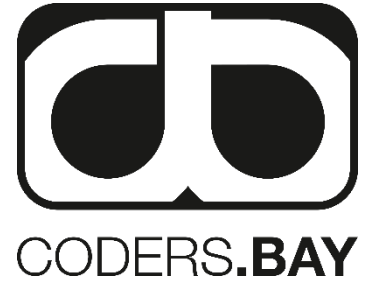


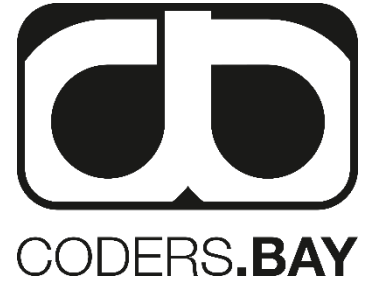
# OBJEKTORIENTIERTE PROGRAMMIERUNG OOP (2)

# INHALT



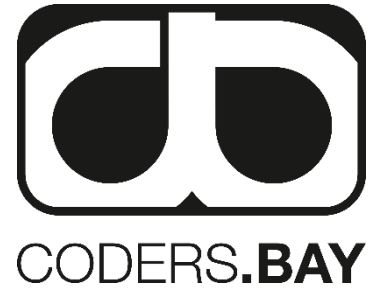
- private
- this
- "getters" & "setters",
  - Beispiel
- 4 fundamentale OOP Konzepte
  - Beispiele

# PRIVATE



- private Variable / Methode
- restriktivster Zugriffsmodifikator
- nicht von außen "sichtbar"
- keine andere Klasse kann darauf DIREKT zugreifen
- Nur die Klasse selbst kann auf solche Variablen / Methoden zugreifen

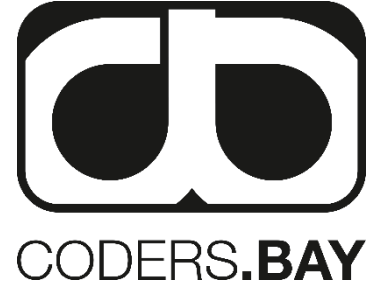
# PRIVATE



```
public class Hauptprogramm {  
  
    public static void main(String[] args) {  
        Person p1 = new Person("Sean", "Parker", 38 );  
  
        //Warum ist das nicht möglich ?  
        //System.out.println(p1.firstname);  
  
        p1.printPersonData();  
    }  
}
```

```
public class Person {  
    //Klassenvariablen (private)  
    private String firstName;  
    private String lastName;  
    private int age;  
  
    //Konstruktor  
    Person(String firstname, String lastname, int ageOfPerson){  
        firstName = firstname;  
        lastName = lastname;  
        age = ageOfPerson;  
    }  
  
    //Methode zur Ausgabe der Personendaten  
    public void printPersonData() {  
        System.out.println("Vorname: " + firstName + "\n");  
        System.out.println("Nachname: " + lastName + "\n");  
        System.out.println("Alter: " + age + "\n");  
    }  
}
```

# THIS

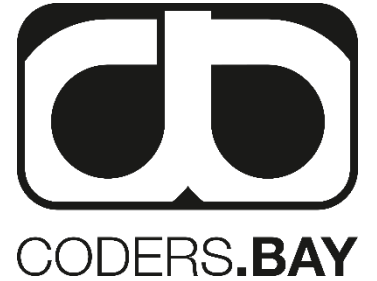


- die `this`-Referenz löst das Problem, wenn Parameter & lokale Variablen denselben Namen haben

```
public class Hauptprogramm {  
    public static void main(String[] args) {  
        Person p = new Person();  
        p.printPerson("Tim", "Berners-Lee", 63);  
    }  
}
```

```
public class Person {  
    String firstName;  
    String lastName;  
    int age;  
  
    Person() {  
        firstName = "TODO";  
        lastName = "TODO";  
        age = 0;  
    }  
  
    Person(String firstName, String lastName, int age) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.age = age;  
    }  
  
    Person(int p_age, String p_lastName, String p_firstName) {  
        firstName = p_firstName;  
        lastName = p_lastName;  
        age = p_age;  
    }  
  
    public void printPerson(String firstName, String lastName, int age) {  
        System.out.println("Vorname: " + firstName);  
        System.out.println("Nachname: " + lastName);  
        System.out.println("Nachname: " + age);  
        System.out.println("THIS-Vorname: " + this.firstName);  
        System.out.println("THIS-Nachname: " + this.lastName);  
        System.out.println("THIS-Nachname: " + this.age);  
    }  
}
```

