Abstrakte Basisklassen

Kapitel 11: Vererbung

11.1 Motivation und Begriffsdefinitionen
11.2 Vorgehensweise und Implementierung
11.3 Arten von Vererbung
11.4 Konstruktoren
11.5 Abstrakte Klasse
11.6 Überschreiben, Überladen, Verschattung
11.7 Wurzelklasse Object
11.8 Zugriffsrechte und Sichtbarkeit
11.9 Schnittstelle

Abstrakte Klassen – Wozu?

- Abstrakte Klassen sind ein wichtiges technisches Konzept der objektorientierten Softwareentwicklung
- Sie dienen dazu, die Schnittstellen künftiger Unterklassen festzulegen (wenn man z.B. noch keine Implementierung angeben kann oder will)
- Sie sind quasi ein Muster (Template Method Pattern), das vorgibt, welche Methoden in Unterklassen implementiert werden müssen
- Erst die Unterklassen wissen, wie sich die Methoden genau verhalten sollen
- Eine abstrakte Klasse wirkt wie ein Steckplatz, in den Objekte der Unterklassen eingesteckt werden
- Software, die solche Steckplätze bereitstellt = Framework

Abstrakte Klasse (1)

- Definition:
  - Klasse, die nicht instanziiert werden kann

- Zwei verschiedene Arten möglich:
  (1) Alle Operationen werden – wie bei konkreten Klassen – vollständig implementiert
  (2) Mindestens eine Operation wir nicht implementiert (abstrakte Operation)
    - Definiert nur Methodensignatur – Methodenrumpf ist leer
    - Spezifiziert lediglich die Schnittstelle
    - Abgeleitete Klassen müssen alle abstrakten Operationen der Oberklasse implementieren

Abstrakte Klasse (2)

- Notation in UML
  - Schlüsselwort `<<abstract>>`
  - Kursive Schrift
  - Bei handschriftlicher Darstellung besser mit Schlüsselwort!

<<abstract>> **Person**
drive() {abstract}

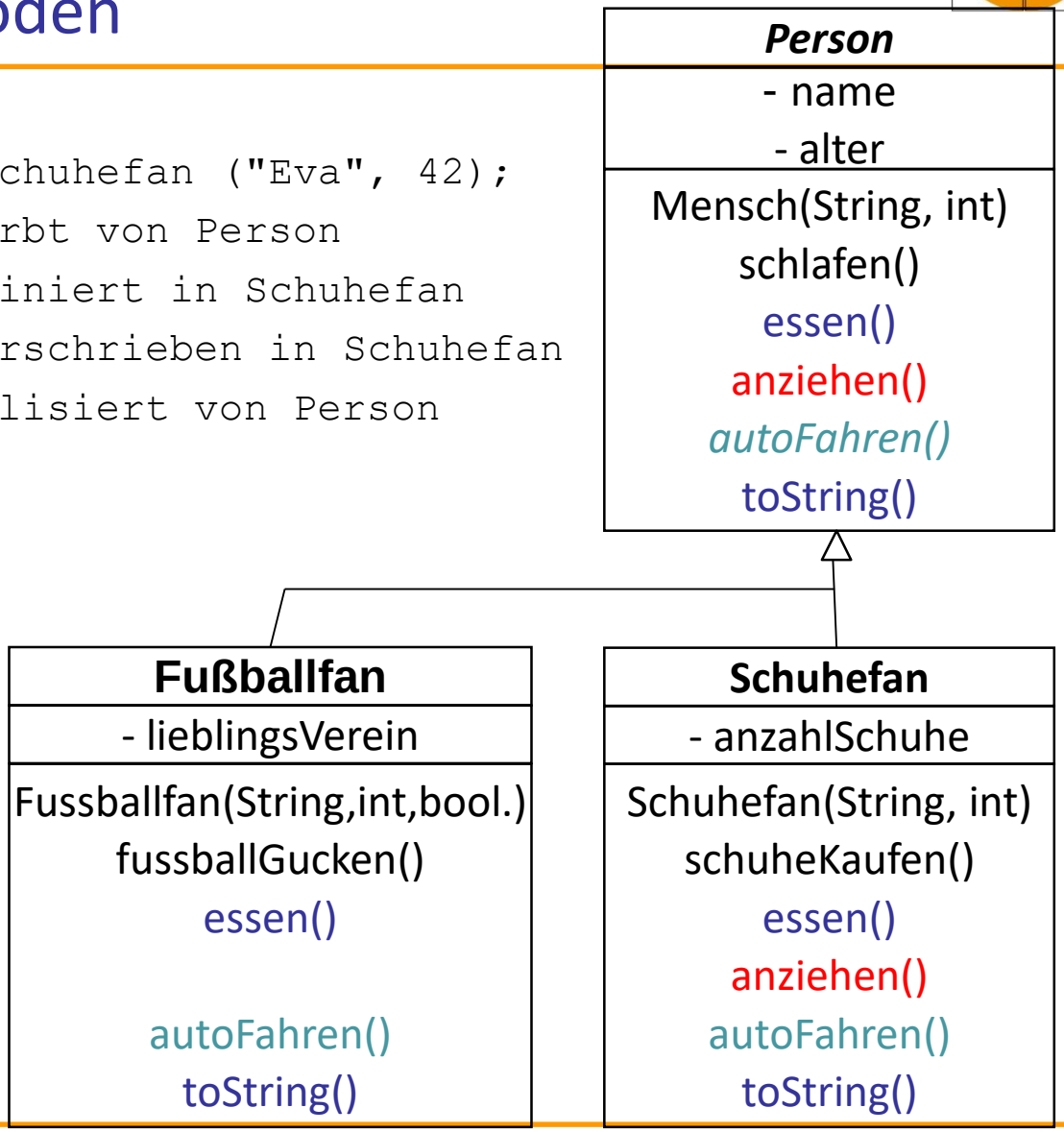
Person
drive()

Aufruf ererbter Methoden

- Beispiel:

```
Schuhefan eva = new Schuhefan ("Eva", 42);
eva.sleep();    // Geerbt von Person
eva.buyShoes();// Definiert in Schuhefan
eva.anziehen();// Überschrieben in Schuhefan
eva.drive();    // Realisiert von Person
```

- Verarbeitung:
  - Compiler stellt sicher, dass die JVM zur Laufzeit eine Implementierung findet
  - JVM sucht erst in Klasse selbst, dann in Basis-klasse, dann in deren Basisklasse, usw.



