



# Erweitern und Festigen der erlernten Konzepte

03. Februar 2025

# Plan für die Woche

---

Montag

- **OOP-Konzepte:**  
Encapsulation
- Methoden-/Überladung
- Var-args
- Konstruktoren-/Überladung

Dienstag

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

Mittwoch

- **OOP-Konzepte:**
  - Wiederholung und Vertiefung Vererbung und Polymorphismus
  - Casting
  - Super

Donnerstag

- Overwritten/Overloaded Methods
- Casting
- Super

Freitag

- Enums
- Programmieren/Erweitern Aufgabe von Montag

# Plan für heute

---

- Kapselung (Encapsulation)
- Wiederholung Methoden und Method-Overloading
- Var- args
- Wiederholung Konstruktoren und Konstruktor-Overloading

# Konzepte der OOP

---

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

# Konzepte der OOP

---

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

# Vertiefung Kapselung

---

FESTIGEN UND ERWEITERN DES WISSENS

BUCHSEITEN VON OCA (1Z0-808: S.88-91)

# Was ist an diesem Code schlecht?

---

```
class Person {  
    public int age;  
}  
  
class Test{  
    public static void main(String[] args) {  
        Person p = new Person();  
        p.age = -10;  
        System.out.println(p.age);  
    }  
}
```



# Was ist an diesem Code schlecht?

Access-Modifier: public

Indirekter Zugriff auf die  
Variable ohne Zustimmung  
der Komponente Person

```
class Person {  
    public int age;  
}  
  
class Test{  
    public static void main(String[] args) {  
        Person p = new Person();  
        p.age = -10;  
        System.out.println(p.age);  
    }  
}
```





# Was ist an diesem Code gut?

---

```
class Person {  
    private int age;  
  
    public void setAge(int a){  
        if(a > 0){  
            this.age = a;  
        }else{  
            System.out.println("Alter muss größer als 0 sein");  
        }  
    }  
  
    public static void main(String[] args) {  
        Person p = new Person();  
        p.setAge(-1); //wird überprüft  
    }  
}
```



# Was ist an diesem Code gut?

Access-Modifier: private

Datenvalidierung

Zugriff mit Zustimmung  
der Komponente

```
class Person {  
    private int age;  
  
    public void setAge(int a){  
        if(a > 0){  
            this.age = a;  
        }else{  
            System.out.println("Alter muss größer als 0 sein");  
        }  
    }  
  
    public static void main(String[] args) {  
        Person p = new Person();  
        p.setAge(-1); //wird überprüft  
    }  
}
```



# Kapselung - Prinzipien

---

- = Encapsulation
- Erlaubt es Komponenten Daten vor anderen Komponenten zu verstecken
  - Genauer: Versteckt die Weise, wie Daten verändert werden
- Verhindert, dass Daten ohne Zustimmung/Wissen der Komponente verändert werden

```
class Person {  
    private int age;  
  
    public void setAge(int a){  
        if(a > 0){  
            this.age = a;  
        }else{  
            System.out.println("Alter muss größer als 0 sein");  
        }  
    }  
  
    public static void main(String[] args) {  
        Person p = new Person();  
        p.setAge(-1); //wird überprüft  
    }  
}
```

# Kapselung - Prinzipien

---

- = Encapsulation
- Erlaubt es Komponenten Daten vor anderen Komponenten zu verstecken
  - Genauer: Versteckt die Weise, wie Daten verändert werden
- Verhindert, dass Daten ohne Zustimmung/Wissen der Komponente verändert werden
- Erlaubt die Validierung von Daten :
  - Werte privater Variablen können durch Setter- und Getter- Methoden validiert werden

```
class Person {  
    private int age;  
  
    public void setAge(int a){  
        if(a > 0){  
            this.age = a;  
        }else{  
            System.out.println("Alter muss größer als 0 sein");  
        }  
    }  
  
    public static void main(String[] args) {  
        Person p = new Person();  
        p.setAge(-1); //wird überprüft  
    }  
}
```

# Kapselung - Prinzipien

---

- bessere Wartbarkeit des Codes
- Warum?:
  - Die Validierung von Daten erfolgt an einer definierten Stelle (Setter-Methode)
  - Änderungen an der Validierung müssen dadurch nur an einer Stelle vorgenommen werden
  - Es ist klar definiert, wo und wie Werte gesetzt werden -> Objektzustand klar
  - Änderungen an der internen Implementierung wirken sich nicht direkt auf andere Klassen aus
  - Ermöglicht zukünftige Erweiterungen

