Universität Osnabrück

VL 04
1

Informatik B – Objektorientierte Programmierung in Java

Vorlesung 04: Objektorientierte Programmierung (Teil 4)

Inhalt

- Modifikatoren
- Zugriffsmodifikatoren
- Kapselung
- Abstrakte Klassen und Operationen
- Polymorphie

Inhalt

- Interfaces
  - Definition
  - Verwendung
  - Implementierung mehrerer Interfaces
  - Konstanten in Interfaces
  - Anwendung
  - Ausgewählte Interfaces
    - Comparable
  - Generizität

Modifikatoren

- Bisher wurden schon einige sog. Modifikatoren wie **`final`** oder **`static`** eingeführt, mit deren Hilfe die Eigenschaften von Klassen, Operationen und Variablen verändert werden.
- Nachfolgend werden einige neue Modifikatoren vorgestellt, die den Zugriff auf Klassen, Attribute und Operationen regeln.

Zugriffsmodifikatoren

- Sichtbarkeit
  - Die in der letzten Vorlesung erwähnte Tatsache, dass in einer abgeleiteten Klasse alle Eigenschaften der Oberklasse übernommen werden, ist nicht in allen Fällen korrekt.
  - Zwar besitzt eine abgeleitete Klasse immer alle Attribute und Operationen der Basisklasse, sie kann aber unter Umständen nicht darauf zugreifen, wenn deren Sichtbarkeit durch Modifikatoren eingeschränkt wurde.
  - Klassen können in sog. Pakete zusammengefasst werden, wodurch sich weitere Möglichkeiten zur Einschränkung der Sichtbarkeit ergeben.

Zugriffsmodifikatoren

- **`private`**
  - Operationen oder Attribute, die als **`private`** deklariert sind, sind nur in ihrer Klasse sichtbar.
  - Sowohl für Aufrufer von Objekten der Klasse als auch für abgeleitete Klassen bleiben sie unsichtbar.

Zugriffsmodifikatoren

- **`protected`**
  - Attribute und Operationen, die als **`protected`** deklariert sind, sind in der aktuellen Klasse und in abgeleiteten Klassen sichtbar.
  - Darüber hinaus sind sie für Operationen anderer Klassen innerhalb desselben Pakets sichtbar.
  - Auf Attribute und Operationen, die als **`protected`** deklariert sind, kann von jeder Klasse aus demselben Paket zugegriffen werden und zusätzlich von abgeleiteten Klassen, auch wenn diese in einem anderen Paket "leben".

Zugriffsmodifikatoren

- **`public`**
  - Attribute und Operationen vom Typ **`public`** sind im Rahmen ihrer Lebensdauer überall sichtbar.
  - Sie können daher in der eigenen Klasse und von beliebigen Operationen anderer Klassen verwendet werden.

Zugriffsmodifikatoren

- Werden Attribute oder Operationen ohne einen der drei Zugriffsmodifikatoren definiert, so sind sie innerhalb desselben Pakets überall sichtbar.
- Der Unterschied zu **`protected`** besteht darin, dass sie in Unterklassen, die in anderen Paketen definiert werden, unsichtbar bleiben.

Zugriffsmodifikatoren

- Sichtbarkeitsebenen für den Zugriff auf Elemente einer Klasse

Modifikator	Beschreibung
`public`	Elemente des Typs **`public`** sind überall sichtbar.
`protected`	Elemente des Typs **`protected`** sind in der Klasse selbst und in abgeleiteten Klassen sichtbar. Der Aufrufer eines Objekts einer Klasse kann auf **`protected`** Elemente der Klasse dagegen nur noch dann zugreifen, wenn er in demselben Paket definiert wurde.
	Elemente ohne Modifikator sind im selben Paket sichtbar.
`private`	Elemente des Typs **`private`** sind lediglich in der Klasse selbst sichtbar.

