



Erweitern und Festigen der erlernten Konzepte

03. Februar 2025

Plan für die Woche

Montag

- Wiederholung
Encapsulation,
Methods,
Konstruktoren

Dienstag

- **OOP-Konzepte:**
 - Abstrakte Klassen vs.
Interfaces
 - Vertiefung Interfaces

Mittwoch

- **OOP-Konzepte:**
 - Vertiefung Vererbung
und Polymorphismus
 - Super vs this
 - Overwritten/Overloaded
Methods
 - Casting

Donnerstag

- Operatoren, Bedingungen,
Ternary
- Parentheses, precedence
- switch

Freitag

- Arrays (1D, 2D)
- Loops, loops, loops

Plan für heute

- Wiederholung und Vertiefung zur Vererbung
- Methodenüberladung und Methodenüberschreibung
- super vs this
- Polymorphismus
- casting

Konzepte der OOP

- steht für objektorientierte Programmierung
- Konzepte:
 - Kapselung (encapsulation)
 - Abstraktion (abstraction)
 - Vererbung (inheritance)
 - Polymorphismus (polymorphism)

Konzepte der OOP

- steht für objektorientierte Programmierung
- Konzepte:
 - Kapselung (encapsulation)
 - Abstraktion (abstraction)
 - **Vererbung (inheritance)**
 - **Polymorphismus (polymorphism)**

Vertiefung Vererbung

FESTIGEN UND ERWEITERN DES WISSENS

Grundlagen

- Sie ermöglicht es einer Klasse, die Eigenschaften und Methoden einer anderen Klasse zu erben
- Hauptzwecke/Benefits der Vererbung:
 - Wiederverwendbarkeit des Codes
 - Hierarchische Struktur
 - Methodenüberschreibung
 - Polymorphismus

Wiederverwendbarkeit des Codes

- alle Variablen und Methoden der Basisklasse werden vererbt, solange sie **nicht private** sind
- in unserem Beispiel hat Abteilungsleiter alle Eigenschaften vom Mitarbeiter

```
public class Mitarbeiter {  
    public String name, vorname;  
    public int personalNummer;  
    public double gehalt;  
}
```

```
public class Abteilungsleiter extends Mitarbeiter {  
    private Mitarbeiter[] untergebene;  
}
```

Schlüsselwort: extends

Was kann wiederverwendet werden?

- private:

- Variablen und Methoden sind niemals sichtbar, auch nicht bei Vererbung. Sie sind nur innerhalb der eigenen Klasse sichtbar

- package-private:

- Variablen oder Methoden sind bei der Vererbung nur sichtbar, wenn die vererbende und erbende Klasse im selben Package liegen

- protected:

- Variablen oder Methoden sind im gleichen Package immer sichtbar
- Wenn die erbende Klasse in einem anderen Package liegt, bleiben protected-Variablen und –Methoden trotzdem sichtbar

- public:

- Variablen oder Methoden sind immer sichtbar, unabhängig davon, ob sie vererbt werden oder in welchem Package sie sich befinden.

Access-Modifier wichtig!

Visibility	Public	Protected	Package-Private	Private
Innerhalb der gleichen Klasse	OK	Ok	Ok	Ok
Innerhalb desselben Packages	Ok	Ok	Ok	X
In einer Subklasse innerhalb desselben Packages	Ok	Ok	Ok	X
Innerhalb einer Subklasse außerhalb desselben Packages	Ok	OK, mittels Vererbung. Auf Referenzvariable achten!	X	X
Innerhalb keiner Subklasse in einem anderen Package	OK	X	X	X

-> überall

-> Nur innerhalb
gleichen packages

-> Nur innerhalb Klasse

Access-Modifier wichtig!

Visibility	Public		Private
Innerhalb der gleichen Klasse		<pre> package p1; import p1.protectedBsp.ProtectedBsp; class MyString extends ProtectedBsp { public static void main(String[] args) { MyString m = new MyString(); System.out.println(m.variable); ProtectedBsp p = new MyString(); //System.out.println(p.variable); } } </pre>	Ok
Innerhalb desselben Packages			X
In einer Subklasse innerhalb desselben Packages			X
Innerhalb einer Subklasse außerhalb desselben Packages			X
Innerhalb keiner Subklasse in einem anderen Package			X
-> überall		ages	-> Nur innerhalb Klasse

Was kann man wiederverwendet werden?

