



Programmierung 2

Vererbung I

Alexander Gepperth, Mai 2025



Vererbung: Grundlagen



Kap. 5.8 (bis 5.14)

OOP: Vererbung für Dummies

- Java-Klassen können ihre Eigenschaften an andere **vererben**!

```
class Oberklasse {  
    int x = 3 ;  
    public int getX() {  
        return this.x ;  
    }  
}
```

```
class Unterklasse extends Oberklasse {  
    public static void main(String[] a) {  
        Unterklasse u = new Unterklasse() ;  
        System.out.println( u.getX() ) ;  
    }  
}
```

?

3



Kap. 5.8 (bis 5.14)

OOP: Vererbung für Dummies

- Java-Klassen können ihre Eigenschaften an andere **vererben**!

```
class Oberklasse {  
    int x = 3 ;  
    public int getX() {  
        return this.x ;  
    }  
}
```

```
class Unterklasse extends Oberklasse {  
    public static void main(String[] a) {  
        Unterklasse u = new Unterklasse() ;  
        System.out.println( u.getX() ) ;  
    }  
}
```

3

4



Kap. 5.8 (bis 5.14)

OOP: Vererbung für Dummies

- Terminologie:
 - **Oberklasse:** vererbt ihre Eigenschaften, steht nach `extends`
 - **Unterklasse:** erbt Eigenschaften, steht vor `extends`
- Synonyme:
 - Oberklasse: *superclass, parent class*
 - Unterklasse: *abgeleitete Klasse, derived class, child class*

```
class Oberklasse {  
    int x = 0 ;  
    public int getX() {  
        return this.x ;  
    }  
}  
  
class Unterklasse extends Oberklasse {  
    public static void main(String[] a){  
        Unterklasse u = new Unterklasse() ;  
        System.out.println( u.getX() ) ;  
    }  
}
```



OOP: Bedeutung von Vererbung

- Unterklasse besitzt **alle** Attribute/Methoden von Oberklasse
- Ausnahme: Konstruktoren werden nicht vererbt (später)
- Unterklasse kann **zusätzliche** Attribute/Methoden definieren
- Unterklasse kann existierende Methoden durch eigene **ersetzen** (Synonyme: *überschreiben*, *override*)

```
class Oberklasse {  
    int x = 0 ;  
    public int getX() {  
        return this.x ;  
    }  
}  
  
class Unterklasse extends Oberklasse {  
    public static void main(String[] a){  
        Unterklasse u = new Unterklasse() ;  
        System.out.println( u.getX() ) ;  
    }  
}
```



OOP: Bedeutung von Vererbung

- Unterklasse hat alles was Oberklasse auch hat
- plus: evtl. zusätzliche Attribute und Methoden
- aber: überschriebene Methoden können natürlich andere Dinge tun
- **Unterklasse kann als Spezialisierung der Oberklasse angesehen werden**
- (später: Unterklasse kann stets anstelle von Oberklasse benutzt werden)



Vererbung: Überschreiben von Methoden

- Überschreiben/Ersetzen einer Methode in einer Unterklasse
 - Name der Methode, Rückgabetyp und Parameter müssen **exakt** übereinstimmen
 - **Annotation** `@Override` erlaubt dies zur Compile-Zeit zu prüfen
 - ursprüngliche Methode in der Unterklasse weiter verfügbar durch `super.methodenname(..)`



Demo

- Einfache Vererbung
- Überschreiben von Methoden
- @Override und wofür es gut ist



Cut: Q&A



Konstruktoren



Konstruktoren

- Spezielle Methoden die bei der Instanziierung einer Klasse aufgerufen werden **müssen**
- Ziel: Initialisierung von Attributen
- Eine Klasse kann mehrere Konstruktoren haben (verschieden Parameter)



Kap. 5.8.6

Konstrukturen

- Terminologie:
 - **Standardkonstruktor** ist ein Konstruktor ohne Parameter
 - **parametrisierter Konstruktor** ist ein Konstruktor mit Parametern
 - Parameter werden ggf. bei der Instanziierung übergeben

```
String standard = new String() ;  
String parametrisiert = new String("xx") ;
```



Kap. 5.8.6

Konstrukturen

- Falls eine Klasse **keinen eigenen Konstruktor** definiert:
 - Standardkonstruktor wird vom Compiler automatisch erzeugt & hinzugefügt
 - dieser automatische Standardkonstruktor tut nichts, außer den Standardkonstruktor der Oberklasse aufzurufen!
- falls **eigener Konstruktor** definiert (parametrisiert oder Standard):
kein automatisch erzeugter Standardkonstruktor!