Zugriffsmodifikatoren

- **`public`**-Modifikator für Klassen
  - Nur Klassen, die als **`public`** deklariert sind, sind außerhalb ihres Pakets sichtbar.
  - In jeder Quelldatei darf nur eine Klasse mit dem Modifikator **`public`** deklariert werden.
  - Der Name der Java-Quelldatei ist der Name dieser Klasse mit dem Modifikator **`public`**.
  - Hinweis: Eine Klasse kann nicht **`protected`** sein.

Kapselung

- Kapselung ist neben der Vererbung einer der Grundpfeiler der objektorientierten Programmierung.
- Unter Kapselung versteht man das Verbergen des internen Aufbaus einer Klasse (*Information Hiding*).
- Dies kann durch die Einschränkung des Zugriffs auf die Attribute einer Klasse erreicht werden.
- Der Zugriff auf diese Attribute ist dann nur noch über Operationen möglich, die von der Klasse zur Verfügung gestellt werden.

Kapselung

- Ein Vorteil der Kapselung besteht darin, dass der interne Aufbau einer Klasse geändert werden kann, ohne dass dies Auswirkungen auf andere (sie benutzende) Klassen hat.
- Die zugänglichen Elemente einer Klasse werden auch als *Schnittstelle* der Klasse bezeichnet.
- Ein weiterer Vorteil darin liegt, dass interne Daten nicht willkürlich geändert werden können, sondern immer unter der Kontrolle der Klasse verbleiben, es also nicht zu einer fehlerhaften Manipulation kommen kann, da eine Kontrolle über die getter- und setter-Methoden erfolgt .

Kapselung

- Zugriffsoperationen
  - Für ein Attribut, das durch den Modifikator **`private`** verborgen ist, können Operationen für den lesenden und schreibenden Zugriff zur Verfügung gestellt werden.
  - Diese ermöglichen einen kontrollierten Zugriff auf das Attribut.
  - Als Konvention für die Namen dieser Operationen haben sich die englischen Vorsilben **`get`** und **`set`** gefolgt vom Namen des Attributes eingebürgert.

Kapselung

- Beispiel

```
class Gekapselt
{
   private int nummer;

   int getNummer()
   {
      return nummer;
   }

   void setNummer(int nummer)
   {
      this.nummer = nummer;
   }
}
```


Abstrakte Klassen und Operationen

- In Java ist es möglich, abstrakte Operationen zu definieren.
- Im Gegensatz zu konkreten Operationen enthalten sie nur die Deklaration des Operationskopfes, aber keine Implementierung des Operationsrumpfes.
- Syntaktisch unterscheiden sich abstrakte Operationen dadurch, dass anstelle der geschweiften Klammern mit den auszuführenden Anweisungen lediglich ein Semikolon steht.

Abstrakte Klassen und Operationen

- Zusätzlich wird die Definition mit dem Schlüsselwort **`abstract`** versehen.
- Abstrakte Operationen definieren nur eine Schnittstelle für eine Operation, die durch Überlagerung in einer abgeleiteten Klasse implementiert werden kann.
- Eine Klasse, die mindestens eine abstrakte Operation enthält, wird selbst als abstrakt angesehen und muss ebenfalls mit dem Schlüsselwort **`abstract`** versehen werden.

Abstrakte Klassen und Operationen

- Abstrakte Klassen können nicht instantiiert werden, da sie Operationen enthalten, die nicht implementiert wurden.
- Statt dessen werden abstrakte Klassen abgeleitet, und in der abgeleiteten Klasse werden eine oder mehrere der abstrakten Operationen implementiert.
- Eine abstrakte Klasse wird konkret, wenn alle ihre Operationen implementiert sind.
- Die Konkretisierung kann dabei auch schrittweise über mehrere Vererbungsstufen erfolgen.

Abstrakte Klassen und Operationen

- Beispiel:
  - Zum Aufbau einer Mitarbeiterdatenbank soll zunächst eine Basisklasse definiert werden, die jene Eigenschaften implementiert, die für alle Mitarbeiter zutreffen, wie beispielsweise persönliche Daten oder der Eintrittstermin in das Unternehmen.
  - Gleichzeitig soll diese Klasse als Basis für spezialisierte Unterklassen verwendet werden, um die Besonderheiten spezieller Mitarbeitertypen, wie Arbeiter, Angestellte oder Manager, abzubilden.

