2 Vererbung

2.1 Wofür benötigen wir Vererbung?

Sie erhalten den Auftrag, für die Webseite eines Internet-Shops einzelne Artikel (z.B. Bücher, CDs, DVDs) als Java-Klassen zu modellieren. Dazu erzeugen Sie eine neue Java-Klasse namens **`Buch`**. Das Klassendiagramm zu dieser Klasse ist gegeben und sieht wie folgt aus:

Buch	
-	titel : String
-	preis : double
-	barcode : long
-	seiten : int
-	autor : String
-	auflage : int
+	Buch(String, double, long, int, String, int)
+	getBeschreibung() : String
+	getPreis() : double
+	getBarcode() : long

Erstellen Sie zunächst eine Testklasse mit dem Namen **`WebshopTest`**. In der `main`-Methode dieser Klasse soll ein neues Buch angelegt und dessen Beschreibung, Preis und Barcode ausgegeben werden:

```
public class WebshopTest {
    public static void main(String[] args) {
        Buch b1 = new Buch("UML 2.0", 9.8, 9783897215214L, 139,
                "Dan Pilone", 2);
        System.out.println(b1.getBeschreibung());
        System.out.println("Preis: " + b1.getPreis() + " Euro");
        System.out.println("Barcode: " + b1.getBarcode());
    }
}
```

Beim Starten des Programms soll die folgende Ausgabe in der Konsole erscheinen:

```
Buch: "UML 2.0" von Dan Pilone, 2. Auflage, 139 Seiten
Preis: 9.8 Euro
Barcode: 9783897215214
```

Aufgaben

1. Setzen Sie die Klasse **`Buch`** gemäß dem Klassendiagramm und der obigen Beispielausgabe in Java um.

2. Erzeugen Sie in der Testklasse ein beliebiges zweites Buch `b2` und lassen Sie sich nur die Beschreibung des teureren Buches in der Konsole ausgeben!

Damit sind wir bereits in der Lage, Bücher zu erfassen und ihre Daten auszulesen. Nach dem selben Prinzip sollen nun auch andere Artikel erfasst werden.

3. Schreiben Sie auf Basis des nachfolgenden UML-Klassendiagramms eine neue Java-Klasse namens **`CD`**.

CD	
-	titel : String
-	preis : double
-	barcode : long
-	laufzeit : int
-	komponist : String
+	CD(String, double, long, int, String)
+	getBeschreibung() : String
+	getPreis() : double
+	getBarcode() : long

4. Erzeugen Sie in Ihrer `main`-Methode eine neue Variable vom Typ **`CD`** namens `cd1` und lassen Sie sich ihre Beschreibung ausgeben (denken Sie sich selbst irgendwelche Daten für die CD aus).

Wir haben jetzt zwei Klassen, mit denen wir unterschiedliche Arten von Artikeln aus dem Angebot unseres Webshops erfassen können. Zum Beispiel könnten wir eine Liste von Büchern verwalten und deren Daten nacheinander ausgeben. Fügen Sie bitte an das Ende Ihrer `main`-Methode die folgenden Zeilen ein:

```
Buch[] liste = new Buch[] {b1, b2};
for(int i = 0; i < liste.length; i++) {
    System.out.println(liste[i].getBarcode() + "\t" +
    liste[i].getBeschreibung());
}
```

So weit, so gut. Doch was passiert, wenn wir uns nun die Daten von Büchern *und* CDs ausgeben lassen wollen? Ändern Sie die Zeile, in der das Array `liste` angelegt wurde, wie folgt ab:

```
Buch[] liste = new Buch[] {b1, b2, c1};
```

Achtung! Diese Zeile kann nicht kompiliert werden und wird mit einer entsprechenden Fehlermeldung quittiert. Eine CD ist nunmal kein Buch!

Wenn ein Array vom Typ **`Buch`** angelegt wurde, können in dieses Array ausschließlich Objekte der Klasse **`Buch`** gelegt werden. Ein Objekt der Klasse **`CD`** ruft hingegen eine

Fehlermeldung hervor. Andersherum wäre es ebenfalls nicht möglich, die `liste` als **`CD`**-Array auszulegen, da die Objekte `b1` und `b2` vom Typ **`Buch`** sind und somit auch nicht als **`CD`** abgelegt werden können. Damit wir sowohl CDs als auch Bücher in einem gemeinsamen Array ablegen können, müssen wir eine Klasse finden, die sowohl CDs als auch Bücher aufnehmen kann.