Kap. 5.8.6

Konstrukturen

- Beispiel:
 - Deklaration von Klasse1
 - kein eigener Konstruktor definiert
 - Instanziierung mit Standard-konstruktor

```
class Klasse1 {  
    int x = 0 ;  
  
    public Klasse1() {  
        this.x = 0;  
    }  
  
    public int getX() {  
        return this.x ;  
    }  
}
```

```
Klasse1 o = new Klasse1() ;
```

OK ?



Konstrukturen

- Beispiel:
 - Deklaration von Klasse1
 - kein eigener Konstruktor definiert
 - Instanziierung mit Standard-konstruktor

```
class Klasse1 {  
    int x = 0 ;  
  
    public Klasse1() {  
        this.x = 0;  
    }  
  
    public int getX() {  
        return this.x ;  
    }  
}
```

```
Klasse1 o = new Klasse1() ;
```



ja, Standardkonstruktor
existiert!



Kap. 5.8.6

Konstrukturen

- Beispiel:
 - Deklaration von Klasse2
 - kein eigener Konstruktor definiert
 - Instanziierung mit Standard-konstruktor

```
class Klasse2 {  
    int x = 0 ;  
  
    public int getX() {  
        return this.x ;  
    }  
}
```

```
Klasse2 o = new Klasse2() ;
```

OK ?



Kap. 5.8.6

Konstrukturen

- Beispiel:
 - Deklaration von Klasse2
 - kein eigener Konstruktor definiert
 - Instanziierung mit Standard-konstruktor

```
class Klasse2 {  
    int x = 0 ;  
  
    public int getX() {  
        return this.x ;  
    }  
}
```

```
Klasse2 o = new Klasse2() ;
```



ja, Standardkonstruktor
automatisch erzeugt!



Kap. 5.8.6

Konstrukturen

- Beispiel:
 - Deklaration von Klasse3
 - jetzt: parametrisierter Konstruktor definiert
 - Instanziierung mit Standard-konstruktor

```
class Klasse3 {  
    int x = 0 ;  
  
    public Klasse3(int x) {  
        this.x = x;  
    }  
  
    public int getX() {  
        return this.x ;  
    }  
}
```

```
Klasse3 o = new Klasse3() ;
```

OK ■



Kap. 5.8.6

Konstrukturen

- Beispiel:
 - Deklaration von Klasse3
 - jetzt: parametrisierter Konstruktor definiert
 - Instanziierung mit Standard-konstruktor

```
Klasse3 o = new Klasse3() ;
```

```
class Klasse3 {  
    int x = 0 ;  
  
    public Klasse3(int x) {  
        this.x = x;  
    }  
  
    public int getX() {  
        return this.x ;  
    }  
}
```

nicht ok, weil kein Standardkonstruktor erzeugt wurde!



Konstrukturen & Vererbung

- Vererbungsmechanismus gilt NICHT für Konstruktoren!
 - jede Unterklasse muss eigene Konstruktoren definieren
 - falls das nicht passiert, wird ein Standardkonstruktor erzeugt der den Std.K. der Oberklasse aufruft
 - ein Konstruktor der Unterklasse muss als erste Aktion einen Konstruktor der Oberklasse aufrufen: mit `super()` oder `super(param1, param2, ...)`
 - falls das nicht getan wird, wird automatisch der Standardkonstruktor der Oberklasse aufgerufen (Fehler falls nicht-existent)



Konstrukturen & Vererbung

- Beispiele:
`KonstruktorTest.java` im E-Learning
- Live-Demo: potentielle Probleme mit
Konstrukturen im Zusammenhang mit
Vererbung



Q&A



Die Klasse Object



Kap. 5.9



Die Klasse `Object`

- Alle Klassen in Java erben von `Object`
 - falls keine Oberklasse → automatisch `Object`



Die Klasse Object

- Alle Klassen in Java erben von Object
 - falls keine Oberklasse → automatisch Object

```
class KeineOberklasse {  
  
    public void methode1() {  
    }  
  
}
```



Die Klasse Object

- Alle Klassen in Java erben von Object
 - falls keine Oberklasse → automatisch Object

```
class KeineOberklasse extends Object {  
  
    public void methode1 () {  
        }  
}
```

äquivalent!!