Elements of Types	Classes	Interfaces
Instance variables	Ok	X
Static variables	Ok	Nur Konstanten
Abstract methods	Ok	Ok
Instance Methods	Ok	Default-Methoden
Static Methods	Ok	Zugreifbar
Constructors	X	X
Initialization blocks	X	X

Prinzipiell: Bei Klassen alles vererbbar außer Konstruktoren und Init-Blocks

Wiederverwendung Konstruktor mittels super()

- die Basisklasse kann mittels super() Methoden und Konstruktoren wiederverwenden/darauf zugreifen

Konstruktor mit Initialisierungen

super() ruft den Konstruktor der Basisklasse auf

```
class Mitarbeiter {  
    public String name, vorname;  
    public int personalNummer;  
    public double gehalt;  
  
    public Mitarbeiter(String name, String vorname){  
        this.name = name;  
        this.vorname = vorname;  
    }  
}  
  
class Abteilungsleiter extends Mitarbeiter {  
    private Mitarbeiter[] untergebene;  
  
    public Abteilungsleiter(String name, String vorname){  
        super(name, vorname);  
    }  
}
```

super() - Aufruf

- der Compiler fügt *super();* in die erste Zeile des Konstruktorkörpers ein
- damit ruft er den Konstruktor der Basisklasse auf

super();
Konstruktoraufruf der
Objekt-Klasse

```
class Mitarbeiter {  
    public String name, vorname;  
    public int personalNummer;  
    public double gehalt;  
  
    public Mitarbeiter(String name, String vorname){  
        this.name = name;  
        this.vorname = vorname;  
    }  
}
```

super() - Aufruf

- wenn es nur einen Konstruktor in der Basisklasse mit Parameter gibt, muss die Subklasse die entsprechenden Parametern dem super() übergeben

Konstruktor mit Parameter

super() muss die gleichen
Parameter übergeben
bekommen.
Sonst: Compilerfehler

```
class Mitarbeiter {  
    public String name, vorname;  
    public int personalNummer;  
    public double gehalt;  
  
    public Mitarbeiter(String name, String vorname){  
        this.name = name;  
        this.vorname = vorname;  
    }  
}  
  
class Abteilungsleiter extends Mitarbeiter {  
    private Mitarbeiter[] untergebene;  
  
    public Abteilungsleiter(String name, String vorname){  
        super(name, vorname);  
    }  
}
```

super() - Aufruf

- wenn es einen weiteren Konstruktor ohne Parameter gibt
wird mittels super() der ohne Parameter gewählt

Konstruktor ohne Parameter

super(); wird vom Compiler
hinzugefügt
Konstruktor ohne Parameter wird
genutzt

```
class Mitarbeiter {  
    public String name, vorname;  
    public int personalNummer;  
    public double gehalt;  
  
    public Mitarbeiter(String name, String vorname){  
        this.name = name;  
        this.vorname = vorname;  
    }  
    public Mitarbeiter(){  
    }  
}  
  
class Abteilungsleiter extends Mitarbeiter {  
    private Mitarbeiter[] untergebene;  
  
    public Abteilungsleiter(String name, String vorname){  
    }  
}
```


Vergleich zu this()

- this() ruft einen anderen Konstruktor in derselben Klasse auf
- übergebene Werte müssen zu der Parameterliste eines Konstruktors passen

Ruft den anderen Mitarbeiter() –
Konstruktor auf

```
class Mitarbeiter {  
    public String name, vorname;  
    public int personalNumber;  
    public double gehalt;  
  
    public Mitarbeiter(String name, String vorname){  
        this.name = name;  
        this.vorname = vorname;  
    }  
    public Mitarbeiter(){  
        this(„Herbert“, „Herbertius“);  
    }  
}  
  
class Abteilungsleiter extends Mitarbeiter {  
    private Mitarbeiter[] untergebene;  
  
    public Abteilungsleiter(String name, String vorname){  
    }  
}
```

super() und this()