Übung – Abstrakte Klasse

- Live Übung
  - Bearbeiten Sie Aufgabe 4 des Blatts Live Übung „Vererbung“
  - Sie haben 5 Minuten Zeit.

Kapitel 11: Vererbung

11.1 Motivation und Begriffsdefinitionen
11.2 Vorgehensweise und Implementierung
11.3 Arten von Vererbung
11.4 Konstruktoren
11.5 Abstrakte Klasse
11.6 Überschreiben, Überladen, Verschattung
11.7 Wurzelklasse Object
11.8 Zugriffsrechte und Sichtbarkeit
11.9 Schnittstelle

Überladen von Methoden

- Beim Überladen einer Methode betrachten wir gleichnamige Methoden innerhalb einer einzigen Klasse
- Eine Methode wird durch drei Eigenschaften festgelegt:
  - Methodenname
  - Parameterliste
  - Rückgabewert
- Wenn wir eine Methode überladen bedeutet das, dass wir mindestens zwei Methoden vom gleichem Namen innerhalb einer Klasse haben.
- **Eine Unterscheidung über den Rückgabewert funktioniert nicht.**
- Somit verbleibt uns zur Unterscheidung lediglich die Parameterliste

Überladen von Methoden: Beispiel

```
public static void foo(int x,double y,double z){
        System.out.println("foo 1");
}

public static void foo(int x,double y){
        System.out.println("foo 2");
}

public static void foo(double x,int y){
        System.out.println("foo 3");
}
```

- Was passiert nun?
  - `> foo(6,7.0,6.7);`
  - `> foo(3.1,5);`
  - `> foo(5,3.1);`
  - `> foo(5.0,5.0);`

Überschreiben von Methoden

- Überschreiben von Methoden behandelt gleichnamige Methoden, die sich innerhalb einer Vererbungshierarchie auf verschiedene Klassen verteilen
- Genau wie beim Überladen ist auch beim Überschreiben von Methoden die Parameterliste das ausschlaggebende Kriterium für die eindeutige Zuordnung eines Aufrufs zu einer Methode.

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

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

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.

- Das Binden von überladenen Methoden ist **statisches Binden** (static binding). Das Binden von überschriebenen Methode ist **dynamisches Binden** (late binding oder dynamic binding)

Verschattung (1)

- Bereits bekannt von „normalen“ Klassen
  - Lokale Variablen bzw. Parameternamen verschatten Attribute
- Neue Art der Verschattung bei Vererbung
  - Attribut der Unterklasse verschattet Attribut der Oberklasse
  - Methode der Unterklasse verschattet Methode der Oberklasse
- Zugriff
  - Auf verschattetes Element `x` der Oberklasse: `super.x`
  - Auf verschattetes Element x der aktuellen Klasse: `this.x`

Verschattung (2)

- Oberklasse

```
public abstract class Person {
   …
   public String eat() {
   return "eat  : Mmmmh, lecker.\n";
   }
}
```

- Unterklasse

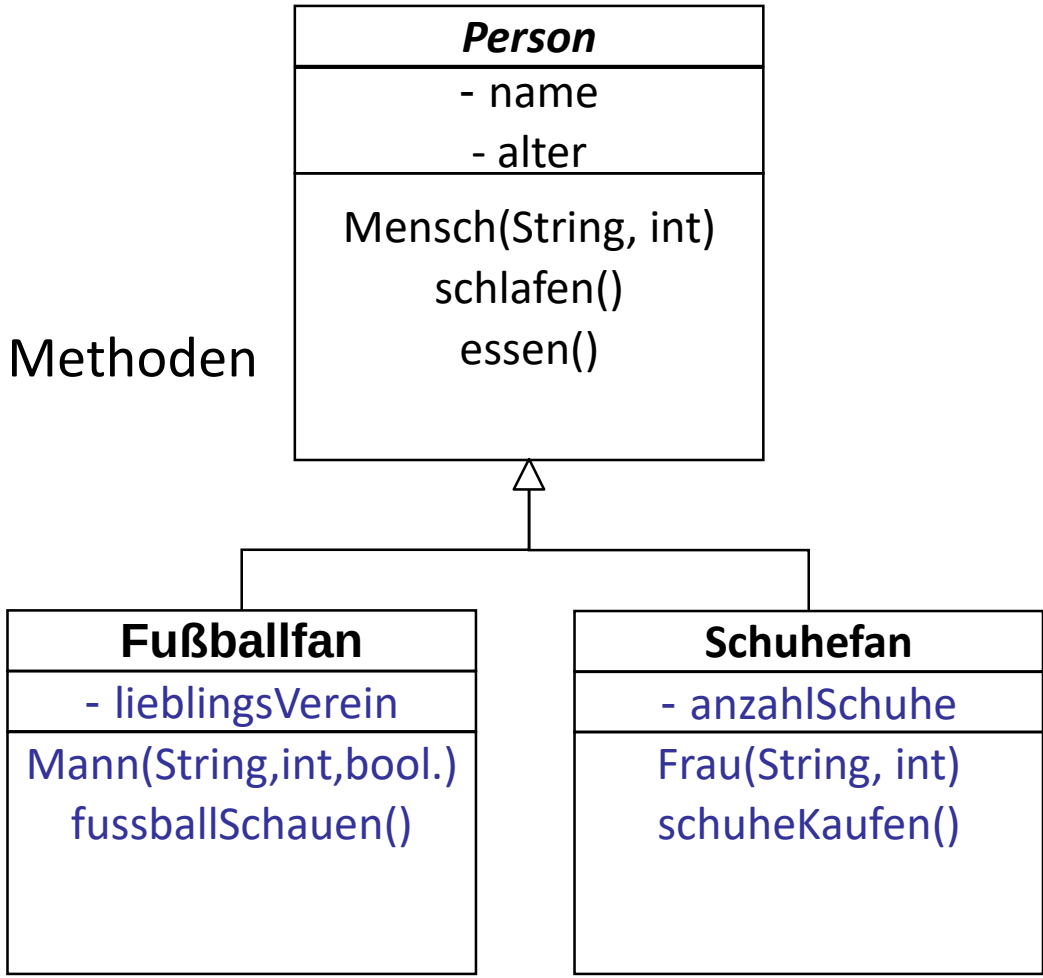
```
public class Fussballfan extends Person {
   …
   public String eat() {
      return super.eat() +
             "Kann ich einen Nachschlag haben?";
   }
}
```

