschiedene Vorteile:

- Die Originalklasse ist üblicherweise gut getestet und es ist nicht klar, welche Konsequenzen Änderungen der Originalklasse haben.
- Bestehende Programme verwenden die Originalklasse bereits und benötigen die neuen Methoden nicht.
- Häufig ist der Code einer Klasse nicht zugänglich. Eine Änderung ist dann nicht möglich.
- In manchen Fällen ist eine Klasse aber auch sehr kompliziert und daher schwer zu verstehen. In diesem Fall ist es nicht empfehlenswert, die bestehende Klasse zu modifizieren.

Vererbung / Inheritance

- Objektorientierte Sprachen wie z.B. Java stellen mit der Möglichkeit Klassen abzuleiten ein sehr mächtiges Konzept zur Verfügung um Klassen zu erweitern ohne den Code der Originalklasse zu modifizieren.
- Diese Technik, die als Vererbung bezeichnet wird, ist eine der grundlegenden Eigenschaften objektorientierter Programmiersprachen.

Ableitung von Klassen durch Vererbung

- Um in Java bestehende Klassen zu erweitern und ihre Verhalten zu erben, verwenden wir das Schlüsselwort `extends` in der Klassendefinition:

```
class BetterBR extends BufferedReader {

      ...

}
```

- Dadurch erbt die Klasse `BetterBR`, die einen erweiterten `BufferedReader` realisieren soll, alle Methoden und Instanzvariablen der Klasse `BufferedReader`.
- Die Vererbung geschieht, ohne dass wir den Source-Code der Klasse `BufferedReader` kopieren müssen.
- In unserem Beispiel ist `BufferedReader` die Superklasse von `BetterBR`.
- Umgekehrt ist `BetterBR` Subklasse von `BufferedReader`.

Realisierung eines erweiterten Buffered Readers `BetterBR`

Unser erweiterter Buffered Reader `BetterBR` soll in folgender Hinsicht eine Erweiterung der `BufferedReader`-Klasse darstellen:

- Wir möchten im Konstruktor einfach einen Dateinamen angeben können.
  `BetterBR(String fileName) {...}`
- Geben wir kein Argument an, soll `System.in` verwendet werden.
  `BetterBR() {...}`
- Die Klasse `BetterBR` soll eine Methode `readInt` zur Verfügung stellen, die einen `int`-Wert zurückliefert:
  `int readInt() {...}`

Implementierung der Methoden: Die Konstruktoren

- Da ein `BetterBR`-Objekt auch ein `BufferedReader`-Objekt beinhaltet, müssen wir dafür sorgen, dass das `BufferedReader`-Objekt korrekt initialisiert wird.
- Normalerweise wird der Konstruktor einer Klasse automatisch aufgerufen, wenn wir ein neues Objekt dieser Klasse mit dem `new`-Operator erzeugen.
- In unserem Fall erzeugt der Benutzer unserer Klasse aber ein `BetterBR`-Objekt und kein `BufferedReader`-Objekt:

```
BetterBR bbr = new BetterBR();
```

- Daher muss der Konstruktor von `BetterBR` dafür sorgen, dass der Konstruktor der `BufferedReader`-Klasse aufgerufen wird:

```
public BetterBR(){
    super(new InputStreamReader(System.in));
}
```

Das Schlüsselwort super

- Der Konstruktor von `BetterBR` muss offensichtlich den Konstruktor von `BufferedReader` aufrufen.
- Generell muss der Konstruktor einer abgeleiteten Klasse immer den Konstruktor seiner Super-Klasse mit allen erforderlichen Argumenten aufrufen.
- Dies geschieht nicht auf die übliche Weise, d.h. mit

  ```
  BufferedReader(new InputStreamReader(System.in))
  ```

  sondern unter Verwendung des Schlüsselworts **`super`**:

  ```
  super(new InputStreamReader(System.in))
  ```

- Wenn der Konstruktor der Superklasse aus dem Konstruktor einer Subklasse heraus aufgerufen wird, wird das Schlüsselwort `super` automatisch durch den Namen der Superklasse ersetzt.

Die Konstruktoren von `BetterBR` (2)

- Nach dem Aufruf des Konstruktors der Superklasse, muss die Subklasse die Initialisierungen, die für Objekte dieser Klasse erforderlich sind, vornehmen.
- In unserem Beispiel, d.h. für `BetterBR` sind jedoch keine weiteren Initialisierungen erforderlich. `BetterBR` hat keine Instanzvariablen.
- Der zweite Konstruktor von `BetterBR` soll einen Dateinamen akzeptieren:

```
public BetterBR(String fileName) throws FileNotFoundException{
       super(new InputStreamReader(
              new FileInputStream(new File(fileName))));
}
```

