Programmieren I

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2009/2010

Definition: Polymorphie

Der Begriff *Polymorphie* (manchmal auch exakter: Laufzeit-Polymorphie) bei der Ausführung von Methoden bedeutet:

- Es wird erst zur Laufzeit bestimmt wird, welche Implementierung für eine bestimmte Methode aufzurufen ist.
- Es kann also vom Programmablauf abhängig sein, aus welcher Klasse die Methode verwendet wird.
- Polymorphie ist ein wichtiger Bestandteil der objektorientierten Programmierung.

Häufig wird Polymorphie als *späte Bindung* bezeichnet.

Beispiel

Betrachten wir als einfaches Beispiel die beiden folgenden (sinnlosen) Klassen:

```
public class Basis {
    public int a = 1;

    public int getA() {
        return a;
    }
}


public class Erweitert extends Basis {
    public int a = 2;

    public int getA() {
        return a;
    }
}
```

Frage

Welche Ausgabe ergibt folgendes Hauptprogramm?

```
public class Anwendung {
    public static void main(String[] args) {
        Erweitert e1 = new Erweitert();
        Basis b1 = e1;
        Basis b2 = new Basis();

        System.out.println(e1.a);
        System.out.println(b1.a);
        System.out.println(b2.a);

        System.out.println(e1.getA());
        System.out.println(b1.getA());
        System.out.println(b2.getA());
    }

}
```

Analyse

Bei den ersten drei Ausgabeanweisungen wird direkt auf das (öffentliche) Attribut a zugegriffen.

Hierbei gilt, dass anhand des *Typs der Objektreferenz* entschieden wird, aus welcher Klasse das Attribut a verwendet wird.

Dies führt zu dem Effekt, dass `e1` und `b1` dasselbe Objekt referenzieren, jedoch die Ausdrücke `e1.a` und `b1.a` unterschiedliche Ergebnisse liefern, da `e1` und `b1` von verschiedenem Typ sind.

In diesem Fall liegt schon zur Kompilierzeit fest, aus welcher Klasse a verwendet wird.

Ausgabe also: 2 1 1

Analyse (Forts.)

Bei Methoden in Java gilt das Prinzip der späten Bindung.

Daher wird erst zur Laufzeit anhand der *realen Instanz*, auf die eine Referenz verweist, entschieden, aus welcher Klasse eine Methode benutzt wird.

Daher ergeben im Beispiel `e1.getA()` und `b1.getA()` dasselbe Resultat, da beide auf eine (sogar dieselbe) Instanz von `Erweitert` verweisen.

Demgegenüber unterscheiden sich `b1.getA()` und `b2.getA()`, obwohl beides Referenzen vom Typ `Basis` sind, da sie tatsächlich auf Instanzen verschiedener Klassen verweisen.

Ausgabe also: 2 2 1

Praktisches Beispiel – Motivation

Ein moderner Supermarkt besitzt Kassen, die sowohl scannen als auch wiegen können:

- Normale Waren werden einzeln über den Scanner gezogen, und pro Stück der Einzelpreis zur Rechnungssumme addiert.
- Bei Waren, die gewogen werden, wird der Kilopreis mit dem Gewicht der Ware (in kg) multipliziert und zur Rechnungssumme addiert.

Es steht also fest: Jede Ware hat einen Preis, es hängt aber davon ab, ob es eine Stückware oder eine Gewichtsware ist, wie sich der Preis berechnet.

Implementierung – Klasse Ware

Zunächst schaffen wir eine allgemeine Klasse Ware, die neben der Beschreibung der Ware erst einmal einen (noch unsinnigen) Preis hat:

```
public class Ware {
    private String beschreibung;

    public Ware(String beschreibung) {
        this.beschreibung = beschreibung;
    }

    public String getBeschreibung() {
        return beschreibung;
    }

    public double getPreis() {
        return 0;
    }
}
```

Implementierung – Klasse StueckWare

Nun kommt die Klasse StueckWare hinzu, bei der als Preis einfach der Stückpreis verwendet wird:

```
public class StueckWare extends Ware {
    private double einzelpreis;

    public StueckWare(String beschreibung, double einzelpreis) {
        super(beschreibung);
        this.einzelpreis = einzelpreis;
    }

    @Override
    public double getPreis() {
        return einzelpreis;
    }
}
```