Modifizieren der Unterklassen

- Erweitern
  - Etwas Neues hinzufügen
  - Unterklasse erweitert Oberklasse um weitere Attribute, Operationen und/oder Beziehungen
- Redefinieren
  - Sich ähnlich verhalten
  - In Unterklasse geerbte Methoden aus der Oberklasse bei Bedarf durch eigene spezifische Implementierung überschreiben
  - Ggf. dabei geerbte Implementierungen verwenden
- Definieren
  - Etwas Versprochenes realisieren
  - Abstrakt deklarierte Operationen der Oberklasse in Unterklasse implementieren

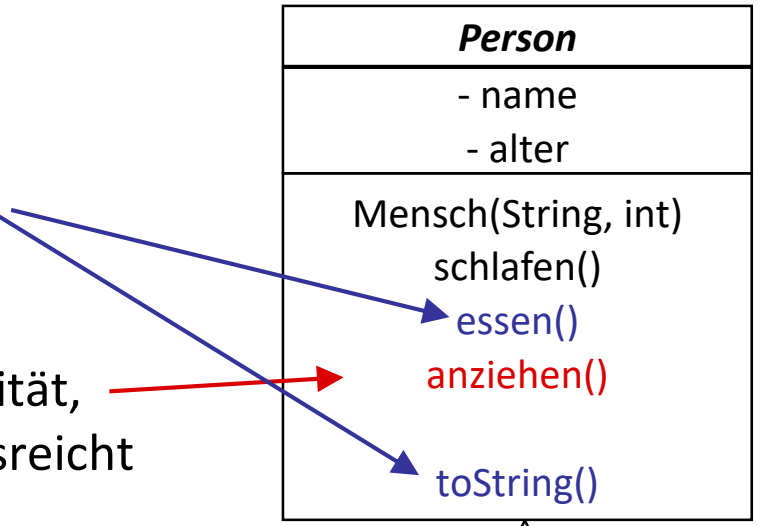
Eigenschaften in Unterklasse erweitern

- Ausgangsbasis
  - Oberklasse Person
  - Gemeinsame Attribute
  - Grundlegende gemeinsame Methoden
  - Konstruktor
- Erweiterung in Unterklassen
  - Spezifische Attribute
  - Spezifische Methoden
  - Insbesondere eigene Konstruktoren

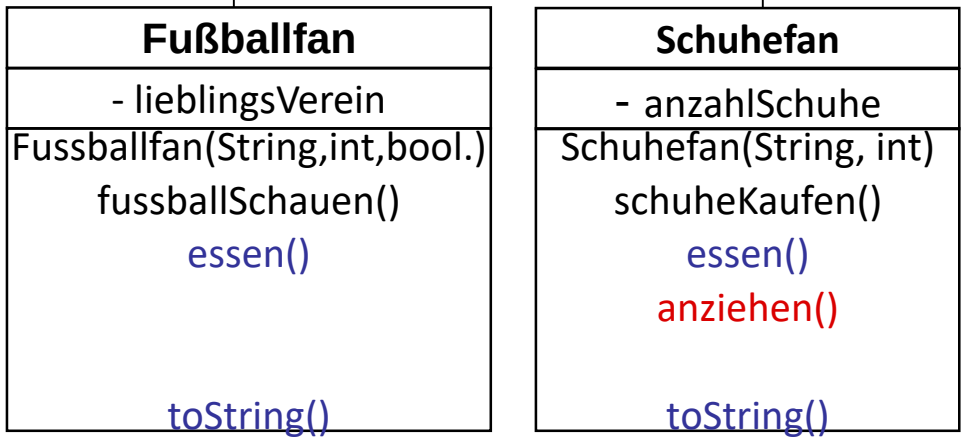


Eigenschaften in Unterklasse redefinieren

- Ausgangsbasis
  - Methode mit Basisfunktionalität, die in allen Unterklassen auftritt, aber ergänzt wird
  - Methode mit Standardfunktionalität, die für einige Unterklassen so ausreicht



- Redefinition in Unterklassen
  - Überschreiben der Methode aus der Oberklasse durch spezifische Implementierung
  - Einbinden der Implementierung aus der Oberklasse über **`super.`**`<methodenname>()`



Beispiel: Redefinieren

- Erweitern der Klasse `Person` um Methode `anziehen()`

```
public abstract class Person { …
  public String anziehen() {
    return "anziehen: Unterhose und Socken";
  } …
}
```

- Redefinieren der Klasse `Schuhefan`

```
public class Schuhefan extends Person { …
  public String anziehen() {
    return „anziehen: Unterhose, Socken und Schuhe";
  } …
}
```

Beispiel: Redefinieren, Basisfunktion nutzen (1)

- Veränderungen gegenüber obigem Beispiel
  - Klassen `Person` unverändert
  - In Klassen `Schuhefan`, `Fussballfan` jeweils Methode `eat()` überschreiben

- Redefinieren der Klasse `Schuhefan`

```
public class Schuhefan extends Person {
  …
  public String eat() {
    String result = super.eat();
    result = result + "\n       Wirklich schade,
      dass das so viele Kalorien hat...";
    return result;
  }…
}
```