- Beide müssen die erste Anweisung im Konstruktor sein
- können nicht zusammen in einem Konstruktor stehen

kein Super()-Aufruf mehr

```
class Mitarbeiter {  
    public String name, vorname;  
    public int personalNumber;  
    public double gehalt;  
  
    public Mitarbeiter(String name, String vorname){  
        this.name = name;  
        this.vorname = vorname;  
    }  
    public Mitarbeiter(){  
        this(„Herbert“, „Herbertius“);  
    }  
}  
  
class Abteilungsleiter extends Mitarbeiter {  
    private Mitarbeiter[] untergebene;  
  
    public Abteilungsleiter(String name, String vorname){  
    }  
}
```

Wiederverwendung Methoden mittels super

- die Basisklasse kann auch einfach erweitert werden, indem Methoden wiederverwendet werden
- Szenario:
 - Ein Abteilungsleiter kriegt einen Bonus von 0.5% auf seinen Gehalt

Funktioniert auch mit Variablen

-
Auf Access-Modifier achten!

Holt sich die Methode der super-Klasse & ändert sie

```
class Mitarbeiter {  
    public String name, vorname;  
    public int personalNumber;  
    public double gehalt;  
  
    public double getGehalt() {  
        return gehalt;  
    }  
}  
  
class Abteilungsleiter extends Mitarbeiter {  
    private Mitarbeiter[] untergebene;  
  
    public double getGehalt() {  
        return super.getGehalt() * 1.005;  
    }  
}
```

Vererbung – Grundlagen

Erstelle eine Klasse **Mitarbeiter**

- private Variablen: name:String, vorname:String, personalNummer:int, gehalt:double
- 1. Konstruktor: **Mitarbeiter**(String name, String vorname)
- 2. Konstruktor: **Mitarbeiter**(String name, String vorname, int personalNummer, double gehalt)
- Der 2. Konstruktor soll den 1. Konstruktor zur Initialisierung von name und vorname nutzen
- Public Methode: *getGehalt()*

Erstelle eine Klasse **Abteilungsleiter**, welche vom Mitarbeiter erbt

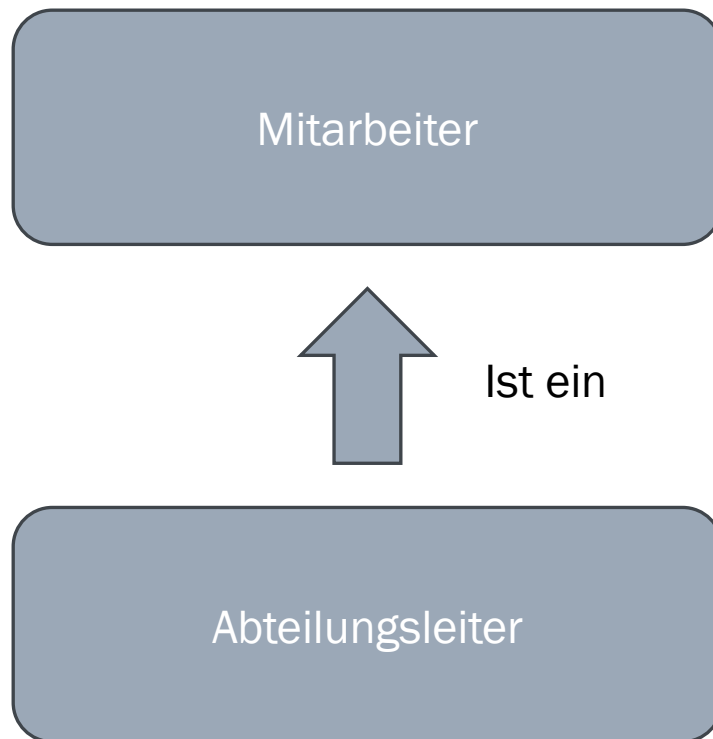
- Konstruktor: **Abteilungsleiter**(String name, String vorname, int personalNummer, double gehalt)
- Der Konstruktor soll den Konstruktor der super-Klasse nutzen
- Public Methode: *getGehalt()*, aber er kriegt 2% Aufschlag zu dem Gehalt eines Mitarbeiters.
- Public Methode: *printDetails()*, gebe alle Daten eines Abteilungsmitarbeiters (name, voname, ...) aus. Musst du was an der Mitarbeiterklasse anpassen?