# Kapselung - Zusammenfassung

---

- Erlaubt die Validierung von Daten in den Setter-Methoden
- direkte Änderung von Daten wird verhindert
- Änderungen an der Implementierung können vorgenommen werden, ohne andere Klassen zu beeinflussen
- Geringere Kopplung zu anderen Klassen, da sie sich nicht um die Implementierung kümmern müssen (z.b. Validierung)



Unnötige Setter und Getter vermeiden

# Wofür braucht man Kapselung?

---

- Datenvalidierung (data validation)
- Daten verstecken (data hiding)
- Speicheroptimierung
- Daten abstrakt gestalten
- Threadsicherheit
- erste Sicherheitsmechanismen



# Wofür braucht man Kapselung?

- Datenvalidierung (data validation)
- Daten verstecken (data hiding)
- ~~Speicheroptimierung~~
- ~~Daten abstrakt gestalten~~
- ~~Threadsicherheit~~
- ~~erste Sicherheitsmechanismen~~



Erste Sicherheitsmechanismen sind die  
JVM und der Bytecode





# Wiederholung Methoden

---

FESTIGEN UND ERWEITERN DES WISSENS

BUCHSEITEN VON OCA (1Z0-808: S.111-114, S.61, S.49-50)

# Methoden - Wiederholung

---

- Methoden können einen oder keinen Rückgabewert (void) haben
- Methoden können Parameter entgegennehmen, die dann innerhalb der Methode sichtbar sind

Rückgabewert

```
public class Calculator {
```

```
    public int add(int a, int b) {
```

```
        return a + b;
```

```
    }
```

```
}
```

Parameter

# Methoden – Vorsicht geboten!

---

- Vorsicht bei gleichnamigen Datenfeldern und Parametern!

```
public class Calculator {  
    public double pi = 3.14159;  
  
    public void add(double pi) {  
        pi = pi;  
    }  
}
```

Was passiert in der Methode add?



# Methoden – Vorsicht geboten!

---

- Vorsicht bei gleichnamigen Datenfeldern und Parametern!

```
public class Calculator {  
    public double pi = 3.14159;  
  
    public void add(double pi) {  
        pi = pi;  
    }  
}
```

Der Parameter wird mit sich selbst gleichgesetzt



# Methoden – Vorsicht geboten!

---

- Vorsicht bei gleichnamigen Datenfeldern und Parametern!

```
public class Calculator {  
    public double pi = 3.14159;  
  
    public void add(double pi) {  
        pi = 100;  
    }  
}
```

Was passiert in der Methode add?



# Methoden – Vorsicht geboten!

---

- Vorsicht bei gleichnamigen Datenfeldern und Parametern!

```
public class Calculator {  
    public double pi = 3.14159;  
  
    public void add(double pi) {  
        pi = 100;  
    }  
}
```

Der übergebene Parameter wird auf 100 gesetzt, nicht die Instanzvariable



# Methoden – Vorsicht geboten!

---

- Deswegen hilft das Schlüsselwort this

```
public class Calculator {  
    public double pi = 3.14159;  
  
    public void add(double pi) {  
        this.pi = pi;  
    }  
}
```

Mittels this. Ist dem Compiler klar, dass die Instanzvariable gemeint ist.

# Methoden – Objekte und andere Parameter

- Methoden können sowohl Objekte als auch primitive Datentypen und Wrapper-Klassen als Parameter erhalten
- bei Objekten wird eine Kopie der Referenzvariable übergeben, nicht das eigentliche Objekt
  - Die Referenz zeigt weiterhin auf dasselbe Objekt im Speicher

„Alice“

```
class Person {  
    String name;  
}  
  
public class Test {  
    public static void changeName(Person p) {  
        p.name = "Alice";  
    }  
    public static void main(String[] args) {  
        Person person = new Person();  
        person.name = "Bob";  
        changeName(person);  
        System.out.println(person.name);  
    }  
}
```



# Methoden – Objekte und andere Parameter

---

- bei primitiven Datentypen wird eine Kopie übergeben, das Original bleibt unverändert

```
public class Test {  
    public static void changeValue(int x) {  
        x = 10;  
    }  
    public static void main(String[] args) {  
        int num = 5;  
        changeValue(num);  
        System.out.println(num);  
    }  
}
```