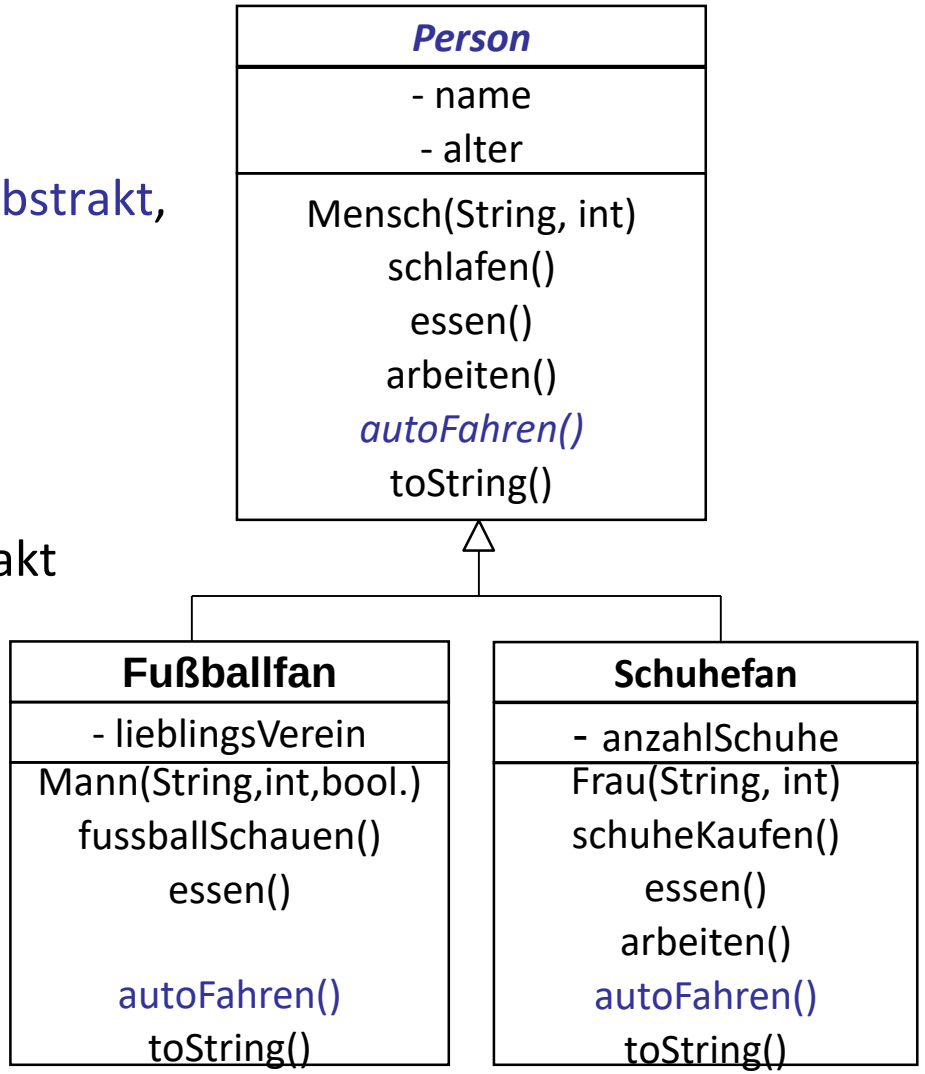
Beispiel: Redefinieren, Basisfunktion nutzen (2)

- Redefinieren der Klasse `Fussballfan`

```
public class Fussballfan extends Person {
  …
  public String eat() {
    String result = super.eat();
    result = result + "\n Kann ich noch einen
      Nachschlag haben?";
    return result;

  }
  …
}
```

Eigenschaften in Unterklasse definieren

- Ausgangsbasis
  - Operation `autoFahren()` ist abstrakt, d.h. definiert nur die Signatur
  - Sichert damit die Existenz dieser Verhaltensweise
  - Keine Implementierung!
  - Oberklasse wird damit auch abstrakt
- Definition in Unterklassen
  - Definieren zu den abstrakten Operationen spezifische Implementierungen
  - In *jeder* nicht abstrakten Unterklasse erforderlich



Beispiel: Definieren (1)

- Veränderungen gegenüber obigem Beispiel
  - Klasse `Person` definiert nur Schnittstelle der Methode `autoFahren();` wird damit zur abstrakten Klasse
  - Klasse `Schuhefan`, `Fussballfan` jeweils erweitert um Implementierung der Methode `autoFahren()`

- Neue Version der Klasse `Person`

```
public abstract class Person {
   …
   public abstract String drive();
}
```

Beispiel: Definieren (2)

- Erweiterung von `Fussballfan` um Definition von `drive()`

```
public class Fussballfan extends Person {
  …
  public String drive() {
    return "Jetz fahr schon, ich will zum Spiel!";
}…}
```

- Erweiterung von `Schuhefan` um Definition von `drive()`

```
public class Schuhefan extends Person {
  …
  public String drive() {
    return "Dann mal los zum Schuhladen!";
}…}
```

Übung – Methoden redefinieren

- Live Übung
  - Bearbeiten Sie Aufgabe 6 des Blatts Live Übung „Vererbung“
  - Sie haben 5 Minuten Zeit.

Kapitel 11: Vererbung

11.1 Motivation und Begriffsdefinitionen
11.2 Vorgehensweise und Implementierung
11.3 Arten von Vererbung
11.4 Konstruktoren
11.5 Abstrakte Klasse
11.6 Überschreiben, Überladen, Verschattung
11.7 Wurzelklasse Object
11.8 Zugriffsrechte und Sichtbarkeit
11.9 Schnittstelle

Klasse `Object`

- Klasse `Object` ist voreingestellte Basisklasse aller Klassen
- Äquivalent:

```
class ClassName
{...}
```

und

```
class ClassName extends Object
{...}
```

- Jede Klasse ist von einer anderen Klasse abgeleitet, außer `Object`
- Alle Klassen (abgesehen von `Object`) haben, direkt oder indirekt, `Object` als gemeinsame Basisklasse

Vordefinierte Methoden in `Object`

- Methoden von `Object` werden an jede Klasse vererbt
- Beispiele:

`public String toString()`	Lesbare Repräsentation
`public boolean equals(Object x)`	`true` wenn dieses Objekt und `x` gleich sind, `false` ansonsten
`public int hashCode()`	Kennnummer
`protected Object clone()`	Erzeugt ein Duplikat
`public Class getClass()`	Typobjekt dieses Objektes

- `Object`-Methoden bieten zum Teil nur minimale Funktionalität
- Methoden sollten in der Regel redefiniert werden!

Kapitel 11: Vererbung

11.1 Motivation und Begriffsdefinitionen
11.2 Vorgehensweise und Implementierung
11.3 Arten von Vererbung
11.4 Konstruktoren
11.5 Abstrakte Klasse
11.6 Verschattung
11.7 Wurzelklasse Object
11.8 Zugriffsrechte und Sichtbarkeit
11.9 Schnittstelle

Zugriffsrechte und Sichtbarkeit (1)

- Zusätzliche Zugriffsrechte und Sichtbarkeitsregeln durch Vererbungskonzept
- 4 Kategorien:
  1. `public (+)`: „weltweiter“ Zugriff sowohl von außen als auch von allen Nachfahren unabhängig von Paketzugehörigkeit
  2. `private (-)`: nur innerhalb der eigenen Klasse sichtbar; werden vererbt, sind aber von der Unterklasse aus nicht zugreifbar
  3. `protected (#)`: von allen Nachfahren darf darauf zugegriffen werden unabhängig davon, ob sich die Nachfahren im gleichen Paket oder in einem anderen Paket befinden und von allen Klassen im gleichen Paket
  4. Implizit: nur innerhalb des Pakets sichtbar, in dem die Klasse definiert ist; gilt für Nachfahren und Nicht-Nachfahren