Grundlagen

- Sie ermöglicht es einer Klasse, die Eigenschaften und Methoden einer anderen Klasse zu erben
- Hauptzwecke/Benefits der Vererbung:
 - Wiederverwendbarkeit des Codes ✓
 - Hierarchische Struktur
 - Methodenüberschreibung
 - Polymorphismus

Hierarchische Struktur



Ein Abteilungsleiter ist ein Mitarbeiter

```
Mitarbeiter mitarbeiter = new Abteilungsleiter();
```

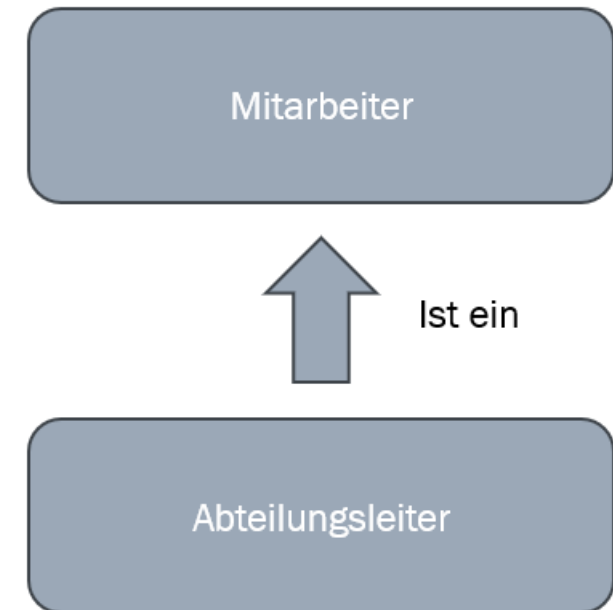
Ein Mitarbeiter ist nicht zwingend ein Abteilungsleiter!

```
Abteilungsleiter abteilungsleiter = new Mitarbeiter();
```

Compilerfehler

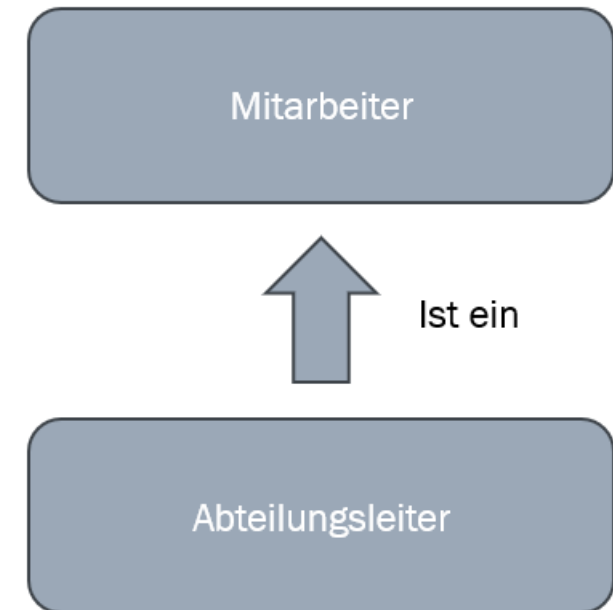
Geht das hierarchisch?

```
class Mitarbeiter {  
  
}  
  
class Abteilungsleiter extends Mitarbeiter {  
  
    public static void main(String[] args) {  
        Abteilungsleiter a = new Abteilungsleiter(); //line 1  
  
        Abteilungsleiter a1 = new Mitarbeiter(); //line 2  
        Mitarbeiter m = new Abteilungsleiter(); //line 3  
        Mitarbeiter m1 = new Mitarbeiter(); //line 3  
  
    }  
}
```