Die Methode `readInt` der Klasse `BetterBR`

- Offensichtlich muss die Methode `readInt` die Methode `readLine` aufrufen, um ein `String`-Objekt einzulesen.
- Anschließend kann sie die Zahl aus dem `String`-Objekt mithilfe von `Integer.parseInt` herauslesen.
- Da `readLine` eine Nachricht ist, die in der Superklasse definiert ist, verwenden wir erneut das Schlüsselwort `super`:

```
public int readInt() throws IOException{
      return Integer.parseInt(super.readLine());
}
```

Die vollständige Klasse `BetterBR`

```
import java.io.*;

class BetterBR extends BufferedReader {
    public BetterBR(){
        super(new InputStreamReader(System.in));
    }

    public BetterBR(String fileName) throws FileNotFoundException{
        super(new InputStreamReader(
            new FileInputStream(
                new File(fileName))));
    }

    public int readInt() throws IOException{
        return Integer.parseInt(super.readLine());
    }

}
```

Subklassen mit Instanzvariablen

- Üblicherweise erfordern Subklassen aber auch eigene Instanzvariablen, um zusätzliche Informationen abzulegen.
- Im Folgenden werden wir anhand einer Klasse `Name` betrachten, wie diese abgeleitet werden kann zu einer Klasse `ExtendedName`.
- Ein Problem stellen hierbei als `private` deklarierte Instanzvariablen der Superklasse dar, da sie den Zugriff auf die Instanzvariablen in der Subklasse unterbinden.
- Um innerhalb einer Subklasse auf die Instanzvariablen der Superklasse zugreifen zu können und dennoch den Zugriff von außen zu unterbinden, verwendet man daher das Schlüsselwort `protected` (anstelle von `private`).

Die Ausgangsklasse Name

```
class Name {
    public Name(String first, String last) {
        this.firstName = first;
        this.lastName = last;
        this.title = "";
    }
    public String getInitials() {
        return this.firstName.substring(0,1) + "."
            + this.lastName.substring(0,1) + ".";
    }
    public String getLastFirst() {
        return this.lastName + ", " + this.firstName;
    }
    public String getFirstLast() {
        return (this.title+" "+this.firstName+" "+ this.lastName).trim();
    }
    public void setTitle(String newTitle) {
        this.title = newTitle;
    }
    protected String firstName;
    protected String lastName;
    protected String title;
}
```

Erweiterung der Klasse `Name`

- Ziel ist die Entwicklung einer Klasse, die zusätzlich zur Klasse `Name` auch einen zweiten Vornamen (`middleName`) zur Verfügung stellt.
- Der Name dieser Klasse soll `ExtendedName` sein.
- Darüber hinaus soll die Klasse eine Methode `formalName` zur Verfügung stellen, die den kompletten Namen einschließlich `middleName` zurückgibt.
- Schließlich soll die Klasse `ExtendedName` alle Methoden der Klasse `Name` beinhalten.

Das Skelett der Klasse `ExtendedName`

Da Vorname, Name und Titel in der Klasse `Name` gespeichert werden, müssen wir in `ExtendedName` lediglich den zweiten Vornamen speichern:

```
class ExtendedName extends Name{
    public ExtendedName(String firstName, String lastName) {
        ...
    }
    public ExtendedName(String firstName,
                        String middleName, String lastName) {
        ...
    }
    public String getFormalName() {
        ...
    }
    protected String middleName;
}
```

Die Konstruktoren von `ExtendedName`

1. Der Konstruktor benötigt beide Vornamen und den Nachnamen. Im Konstruktor müssen wir den Konstruktor von `Name` aufrufen und anschließend Variable `middleName` setzen:

```
public ExtendedName(String firstName,
                    String middleName, String lastName) {
    super(firstName, lastName);
    this.middleName = middleName;
}
```

2. Der zweite Konstruktor akzeptiert lediglich den ersten Vornamen und den Nachnamen. Der zweite Vorname ist dann leer.

```
public ExtendedName(String firstName, String lastName) {
    super(firstName, lastName);
    this.middleName = "";
}
```

Die Methode `getFormalName`

- Die Methode `getFormalName` liefert den vollständigen Namen einschließlich des Titels und der Vornamen.
- Hierbei berücksichtigen wir, dass der zweite Vorname und der Titel evtl. leer sein können.

```
public String getFormalName() {
    if (this.middleName.equals(""))
        return super.getFirstLast();
    else
        return (super.title + " " + super.firstName + " "
                                   + this.middleName + " "
                                   + super.lastName).trim();
}
```