5

# Methoden – Objekte und andere Parameter

---

- bei Wrapper-Klassen ist das Verhalten wie bei Objekten. Jedoch sind Wrapper-Klassen immutable
  - Bei Änderungen wird ein neues Objekt erstellt

```
public class Test {  
    public static void changeValue(Integer x) {  
        x = 10; // Ein neues Integer-Objekt wird erstellt  
    }  
  
    public static void main(String[] args) {  
        Integer num = 5;  
        changeValue(num);  
        System.out.println(num); // Bleibt 5, da Integer  
        immutable ist  
    }  
}
```

# Methoden – Objekte und andere Parameter

---

- Methoden können sowohl Objekte als auch primitive Datentypen und Wrapper-Klassen als Parameter erhalten
- bei Objekten wird eine Kopie der Referenzvariable übergeben, nicht das eigentliche Objekt
  - Die Referenz zeigt weiterhin auf dasselbe Objekt im Speicher
- bei primitiven Datentypen wird eine Kopie übergeben, das Original bleibt unverändert
- bei Wrapper-Klassen ist das Verhalten wie bei Objekten. Jedoch sind Wrapper-Klassen immutable
  - Bei Änderungen wird ein neues Objekt erstellt

# Methoden Wiederholungen

- Übergabewerte: Was ist das Ergebnis?

0  
5  
5

```
class MyObject {  
    int value;  
  
    public void changePrimitive(int value) {  
        value = value;  
    }  
  
    public void changeVariable(int value) {  
        this.value = value;  
    }  
}  
  
public class Main {  
  
    public static void main(String[] args) {  
        MyObject myObj = new MyObject();  
        int num = 5;  
        myObj.changePrimitive(num);  
        System.out.println(myObj.value);  
        System.out.println(num);  
  
        myObj.changeVariable(num);  
        System.out.println(myObj.value);  
    }  
}
```



# Methoden Wiederholungen

- Übergabewerte: Was ist das Ergebnis?

0

5

10

```
class MyObject {
    int value;

    public void changePrimitive(int value) {
        value = value;
    }

    public void changeObject(MyObject obj) {
        obj.value = 10;
    }
}

public class Main {

    public static void main(String[] args) {
        MyObject myObj = new MyObject();
        MyObject myObj2 = new MyObject();
        myObj2.value = 15;
        int num = 5;
        myObj.changePrimitive(num);
        System.out.println(myObj.value);
        System.out.println(num);

        myObj.changeObject(myObj2);
        System.out.println(myObj2.value);
    }
}
```



# Methoden – Var-args - Parameter

---

- = Variable Argument Lists
- dadurch ist es möglich, Methoden zu implementieren, die eine variable Anzahl an Argumenten entgegennehmen kann
  - Intern wird ein Array mit den übergebenen Argumenten erzeugt
- Fällt euch so eine Art Methode ein, die ihr oft nutzt?

# Methoden – Var-args - Parameter

---

- = Variable Argument Lists
- dadurch ist es möglich, Methoden zu implementieren, die eine variable Anzahl an Argumenten entgegennehmen kann
  - Intern wird ein Array mit den übergebenen Argumenten erzeugt
- Die Main-Methode!
  - `public static void main(String[] args)` kann auch als `public static void main(String... args)` geschrieben werden

# Methoden – Var-args - Parameter

---

– Syntax

Datentyp

„Ellipsis“

Name des Arrays

```
public static void main(String... args)
```

The diagram shows the Java method signature `public static void main(String... args)` on a dark background. Three red circles highlight specific parts: the first circle is around `String`, with a red line pointing to the label 'Datentyp'; the second circle is around the ellipsis `...`, with a red line pointing to the label '„Ellipsis“'; and the third circle is around `args`, with a red line pointing to the label 'Name des Arrays'. The keywords `public`, `static`, and `void` are in orange, `main` is in blue, and the parameter part `(String... args)` is in white.



# Methoden – Var-args - Parameter

---

- eine Methode darf höchstens einen var-args Parameter besitzen
- Zusätzlich zu var-args können weitere Parameter vorhanden sein
- Der var-args Parameter muss dann am Ende der Parameterliste stehen

# Methoden – Var-args - Parameter

---

– Welche Methoden sind korrekt?

```
public void test(int... zahlen, String text) {           //m1
}
public void test(String prefix, int... zahlen) {          //m2
}
public void test(String ..., int... zahlen) {            //m3
}
public void test(int... zahlen) {                        //m4
}
public void test(int zahlen...) {                        //m5
}
```

- eine Methode darf höchstens einen var-args Parameter besitzen
- Zusätzlich zu var-args können weitere Parameter vorhanden sein
- Der var-args Parameter muss dann am Ende der Parameterliste stehen