Geht das hierarchisch?

```
class Mitarbeiter {  
  
}  
  
class Abteilungsleiter extends Mitarbeiter {  
  
    public static void main(String[] args) {  
        Abteilungsleiter a = new Abteilungsleiter(); //line 1  
  
        Abteilungsleiter a1 = new Mitarbeiter(); //line 2 NEIN!  
        Mitarbeiter m = new Abteilungsleiter(); //line 3  
        Mitarbeiter m1 = new Mitarbeiter(); //line 3  
    }  
}
```



Grundlagen

- Sie ermöglicht es einer Klasse, die Eigenschaften und Methoden einer anderen Klasse zu erben
- Hauptzwecke/Benefits der Vererbung:
 - Wiederverwendbarkeit des Codes ✓
 - Hierarchische Struktur ✓
 - Methodenüberschreibung
 - Polymorphismus

Methodenüberladung

- Methoden können bei Vererbung auch überladen werden

```
class Mitarbeiter {  
    public void arbeitet(){  
        System.out.println("Ich arbeite gerade");  
    }  
}  
  
class Abteilungsleiter extends Mitarbeiter {  
  
    public void arbeitet(String s){ //Überladung der arbeitet()-Methode  
        System.out.printf("Ich arbeite gerade an %s", s);  
    }  
  
    public static void main(String[] args) {  
        Abteilungsleiter a = new Abteilungsleiter();  
        a.arbeitet();  
        a.arbeitet("Stacking");  
    }  
}
```

Methodenüberschreibung @Override

- Methoden, die vererbt werden, können IMMER überschrieben werden
 - Außer, wenn die Methode final oder static ist
 - Methodenkopf muss beibehalten werden, sonst überschrieben
- dient dazu, Funktionalität anzupassen

```
class Mitarbeiter {  
    public void arbeitet(){  
        System.out.println("Ich arbeite einfach");  
    }  
}  
  
class Abteilungsleiter extends Mitarbeiter {  
    private Mitarbeiter[] untergebene;  
  
    public void arbeitet(){ // @Override  
        System.out.println("Ich leite eine Abteilung");  
    }  
  
    public static void main(String[] args) {  
        Abteilungsleiter a = new Abteilungsleiter();  
        a.arbeitet();  
    }  
}
```

Methodenüberschreibung @Override

- zu beachten:

- Überschriebene Methoden dürfen nicht restriktiver sein!
 - Z.B. wenn Methode protected ist, darf sie nicht plötzlich private sein
 - In die andere Richtung ist ok! (protected -> public)

Restriktiver!

```
class Mitarbeiter {  
    protected void arbeitet(){  
        System.out.println("Ich arbeite einfach");  
    }  
}  
  
class Abteilungsleiter extends Mitarbeiter {  
    private Mitarbeiter[] untergebene;  
  
    private void arbeitet(){ //CompilerFehler!  
        System.out.println("Ich leite eine Abteilung");  
    }  
  
    public static void main(String[] args) {  
        Abteilungsleiter a = new Abteilungsleiter();  
        a.arbeitet();  
    }  
}
```



Methodenüberschreibung @Override

- zu beachten:

- Überschriebene Methoden dürfen nicht restriktiver sein!
 - Z.B. wenn Methode protected ist, darf sie nicht plötzlich private sein
 - In die andere Richtung ist ok! (protected -> public)

Nicht
restriktiver!

```
class Mitarbeiter {  
    protected void arbeitet(){  
        System.out.println("Ich arbeite einfach");  
    }  
}  
  
class Abteilungsleiter extends Mitarbeiter {  
    private Mitarbeiter[] untergebene;  
  
    public void arbeitet(){ //funktioniert!  
        System.out.println("Ich leite eine Abteilung");  
    }  
  
    public static void main(String[] args) {  
        Abteilungsleiter a = new Abteilungsleiter();  
        a.arbeitet();  
    }  
}
```


Methodenüberschreibung @Override

- zu beachten:

- Überschriebene Methoden dürfen nicht restriktiver sein!
 - Z.B. wenn Methode protected ist, darf sie nicht plötzlich private sein
 - In die andere Richtung ist ok! (protected -> public)
- Überschriebene Methoden dürfen nicht plötzlich Exceptions werfen!