Overriding: Überschreiben von Methoden

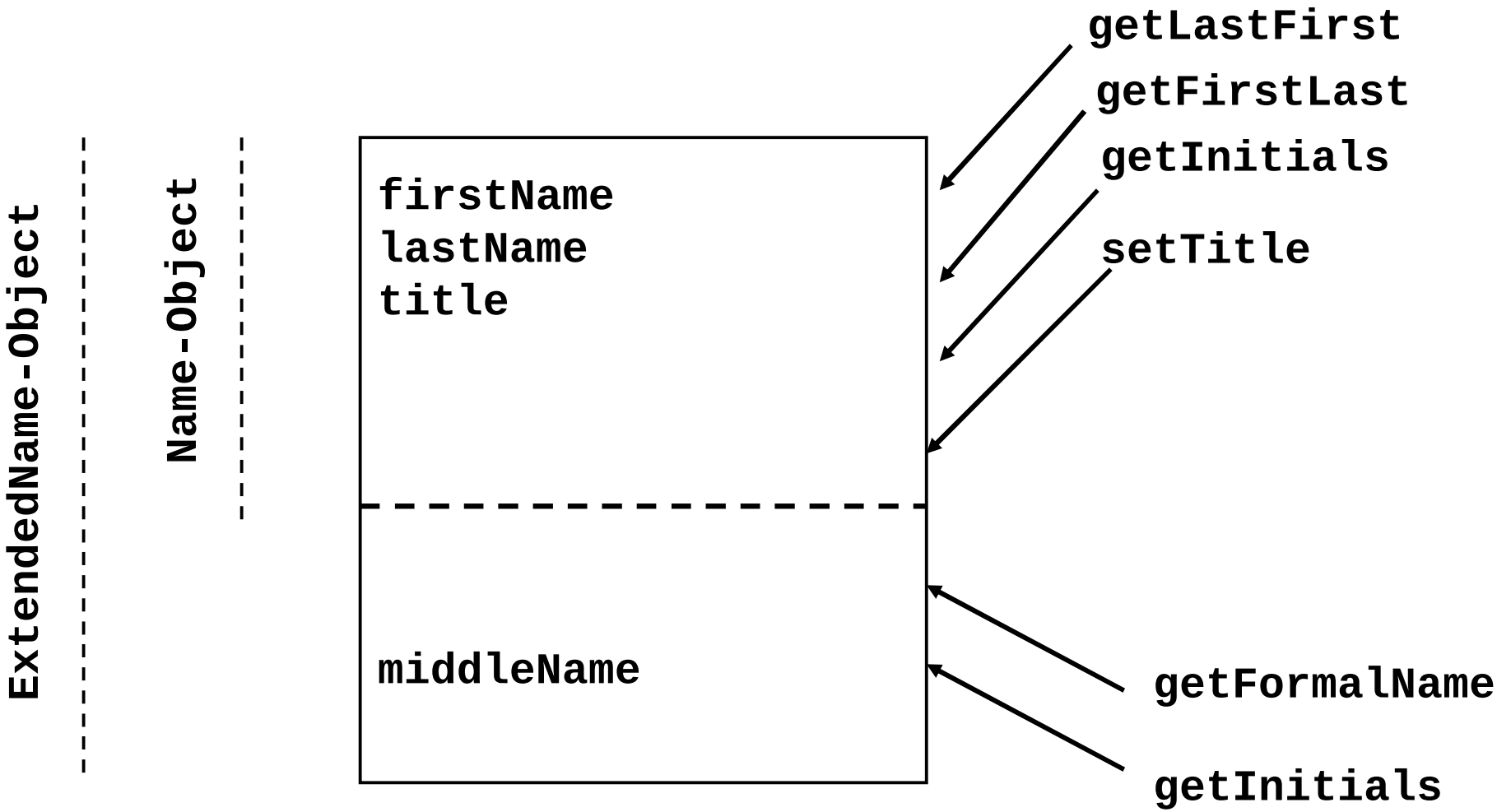
- Die Klasse `ExtendedName` übernimmt alle Methoden der Klasse `Name`.
- Da die `Name`-Klasse nur den ersten Vornamen und den Nachnamen kennt, liefert die Methode `getInitials` der Klasse `Name` das falsche Ergebnis.
- Wir müssen daher die Methode `getInitials` der Klasse `Name` in der Klasse `ExtendedName` durch eine korrekte Methode ersetzen.
- Diesen Prozess der Ersetzung bzw. Überschreibung von Methoden nennt man Overriding.