# 4 FUNDAMENTALE OOP KONZEPTE



- Datenkapselung
  - Getters & Setters
- Vererbung
  - dynamische Bindung
- abstrakte Klassen
- Mehrfachvererbung
  - in Java → Interfaces

# "GETTERS" & "SETTERS"

- Zugriffsmethoden
- für jede Variable eine Schreib- und eine Lesemethode (set, get)
  - Variable → `private`
  - Methode → öffentlich → `public`
- "Getter" Rückgabetyp (`return`) = Parametertyp
- Bei boolean-Attributen darf / soll die Methode `isXXX()` heißen  
→ `isFullAged()` **oder** `isHuman()`

# "GETTERS" & "SETTERS"

- waren die Konsistenz der Datenstruktur
    - z.B.: kann es wichtig sein das eine variable in Abhängigkeit zu einer anderen mit geändert werden muss. Diese Aufgabe wird im setter erledigt
- Datenkapselung (engl. Encapsulation)



# "GETTERS" & "SETTERS"

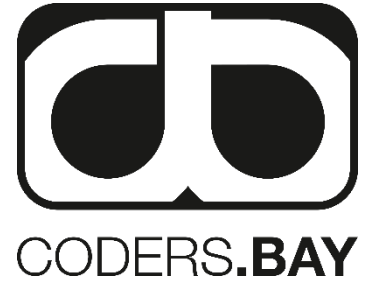
- exakter Zugriff auf (private) Klassenvariable

Eclipse → Person.java → Rechtsklick →  
Source → Generate Getters and Setters

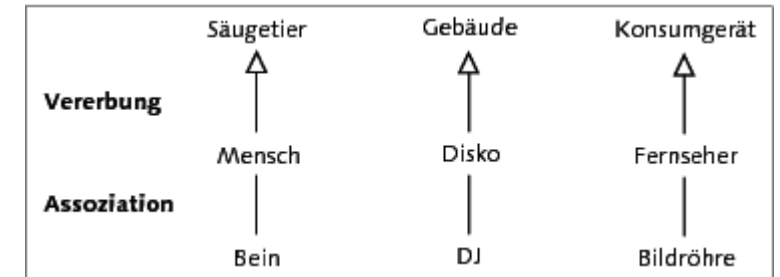
```
public class Hauptprogramm {  
    public static void main(String[] args) {  
        Person p1 = new Person("Steve", "Wozniak", 67);  
        System.out.println("Der Vorname ist: " + p1.getFirstName());  
        System.out.println("Der Nachname von " + p1.getFirstName() + " ist: " + p1.getLastName());  
        p1.setLastName("Jobs");  
        System.out.println("Der Nachname von " + p1.getFirstName() + " ist nun: " + p1.getLastName());  
    }  
}
```

```
public class Person {  
    private String firstName;  
    private String lastName;  
    private int age;  
  
    Person(String firstName, String lastName, int age) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.age = age;  
    }  
  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
  
    public String getLastName() {  
        return lastName;  
    }  
  
    public void setLastName(String lastName) {  
        this.lastName = lastName;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public void setAge(int age) {  
        this.age = age;  
    }  
}
```