Zugriffsrechte und Sichtbarkeit (2)

- Regeln zum Ändern der Zugriffskategorie beim Überschreiben:
  - Zugriffsrechte dürfen nur erweitert, aber nicht weiter eingeschränkt werden
  - `public`-Operationen müssen `public` bleiben
  - `private`-Operationen, die in Unterklassen neu definiert werden, dürfen eine beliebige Zugriffskategorie haben, da es sich um neue Operationen handelt
  - Operationen ohne explizite Zugriffskategorie können so bleiben oder als `protected` oder `public` überschrieben werden
  - `protected` darf als `public` überschrieben werden

Kapitel 11: Vererbung

11.1 Motivation und Begriffsdefinitionen
11.2 Vorgehensweise und Implementierung
11.3 Arten von Vererbung
11.4 Konstruktoren
11.5 Abstrakte Klasse
11.6 Verschattung
11.7 Wurzelklasse Object
11.8 Zugriffsrechte und Sichtbarkeit
11.9 Schnittstelle

Schnittstelle – Bedeutung

- Definition: Schnittstelle, Interface
  - Spezielle Form von Klasse
  - Keine Objekte direkt von Interface ableitbar
- Verhalten
  - Definiert nur abstrakte Operationen, keine Implementierungen
  - Legt also nur Anforderungen fest
  - Keine ausführbaren Anweisungen (seit Java8: `static` und `default` möglich)
  - Keine Konstruktoren
- Eigenschaften
  - Enthält keine veränderbaren Attribute
  - Öffentlich sichtbare Konstanten als Attribute möglich
- Alle Methoden / Datenelemente haben implizit Sichtbarkeit `public`!

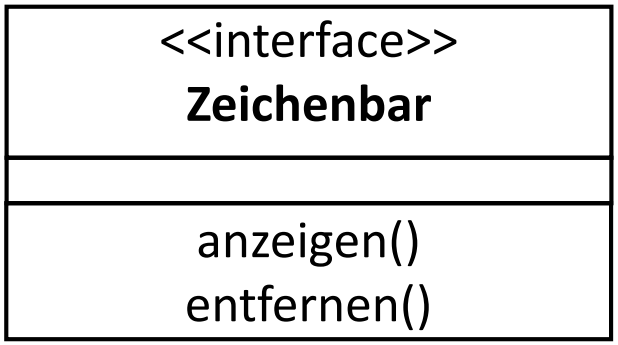
Schnittstelle – Umsetzung in Java

- Bedeutung von Java
  - Ermöglicht klare Trennung von Implementierung und Schnittstelle
  - Mehrfachvererbung von konkreten Klassen in Java nicht erlaubt
  - Implementierung von mehreren Schnittstellen ist aber möglich!!!

- Umsetzung in Java:
  - Reserviertes Wort `interface` (statt `class`)
  - Je Interface eigene `.java`-Datei, wird übersetzt zu `.class`–Datei

Schnittstelle in UML

- Schnittstelle in UML
  - Symbol analog zu Klasse
  - Stereotyp `<<interface>>` oberhalb des Klassennamens
  - Schnittstelle ist immer auch abstrakt,
    muss nicht explizit als abstrakt gekennzeichnet werden

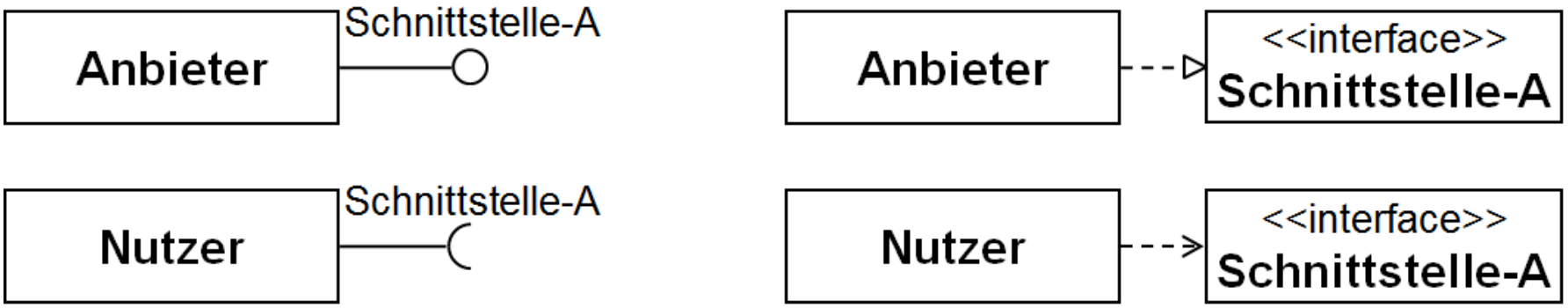


Begriffe – Anbieter und Nutzer

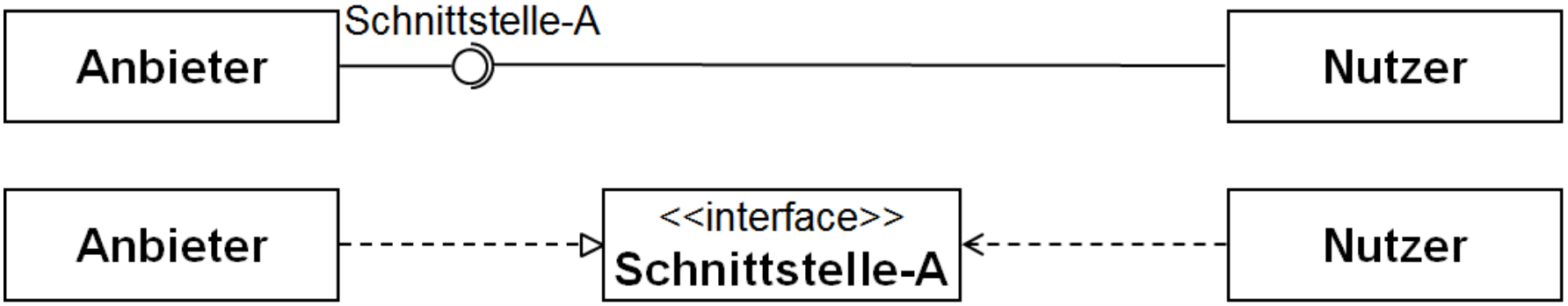
- Anbieter einer Schnittstelle
  - Realisiert die Schnittstelle, d.h. implementiert die Operationen
- Nutzer einer Schnittstelle
  - Verwendet die Schnittstelle, d.h. ruft die Operation auf
  - Kennt konkrete Implementierung nicht!