Abstrakte Klassen und Operationen

- Da die Berechnung des monatlichen Gehalts zwar für jeden Mitarbeiter erforderlich, in ihrer konkreten Realisierung aber abhängig vom Typ des Mitarbeiters ist, soll eine abstrakte Operation **`monatsBrutto`** in der Basisklasse definiert werden, die in den abgeleiteten Klassen konkretisiert wird.
- Das folgende Beispiel zeigt die Implementierung der Klassen **`Mitarbeiter`**, **`Arbeiter`**, **`Angestellter`** und **`Manager`** zur Realisierung der verschiedenen Mitarbeitertypen.
- Zusätzlich wird die Klasse **`Mitarbeiterverwaltung`** definiert, um das Hauptprogramm zur Verfügung zu stellen, in dem die Gehaltsberechnung durchgeführt wird.

Abstrakte Klassen und Operationen

- Dazu wird ein Feld **`ma`** mit konkreten Untertypen der Klasse **`Mitarbeiter`** gefüllt (hier nur angedeutet) und dann für alle Elemente das Monatsgehalt durch Aufruf von **`monatsBrutto`** ermittelt.

Abstrakte Klassen und Operationen

```
abstract class Mitarbeiter
{
   String name;
   int persnr;

   public Mitarbeiter()
   {
   }

   public abstract double monatsBrutto();
}
```


Abstrakte Klassen und Operationen

```
class Arbeiter
extends Mitarbeiter
{
   double stundenlohn;
   double anzahlstunden;
   double ueberstundenzuschlag;
   double anzahlueberstunden;
   double schichtzulage;

   public double monatsBrutto()
   {  return stundenlohn*anzahlstunden+
             ueberstundenzuschlag*anzahlueberstunden+
             schichtzulage;
   }
}
```


Abstrakte Klassen und Operationen

```
class Angestellter
extends Mitarbeiter
{
   double grundgehalt;
   double ortszuschlag;
   double zulage;

   public double monatsBrutto()
   {
      return grundgehalt+
             ortszuschlag+
             zulage;
   }
}
```

Abstrakte Klassen und Operationen

```
class Manager
extends Mitarbeiter
{
   double fixgehalt;
   double provision1;
   double provision2;
   double umsatz1;
   double umsatz2;

   public double monatsBrutto()
   {  return fixgehalt+
             umsatz1*provision1/100+
             umsatz2*provision2/100;
   }
}
```


Abstrakte Klassen und Operationen

```
public class Mitarbeiterverwaltung
{
   static final int ANZ_MA = 5;
   static Mitarbeiter ma[];
   static double bruttosumme;

   public static void main(String[] args)
   {
      ma = new Mitarbeiter[ANZ_MA];

      //Mitarbeiter-Feld füllen, z.B.
      ma[0] = new Manager();
      ma[1] = new Arbeiter();
      ma[2] = new Angestellter();
      ma[3] = new Manager();
      //...
```


Abstrakte Klassen und Operationen

```
      //Bruttosumme berechnen
      bruttosumme = 0.0;
      for (int i=0; i<ma.length; ++i) {
         bruttosumme += ma[i].monatsBrutto();
      }
      System.out.println("Bruttosumme="+bruttosumme);
   }
} // siehe Beispiel Mitarbeiterverwaltung.java
```


Abstrakte Klassen und Operationen

- Unabhängig davon, ob in einem Feldelement ein Arbeiter, Angestellter oder Manager gespeichert wird, führt der Aufruf der Operation **`monatsBrutto`** dank der dynamischen Operationensuche die zum Typ des konkreten Objekts passende Berechnung aus.
- Auch weitere Verfeinerungen der Klassenhierarchie durch Ableiten neuer Klassen erfordern keine Veränderung der Berechnung der monatlichen Bruttosumme.