Überschreibung der Methode `getInitials`

- Die Methode `getInitials` muss in der Klasse `ExtendedName` neu definiert werden.
- Lediglich in dem Fall, dass `middleName` leer ist, können wir auf die entsprechende Methode der Superklasse zurückgreifen:

```
public String getInitials() {
    if (this.middleName.equals(""))
        return super.getInitials();
    else
        return super.firstName.substring(0,1) + "."
                    + this.middleName.substring(0,1) + "."
                    + super.lastName.substring(0,1) + ".";
}
```

Die resultierende Klasse ExtendedName



Der Programmcode von ExtendedName

```
class ExtendedName extends Name{
    public ExtendedName(String firstName, String lastName) {
        super(firstName, lastName);
        this.middleName = "";
    }
    public ExtendedName(String firstName, String middleName, String lastName) {
        super(firstName, lastName);
        this.middleName = middleName;
    }
    public String getInitials() {
        if (this.middleName.equals(""))
            return super.getInitials();
        else
            return super.firstName.substring(0,1) + "."
                                + this.middleName.substring(0,1) + "."
                                + this.lastName.substring(0,1) + ".";
    }
    public String getFormalName() {
        if (this.middleName.equals(""))
            return super.getFirstLast();
        else
            return (super.title + " " + super.firstName + " "
                                + this.middleName + " " + super.lastName).trim();
    }
    protected String middleName;
}
```

Anwendung der Klasse ExtendedName

```
class useExtendedName {
    public static void main(String [] args) {
       ExtendedName name1 = new ExtendedName("Peter", "Mueller");
       ExtendedName name2 = new ExtendedName("Peter", "Paul", "Meyer");
       System.out.println(name1.getFormalName());
       System.out.println(name1.getInitials());
       System.out.println(name2.getFormalName());
       System.out.println(name2.getInitials());
       name2.setTitle("Dr.");
       System.out.println(name2.getFormalName());
       ystem.out.println(name2.getInitials());
    }
}
```

liefert:

```
Peter Mueller
P.M.
Peter Paul Meyer
P.P.M.
Dr. Peter Paul Meyer
P.P.M.
```

Inheritance versus Komposition

- Bisher haben wir neue Klassen immer definiert, indem wir Objekte anderer Klassen verwendet haben.
- Beispielsweise haben wir zur Realisierung der Klasse `Name` Objekte der Klasse `String` verwendet. Ähnliches gilt für die Klasse `Set`, bei deren Definition wir auf Methoden der Klasse `Vector` zurückgegriffen haben.
- Dieses Verfahren, bei dem man Referenzvariablen auf Objekte anderer Klassen verwendet, bezeichnet man als Komposition.
- Im Gegensatz dazu werden bei der Ableitung von Klassen Instanzvariablen und Methoden von der Superklasse geerbt.

Eine Erweiterung von Name mittels Komposition

```
class ExtendedNameComposition {
    public ExtendedNameComposition(String firstName,
                                   String middleName,
                                   String lastName) {

        this.name = new Name(firstName, lastName);
        this.middleName = middleName;
    }

    // ...

    private String middleName;
    private Name name;
}
```

