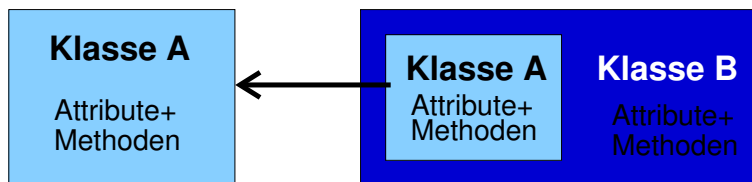


Vererbung	Type Casts	Konstruktor	Beispiel	Sichtbarkeit
<h1>Programmieren I</h1> <p>Dr. Klaus Höppner</p> <p>Hochschule Darmstadt – Wintersemester 2009/2010</p>				
1/21				
<p>Vererbung</p> <p>Type Casts</p> <p>Konstruktor</p> <p>Beispiel</p> <p>Sichtbarkeit</p>				
2/21				
Vererbung	Type Casts	Konstruktor	Beispiel	Sichtbarkeit
<h2>Vererbung</h2> <p>Eine Klasse kann eine speziellere Version einer allgemeineren Klasse sein, z. B.</p> <ul style="list-style-type: none">• Kfz (allgemein) – Pkw, Lkw (speziell)• Konto (allgemein) – Girokonto, Sparkonto (speziell) <p>Diesen Mechanismus nennt man <i>Vererbung</i>, d. h. die spezielle Klasse <i>erbt</i> die Attribute und Methoden der allgemeinen Klasse.</p> <p>In der abgeleiteten Klasse können neue Attribute/Methoden hinzu kommen, aber auch Attribute/Methoden der Basisklasse redefiniert werden.</p> <p>Beispiel: In einer Leihbibliothek gibt es Waren in Form von Büchern, DVDs, CDs etc.:</p> <p style="padding-left: 40px;"><i>Inventar – Buch, DVD, CD, ...</i></p>				
3/21				

Schaubild



4/21

Einfaches Beispiel

Betrachten wir zwei einfache Klassen:

```

public class Tier {
    public void atme() {
        System.out.println("Ich atme");
    }
}
  
```

```

public class Ente extends Tier {
    public void quak() {
        System.out.println("quak");
    }
}
  
```

5/21

Einfaches Beispiel (Forts.)

In diesem Beispiel wird durch

class Ente extends Tier

Ente als Kindklasse von **Tier** definiert. Dabei erbt **Ente** dann alle Attribute und Methoden der Elternklasse **Tier**, in diesem Fall also die Methode **void atme()**.

Tatsächlich ist jede Instanz von **Ente** auch eine Instanz von **Tier**. Daher kann man Enten überall da verwenden, wo allgemein ein Tier erwartet wird.

6/21

Einfaches Beispiel (Forts.)

Beispielprogramm:

```
public class App {
    public static void main(String[] args) {
        Tier t1 = new Tier();
        Ente e = new Ente();
        Tier t2 = e; // zulässig, da jede Ente ein Tier ist

        t1.atme();
        e.atme(); // atme wird von Tier geerbt
        e.quak(); // geht
        t2.quak(); // FEHLER, t2 ist "nur" ein Tier
        System.out.println(e instanceof Tier); // true
    }
}
```

7/21

Type Casts – Typumwandlung

Da eine Ente ein Tier ist, kann jede Objektreferenz vom Typ **Tier** auf eine Instanz von **Ente** verweisen. Daher ist die Anweisung `Tier t2 = e;` zulässig. Dies nennt man *up cast*. Ein solcher ist immer zulässig.

`t2` „kann“ dann natürlich nur das, was alle Tier können!

Man kann aus einer Objektreferenz vom Typ **Tier** wieder eine Referenz vom Typ **Ente** machen, aber nur, wenn diese Referenz auf eine Instanz von **Ente** verweist! Dies nennt man *down cast*. Solche Casts müssen explizit durchgeführt werden, z. B.

```
Ente e2 = (Ente) t2;
```

8/21

Vererbung und Konstruktoren

Erweitern wir den Code von **Tier** so, dass jedes Tier einen Namen hat, der beim Instanzieren übergeben wird:

```
public class Tier {
    private String name;

    public Tier(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void atme() {
        System.out.println("Ich atme");
    }
}
```

9/21

Vererbung und Konstruktoren (Forts.)

Nun gibt es einen Fehler in **Ente**! Warum?

Beim Instanzieren eines neuen Objektes der Kindklasse wird implizit *zuerst* der Standardkonstruktor der Elternklasse ausgeführt. Dieser existiert im Beispiel nun aber nicht mehr, nachdem der Klasse **Tier** der Konstruktor mit einem Namen als Parameter definiert wurde!

Im Konstruktor einer Klasse darf als *erste* Anweisung *einer* der folgenden Aufrufe stehen:

this(...) Aufruf eines anderen Konstruktors derselben Klasse, oder

super(...) Angabe, welcher Konstruktor der Elternklasse aufgerufen werden soll.

Fehlt beides, so wird implizit **super()** angenommen.