Außerdem fällt auf, dass einige Attribute und Methoden sowohl in der Klasse **`CD`** als auch in **`Buch`** definiert wurden. Beispielsweise liegen die Methoden `getBarcode` und `getPreis` in beiden Klassen in identischer Form vor. Diese Redundanz ist zum einen unschön (der Programmierer hat die doppelte Arbeit) und zum anderen gefährlich (wenn sich eine der Methoden ändert, muss daran gedacht werden, die Änderungen auch in die andere Klasse zu übernehmen).

Nun kommt die **Vererbung** ins Spiel, die es uns erlaubt unser Programm etwas aufzuräumen und für zukünftige Erweiterungen offen zu halten.

2.2 Eine Oberklasse

Für die beiden Klassen **`Buch`** und **`CD`** muss nun zunächst einmal ein Name für eine neue Oberklasse gefunden werden, die beide Klassen generalisiert beschreibt. Da es sich bei Büchern und CDs um Artikel aus unserem Warensortiment handelt, wählen wir den Klassennamen **`Artikel`**. Legen Sie also zunächst eine neue Klasse mit diesem Namen an.

In **`Artikel`** werden jetzt zunächst alle Attribute gesammelt, die jeder Artikel (egal ob es sich um ein Buch, eine CD oder eine andere Art von Artikel handelt) besitzt. Wir stellen fest, dass sowohl in **`Buch`** als auch in **`CD`** die Attribute `titel`, `preis` und `barcode` vorhanden sind. Es ist zu erwarten, dass auch jede weitere Art von Artikel, die wir in Zukunft hinzufügen werden, einen Titel, einen Preis und einen Barcode besitzen wird. Daher erhalten wir folgenden Quelltext für unsere Oberklasse:

```
public class Artikel {
    protected String titel;
    protected double preis;
    protected long barcode;
}
```

Achtung!

Auf die Attribute dieser Klasse soll von allen Unterklassen zugegriffen werden können – `private` Attribute sind aber gegen Zugriff durch andere Klassen versteckt. Daher müssen wir für vererbte Attribute das neue Zugriffsrecht `protected` verwenden, das soviel heißt wie *"zugreifbar innerhalb der Klasse und aller ihrer Unterklassen."* Im Klassendiagrammen werden `protected`-Methoden oder -Attribute durch ein `#`-Zeichen gekennzeichnet.

Nun können wir beginnen, die Vererbungen festzulegen. Ändern Sie die erste Zeile der Klasse **`Buch`** wie folgt ab:

```
public class Buch extends Artikel {
```

Ebenso können wir die Klasse **CD** abändern:

```
public class CD extends Artikel {
```

Durch diese Änderungen haben wir dem Compiler nun mitgeteilt, dass **Buch** und **CD** Unterklassen von **Artikel** sind.

Tipp

Will man sichergehen, dass eine Vererbung Sinn ergibt, kann man das Schlüsselwort `extends` deuten als *"…ist eine Art von…"*. Für die Klasse Buch würde das zu folgendem Satz führen: *"ein Buch ist eine Art von Artikel"*. Andere Beispiele wären: *"ein Tisch ist eine Art von Möbel"* oder *"ein Goldfisch ist eine Art von Lebewesen"*. Wenn der Satz keinen Sinn ergibt, hat man etwas falsch gemacht: *"ein Obst ist eine Art von Apfel"* ist ebenso falsch wie *"ein Auto ist eine Art von Nahrungsmittel"*.

Die drei Attribute `titel`, `preis` und `barcode` wurden nun bereits in der Oberklasse definiert; eine nochmalige Definition in den Unterklassen ist überflüssig.

Aufgaben

1. Entfernen Sie in den Klassen **Buch** und **CD** die Zeilen, in denen die bereits in der Oberklasse festgelegten Attribute festgelegt wurden.
2. Verschieben Sie die beiden Methoden `getPreis` und `getBarcode` aus den beiden Unterklassen in die Oberklasse.

2.3 Abstrakte Klassen und Methoden

Als letzte Gemeinsamkeit zwischen **Buch** und **CD** müssen wir nun noch die Methode `getBeschreibung` betrachten. Auch diese ist in beiden Klassen mit identischem Methodenkopf vorhanden, unterscheidet sich aber im eigentlichen Inhalt der Methode: Beide Klassen liefern klar unterschiedliche Beschreibungstexte. Daher können wir diese Methode nun nicht ohne Weiteres in der Oberklasse vorgeben, da sich die zurückgegebenen Werte in den Unterklassen unterscheiden sollen. Wir müssen also in der Oberklasse vorgeben, dass jeder Artikel eine Methode `getBeschreibung` haben soll, legen aber noch nicht fest *wie* der Inhalt dieser Methode aussehen soll. Eine solche Methode nennt man **abstrakte Methode**.