Abstrakte Klassen und Operationen

- So könnte beispielsweise eine neue Klasse **`GFManager`** (ein Manager, der Mitglied der Geschäftsführung ist) aus **`Manager`** abgeleitet und problemlos in die Gehaltsberechnung integriert werden:

Abstrakte Klassen und Operationen

```
class GFManager
extends Manager
{
    double gfzulage;

    public double monatsBrutto()
    {
        return super.monatsBrutto() + gfzulage;
    }
}
```


Polymorphie

- Polymorphie bedeutet Vielgestaltigkeit.
- Polymorphie ist ein weiterer Grundpfeiler der objektorientierten Programmierung.
- Polymorphie von Operationen bedeutet, dass ein und derselbe Operationsaufruf in verschiedenen Klassen durch eine jeweils klassenspezifische Anweisungsfolge abgearbeitet wird.

Polymorphie

- Eine Polymorphie von Objekten gibt es nur bei Vererbungshierarchien.
- An die Stelle eines Objektes in einem Programm kann stets auch ein Objekt einer abgeleiteten Klasse treten.
- Der Grund dafür ist, dass ein Objekt einer abgeleiteten Klasse polymorph ist.
- Es kann sich als Objekt der abgeleiteten Klasse, aber auch als ein Objekt irgendeiner Oberklasse verhalten.

Inhalt

- Interfaces
  - Definition
  - Verwendung
  - Implementierung mehrerer Interfaces
  - Konstanten in Interfaces
  - Anwendung
  - Ausgewählte Interfaces
    - Comparable
  - Generizität

Definition eines Interfaces

- In Java gibt es keine Mehrfachvererbung von Klassen:
  - Die möglichen Schwierigkeiten im Umgang mit mehrfacher Vererbung und die Einsicht, dass das Erben von nichttrivialen Operationen aus mehr als einer Klasse in der Praxis selten zu erreichen ist, haben die Designer dazu veranlasst dieses Feature nicht zu realisieren.
  - Andererseits sah man es sehr wohl als wünschenswert an, Operationsdeklarationen von mehr als einer Klasse zu erben und hat mit den Interfaces ein Ersatzkonstrukt geschaffen, das genau dieses Feature bietet.

Definition eines Interfaces

- Ein Interface enthält ausschließlich abstrakte Operationen und Konstanten.
- Anstelle von **`class`** wird zur Definition eines Interfaces das Schlüsselwort **`interface`** verwendet.
- Ein Interface ist implizit **`public`**.
- Alle Operationen sind standardmäßig **`abstract`** und **`public`**.

Definition eines Interfaces

- Das folgende Listing definiert ein Interface **`Fortbewegungsmittel`**, das die Operationen **`kapazitaet`** und **`kilometerPreis`** definiert:

```
interface Fortbewegungsmittel
{
   public int kapazitaet();
   public double kilometerPreis();
}
```

Verwendung von Interfaces

- Was bei der Vererbung von Klassen als Ableitung bezeichnet wird, nennt man bei Interfaces Implementierung.
- Durch das Implementieren eines Interfaces verpflichtet sich die Klasse, alle Operationen, die im Interface definiert sind, zu implementieren.
- Fehlt eine Operation, so gibt es einen Compilerfehler, sofern die Klasse nicht **`abstract`** ist.
- Die Implementierung eines Interfaces wird durch das Schlüsselwort **`implements`** bei der Klassendefinition angezeigt.