# Methoden – Var-args - Parameter

---

– Welche Methoden sind korrekt?

```
public void test(int... zahlen, String text) {           //m1
}
public void test(String prefix, int... zahlen) {          //m2
}
public void test(String ..., int... zahlen) {            //m3
}
public void test(int... zahlen) {                        //m4
}
public void test(int zahlen...) {                        //m5
}
```

- eine Methode darf höchstens einen var-args Parameter besitzen
- Zusätzlich zu var-args können weitere Parameter vorhanden sein
- Der var-args Parameter muss dann am Ende der Parameterliste stehen

**M1!** -> var-args muss am Ende stehen

M2 -> passt

**M3!** -> nur ein var-args ist erlaubt

M4 -> passt

**M5!** -> die Punkte müssen direkt hinter das int

# Methoden Überladung - Wiederholung

- Methoden mit Argumenten und Rückgabewerten, einschließlich überladener Methoden

Rückgabewert

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
    public int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

Parameter

Überladene Methode:

- Parameterliste MUSS geändert werden, sonst nicht überladen
  - Datentypen oder Anzahl der Parameter ändern
  - Reihenfolge der übergebenen Parameter ändern bewirkt ebenfalls überladen
- Können Rückgabewert ändern
- Können den Access-Modifizier ändern

# Methoden Überladung - Übung

---

– Sind folgende Methoden überladen?

```
public class Calculator {  
  
    public void add(int a, int b) {  
    }  
  
    public int add(int a, int b) {  
        return a + b ;  
    }  
}
```



# Methoden Überladung - Übung

---

- Nein! Die Parameterliste hat sich nicht geändert! Das Ändern des Rückgabewert allein ändert nichts

```
public class Calculator {  
  
    public void add(int a, int b) {  
    }  
  
    public int add(int a, int b) {  
        return a + b ;  
    }  
}
```



# Methoden Überladung - Übung

---

– Sind folgende Methoden überladen?

```
public class Calculator {  
  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    public int add(Integer a, Integer b) {  
        return a + b ;  
    }  
}
```



# Methoden Überladung - Übung

---

- Ja! Integer ist eine Wrapper-Klasse und somit trotzdem ein anderer Datentyp als der primitive Datentyp int.

```
public class Calculator {  
  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    public int add(Integer a, Integer b) {  
        return a + b ;  
    }  
}
```





# Methoden Überladung - Übung

---

– Sind folgende Methoden überladen?

```
public class Calculator {  
  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    private int add(int a, int b) {  
        return a + b ;  
    }  
}
```



# Methoden Überladung - Übung

---

- Nein! Das Ändern des Access-Modifiers überlädt eine Methode nicht! Die Parameterliste ist entscheidend

```
public class Calculator {  
  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    private int add(int a, int b) {  
        return a + b ;  
    }  
}
```



# Methoden Überladung - Übung

---

– Sind folgende Methoden überladen?

```
public class Calculator {  
  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    private int add(int a, int b, int c) {  
        return a + b + c;  
    }  
}
```



# Methoden Überladung - Übung

---

- Ja! Die Parameterliste wurde erfolgreich angepasst

```
public class Calculator {  
  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    private int add(int a, int b, int c) {  
        return a + b + c;  
    }  
}
```



# Methoden Überladung - Exceptions

Rückgabewert

```
public class Calculator {
```

```
    public int add(int a, int b) {
```

```
        return a + b;
```

```
    }
```

```
    public int add(int a, int b, int c) throws Exception {
```

```
        return a + b + c;
```

```
    }
```

```
}
```

Parameter

Überladene Methode:

- Nur möglich wenn mehr Parameter übergeben
- Andere Datentypen der Parameter

Überladene Methode:

- Können neue Exceptions werfen

# Methoden Überladung - Anwendungsbeispiel

- welche Methoden werden gewählt?

```
class Calculator {  
    public int add(int a, int b) {  
        return a + b;  
    }  
    public double add(double a, double b) {  
        return a + b;  
    }  
    public float add(float a, float b) {  
        return a + b;  
    }  
    public Integer add(Integer a, Integer b) {  
        return a + b;  
    }  
    public static void main(String[] args) {  
        Calculator c = new Calculator();  
        c.add(10, 10);  
        c.add(10.0, 11.0);  
        c.add(10.0F, 11.0F);  
    }  
}
```

# Methoden Überladung - Anwendungsbeispiel

- welche Methoden werden gewählt?
    - In diesem Beispiel gibt es keine Methode für den primitiven Datentypen int
    - Es wird die Methode mit dem nächst-größeren Datentypen gewählt
- ➔ Auch bei den anderen Datentypen ähnlich!