Fügen Sie bitte in Ihre Klasse **Artikel** die folgende Methodendefinition ein:

```
public abstract String getBeschreibung();
```

Damit haben wir festgelegt, dass jede **`Artikel`**-Unterklasse eine Methode `getBeschreibung` beinhalten muss – wie diese Methode auszusehen hat, ist der jeweiligen Unterklasse überlassen.

Achtung!

Beachten Sie bitte, dass eine abstrakte Methodendefinition statt mit einer geschweiften Klammer **immer** mit einem Semikolon abgeschlossen wird. Zusätzlich muss die Methode mit dem Schlüsselwort `abstract` gekennzeichnet werden.

Jetzt liefert unsere Klasse **`Artikel`** jedoch eine Fehlermeldung: Dadurch, dass wir eine abstrakte Methode eingebaut haben, kann jetzt kein Objekt dieser Klasse mehr erzeugt werden, da ein solches Objekt ja nicht wüsste, wie es beim Aufruf der abstrakten Methoden zu reagieren hätte. Die Klasse **`Artikel`** ist durch die Einbindung abstrakter Methoden selbst zu einer **abstrakten Klasse** geworden. Dies müssen wir in der Klassendefinitionszeile entsprechend kennzeichnen, indem wir das Schlüsselwort `abstract` einfügen:

```
public abstract class Artikel {
```

Das endgültige Klassendiagramm sieht nun wie in Abbildung 1 aus.

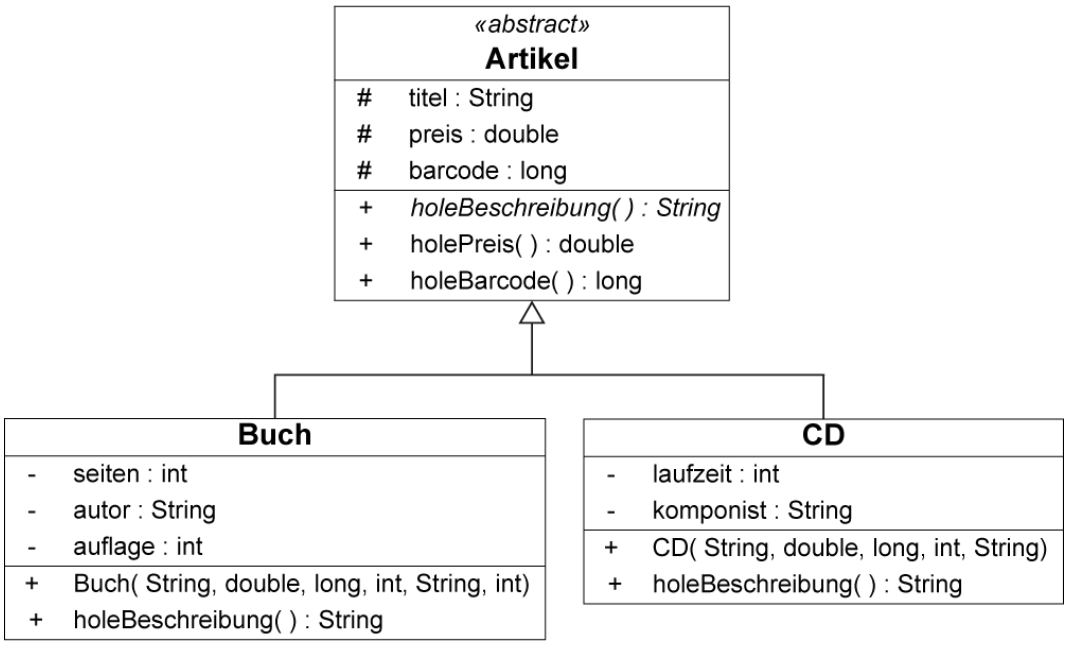

Abbildung 1: Klassendiagram des Webshops

In der `main`-Methode unserer Klasse **`WebshopTest`** können wir nun endlich testen, wie verschiedene Artikel in einem Array gesammelt und ausgegeben werden können:

```
Artikel[] liste = new Artikel[] {b1, b2, c1};
for(int i = 0; i < liste.length; i++) {
    System.out.println(liste[i].getBarcode() + "\t" +
    liste[i].getBeschreibung());
}
```

Aufgaben

1. Schreiben Sie eine neue **Artikel**-Unterklasse namens **DVD**. Eine DVD soll neben den **Artikel**-Attributen einen Regisseur und ein Erscheinungsjahr besitzen. Fügen Sie testweise einige DVDs zu `liste` hinzu.

2. Schreiben Sie eine Klasse **Bluray**, die von der Klasse **DVD** abgeleitet ist, und sich nur in der Methode `getBeschreibung` von der DVD unterscheidet.

3. Berechnen Sie in der `main`-Methode den Gesamtpreis aller Artikel des Arrays `liste` und lassen Sie ihn in der Konsole ausgeben.