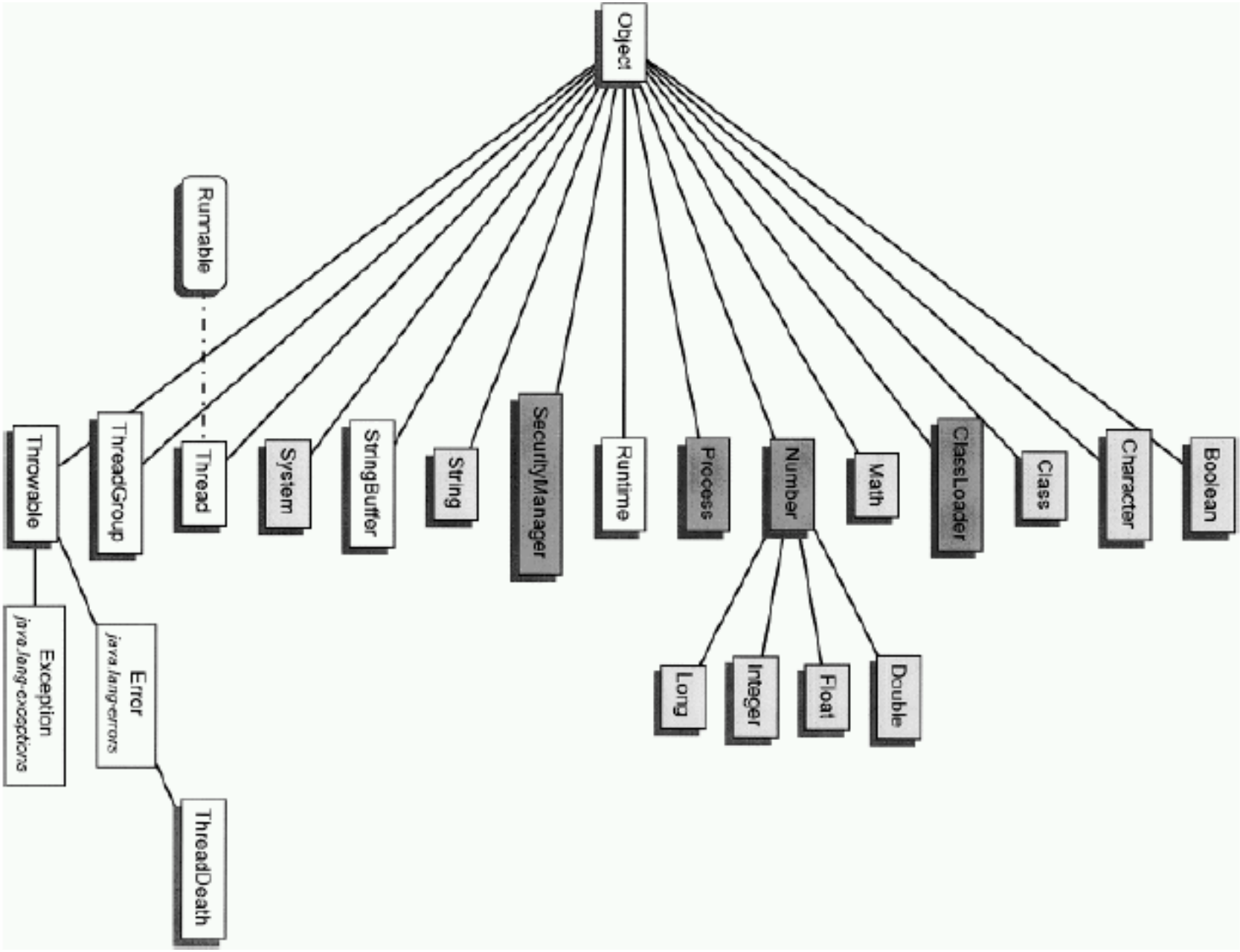
Nachteile dieses Vorgehens

- Die Klasse `ExtendedNameComposition` erbt nicht die Methoden der Klasse `Name`.
- Stattdessen muss der Programmierer für alle Methoden der Superklasse `Name`, die auch durch `ExtendedNameComposition` zur Verfügung gestellt werden sollen, entsprechenden Code programmieren.
- Innerhalb der Klasse `ExtendedNameComposition` kann man nicht auf die in `Name` als `protected` definierten Variablen zugreifen.
- Daher können die Methoden `getFormalName` nicht so, wie in `ExtendedName`, programmiert werden, da nicht auf die Instanzvariablen von `Name` zugegriffen werden kann.
- Eine Lösung besteht darin, die Instanzvariablen als `public` zu deklarieren, was allerdings ein Sicherheitsrisiko in sich birgt.

Klassenhierarchien

- In unseren Beispielen haben wir immer nur eine einfache Ableitung vorgenommen.
- Prinzipiell gibt es jedoch keinen Grund dafür, dass neue Klassen nicht auch von Subklassen abgeleitet werden können.
- Darüber hinaus ist es natürlich auch möglich, mehrere Klassen von einer Klasse abzuleiten.
- Dabei unterstützt Java lediglich Single Inheritance (einfache Vererbung), d.h. jede Klasse kann lediglich eine Superklasse haben.
- Dies führt zu stammbaumähnlichen Strukturen, die man als Klassenhierarchie bezeichnet.

Ein Ausschnitt aus der Klassenhierarchie von Java



Polymorphismus (1)

Das Prinzip des **Overriding**:

- Wir haben gesehen, dass eine abgeleitete Klasse eine Methode ihrer Superklasse **überschreiben** kann.

- Im Beispiel

```
Name n = new Name("Shlomo", "Weiss");
ExtendedName en = new ExtendedName("William", "Tecumseh",
                                   "Sherman");
String initals = n.getInitials();
initials = en.getInitials();
```

wird beim ersten Aufruf von `getInitials` die Methode der Klasse `Name` und beim zweiten Aufruf die Methode der Klasse `ExtendedName` aufgerufen.

Polymorphismus (2)

- Ist hingegen die Methode in der abgeleiteten Klasse nicht definiert, wie z.B. bei der Nachricht

  ```
  String s = en.getLastFirst();
  ```

  dann sucht der Java-Interpreter zunächst nach der Methode

  `getLastFirst` in der Klasse `extendedName`.

- Findet er sie dort, wird sie ausgeführt.
- Findet er sie nicht, wird zur Superklasse `Name` übergegangen und dort nach der Methode gesucht.
- Dieses Verfahren wird **rekursiv** fortgesetzt, bis eine Superklasse gefunden wurde, in der die Methode enthalten ist.

Polymorphismus (3): Dynamisches Binden

- Betrachten wir die folgende Deklaration:

  ```
  Name n = new ExtendedName("Thomas", "Alva", "Edison");
  ```

- Dieses Deklaration ist zulässig, **weil wir immer ein Objekt einer Subklasse anstelle eines Objektes der Klasse selbst verwenden dürfen**[a].

- Wenn wir n jetzt die Nachricht

  ```
  String s = n.getInitials();
  ```

  senden, so verwendet Java die Methode der Klasse `ExtendedName` und nicht die der Klasse `Name`.