Kap. 5.9

Die Klasse `Object`

- Alle Klassen in Java erben von `Object`
- Folglich: jede Klasse besitzt alle Methoden von `Object`, z.B.
 - `equals()`
 - `toString()`
 - `clone()`
 - `hashCode()`
 - ...



Demo

- Benutzung von Methoden die von implizit von `Object` geerbt wurden



Q&A



Kap. 5.9



Das Substitutionsprinzip



Das Substitutionsprinzip

- Eine Unterklasse ist Spezialfall ihrer Oberklasse: kann **formal** alles was die Oberklasse auch kann (und evtl. mehr)
 - da, wo Referenz auf Oberklasse erwartet wird, kann auch Referenz auf Unterklasse stehen
 - allerdings: über die Referenz auf Oberklasse dürfen nur Methoden aus Oberklasse aufgerufen werden (**WARUM?**)
 - insbesondere: da wo Referenz auf `Object` erwartet wird, kann jede Klasse stehen (aber: nur Methoden aus `Object` verfügbar)



Kap. 5.9

Das Substitutionsprinzip

```
class Oberklasse {
    public void printSomething() {
        System.out.println("Oberklasse");
    }
}

class Unterklasse extends Oberklasse {
    @Override
    public void printSomething() {
        System.out.println("Unterklasse");
    }

    void printSomethingElse() {
        System.out.println("ZUSÄTZLICH");
    }
}

public class Main {
    public static void main(String[] args) {
        Oberklasse ref = new Unterklasse();
        ref.printSomething();
        ref.printSomethingElse();
    }
}
```

Legal?



Kap. 5.9

Das Substitutionsprinzip

```
class Oberklasse {
    public void printSomething() {
        System.out.println("Oberklasse");
    }
}

class Unterklasse extends Oberklasse {
    @Override
    public void printSomething() {
        System.out.println("Unterklasse");
    }

    void printSomethingElse() {
        System.out.println("ZUSÄTZLICH");
    }
}

public class Main {
    public static void main(String[] args) {
        Oberklasse ref = new Unterklasse(); ✓
        ref.printSomething(); ✓
        ref.printSomethingElse(); ⚡
    }
}
```



Code-Demo



CUT: Q&A



Polymorphie



Polymorphie

```
Object ref = new String("Ich bin die Unterklasse") ;  
System.out.println("Der Hash Code ist " + ref.hashCode()) ;
```

- Wenn eine Referenz auf `Object` vorliegt...
- ... diese allerdings auf ein Objekt vom Typ `String` zeigt...
- ... welches die Methode `hashCode()` überschrieben hat...
- ... wie weiß der Compiler (zur Compile-Zeit), welche Methode aufgerufen werden muss?
 - `Object.hashCode()`
 - `String.hashCode()`



Polymorphie

```
Object ref = new String("String") ;  
System.out.println("toString() liefert " + ref.toString()) ;
```

- Wenn eine Referenz auf `Object` vorliegt...
- ... diese allerdings auf ein Objekt vom Typ `String` zeigt...
- ... welches die Methode `toString()` überschrieben hat...
- ... wie weiss der Compiler (zur Compile-Zeit), welche Methode aufgerufen werden muss?
 - `Object.toString()`
 - `String.toString()`
- **kann er nicht wissen!! Demo!**



Kap. 5.11

Polymorphie

- Wenn eine Referenz auf Oberklasse vorliegt...
- ... diese allerdings auf ein Objekt vom Typ Unterklasse zeigt...
- ... welches die Methode `printSomething()` überschrieben hat...
- ... wie weiss der Compiler (zur Compile-Zeit), welche Methode aufgerufen werden muss?
 - `Oberklasse.printSomething()`
 - `Unterklasse.printSomething()`
- kann er nicht wissen!!

```
class Oberklasse {  
    public void printSomething() {  
        System.out.println("Oberklasse");  
    }  
}  
  
class Unterklasse extends Oberklasse {  
    @Override  
    public void printSomething() {  
        System.out.println("Unterklasse");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Oberklasse ref = new Unterklasse();  
        ref.printSomething();  
    }  
}
```

Ausgabe?