2.4 Der instanceof-Operator

In einer Variable vom Typ **Artikel** können sich – Dank der Vererbung – Objekte verschiedener Klassen befinden: Sei es ein **Buch**, eine **CD**, eine **DVD** oder eine **Bluray**-Disk. Manchmal (wenn auch eher selten) ist es wünschenswert herauszufinden, ob ein Objekt von einem bestimmten Typ ist.

Nehmen wir an, wir wollen aus unserer `liste` nur diejenigen Objekte ausgeben, die vom Typ **CD** sind. Fügen Sie dazu die folgenden Zeilen am Ende der `main`-Methode ein:

```
for(int i = 0; i < liste.length; i++) {
    if(liste[i] instanceof CD) {
        System.out.println(liste[i].getBeschreibung());
    }
}
```

Der neue Operator `instanceof` überprüft, ob ein Objekt von einem bestimmten Typ ist. Wenn dies der Fall ist, liefert der Operator den Wert `true` zurück, ansonsten den Wert `false`.

Aufgaben

1. Lassen Sie sich die Barcodes aller Bücher in `liste` ausgeben.

2. Berechnen Sie den Gesamtpreis aller DVDs (aber nicht der Blu-ray-Disks!) von `liste` und lassen Sie sich die Summe in der Konsole ausgeben.

Wenn man nun weiß, dass ein Objekt von einer bestimmten Klasse ist, kann man durch sogenanntes **Casting** wieder auf die speziellen Attribute und Methoden eines Objektes zugreifen. Nehmen wir an, wir wollen uns jetzt nur die Autoren aller Bücher ausgeben lassen. Fügen Sie bitte in die Klasse **Buch** eine neue Methode `getAutor()` ein, die den Autor des Buches zurückgibt. In der `main`-Methode fügen wir folgende Zeilen ein:

```
for(int i = 0; i < liste.length; i++) {
    if(liste[i] instanceof Buch) {
        Buch b = (Buch)liste[i];
        System.out.println(b.getAutor());
    }
}
```

Damit wir auf die speziellen Methoden der Klasse **Buch** zugreifen können, müssen wir dem Compiler mitteilen, dass er den **Artikel**, der sich in der `liste` befindet, als **Buch** betrachten soll. Dafür **casten** wir das Objekt `liste[i]` auf die Klasse **Buch**, indem wir den Klassennamen in runde Klammern vor das Objekt schreiben. Für das so erhaltene Buch `b` können wir dann die Methode `getAutor` aufrufen.

Achtung! Wenn das gecastete Objekt kein Buch ist, wird das Programm mit einer `Class-Cast-Exception` abgebrochen.

2.5 Interfaces

Nehmen wir an, wir wollen in unseren Webshop eine Umtauschmöglichkeit einbauen. Dabei sollen Bücher 30 Tage lang, CDs 14 Tage lang und DVDs bzw. Blu-Ray-Disks gar nicht umgetauscht werden können. Hierzu schreiben wir uns ein neues Interface namens **Umtauschbar**:

```
public interface Umtauschbar {
    public int umtauschfrist();
}
```

Der einzige Unterschied zwischen diesem Interface und einer abstrakten Klasse ist das Schlüsselwort `interface`, und die Tatsache, dass die abstrakte Methode `umtauschfrist` nicht als abstrakt gekennzeichnet werden muss, da in einem Interface automatisch alle Methoden abstrakt sind.

Um das Interface in unsere Klassen **Buch** bzw. **CD** einzubauen, verwenden wir das neue Schlüsselwort `implements`, das äquivalent zum `extends`-Schlüsselwort für Oberklassen angewendet wird. Einziger Unterschied: Es können hinter `implements` mehrere Interfaces durch Komma getrennt angegeben werden.

```
public class Buch extends Artikel implements Umtauschbar {
```

Aufgaben

1. Bauen Sie das Interface **Umtauschbar** wie oben gezeigt in die Klassen **Buch** und **CD** ein. Fügen Sie die notwendigen Umsetzungen der abstraken Methoden `umtauschfrist` mit den oben genannten Fristen ein.

2. Lassen Sie sich in der `main`-Methode die Umtauschfristen aller umtauschbaren Artikel der `liste` ausgeben.

3. Schreiben Sie ein Interface **`Abspielbar`** mit einer Methode `getAbspieldauer()` (liefert die Abspieldauer in Minuten) und einer Methode `spieleAb()` (gibt einen kurzen Text in der Konsole aus). Lassen Sie die Klassen **`CD`** und **`DVD`** das Interface implementieren.

4. Lassen Sie in der `main`-Methode alle abspielbaren Artikel abspielen, deren Abspieldauer größer als 90 Minuten ist.