# VERERBUNG

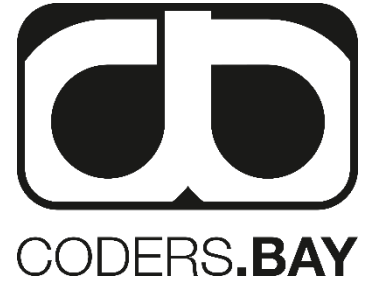


- Unterscheidung Sub- und Superklasse
- Superklasse = Eltern-, Ober-, Basisklasse
- Subklasse = Kind-, Unter-, Childklasse
  - bekommt von Superklasse sämtliche Attribute und Methoden vererbt
  - um eigene Attribute und Methoden erweiterbar
- Schlüsselwort `extends`



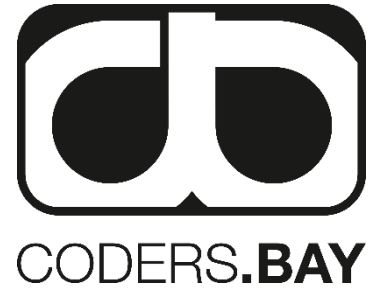
```
Modifikator class Subklasse extends Superklasse {  
    ...  
}
```

# VERERBUNG



- alle `private` Attribute und Methoden werden nicht vererbt
- vererbte Methoden der Superklasse können in Subklasse überladen und “ersetzt” werden
  - Sichtbarkeitsmodifikator darf nicht „privater“ sein als der der Superklasse
    - gilt auch für Attribute
- Subklasse kann explizit Methoden der Superklasse aufrufen
  - Schlüsselwort → `super`
- Superklasse kennt Subklassen nicht

# VERERBUNG

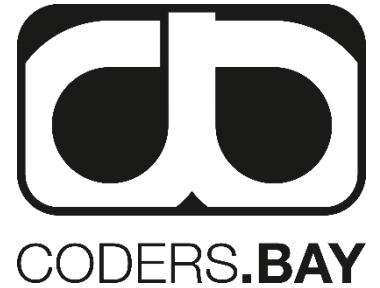


```
/**
 * @author Simon Haidinger
 * @version 1.0
 */
public class Super {

    String s1 = "SUPER";

    public void superMethod() {
        System.out.println("Das ist die Methode superMethod()");
    }
}
```

# VERERBUNG



```
/**
 * @author Simon Haidinger
 * @version 1.0
 */
public class Sub extends Super {
    String s1 = "SUB";
    //zuweisung der gleichnamigen Variable s1, jedoch aus der "super" Klasse
    String s2 = super.s1;

    public void subMethod() {
        System.out.println("Das ist die Methode subMethod()");
        //Aufruf der gleichnamigen Methode superMethod, jedoch aus der "super" Klasse
        super.superMethod();
    }

    public void superMethod() {
        System.out.println("Das ist die Methode superMethod() in Sub.java");
    }
}
```

```
/**
 * @author Simon Haidinger
 * @version 1.0
 */
public class Hauptprogramm {

    public static void main(String[] args) {

        Super super1 = new Super();
        System.out.println(super1.s1);
        super1.superMethod();

        System.out.println();

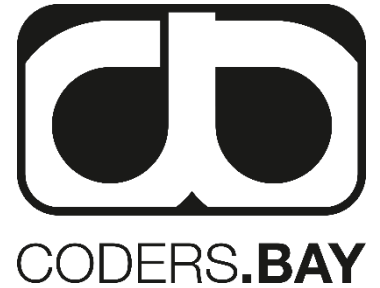
        Sub sub1 = new Sub();
        System.out.println(sub1.s1);
        System.out.println(sub1.s2);
        sub1.subMethod();

        System.out.println();

        Super sub2 = new Sub();
        System.out.println(sub2.s1);
        sub2.superMethod();

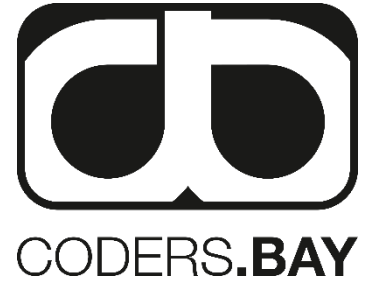
        if( super1 instanceof Super) {
            System.out.println("super1 = Super");
        }
        if( sub1 instanceof Sub) {
            System.out.println("sub1 = Sub");
        }
        if( sub2 instanceof Sub) {
            System.out.println("sub2 = Sub");
        }else {
            System.out.println("sub2 = Super");
        }
    }
}
```

# VERERBUNG



Fällt etwas auf?