- Allgemein gilt: **Die Suche nach der Methode beginnt immer in der Klasse des Objektes und nicht in der Klasse der Referenzvariable**.

[a] Wir haben dies bereits bei der Definition der Klasse Set (`Object`) ausgenutzt

Dynamic Binding am Beispiel der Methode `toString`

- Ein typisches Beispiel für **Polymorphismus** ist die Methode `toString`.
- Diese Methode ist in der Klasse **Object** definiert und soll von abgeleiteten Klassen überschrieben werden.
- Sinn der Methode `toString` ist die **Erzeugung einer `String`-Repräsentation** des Objektes:

```
Enumeration e = v.elements();
while (e.hasMoreElements())
System.out.println((e.nextElement()).toString());
```

- **Während der Ausführung sucht der Java-Interpreter dann mit dem oben beschriebenen rekursiven Verfahren nach der `toString`-Methode**.

Zusammenfassung gemeinsamer Eigenschaften durch Vererbung

- Betrachten Sie die Situation, dass Sie eine Inventarverwaltung für ein Fotogeschäft realisieren sollen.
- Durch dieses System sollen verschiedene Objekte wie Objektive (`Lens`), Filme (`Film`) und Kameras (`Camera`) etc. verwaltet werden.
- Wenngleich alle Objekte verschiedene Eigenschaften haben, gibt es auch bestimmte Attribute, die bei allen Objekten abgelegt werden. Hierzu sollen in unserem Beispiel gehören:
  - Bezeichnung (`description`),
  - Identifikationsnummer (`id`),
  - Anzahl (`quantity`) und
  - Preis (`price`).

Das Skelett der Klasse Lens

```
class Lens {
   public Lens(...) {...}      // Konstructor
   public String getDescription() {
        return this.description;
   }
   public int getQuantity() {
        return this.quantity;
   }
   ...                         // weitere für Lens spezifische Methoden
   private String description; // allgemeine Instanzvariablen
   private int id;
   private int quantity;
   private int price;
   private boolean isZoom;     // Instanzvariablen der Klasse Lens
   private double focalLength;
}
```

Das Skelett der Klasse `Film`

```
class Film {
    public Film(...) {...}     // Konstructor
    public String getDescription() {
        return this.description;
    }
    public int getQuantity() {
        return this.quantity;
    }
    ...                        // weitere für Film spezifische Methoden
    private String description;// allgemeine Instanzvariablen
    private int id;
    private int quantity;
    private int price;
    private int speed;         // Instanzvariablen der Klasse Film
    private int numberOfExposures;
}
```

Analog für `Camera . . .`

Zusammenfassung gemeinsamer Eigenschaften in einer Superklasse

- Offensichtlich **unterscheiden sich die Klassen** `Film`, `Lens` und `Camera` **in bestimmten Instanzvariablen und Methoden**.
- Allerdings haben auch **alle drei Klassen einige Verhalten gemeinsam**.
- Hier können wir jetzt Vererbung verwenden, und die **gemeinsamen Eigenschaften in einer Superklasse `InventoryItem` zusammenfassen**.

Die Klasse `InventoryItem`

```
class InventoryItem {
    public InventoryItem(...) {...} // Konstruktor
    public String getDescription() {
        return this.description;
    }
    public int getQuantity() {
        return this.quantity;
    }
    ...                                 // weitere allgemeine Methoden von
                                        // InventoryItem
    protected String description;   // Instanzvariablen
    protected int id;
    protected int quantity;
    protected int price;
}
```

Ableitung der Klassen `Lens` und `Film`

- Jetzt, da wir alle gemeinsamen Eigenschaften in der Klasse `InventoryItem` zusammengefasst haben, können wir die eigentlich benötigten Klassen daraus ableiten.
- Dabei können wir uns in der Definition dann auf die spezifischen Eigenschaften der Subklassen konzentrieren:

```
class Film extends InventoryItem {
    public Film(...) {...} // Konstructor
    ...                    // Für Film spezifische Methoden
    private int speed;     // Instanzvariablen der Klasse Film
    private int numberOfExposures;
}
```

Die abgeleitete Klasse Lens

```
class Lens extends InventoryItem {
    public Lens(...) {...}        // Konstructor
    ...                           // Für Lens spezifische Methoden
    private boolean isZoom;       // Instanzvariablen der Klasse Lens
    private double focalLength;
}
```

Ausnutzung des Polymorphismus

Wenn wir jetzt **Kollektionen** von InventoryItem-Objekten verwalten, können wir den **Polymorphismus** ausnutzen:

```
InventoryItem[] inv = newInventoryItem[3];

inv[0] = new Lens(...);
inv[1] = new Film(...);
inv[2] = new Camera(...);

for (int i = 0; i < inv.length; i++)
    System.out.println(inv[i].getDescription() + ": "
                                        + inv[i].getQuantity());
```