Anbieter und Nutzer in UML

Bereitstellung und Nutzung der Schnittstelle-A



Interaktion über Schnittstelle-A

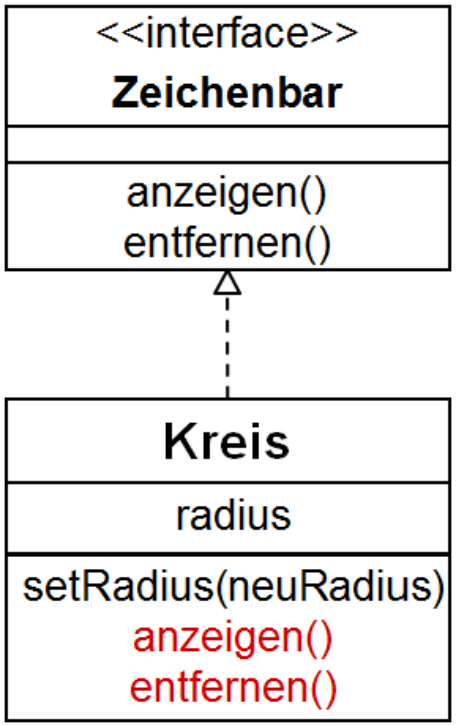


Begriffe – Realisierung und Vererbung

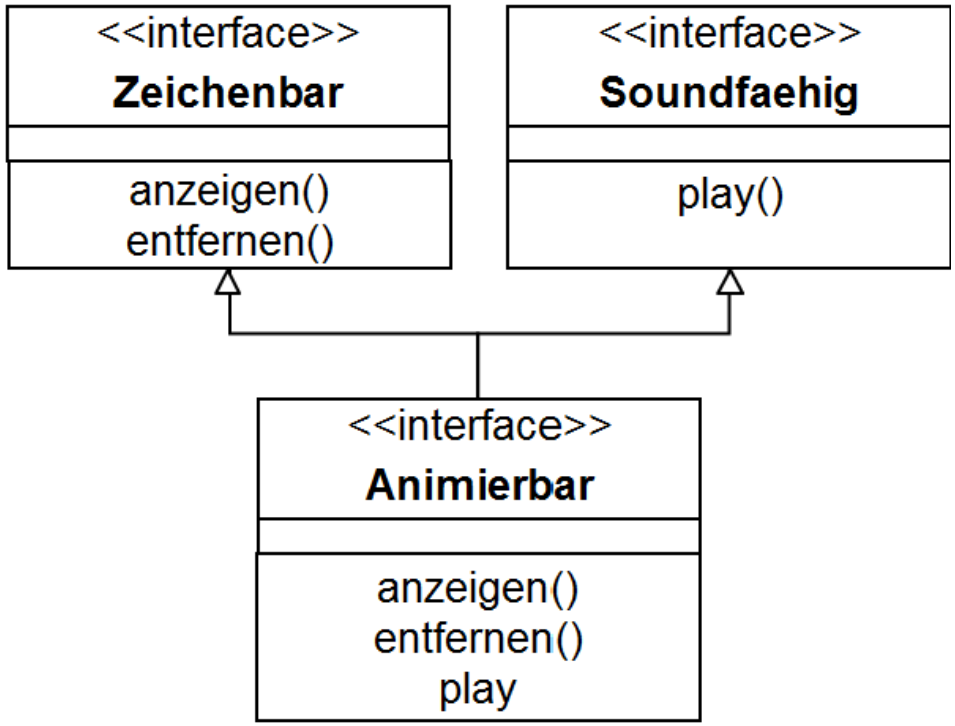
- Realisierung
  - Schnittstelle alleine nicht ausführbar
  - Konkrete Klasse ist von Schnittstelle abgeleitet
  - Sprachgebrauch: „Konkrete Klasse implementiert das Interface“
  - Implementiert dabei alle definierten Operationen der Schnittstelle

- Vererbung zwischen Schnittstellen
  - Neue Schnittstelle erweitert alte Schnittstelle
  - Dabei lediglich Hinzufügen von abstrakten Operationen
  - In Java: Interface kann mehrere Interfaces erweitern
  - D.h. Mehrfachvererbung zwischen Schnittstellen möglich!