Woher kommt
die Exception?

```
class Mitarbeiter {  
    protected void arbeitet(){  
        System.out.println("Ich arbeite einfach");  
    }  
}  
  
class Abteilungsleiter extends Mitarbeiter {  
    private Mitarbeiter[] untergebene;  
  
    public void arbeitet() throws Exception{ //Compilerfehler  
        System.out.println("Ich leite eine Abteilung");  
    }  
  
    public static void main(String[] args) {  
        Abteilungsleiter a = new Abteilungsleiter();  
        a.arbeitet();  
    }  
}
```



Methodenüberschreibung

@Override

- zu beachten:

- Überschriebene Methoden dürfen nicht restriktiver sein!
 - Z.B. wenn Methode protected ist, darf sie nicht plötzlich private sein
 - In die andere Richtung ist ok! (protected -> public)
- Überschriebene Methoden dürfen nicht plötzlich Exceptions werfen!

```
class Mitarbeiter {  
    protected void arbeitet() throws Exception{  
        System.out.println("Ich arbeite einfach");  
    }  
}  
  
class Abteilungsleiter extends Mitarbeiter {  
    private Mitarbeiter[] untergebene;  
  
    public void arbeitet() throws Exception{ //passt  
        System.out.println("Ich leite eine Abteilung");  
    }  
  
    public static void main(String[] args) {  
        Abteilungsleiter a = new Abteilungsleiter();  
        a.arbeitet();  
    }  
}
```

Methodenüberschreibung

@Override

- zu beachten:

- Überschriebene Methoden dürfen nicht restriktiver sein!
 - Z.B. wenn Methode protected ist, darf sie nicht plötzlich private sein
 - In die andere Richtung ist ok! (protected -> public)
- Überschriebene Methoden dürfen nicht plötzlich Exceptions werfen!
 - Überschriebene Methoden müssen nicht unbedingt Exceptions der Methode in der Basisklasse werfen!

```
class Mitarbeiter {  
    protected void arbeitet() throws Exception{  
        System.out.println("Ich arbeite einfach");  
    }  
}  
  
class Abteilungsleiter extends Mitarbeiter {  
    private Mitarbeiter[] untergebene;  
  
    public void arbeitet() { //passt  
        System.out.println("Ich leite eine Abteilung");  
    }  
  
    public static void main(String[] args) {  
        Abteilungsleiter a = new Abteilungsleiter();  
        a.arbeitet();  
    }  
}
```


Methodenüberschreibung

@Override

- zu beachten:

- Überschriebene Methoden dürfen nicht restriktiver sein!
 - Z.B. wenn Methode protected ist, darf sie nicht plötzlich private sein
 - In die andere Richtung ist ok! (protected -> public)
- Überschriebene Methoden dürfen nicht plötzlich Exceptions werfen!
 - Überschriebene Methoden müssen nicht unbedingt Exceptions der Methode in der Basisklasse werfen!
 - Wenn, dann darf eine spezifischere Exception neu deklariert werden

```
class Mitarbeiter {  
    protected void arbeitet() throws Exception {  
        System.out.println("Ich arbeite einfach");  
    }  
}  
  
class Abteilungsleiter extends Mitarbeiter {  
    private Mitarbeiter[] untergebene;  
  
    public void arbeitet() throws IOException { //passt  
        System.out.println("Ich leite eine Abteilung");  
    }  
  
    public static void main(String[] args) {  
        Abteilungsleiter a = new Abteilungsleiter();  
        a.arbeitet();  
    }  
}
```

Methodenüberschreibung

@Override

- zu beachten:

- Überschriebene Methoden dürfen nicht restriktiver sein!
 - Z.B. wenn Methode protected ist, darf sie nicht plötzlich private sein
 - In die andere Richtung ist ok! (protected -> public)
- Überschriebene Methoden dürfen nicht plötzlich Exceptions werfen!
 - Überschriebene Methoden müssen nicht unbedingt Exceptions der Methode in der Basisklasse werfen!
 - Wenn, dann darf eine spezifischere Exception neu deklariert werden
 - RuntimeExceptions dürfen immer geworfen werden

```
class Mitarbeiter {  
    protected void arbeitet() {  
        System.out.println("Ich arbeite einfach");  
    }  
}  
  
class Abteilungsleiter extends Mitarbeiter {  
    private Mitarbeiter[] untergebene;  
  
    public void arbeitet() throws NullPointerException { //passt  
        System.out.println("Ich leite eine Abteilung");  
    }  
  
    public static void main(String[] args) {  
        Abteilungsleiter a = new Abteilungsleiter();  
        a.arbeitet();  
    }  
}
```

Geht das?

```
class BaseClass {  
    public void process(int data) throws FileNotFoundException {  
        System.out.println("Processing int data in BaseClass");  
    }  
}  
  
class SubClass extends BaseClass {  
    public void process(int data) throws IOException {  
        System.out.println("Processing String data in SubClass");  
    }  
}  
  
class Test{  
    public static void main(String[] args) {  
        SubClass c = new SubClass();  
        try {  
            c.process("12");  
        }catch (IOException e){  
  
        }  
    }  
}
```

- zu beachten:

- Überschriebene Methoden dürfen nicht restriktiver sein!
 - Z.B. wenn Methode protected ist, darf sie nicht plötzlich private sein
 - In die andere Richtung ist ok! (protected -> public)
- Überschriebene Methoden dürfen nicht plötzlich Exceptions werfen!
 - Überschriebene Methoden müssen nicht unbedingt Exceptions der Methode in der Basisklasse werfen!
 - Wenn, dann darf eine spezifischere Exception neu deklariert werden
 - RuntimeExceptions dürfen immer geworfen werden

Nein, die überschreibende Methode muss eine spezifischere Exception haben!



Geht das?

```
class BaseClass {  
    public void process(int data) {  
        System.out.println("Processing int data in BaseClass");  
    }  
}  
  
class SubClass extends BaseClass {  
    public void process(int data) throws IOException {  
        System.out.println("Processing String data in SubClass");  
    }  
}  
  
class Test{  
    public static void main(String[] args) {  
        SubClass c = new SubClass();  
        try {  
            c.process("12");  
        }catch (IOException e){  
  
        }  
    }  
}
```

- zu beachten:

- Überschriebene Methoden dürfen nicht restriktiver sein!
 - Z.B. wenn Methode protected ist, darf sie nicht plötzlich private sein
 - In die andere Richtung ist ok! (protected -> public)
- Überschriebene Methoden dürfen nicht plötzlich Exceptions werfen!
 - Überschriebene Methoden müssen nicht unbedingt Exceptions der Methode in der Basisklasse werfen!
 - Wenn, dann darf eine spezifischere Exception neu deklariert werden
 - RuntimeExceptions dürfen immer geworfen werden

Nein, die überschreibende Methode darf nicht plötzlich eine Exception werfen!



Geht das?

```
class BaseClass {  
    public void process() {  
        System.out.println("Processing in BaseClass");  
    }  
}  
  
class SubClass extends BaseClass {  
  
    public void process() throws NullPointerException{  
        System.out.println("Processing in SubClass");  
    }  
}  
  
public class Test{  
    public static void main(String[] args) {  
        SubClass c = new SubClass();  
        c.process();  
    }  
}
```

- zu beachten:

- Überschriebene Methoden dürfen nicht restriktiver sein!
 - Z.B. wenn Methode protected ist, darf sie nicht plötzlich private sein
 - In die andere Richtung ist ok! (protected -> public)
- Überschriebene Methoden dürfen nicht plötzlich Exceptions werfen!
 - Überschriebene Methoden müssen nicht unbedingt Exceptions der Methode in der Basisklasse werfen!
 - Wenn, dann darf eine spezifischere Exception neu deklariert werden
 - RuntimeExceptions dürfen immer geworfen werden