Implementierung – Klasse GewichtsWare

Nun die Klasse GewichtsWare hinzu, bei der Preis aus Gewicht und Kilopreis berechnet wird:

```
public class GewichtsWare extends Ware {
    private double gewicht;
    private double kilopreis;

    public GewichtsWare(String beschreibung, double gewicht,
            double kilopreis) {
        super(beschreibung);
        this.gewicht = gewicht;
        this.kilopreis = kilopreis;
    }

    @Override
    public double getPreis() {
        return gewicht*kilopreis;
    }
}
```

Anwendung

Somit kann für einen Array von Waren verschiedenen Typs einfach der Preis aufsummiert werden, da dynamisch entschieden wird, aus welcher Klasse `getPreis` verwendet wird:

```
public class Anwendung {
    public static void main(String[] args) {
        Ware[] waren = new Ware[2];
        waren[0] = new Stueckware("Chips",1.89);
        waren[1] = new GewichtsWare("Rinderhack",6.99,.5);

        double summe = 0;
        for (Ware w : waren) {
            summe += w.getPreis();
        }
        System.out.println("Gesamtsumme: "+summe);
    }
}
```

Kritikpunkte am Beispiel

Das Beispiel dient zwar zur Demonstration der späten Bindung, allerdings gibt es zwei Kritikpunkte:

- Es wäre möglich, im Beispiel Instanzen von `Ware` zu erzeugen, obwohl in der Realität nur Instanzen von `StueckWare` und `GewichtsWare` vorkommen sollen.
- In `Ware` ist die Methode `getPreis` mit sinnlosem Code definiert, nur damit sie dort schon bekannt ist (und für Referenzen vom Typ `Ware` benutzt werden kann).

Abstrakte Klassen

Klassen in Java können Methoden besitzen, die *abstrakt* sind.

Abstrakte Methoden besitzen nur einen (als abstrakt gekennzeichneten) Deklarationskopf, aber keinen Code.

Klassen mit mindestens einer abstrakten Methode *müssen* als abstrakt markiert sein.

Von abstrakten Klassen können keine Instanzen erzeugt werden.

Neue Version der Klasse Ware

```
public abstract class Ware {
    private String beschreibung;

    public Ware(String beschreibung) {
        this.beschreibung = beschreibung;
    }

    public String getBeschreibung() {
        return beschreibung;
    }

    public abstract double getPreis();
}
```

Interfaces

Manchmal implementieren abstrakte Klassen keinerlei Code, sondern *deklarieren* nur nach außen sichtbare Methoden.

Für solche Zwecke gibt es in Java ein spezielles Sprachelement, die so genannten *Interfaces*.

Ein Interface wird in Java wie eine Klasse definiert (nur mit dem Schlüsselwort **`interface`** statt **`class`**).

Namen von Interfaces enden häufig auf `-able`.