Beispiel: Realisierung und Vererbung

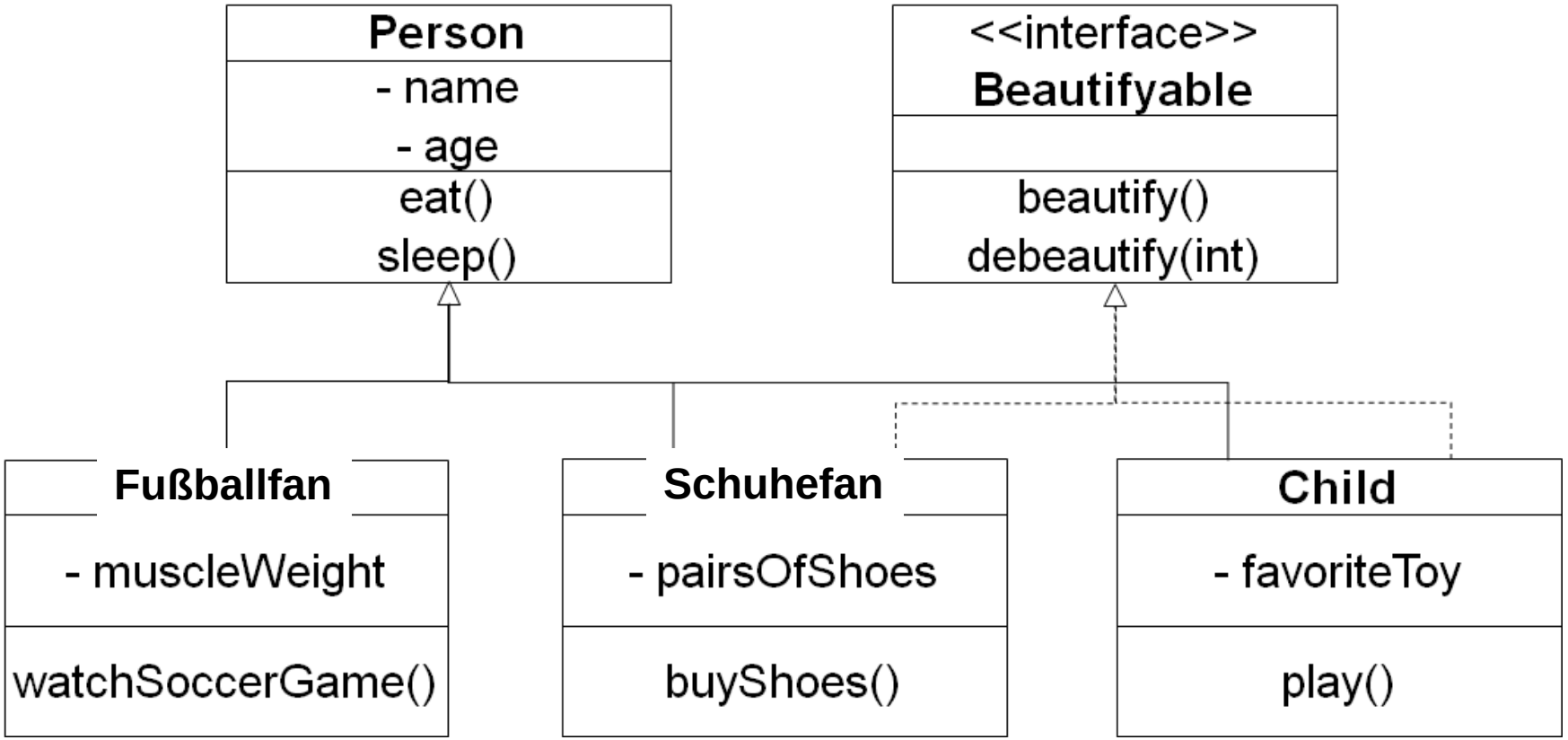


Realisierung



Vererbung

Beispiel


Person
- name
- age
eat()
sleep()
<<interface>>
Beautifyable
beautify()
debeautify(int)
Fußballfan
- muscleWeight
watchSoccerGame()
Schuhefan
- pairsOfShoes
buyShoes()
Child
- favoriteToy
play()

Implementierung des Interfaces

- Veränderungen gegenüber obigem Beispiel
  - Klassen `Fussballfan`, `Person` unverändert
  - Erweiterung der Klasse `Main`
  - Neues Interface `Beautifyable`
  - Klasse `Schuhefan` implementiert Interface `Beautifyable`
  - Neue Klasse `Child` implementiert Interface `Beautifyable`

- Neues Interface `Beautifyable`

```
public interface Beautifyable {
  public String beautify();
  public String debeautify(int minutes);
}
```

Veränderungen der `Main`-Klasse

- Erweiterte Klasse `Main`

```
public class Main{
  public static void main(
    String[] args) {

    Fussballfan adam = new Fussballfan();
    Schuhefan eva = new Schuhefan();
    Child kain = new Child();

    adam.processPerson();
    System.out.println(
      adam.watchSoccerGame());
    System.out.println();
    …
```

```
…
    eva.processPerson();
    System.out.println(
      eva.buyShoes());
    System.out.println(
      eva.beautify());
    System.out.println(
      eva.debeautify(3));
    System.out.println();

    kain.processPerson();
    System.out.println(
      kain.play());
    System.out.println(
      kain.beautify());
    System.out.println(
      kain.debeautify(3));
    System.out.println();
} }
```

Veränderung der Klasse `Woman`

- Erweiterte Klasse `Schuhefan`

```
public class Schuhefan extends Person implements Beautifyable{
  …
  public String beautify() {
    return "Oh, mein Haar ist schon wieder durcheinander!.");
  }
  public String debeautify(int minutes) {
    return "Das schaff ich nie, mich in "
      + minutes + " Minuten zu stylen.");
  }
}
```

Implementierung der neuen Klasse `Child`

```
public class Child extends Person implements Beautifyable{
  private String favoriteToy;
  public String play() {
    return "play : Brrruummmm… Miau… Fiep…";
  }
  public String beautify() {
    return "Guck mal, ich bin ein Ritter" +
      " mit Prinzessinen-Krone!";
  }
  public String debeautify(int minutes) {
    return "Och, muss ich in "
      + minutes + " Minuten schon aufhören?";
  }
}
```

Ausgabe bei Ausführung von `Main`

```
Adam ist 21 Jahre alt und kann superfix Regale
zusammenschrauben.
sleep: Chrrrrr.... chrrrr...
eat  : Mmmmh, lecker.
play : Ja... JAA... TOOOOOOOR!!!
Eva ist 19 Jahre alt und hat 0 Paar Schuhe.
sleep: Chrrrrr.... chrrrr...
eat  : Mmmmh, lecker.
shop : DIE sind ja schick...; Paar Nummer 1
Oh, mein Haar ist schon wieder durcheinander!
Das schaff ich nie, mich in 3 Minuten zu stylen.
Kain ist 4 Jahre alt und hat als Lieblingsspielzeug Teddy
sleep: Chrrrrr.... chrrrr...
eat  : Mmmmh, lecker.
play: Brrruummmm... Miau... Fiep...
Guck mal, ich bin ein Ritter mit Prinzessinen-Krone!
Och, muss ich in 10 Minuten schon aufhören?
```

Interfaces als `Typ`

- Interface als Typ
  - Definiert Referenztyp (analog zu Klasse)
  - Zulässig für Deklaration von Variable, Parameter, Rückgabetyp von Methode
  - Alle implementierenden Klassen kompatibel zum Interface

- Beispiel

```
Beautifyable b;
b = new Child("Sabine", "Bobbycar");
b.beautify();
b = new Schuhefan("Peter", 42);
b.beautify();
```

- Auswahl der Implementierung der Methode dynamisch zur Laufzeit!

Interface vs. Abstrakte Klasse

- Während der Designphase eines komplexen Softwaresystems: häufig schwierig, sich für eine von beiden Varianten zu entscheiden
- **Pro Interfaces**: größere Flexibilität durch die Möglichkeit, in unterschiedlichen Klassenhierarchien verwendet zu werden
- **Pro Klasse**: Möglichkeit, bereits ausformulierbare Teile der Implementation zu realisieren und die Fähigkeit, statische Bestandteile und Konstruktoren unterzubringen
- Kombination der beiden Ansätze: zunächst zur Verfügung stellen eines Interfaces, dann Vereinfachung seiner Anwendung durch Verwendung einer Hilfsklasse

Interface vs. Abstrakte Klasse

- Beispiel: Nachträgliche Änderungen
  - Auch nachträgliche Änderungen an Schnittstellen sind nicht einfach: Einer abstrakten Klasse kann eine konkrete Methode mitgegeben werden, was zu keiner Quellcodeanpassung für Unterklassen führt.