Ja, Runtime Exceptions dürfen immer geworfen werden



Geht das?

```
import java.io.IOException;

class BaseClass {
    public void loadData() throws IOException {
        System.out.println("Loading data in BaseClass");
    }
}

class SubClass extends BaseClass {
    @Override
    public void loadData() throws NullPointerException{
        System.out.println("Loading data in SubClass");
    }
}

public class Test{
    public static void main(String[] args) {
        SubClass c = new SubClass();
        c.loadData();
    }
}
```

- zu beachten:

- Überschriebene Methoden dürfen nicht restriktiver sein!
 - Z.B. wenn Methode protected ist, darf sie nicht plötzlich private sein
 - In die andere Richtung ist ok! (protected -> public)
- Überschriebene Methoden dürfen nicht plötzlich Exceptions werfen!
 - Überschriebene Methoden müssen nicht unbedingt Exceptions der Methode in der Basisklasse werfen!
 - Wenn, dann darf eine spezifischere Exception neu deklariert werden
 - RuntimeExceptions dürfen immer geworfen werden

Ja, Runtime Exceptions dürfen immer geworfen werden.
Überschreibende Methoden müssen nicht zwingend die Exceptions übernehmen



Geht das?

```
import java.io.IOException;

class BaseClass {
    public void process(int data) {
        System.out.println("Processing int data in BaseClass");
    }
}

class SubClass extends BaseClass {
    public void process(String data) throws IOException {
        System.out.println("Processing String data in SubClass");
    }
}

public class Test{
    public static void main(String[] args) {
        BaseClass c = new SubClass();
        c.process(12);
    }
}
```

- zu beachten:

- Überschriebene Methoden dürfen nicht restriktiver sein!
 - Z.B. wenn Methode protected ist, darf sie nicht plötzlich private sein
 - In die andere Richtung ist ok! (protected -> public)
- Überschriebene Methoden dürfen nicht plötzlich Exceptions werfen!
 - Überschriebene Methoden müssen nicht unbedingt Exceptions der Methode in der Basisklasse werfen!
 - Wenn, dann darf eine spezifischere Exception neu deklariert werden
 - RuntimeExceptions dürfen immer geworfen werden

Ja, unsere Methode ist nämlich überladen, nicht überschrieben



Geht das?

```
import java.io.FileNotFoundException;
import java.io.IOException;

class BaseClass {
    public void process() throws IOException {
        System.out.println("Processing int data in BaseClass");
    }
}

class SubClass extends BaseClass {
    public void process() throws FileNotFoundException {
        System.out.println("Processing String data in SubClass");
    }
}

public class Test{
    public static void main(String[] args) {
        BaseClass c = new SubClass();
        try {
            c.process();
        } catch (IOException e) {
            e.getMessage();
        }
    }
}
```

- zu beachten:

- Überschriebene Methoden dürfen nicht restriktiver sein!
 - Z.B. wenn Methode protected ist, darf sie nicht plötzlich private sein
 - In die andere Richtung ist ok! (protected -> public)
- Überschriebene Methoden dürfen nicht plötzlich Exceptions werfen!
 - Überschriebene Methoden müssen nicht unbedingt Exceptions der Methode in der Basisklasse werfen!
 - Wenn, dann darf eine spezifischere Exception neu deklariert werden
 - RuntimeExceptions dürfen immer geworfen werden

Ja, FileNotFoundException ist nämlich spezifischer als IOException



Übung

- Erstelle eine Klasse Rechteck mit den Variablen `breite:double`, `laenge :double`
- Füge einen Konstruktor hinzu, der `breite` und `laenge` setzt
- Erstelle eine Methode `berechneFlaeche()`, die die Fläche des Rechtecks berechnet

- Erstelle eine Klasse Quadrat, die von Rechteck erbt
- Füge einen Konstruktor hinzu, der nur `seite` als Parameter entgegennimmt
- Überschreibe die Methode `berechneFlaeche()`
- Überlade die Methode `berechneFlaeche()`, sodass die Fläche mit neuer Seitenlänge berechnet werden kann.