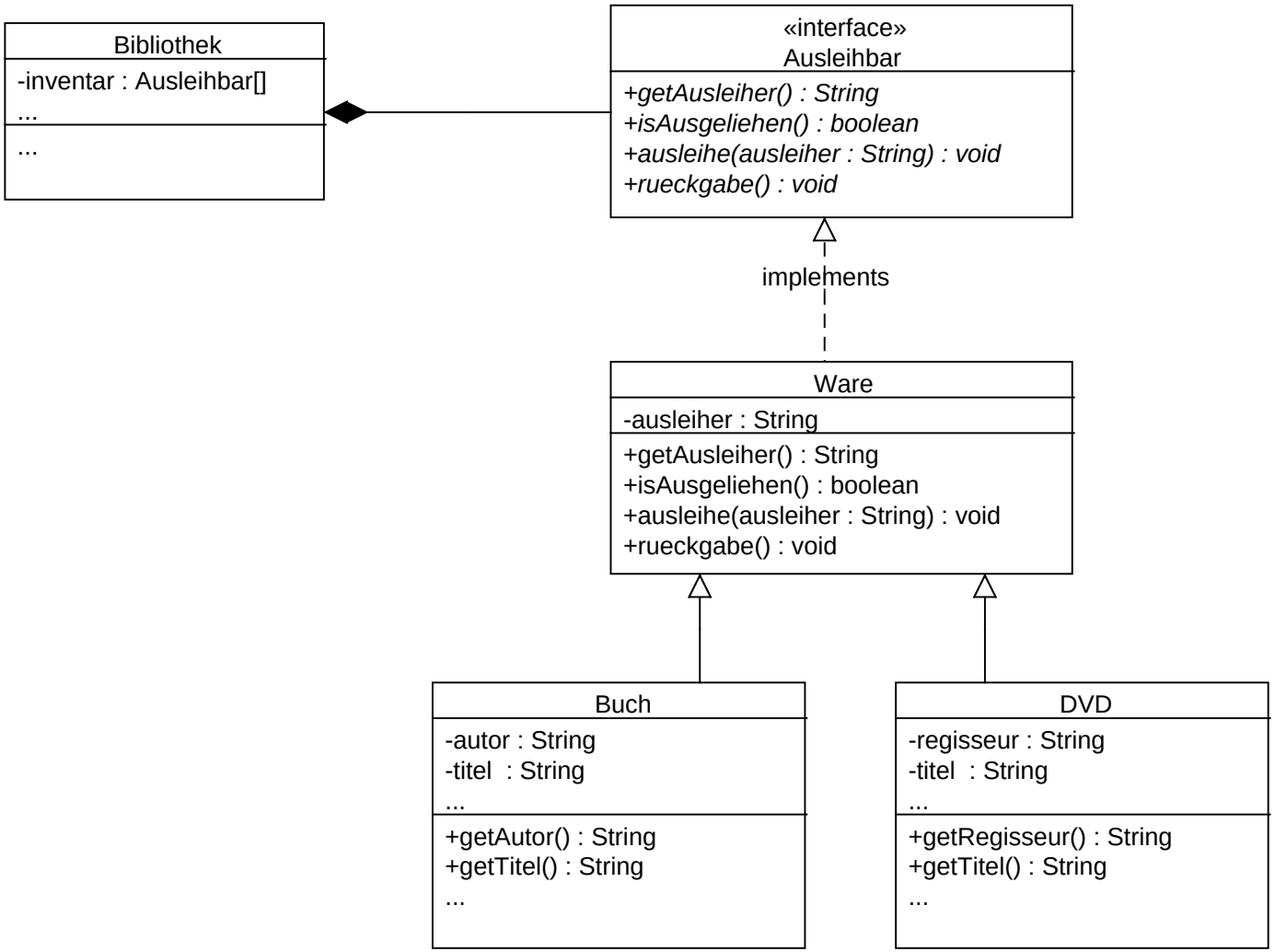
Programmieren gegen Interfaces

Interfaces in Java dienen dazu, dem Programmierer die Möglichkeit zur Trennung von *Schnittstelle* und *Implementierung* zu geben.

Hierbei bietet es sich an, im Code dann Objektreferenzen vom Typ des *Interfaces* zu benutzen.
Vorteile:

- Der Programmierer besitzt die Garantie, dass die referenzierten Objekte die im Interface deklarierten Methoden unterstüzten,
- aber die eigentliche Implementierung befindet sich in der Klasse, von der das referenzierte Objekt instanziert wurde, und bleibt so gekapselt.

Beispiel: Bibliothek



Unterschied: Abstrakte Klassen – Interface

	Abstrakte Klassen	Interfaces
Methoden	mindestens eine abstrakte Methode, aber zusätzlich nichtabstrakte erlaubt	nur abstrakte Methoden erlaubt
Attribute	beliebig	nur statisch
Vererbung	eine Klasse kann nur von einer einzigen abstrakten Klasse erben	eine Klasse darf beliebig viele Interfaces implementieren (und zusätzlich von einer Klasse erben)

Beispiel: Verwendung von Comparable

```
public class GewichtsWare extends Ware implements Comparable {
    ...

    public int compareTo(Object arg) {
        if (arg instanceof GewichtsWare) {
            GewichtsWare other = (GewichtsWare) arg;
            if (gewicht < other.gewicht) {
                return -1;
            } else if (gewicht == other.gewicht) {
                return 0;
            } else return 1;
        }
        throw new IllegalArgumentException("Illegal Argument");
    }
}
```