Es wird **NICHT** die Methode mit der Wrapper-Klasse gewählt  
Erst, wenn es keine anderen Methoden gibt, die einen größeren Datentypen als Parameter übernehmen

```
class Calculator {  
    public double add(double a, double b) {  
        return a + b;  
    }  
    public float add(float a, float b) {  
        return a + b;  
    }  
    public Integer add(Integer a, Integer b) {  
        return a + b;  
    }  
    public static void main(String[] args) {  
        Calculator c = new Calculator();  
        c.add(10, 10);  
        c.add(10.0, 11.0);  
        c.add(10.0F, 11.0F);  
    }  
}
```

# Methoden Überladung - Übung

---

Was ist die Ausgabe vom folgenden Code?

```
class Calculator {  
  
    public Integer add(Integer a, Integer b) {  
        return a + b;  
    }  
  
    public static void main(String[] args) {  
        Calculator c = new Calculator();  
        c.add(10, 10);  
        c.add(10.0, 11.0);  
        c.add(10.0F, 11.0F);  
    }  
}
```





# Methoden Überladung - Übung

Da wir nur eine Methode haben, dessen Parameter die Integer-Wrapperklasse als Datentyp entgegennimmt, kann die Code-Zeile n1 ausgeführt werden.

Für die Code-Zeilen n2 und n3 gibt es keine passenden Methoden – aus diesem Grund kommt es hier zum Compilerfehler

```
class Calculator {  
    public Integer add(Integer a, Integer b) {  
        return a + b;  
    }  
    public static void main(String[] args) {  
        Calculator c = new Calculator();  
        c.add(10, 10);           //n1  
        c.add(10.0, 11.0);      //n2  
        c.add(10.0F, 11.0F);    //n3  
    }  
}
```



# Statische Methoden – Good To Know

- Statische Methoden und Datenfelder:

Klassenvariable

Methode

Instanzvariablen

können nicht in einem statischen  
Kontext aufgerufen werden

Klassenvariablen können  
jedoch in einem nicht-statischen  
Kontext problemlos aufgerufen werden

```
public class Test {
    public static double aPI = 3.14159;
    public double bPI = 3.14159;

    public static int square(int number) {
        double v = aPI * number;
        double w = bPI * number; //FEHLER!
        return number * number;
    }

    public int square() {
        int number = 10;
        double v = aPI * number;
        double w = bPI * number;
        return number * number;
    }
}
```

# Wiederholung Konstrukturen

---

FESTIGEN UND ERWEITERN DES WISSENS

BUCHSEITEN VON OCA (1Z0-808: S.50-51, S.132-137, 140-145)

# Konstruktor - Grundlagen

---

- Jede Klasse hat einen Konstruktor
- Konstruktor müssen den gleichen Namen haben, wie die Klasse
- Sie haben **KEINEN** Rückgabewert
- Sie können Parameter entgegennehmen
- können **NICHT** statisch, final oder abstract sein
- können privat sein

```
class Test{  
    //legal  
    Test(){}  
    private Test(int a, String... args){}  
    //illegal  
    void Test(){}  
    Test2(){}  
    static Test(){}  
    final Test(){}  
    abstract Test(){}  
}
```

# Konstruktor - Grundlagen

---

- Wenn man keinen explizit implementiert, wird der Compiler automatisch einen erzeugen (default-Konstruktor)
- Sobald ein eigener Konstruktor geschrieben wird, wird der default-Konstruktor überschrieben und wird nicht mehr automatisch erzeugt!

Compilerfehler!  
Konstruktor ohne  
Parameter existiert nicht  
mehr!

```
class Test{  
    Test(int a, int b){}  
  
    public static void main(String[] args) {  
        Test t = new Test();  
    }  
}
```

# Konstrukturen - Überladen

---

- Konstrukturen können beliebig überladen werden, dabei gelten die gleichen Regeln wie bei Methoden

```
class Test{  
    Test(){}  
    Test(int a, int b){}  
    Test(int a, String b, double c){}  
}
```

# Konstruktor - Überladen

---

- Konstruktor können andere Konstruktor aufrufen
  - this() – innerhalb der Klasse
  - super() – Aufruf des Konstruktors der Superklasse (wird später behandelt)

```
class Test{  
    private int a, b;  
  
    Test(){  
        this(10, 100);  
    }  
    Test(int a, int b){  
        this.a = a;  
        this.b = b;  
    }  
    Test(int a, String b, double c){}  
}
```