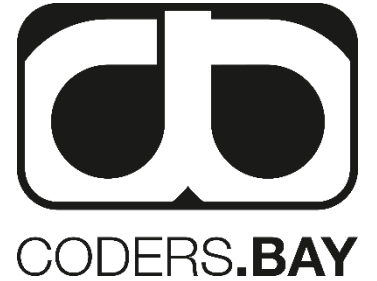
# VERERBUNG



- Unterklasse erbt alle sichtbaren Eigenschaften
  - alle `public`-Elemente und, falls sich Unterklasse und Oberklasse im gleichen Paket befinden, auch die paketsichtbaren Eigenschaften
- Vererbung kann durch `private` eingeschränkt werden
  - keine andere Klasse sieht die Eigenschaften → weder fremde noch Unterklassen
- **protected**
  - `protected`-Eigenschaften werden an alle Unterklassen vererbt
  - Klassen, die sich im gleichen Paket befinden, können alle `protected`-Eigenschaften sehen
  - sind weitere Klassen im gleichen Paket und Eigenschaften `protected`, ist die Sichtbarkeit für sie `public`
  - Für andere Nicht-Unterklassen in anderen Paketen sind die `protected`-Eigenschaften `private`

**public > protected > paketsichtbar > private**

# VERERBUNG



## Dynamische Bindung

- **statischer Typ** → mit dem die Variable deklariert wurde
  - bestimmt welche Elemente der Klasse sichtbar sind
- **dynamischer Typ** → mit dem das Objekt zur Laufzeit referenziert wurde
  - bestimmt welche Methode aufgerufen wird → **dynamische Bindung**

```
//statischer Typ
Person p1;
//dynamischer Typ
p1 = new Mechaniker();
```



# VERERBUNG – DYNAMISCHE BINDUNG

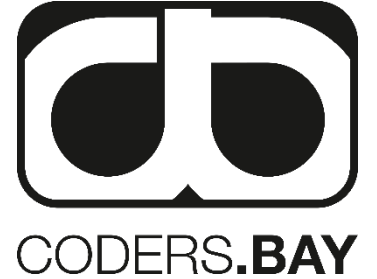


```
/**
 * @author Simon Haidinger
 * @version 1.0
 */
public class Super {

    public String s1 = "SUPER";

    @Override
    public String toString() {
        return String.format( "Super[s1=%s]", s1 );
    }
}
```

# VERERBUNG – DYNAMISCHE BINDUNG



```
/**
 * @author Simon Haidinger
 * @version 1.0
 */
public class Sub extends Super {
    String s1 = "SUB";
    //zuweisung der gleichnamigen Variable s1, jedoch aus der "super" Klasse
    String s2 = super.s1;

    @Override
    public String toString(){
        return String.format( "Sub[s1=%s, s2=%s]", s1, s2 );
    }
}
```

```

/**
 * @author Simon Haidinger
 * @version 1.0
 */
public class Hauptprogramm {

    public static void main(String[] args) {

        Super super1 = new Super();
        System.out.println("super1: " + super1.toString());

        System.out.println();

        Sub sub1 = new Sub();
        System.out.println("sub1: " + sub1.toString());

        System.out.println();

        //Dynamische Bindung
        Super super2 = new Sub();
        System.out.println("super2: " + super2.toString());

        System.out.println();

        Object o1 = new Super();
        System.out.println("o1: " + o1.toString());

        System.out.println();

        Object o2 = new Sub();
        System.out.println("o2: " + o2.toString());

    }
}

```



super1: Super[s1=SUPER]