Kap. 5.11



Polymorphie

- Wenn eine Referenz auf Oberklasse vorliegt...
- ... diese allerdings auf ein Objekt vom Typ Unterklasse zeigt...
- ... welches die Methode `printSomething()` überschrieben hat...
- ... wie weiss der Compiler (zur Compile-Zeit), welche Methode aufgerufen werden muss?
 - `Oberklasse.printSomething()`
 - `Unterklasse.printSomething()`
- Lösung: JVM verifiziert zur **Laufzeit**, auf welchen Typ eine Referenz **wirklich** zeigt
 - hier: `Unterklasse`
 - JVM ruft `Unterklasse.printSomething()` auf

```
class Oberklasse {  
    public void printSomething() {  
        System.out.println("Oberklasse");  
    }  
}  
  
class Unterklasse extends Oberklasse {  
    @Override  
    public void printSomething() {  
        System.out.println("Unterklasse");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Oberklasse ref = new Unterklasse();  
        ref.printSomething();  
    }  
}
```

Unterklasse



Formal: Polymorphie

- Bei Methodenaufrufen weiß der Compiler aufgrund des Substitutionsprinzips nicht, auf welche Instanz eine Referenz zeigt (Oberklasse oder abgeleitete Klassen)
- Bei Methodenaufrufen wird zur Laufzeit geprüft, auf was eine Referenz tatsächlich zeigt
- Es wird immer die Methode der Klasse aufgerufen, auf die die Referenz **jetzt gerade** zeigt
- Folge: selbe Referenz kann bei Methodenaufrufen unterschiedliches Verhalten haben (Polymorphie)



Beispiele für Typsubstitution und Polymorphie aus dem Projekt

```
class SpaceInvadersGame extends GameLoop {  
    // ...  
    public static void main(String[] args) {  
        GameLoop game = new SpaceInvadersGame();  
        game.runGame(args);  
    }  
}
```



Cut: Q&A



Zugriffsrechte & Vererbung



Kap. 5.2

Zugriffsrechte und Vererbung

- Betrifft hpts. Methoden und Attribute
- Bisher kennen wir die Zugriffsrechte
 - `public`: jeder darf zugreifen
 - `package-public`: nur Klassen des eigenen Packages können zugreifen
 - für Vererbung ist dies nicht ausreichend!



Zugriffsrechte und Vererbung

- Betrifft hpts. Methoden und Attribute
- Bisher kennen wir die Zugriffsrechte
 - `public`: jeder darf zugreifen
 - `package-public`: nur Klassen des eigenen Packages können zugreifen
 - für Vererbung ist dies nicht ausreichend!
warum?



Zugriffsrechte und Vererbung

- Vier Typen von Zugriffsrechten für Klassen, Methoden und Attribute
 - `public` (Schlüsselwort `public`): alle Klassen dürfen zugreifen
 - `package-public` (kein Schlüsselwort): alle Klassen im selben Package dürfen zugreifen
 - `protected` (Schlüsselwort `protected`): Unterklassen dürfen zugreifen
 - `private` (Schlüsselwort `private`): nur eigene Klasse darf zugreifen



Motivation für Vererbung

- Copy&Paste vermeiden,
Wiederverwendung gemeinsamen Codes!
- Baukastensystem durch Überschreiben
- Copy&Paste vermeiden!



Motivation für Vererbung

- Copy&Paste vermeiden,
Wiederverwendung gemeinsamen Codes!
- Baukastensystem durch Überschreiben
- Copy&Paste vermeiden!
- Copy&Paste vermeiden!



Motivation für Vererbung

- Copy&Paste vermeiden,
Wiederverwendung gemeinsamen Codes!
- Baukastensystem durch Überschreiben
- Copy&Paste vermeiden!
- Copy&Paste vermeiden!
- Copy&Paste vermeiden!