Verwendung von Interfaces

- Als Beispiel wird die Klasse **`Auto`** um das neue Interface **`Fortbewegungsmittel`** erweitert und die Operationen **`kapazitaet`** und **`kilometerPreis`** implementiert:

Verwendung von Interfaces

```
class Auto
implements Fortbewegungsmittel
{
              String       name;
              int          erstzulassung;
              int          leistung;
    private   int          anzahlSitze;
    private   double       spritVerbrauch;
    private   double       spritPreis;

    public int kapazitaet()
    {  return anzahlSitze;
    }
    public double kilometerPreis()
    {  return spritVerbrauch * spritPreis / 100;
    }
}
```


Verwendung von Interfaces

- Ebenso wie die Klasse **`Auto`** könnte auch jede andere Klasse das Interface **`Fortbewegungsmittel`** implementieren und die beiden Operationen realisieren.
- Nützlich ist dies insbesondere für Klassen, die in keinem direkten Zusammenhang mit der Klasse **`Auto`** und ihrer Vererbungshierarchie stehen.
- Um ihr gewisse Eigenschaften eines **`Fortbewegungsmittels`** zu verleihen, könnte also beispielsweise auch die Klasse **`Teppich`** dieses Interface implementieren.

Verwendung von Interfaces

- Eine Klasse kann auch dann ein Interface implementieren, wenn sie bereits von einer anderen Klasse abgeleitet ist.
- In diesem Fall erbt die neue Klasse wie gewohnt alle Eigenschaften der Basisklasse und hat zusätzlich die Aufgabe, die abstrakten Operationen des Interfaces zu implementieren.
- Das folgende Interface **`Sammlerstueck`** mag bei gewöhnlichen Autos keine Anwendung finden, ist bei einem Oldtimer aber durchaus sinnvoll:

Verwendung von Interfaces

```
interface Sammlerstueck
{
  public double sammlerWert();
  public String bisherigeAusstellungen();
}

class Oldtimer
extends Auto
implements Sammlerstueck
{
  // ...
} // siehe Beispiel Schnittstellen.java
```


Verwendung von Interfaces

- Da ein **`Sammlerstueck`** aber durchaus auch in ganz anderen Vererbungshierarchien auftauchen kann als bei Autos (beispielsweise bei Briefmarken, Schmuck oder Telefonkarten), macht es keinen Sinn, diese Operationen in den Ableitungsbäumen all dieser Klassen wiederholt zu deklarieren.
- Statt dessen sollten die Klassen das Interface **`Sammlerstueck`** implementieren und so garantieren, dass die Operationen **`sammlerWert`** und **`bisherigeAusstellungen`** zur Verfügung stehen.

Implementieren mehrerer Interfaces

- Eine Klasse kann nicht nur ein einzelnes, sondern eine beliebige Anzahl an Interfaces implementieren.
- So ist es beispielsweise problemlos möglich, eine aus **`Flugzeug`** abgeleitete Klasse **`Doppeldecker`** zu definieren, die sowohl **`Sammlerstueck`** als auch **`Fortbewegungsmittel`** implementiert:

Implementieren mehrerer Interfaces

```
class Doppeldecker
extends Flugobjekt
implements Fortbewegungsmittel,Sammlerstueck
{
  // ...
}
```

- Die Klasse Doppeldecker muss dann alle in **`Sammlerobjekt`** und **`Fortbewegungsmittel`** deklarierten Operationen implementieren.

Interfaces und Klassen

- Interfaces besitzen zwei wichtige Eigenschaften, die auch Klassen haben:
  - Sie lassen sich vererben. Wird ein Interface B aus einem Interface A abgeleitet, so erbt es alle Deklarationen von A und kann diese um eigene Deklarationen erweitern.
  - Variablen können vom Typ eines Interfaces sein. In diesem Fall kann man ihnen Objekte zuweisen, die aus Klassen abstammen, die dieses oder eines der daraus abgeleiteten Interfaces implementieren.

Interfaces und Klassen

- Es macht also Sinn, ein Interface als eine Typvereinbarung anzusehen.
- Eine Klasse, die dieses Interface implementiert, ist dann vom Typ des Interfaces.
- Wegen der Mehrfachvererbung von Interfaces kann eine Variable damit zu mehr als einem Typ zuweisungskompatibel sein.