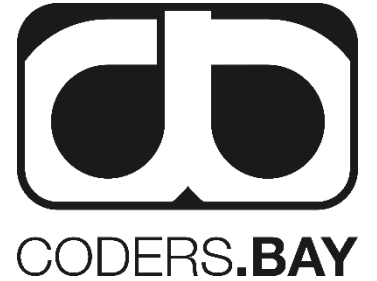
sub1: Sub[s1=SUB, s2=SUPER]

super2: Sub[s1=SUB, s2=SUPER]

o1: Super[s1=SUPER]

o2: Sub[s1=SUB, s2=SUPER]

# VERERBUNG – DYNAMISCHE BINDUNG - KONSTRUKTOREN

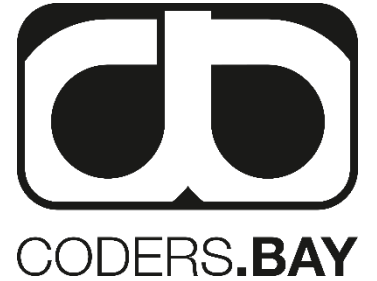


```
/**
 * @author Simon Haidinger
 * @version 1.0
 */
public class Super {

    Super ()
    {
        whoAmI ();
    }

    void whoAmI ()
    {
        System.out.println( "Ich weiß es noch nicht :-( aber ich bin super :-( " );
    }
}
```

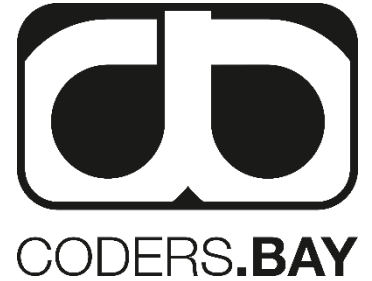
# VERERBUNG – DYNAMISCHE BINDUNG - KONSTRUKTOREN



```
/**
 * @author Simon Haidinger
 * @version 1.0
 */
public class Sub extends Super {
    String who = "Ich bin eine Subklasse";

    @Override
    void whoAmI ()
    {
        System.out.println( who );
    }
}
```

# VERERBUNG – DYNAMISCHE BINDUNG - KONSTRUKTOREN



```
/**
 * @author Simon Haidinger
 * @version 1.0
 */
public class Hauptprogramm {
```

```
    public static void main( String[] args )
    {
        Super super1 = new Super();
        super1.whoAmI();

        Sub sub1 = new Sub();
        sub1.whoAmI();
    }
```

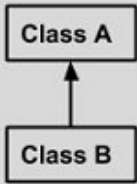
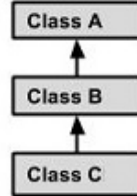
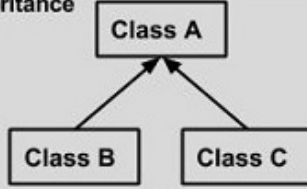
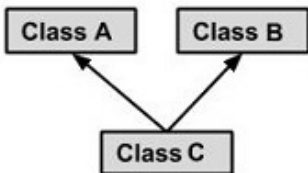
Ich weiß es noch nicht :-( aber ich bin super :-)

Ich weiß es noch nicht :-( aber ich bin super :-)

null

Ich bin eine Subklasse

# VERERBUNG

<b>Single Inheritance</b> 	<pre>public class A {     ..... } public class B extends A {     ..... }</pre>
<b>Multi Level Inheritance</b> 	<pre>public class A { .....} public class B extends A {.....} public class C extends B {.....}</pre>
<b>Hierarchical Inheritance</b> 	<pre>public class A { .....} public class B extends A {.....} public class C extends A {.....}</pre>
<b>Multiple Inheritance</b> 	<pre>public class A { .....} public class B {.....} public class C extends A,B {     ..... } // Java does not support mutiple Inheritance</pre>