- Ist eine Schnittstelle oder eine abstrakte Klasse besser, um folgende Operation zu deklarieren?

```
abstract class Time {
  abstract long getTimeInMillis()
}
```

```
interface Time {
    long getTimeInMillis()
}
```

Interface vs. Abstrakte Klasse

- Was ist zu tun, wenn man eine Hilfsfunktion `getTimeInSeconds`() einführen will?
- In Schnittstelle → alle implementierenden Klassen müssen diese Implementierung neu einführen.
- In abstrakter Oberklasse → einfach in der abstrakten Klasse einfügen, keine Änderungen der Klassen-Verwender notwendig.

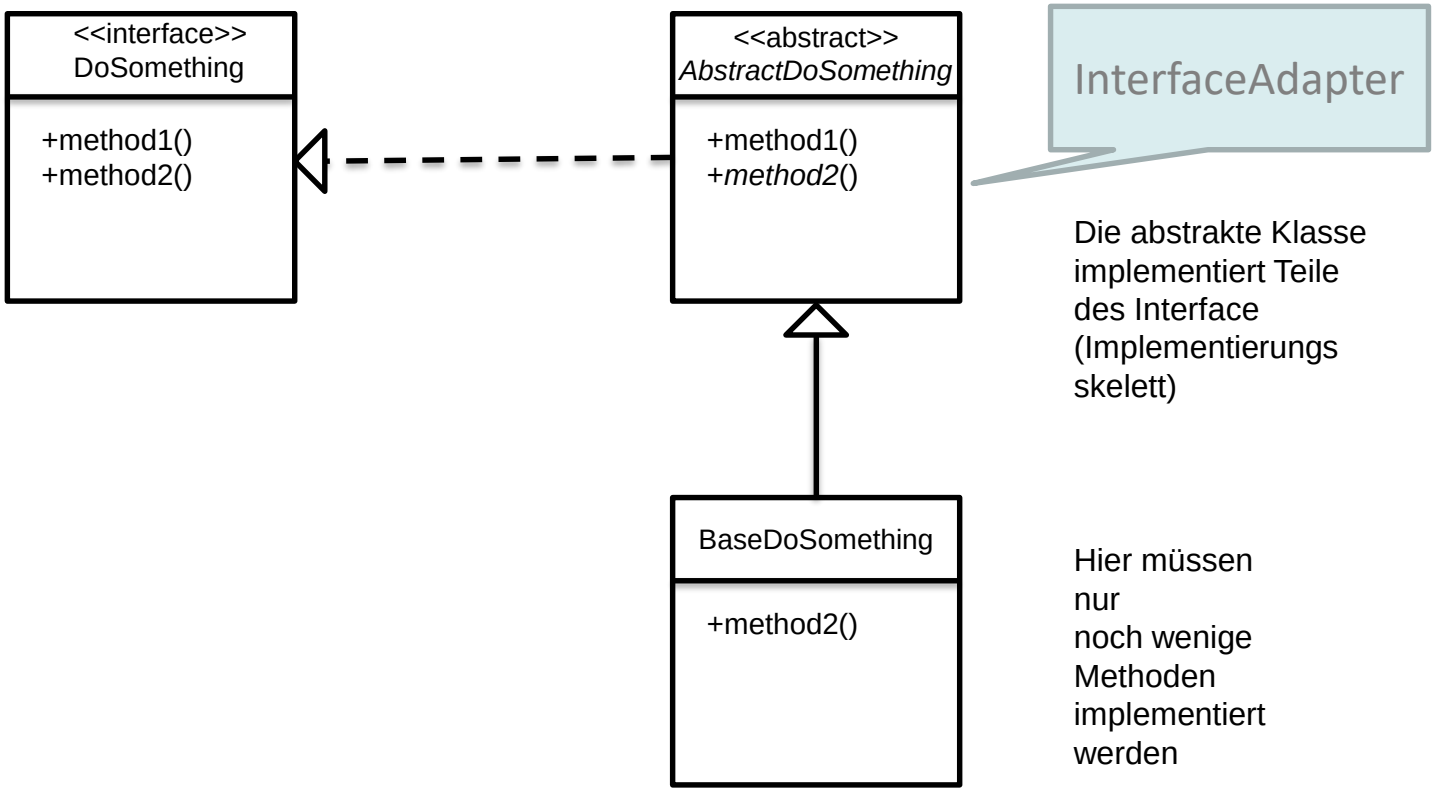
```
abstract class Timer {
  abstract long getTimeInMillis();

  long getTimeInSeconds() {
    return getTimeInMillis() / 1000;
  }
}
```

Interface vs. Abstrakte Klasse

- Wenn man die Möglichkeit haben will in mehr als einen Basistyp zu verallgemeinern (upcast) → Interface
- Wenn der Client-Programmierer kein Objekt erzeugen können soll → Interface oder abstrakte Klasse
- Wenn möglich, d.h. wenn eine Klasse ohne Methoden-Definitionen oder Member-Variablen vereinbart werden kann, dann sollte man ein Interface der abstrakten Klasse vorziehen
- Abstrakte Klassen werden eingesetzt, um gemeinsame Methoden in der Vererbungshierachie an der „richtigen“ Stelle zu platzieren

Kombination von Interfaces und abstrakten Klassen

- Adapter stellen eine Basisimplementierung eines Interfaces zur Verfügung



Übung – Schnittstellen

- Live Übung
  - Bearbeiten Sie Aufgabe 7 des Blatts Live Übung „Vererbung“
  - Sie haben 8 Minuten Zeit.

Typinformation zur Laufzeit

```
AbstrakteBasis ab1 = new NormaleBasis();
AbstrakteBasis ab2 = new KonkreteA();
NormaleBasis nb1 = new KonkreteB();
KonkreteA ka1 = new KonkreteA();
KonkreteB kb1 = new KonkreteB();
```

- Instanzen von Unterklassen können in Oberklassen gespeichert warden
- Laufzeitinformation über
  - **`instanceof`**: z.B. `if(ref instanceof MyClass){ … }`
  - Via `Object.getClass()`