Abstrakte Methoden und Klassen (Motivation)

- Die Klasse `InventoryItem` haben wir lediglich dafür benutzt, um gemeinsame Eigenschaften ihrer Subklassen zusammenzufassen.
- Darüber hinaus haben wir in der Schleife

```
for (int i = 0; i < inv.length; i++)
    System.out.println(inv[i].getDescription() + ": "
                                + inv[i].getQuantity());
```

  nur auf die Methoden der Klasse `InventoryItem` zugegriffen.
- Was aber können wir tun, wenn wir in der Schleife eine `print`-Methode verwenden wollen, die detailliertere Ausgaben macht (und z.B. auch die Empfindlichkeit eines Films ausgibt)?
- Hier taucht das Problem auf, dass die Nachrichten an dieser Stelle an die `InventoryItem`-Objekte `inv[i]` geschickt werden, so dass diese Klasse auch eine (eigentlich überflüssige) `print`-Methode enthalten muss (die immer von den Subklassen überschrieben wird).

Das Schlüsselwort `abstract`

- Java bietet eine einfache Möglichkeit die Definition dieses Problem zu vermeiden.
- Durch das **Voranstellen des Schlüsselworts `abstract` an die Klassendefinition und den Prototypen der Methode** wie in

```
abstract class InventoryItem{
    ...
    abstract void print();
    ...
}
```

spezifizieren wir, dass es sich **bei der Methode `print` und damit auch bei der Klasse `InventoryItem`** um eine **abstrakte Methode bzw. Klasse handelt**.

- **Für abstrakte Methoden müssen die Subklassen entsprechenden Code enthalten.**

Verwendung abstrakter Klassen

- **Es können keine Objekte erzeugt werden können, die Instanzen abstrakter Klassen sind.** Folgendes Statement führt daher zu einem **Fehler**:

```
InventoryItem inv = new InventoryItem(...)
```

- Allerdings können wir **Referenzvariablen für abstrakte Klassen deklarieren und diesen eine Referenz auf Objekte von Subklassen** zuweisen:

```
InventoryItem inv = new Lens(...)
```

- Darüber hinaus können wir **Objekten von Subklassen abstrakter Klassen eine abstrakte Nachricht senden**:

```
InventoryItem inv[];
...
for (i = 0; i < inv.length; i++)
    inv[i].print();
```

Interfaces: Gleiches Verhalten verschiedener Klassen

- Die **Vererbung** stellt bereits ein mächtiges Konzept für die Beschreibung gleichen Verhaltens dar.
- In dem oben angegebenen Inventarisierungsbeispiel macht die gemeinsame Superklasse auch deshalb Sinn, weil zwischen dem `InventoryItem` und seinen Subklassen eine **Is-a-Beziehung** besteht.
- Was aber können wir tun, wenn wir generische Methoden spezifizieren wollen, die wir auf beliebigen Objekten ausführen möchten.
- Ein Beispiel sind **Enumerations**, die jeweils das nächste Objekt (unabhängig von seiner Klasse) aus einer Kollektion liefern.
- Dieses Problem taucht auch beim Sortieren auf: Wie können wir Vektoren beliebiger Objekte sortieren kann?
- Um zu vermeiden, dass **nicht zusammenhängende Objekte** zu diesem Zweck unter einer (abstrakten) Superklasse zusammengefasst werden müssen, stellt Java so genannte **Interfaces** zur Verfügung.

Interfaces und ihre Anwendung

- Interfaces dienen dazu, ein Verhalten zu spezifizieren, das von Klassen implementiert werden soll:

```
interface Enumeration {
    Object nextElement();
    boolean hasMoreElements();
}
```

- Durch diese Definition wird festgelegt, welche Methoden eine Klasse, die ein solches Interface besitzt, realisieren muss.
- In der Klassendefinition kennzeichnet man die Interfaces mit dem **Schlüsselwort `implements`**:

```
class VectorEnumeration implements Enumeration {
    ...
    Object nextElement() {...};
    boolean hasMoreElements() {...};
}
```

Interfaces und die Klassenhierarchie

- Interfaces stellen keine Klassen dar.
- Sie erben nichts und sie vererben nichts.
- Klassen können Interfaces auch nicht erweitern.
- **Klassen können Interfaces lediglich implementieren**.
- Daher können Klassen gleichzeitig Interfaces realisieren und Subklassen sein:

```
class ACSStudent extends Student implements Comparable {...}
```

- Wenn eine Klasse ein Interface implementiert, kann sie behandelt werden, als wäre sie eine Subklasse des Interfaces.