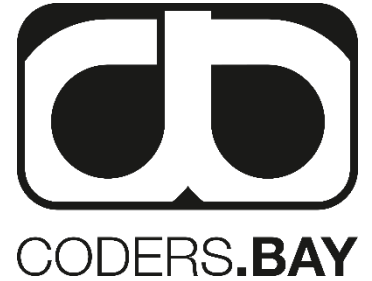
# ABSTRAKTE KLASSEN



- Klasse soll / muss nicht immer sofort ausprogrammiert werden
  - z.B.: wenn Oberklasse lediglich Methoden für die Unterklassen vorgeben möchte
    - nicht weiß, wie sie diese implementieren soll.
  - Java → zwei Konzepte
    - **abstrakte Klassen**
    - Schnittstellen (engl. `interfaces`)
- Schlüsselwort `abstract`
- Attribute und Methoden können auch als `abstract` deklariert werden

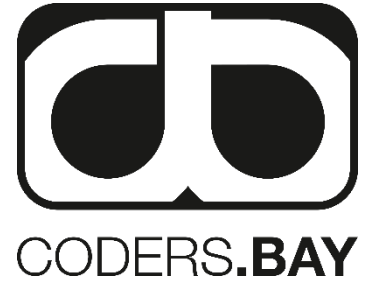


# ABSTRAKTE KLASSEN



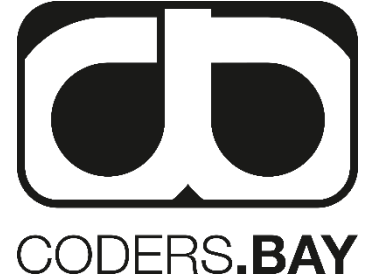
- Objekte erzeugen ist nicht immer sinnvoll
  - z.B.: soll es untersagt werden, wenn eine Klasse nur als Oberklasse in einer Vererbungshierarchie existieren soll
    - sie kann als Modellierungsklasse eine Ist-eine-Art-von-Beziehung ausdrücken und Signaturen für die Unterklassen vorgeben.
- eine Oberklasse besitzt Vorgaben für die Unterklasse
  - alle Unterklassen erben die Methoden
  - ein Instanz der Oberklasse muss nicht existieren
  - **eine Instanz einer abstrakten Klasse darf nicht existieren**

# ABSTRAKTE KLASSEN



```
/**
 * @author Simon Haidinger
 * @version 1.0
 */
public abstract class GeometrischeFigur {
    public abstract void berechneFlaecheninhalt();
}
```

# ABSTRAKTE KLASSEN



```
/**
 * @author Simon Haidinger
 * @version 1.0
 */
public class Quadrat extends GeometrischeFigur {
    private int a;

    public Quadrat(int a) {
        this.setA(a);
    }

    public int getA() {
        return a;
    }

    public void setA(int a) {
        this.a = a;
    }

    @Override
    public void berechneFlaecheninhalt() {
        System.out.println("berechneFlaecheninhalt() Quadrat: Die Fläche ist: " + a * a);
    }
}
```

# ABSTRAKTE KLASSEN



```
/**
 * @author Simon Haidinger
 * @version 1.0
 */
public class Rechteck extends GeometrischeFigur {
    private int a;
    private int b;

    Rechteck(int a, int b) {
        this.setA(a);
        this.setB(b);
    }

    public int getA() {
        return a;
    }

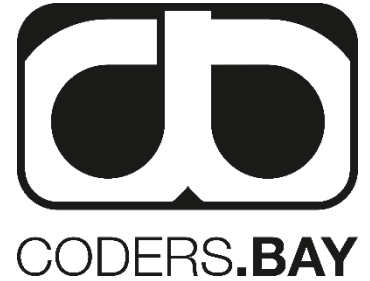
    public void setA(int a) {
        this.a = a;
    }

    public int getB() {
        return b;
    }

    public void setB(int b) {
        this.b = b;
    }

    @Override
    public void berechneFlaecheninhalt() {
        System.out.println("berechneFlaecheninhalt() Quadrat: Die Fläche ist: " + a * b);
    }
}
```