10/21

Beispiel

Im vorigen Beispiel kann der Compiler-Fehler nun dadurch beseitigt werden, dass der Konstruktor von **Tier** mit Parameter aufgerufen wird:

```
package tier;
```

```
public class Ente extends Tier {
    public Ente(String name) {
        super(name);
    }

    public void quak() {
        System.out.println("quak");
    }
}
```

11/21

Diagramm der Klasse Buch

In einer früheren Vorlesung wurde eine Klasse Buch definiert:

Buch
-String autor -String titel -String verlag -int jahr -String isbn -String ausleiher
+boolean isAusgeliehen() +void ausleihe(ausleiher : String) +void rueckgabe() +String getAusleiher() +String getAutor() +String getTitel() +String getVerlag() +int getJahr() +String getISBN()

12/21

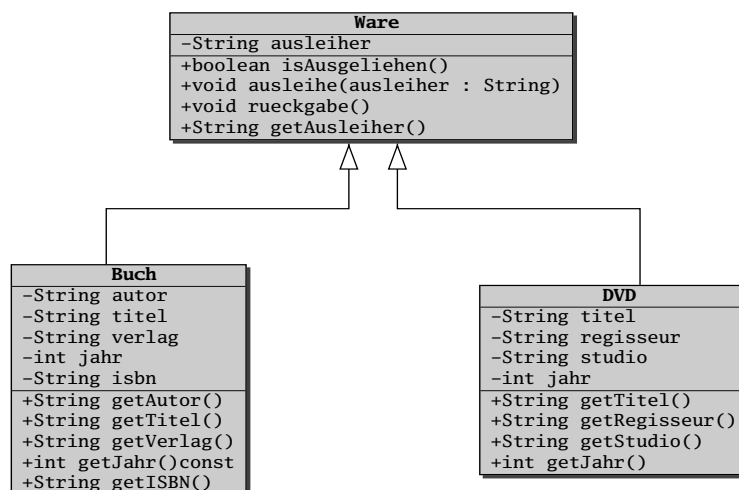
Erweiterung

Die Klasse Buch enthält Teile, die den Vorgang des Ausleihens betreffen, als auch buchspezifische Details. Möchte eine Bibliothek neben Büchern auch noch weitere Gegenstände anbieten (z. B. DVDs, Musik-CDs), so gibt es zwei Möglichkeiten:

- In neuen Klassen werden sowohl die Ausleihe als auch die Produktdetails definiert. Das ist unpraktisch, da für jede neue Warenart die Funktionalität der Ausleihe erneut definiert werden muss.
- Die Funktionalität, die für alle Waren identisch ist – also die Ausleihe –, wird in eine Basisklasse ausgelagert, von der für jede Warenart eine Kindklasse erbt.

13/21

Klassendiagramm: Bücher und DVDs als Warenarten



14/21

Implementierung von Ware.java

```

public class Ware {
    private String ausleiher = null;

    public String getAusleiher() {
        return ausleiher;
    }
    public void ausleiher(String ausleiher) {
        this.ausleiher = ausleiher;
    }
    public void rueckgabe() {
        ausleiher = null;
    }
    public boolean isAusgeliehen() {
        return ausleiher == null;
    }
}
  
```

15/21

Implementierung von Buch.java

```
public class Buch extends Ware {
    private String autor;
    private String titel;
    private String verlag;
    private int jahr;
    private String isbn;

    public Buch(String autor, String titel, String verlag,
                int jahr, String isbn) {
        super();
        this.autor = autor;
        this.titel = titel;
        this.verlag = verlag;
        this.jahr = jahr;
        this.isbn = isbn;
    }
}
```

16/21

Implementierung von Buch.java (Forts.)

```
public String getAutor() {
    return autor;
}
public String getTitel() {
    return titel;
}
public String getIsbn() {
    return isbn;
}
public int getJahr() {
    return jahr;
}
public String getVerlag() {
    return verlag;
}
}
```

17/21

Implementierung von DVD.java

```
public class DVD extends Ware {
    private String regisseur;
    private String titel;
    private String studio;
    private int jahr;

    public DVD(String regisseur, String titel,
               String studio, int jahr) {
        super();
        this.regisseur = regisseur;
        this.titel = titel;
        this.studio = studio;
        this.jahr = jahr;
    }
}
```

18/21

Implementierung von DVD.java

```

public String getRegisseur() {
    return regisseur;
}
public String getTitel() {
    return titel;
}
public int getJahr() {
    return jahr;
}
public String getStudio() {
    return studio;
}
}

```

19/21

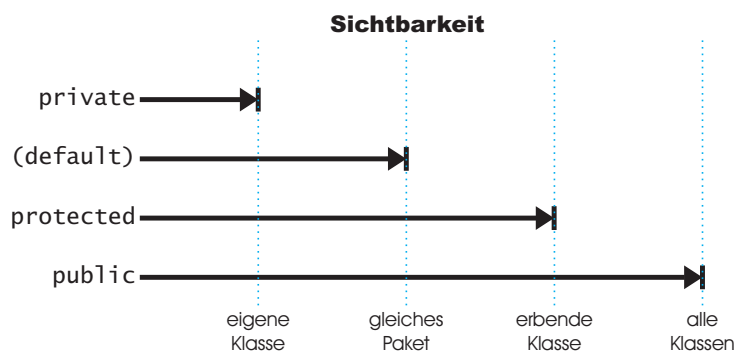
Sichtbarkeit von Klassen, Methoden, Attributen

Man kann in Java für Klassen, Methoden und Attribute entscheiden, in welchem Kontext sie sichtbar sind. Die Sichtbarkeitsstufen lauten:

- private** Sichtbarkeit nur innerhalb derselben Klasse,
- (keine Angabe) Sichtbarkeit innerhalb des selben Paketes
(**package**),
- protected** Sichtbarkeit in allen erbenden Klassen (zusätzlich zur Sichtbarkeit im selben Paket!)
- public** Sichtbarkeit in allen Klassen

20/21

Schaubild



21/21