Konstanten in Interfaces

- Neben abstrakten Operationen können Interfaces auch Konstanten, also Variablen mit den Modifikatoren **`static`** und **`final`**, enthalten.
- Alle Variablen eines Interface sind implizit **`final`** und **`static`**.
- Wenn eine Klasse ein solches Interface implementiert, erbt es gleichzeitig auch all seine Konstanten. Es ist auch erlaubt, dass ein Interface ausschließlich Konstanten enthält.
- Siehe Beispiel **`Konstanten.java`**

Konstanten in Interfaces

- Dieses Feature kann zum Beispiel nützlich sein, wenn ein Programm sehr viele Konstanten definiert. Anstatt diese in ihren korrespondierenden Klassen zu belassen und mit **`Klasse.Name`** aufzurufen, könnte ein einzelnes Interface definiert werden, das alle Konstantendefinitionen vereinigt.
- Wenn nun jede Klasse, die eine der Konstanten benötigt, dieses Interface implementiert, so stehen alle darin definierten Konstanten direkt zur Verfügung und können ohne die Qualifizierung mit einem Klassennamen aufgerufen werden.

Anwendung von Interfaces

- Interfaces werden benutzt, um Eigenschaften auszudrücken, die auf Klassen aus unterschiedlichen Klassenhierarchien zutreffen können.
- Dies lässt sich z.B. an einigen Interfaces aus der Java-Klassenbibliothek erkennen, deren Namen (substantivierte) Eigenschaftswörter sind:
  - **`Cloneable`**
  - **`Comparable`**
  - **`Runnable`**

Ausgewählte Interfaces

- Comparable
  - Das Interface **`Comparable`** ist in der Java-Klassenbibliothek im Paket **`java.lang`** enthalten.
  - Es besitzt folgenden Aufbau:

```
public interface Comparable
{
 public int compareTo(Object o);
}
```

  - Dieses Interface kann von Klassen implementiert werden, deren Objekte paarweise vergleichbar sind.

Ausgewählte Interfaces

- Comparable
  - In der Klassenbibliothek gibt es eine Reihe von Klassen, die das Interface **`Comparable`** implementieren, beispielsweise **`String`** und **`Character`**.

Ausgewählte Interfaces

- Comparable
  - Die Operation **`compareTo`** liefert genau dann einen Wert kleiner 0, wenn das Objekt „kleiner“; größer 0, wenn es „größer“, und gleich 0, wenn es „gleich“ dem als Argument übergebenen Objekt **`o`** ist.
  - Mit Hilfe von **`Comparable`** kann die Reihenfolge der Objekte einer Klasse ermittelt werden.
  - Aus dem paarweisen Vergleich lässt sich eine (nicht notwendigerweise eindeutige) implizite Ordnung der Elemente ableiten, denn für alle aufeinanderfolgenden Objekte **`a`** und **`b`** muss **`a.compareTo(b) <= 0`** gelten.

Ausgewählte Interfaces

- Comparable
  - Damit ist es möglich, Operationen zu schreiben, die das kleinste oder größte Element einer Menge von Objekten ermitteln oder diese sortieren:

```
public static Object
getSmallest(Comparable[] objects)
{
 Object smallest = objects[0];
 for (int i = 1; i < objects.length; i++){
    if (objects[i].compareTo(smallest)<0) {
       smallest = objects[i];
    }
 }
 return smallest;
}
```


Ausgewählte Interfaces

- Comparable

```
public static void bubbleSort(Comparable[] objects)
{
    boolean sorted;
    do {
      sorted = true;
      for (int i = 0; i < objects.length - 1; i++)
          if(objects[i].compareTo(objects[i+1]) > 0) {
           Comparable tmp = objects[i];
           objects[i] = objects[i+1];
           objects[i+1] = tmp;
           sorted = false;
          }
      }
    } while (!sorted);
}
```


Generiziät

- Bei den Operationen **`getSmallest`** und **`bubbleSort`** spielt es keine Rolle, welche Art von Objekten bearbeitet werden, solange alle das Interface **`Comparable`** implementieren.
- Diese Eigenschaft der gefunden Lösung wird auch als Generizität (Typunabhängigkeit) bezeichnet.
- Siehe Beispiel **`Generizitaet.java`**