# ABSTRAKTE KLASSEN



```
/**
 * @author Simon Haidinger
 * @version 1.0
 */
public class Hauptprogramm {

    public static void main(String[] args) {
        System.out.println("RECHTECK");
        Rechteck r1 = new Rechteck(2, 3);
        r1.berechneFlaecheninhalt();

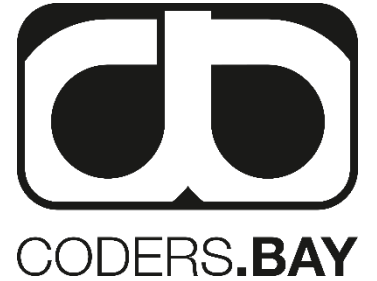
        System.out.println();

        System.out.println("QUADRAT");
        Quadrat q1 = new Quadrat(3);
        q1.berechneFlaecheninhalt();
    }
}
```

RECHTECK  
berechneFlaecheninhalt() Quadrat: Die Fläche ist: 6

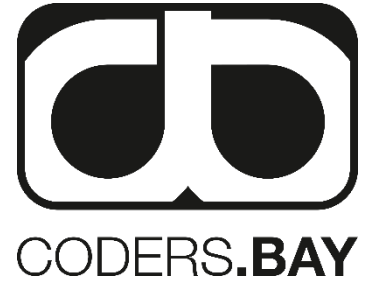
QUADRAT  
berechneFlaecheninhalt() Quadrat: Die Fläche ist: 9

# MEHRFACHVERERBUNG



- Mehrfachvererbung = Polymorphismus (engl. Polymorphism)
- Java kann nur Einfachvererbung
  - dafür gibt es Interfaces

# INTERFACES



- In Java existieren neben **Klassen** auch **Interfaces**
- `interface` → anstatt dem Schlüsselwort `class`
- Aufbau sehr ähnlich zu einer Klasse
  - nahezu vergleichbar zu abstrakten Klassen
    - welche **nur** Methodendeklarationen enthalten
  - ein Interface besitzt ebenfalls keine Implementierung
    - = einzige Unterschied zu einer Klasse
      - sondern nur **Methodenköpfe und Konstanten**

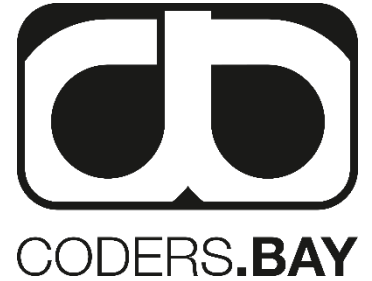
# INTERFACES

- Schlüsselwort `implements`
- kann mehrfach vorkommen → durch `,` getrennt
  - simuliert somit die Mehrfachvererbung

```
Modifikator class Klasse extends Superklasse implements MyInterface
{
    // Anweisungen
}
```



# INTERFACES

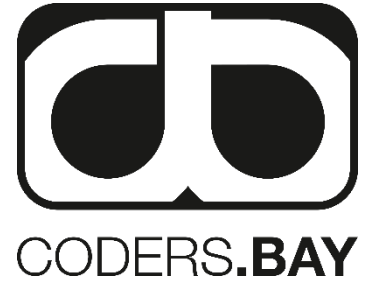


```
/**
 * @author Simon Haidinger
 * @version 1.0
 */
public interface MeinErstesInterface {
    final public int VERSION = 1;
    //Attribut wird zur Konstante
    public int eineNummer = 1;

    public void meineErstePublicInterfaceMethode ();

    abstract void meineErsteAbstrakteInterfaceMethode ();
}
```

# INTERFACES



```
/**
 * @author Simon Haidinger
 * @version 1.0
 */
public interface MeinZweitesInterface {
    final public int VERSION = 2;
    // Attribut wird zur Konstante
    public int eineNummer = 2; ✗

    public String meineErstePublicStringInterfaceMethode();

    abstract int meineErsteAbstrakteIntInterfaceMethode();
}
```