Beispiel: Das Interface `Comparable`

„**Vergleichbare**“ Objekte sollen die Methoden `lessThan` und `equal` zur Verfügung stellen:

```
interface Comparable{
    boolean lessThan(Comparable comp);
    boolean equal(Comparable comp);
}
```

Verwendung von `Comparable`: Die Klasse `Student`

```
class Student implements Comparable {
    public Student(String firstName, String name, int id) {
        this.name = name;
        this.firstName = firstName;
        this.id = id;
    }
    public boolean lessThan(Comparable comp){
        Student s1 = (Student) comp;
        int val;
        if ((val = this.name.compareTo(s1.name)) == 0)
            return this.firstName.compareTo(s1.firstName) < 0;
        else
            return val < 0;
    }
    public boolean equal(Comparable comp){
        return this.id == ((Student) comp).id;
    }
    public String toString(){
        return firstName + " " + name;
    }
    private String name;
    private String firstName;
    private int id;
}
```

Eine weitere Klasse, die `Comparable` implementiert

```
class SortableInteger implements Comparable{
    SortableInteger(int val){
        this.val = val;
    }
    public int intValue(){
        return this.val;
    }
    public boolean lessThan(Comparable comp){
        return this.val < ((SortableInteger) comp).val;
    }
    public boolean equal(Comparable comp){
        return ((SortableInteger) comp).val == this.val;
    }
    public String toString(){
        return new String(""+this.val);
    }
    private int val;
}
```

Eine generische Methode `getSmallest`

```
static int getSmallest(Vector v, int k) {
    if (v==null || v.size()<=k)
        return -1;
    int i = k+1;
    int small = k;
    Comparable smallest = (Comparable) v.elementAt(small);
    while (i != v.size()) {
        Comparable current = (Comparable) v.elementAt(i);
        if (current.lessThan(smallest)){
            small = i;
            smallest = (Comparable) v.elementAt(i);
        }
        i++;
    }
    return small;
}
```

Anwendung der Generischen Sortiermethode

Wenn wir jetzt die `getSmallest`-Methode in unserer Sortiermethode verwenden, können wir beliebige Objekte sortieren, sofern sie das `Comparable`-Interface implementieren.

```
Vector v1 = new Vector();
Vector v2 = new Vector();
Vector v3 = new Vector();
v1.addElement(new Student("Albert", "Ludwig", 0));
v1.addElement(new Student("Dirk", "Becker", 10101));
v1.addElement(new Student("Dieter", "Becker", 10102));
v2.addElement(new SortableInteger(1));
v2.addElement(new SortableInteger(3));
v2.addElement(new SortableInteger(2));
sort(v1);
sort(v2);
System.out.println(v1.toString());
System.out.println(v2.toString());
```

Liefert:

```
[Dieter Becker, Dirk Becker, Albert Ludwig]
[1, 2, 3]
```

Zusammenfassung (1)

- **Vererbung** ist ein Mechanismus um **neue Klassen aus bestehenden Klassen abzuleiten**.
- Vererbung stellt eine Art **is-a-Beziehung** zwischen der Sub- und der Superklasse her.
- Dabei **erben** die neuen **Subklassen alle Instanzvariablen und Methoden von der Superklasse**
- Darüber hinaus können **Methoden** in einer Subklasse **überschrieben** werden (**Overriding**).
- Vererbung kann aber auch dazu genutzt werden um **gemeinsames Verhalten von Klassen in einer gemeinsamen Superklasse zu implementieren**.

Zusammenfassung (2)

- Ist eine Referenzvariable für ein Superklassenobjekt deklariert aber an ein Subklassenobjekt gebunden und wird dieser Variablen ein Nachricht geschickt, so wird zunächst beidem Subklassenobjekt nach dieser Methode gesucht. Dies nennt man **Polymorphismus**.
- **Abstrakte Methoden und Klassen** wiederum befreien den Programmierer davon, Methoden in der Superklasse zu definieren, die nur in Subklassen benötigt werden.
- Sind alle Methoden in einer Klasse abstrakt, verwendet man üblicherweise **Interfaces**.
- Während in Java jede Klasse nur eine Superklasse haben kann (**Single Inheritance**), **kann eine Klasse mehrere Interfaces implementieren**.
- Mithilfe von **Interfaces** können generische Methoden realisiert werden, die auf einer Vielzahl verschiedener Objekte operieren (Enumerations, Sortieren, . . .).

Und es gibt auch ein paar Grenzen . . .

- Allerdings sind diese Mechanismen in Java beschränkt:
  - Keine Mehrfachvererbung.
  - Eingebaute wie Klassen wie `Integer` können nicht weiter abgeleitet werden.
  - Dadurch können mit generischen Methoden keine `Integer`-Objekte sortiert werden.
  - . . .

Das Problem der Mehrfachvererbung

1. Welche Methode wird geerbt, wenn beide Superklassen die gleiche Methode implementieren?
2. Welcher Wert einer Instanzvariable wird verwendet, wenn Variablen mit gleichem Namen in beiden Superklassen definiert sind?