# INTERFACES



```
/**
 * @author Simon Haidinger
 * @version 1.0
 */
public class EineKlasse implements MeinErstesInterface, MeinZweitesInterface {
```

```
    @Override
    public void meineErstePublicInterfaceMethode() {
        System.out.println("Das ist eine Methode");
    }
```

```
    @Override
    public void meineErsteAbstrakteInterfaceMethode() {
        System.out.println("Das ist eine weitere Methode");
    }
```

```
    @Override
    public String meineErstePublicStringInterfaceMethode() {
        String s = "Das ist ein String";
        return s;
    }
```

```
    @Override
    public int meineErsteAbstrakteIntInterfaceMethode() {
        int i = 10;
        return i;
    }
```

```
//eine Methode von EineKlasse, in keinem Interface enthalten
```

```
public void printAttributes() {
    //MeinErstesInterface.version = 2;
    //MeinErstesInterface.eineNummer = 3;
    System.out.println("MeinErstesInterface.version --> " + MeinErstesInterface.VERSION);
    System.out.println("MeinErstesInterface.eineNummer --> " + MeinErstesInterface.eineNummer);
    System.out.println("MeinZweitesInterface.version --> " + MeinZweitesInterface.VERSION);
    System.out.println("MeinZweitesInterface.eineNummer --> " + MeinZweitesInterface.eineNummer);
}
```

Probleme beim setzen der "Variablen / Konstanten"  
Attribute in einem `interface` werden automatisch zu Konstanten  
Konstanten --> unveränderbar

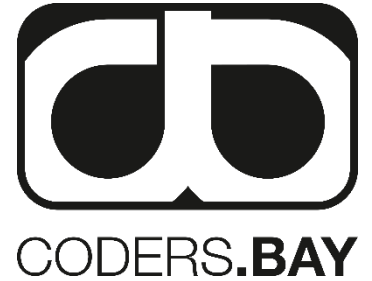
# INTERFACES



```
/**
 * @author Simon Haidinger
 * @version 1.0
 */
public class Hauptprogramm {

    public static void main(String[] args) {
        EineKlasse ek = new EineKlasse();
        ek.meineErstePublicInterfaceMethode();
        ek.meineErsteAbstrakteInterfaceMethode();
        System.out.println(ek.meineErstePublicStringInterfaceMethode());
        System.out.println(ek.meineErsteAbstrakteIntInterfaceMethode());
        //System.out.println(ek.version);
        //System.out.println(ek.eineNummer);
        System.out.println("MeinErstesInterface.VERSION --> " + MeinErstesInterface.VERSION);
        System.out.println("MeinErstesInterface.eineNummer --> " + MeinErstesInterface.eineNummer);
        System.out.println("MeinZweitesInterface.VERSION --> " + MeinZweitesInterface.VERSION);
        System.out.println("MeinZweitesInterface.eineNummer --> " + MeinZweitesInterface.eineNummer);
        ek.printAttributes();
    }
}
```

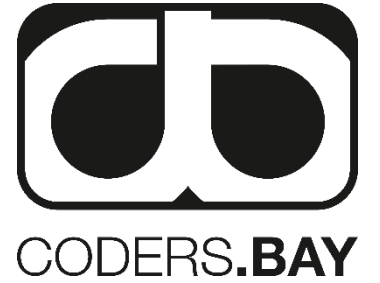
# VOLLSTÄNDIGER MODIFIKATOREN ÜBERBLICK

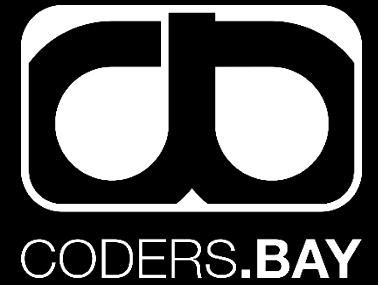


	Attribut	Methode	Konstruktor	Klasse	Interface
<b>abstract</b>		X		X	X
<b>final</b>	X	X		X	
<b>native</b>		X			
<b>private</b>	X	X	X		
<b>protected</b>	X	X	X		
<b>public</b>	X	X	X	X	X
<b>static</b>	X	X			
<b>synchronized</b>		X			
<b>transient</b>	X				
<b>volatile</b>	X				

# AUSBLICK

- Rekursion
  - Beispiel





**VIELEN DANK FÜR EURE  
AUFMERKSAMKEIT !**