

Java Fundamentals

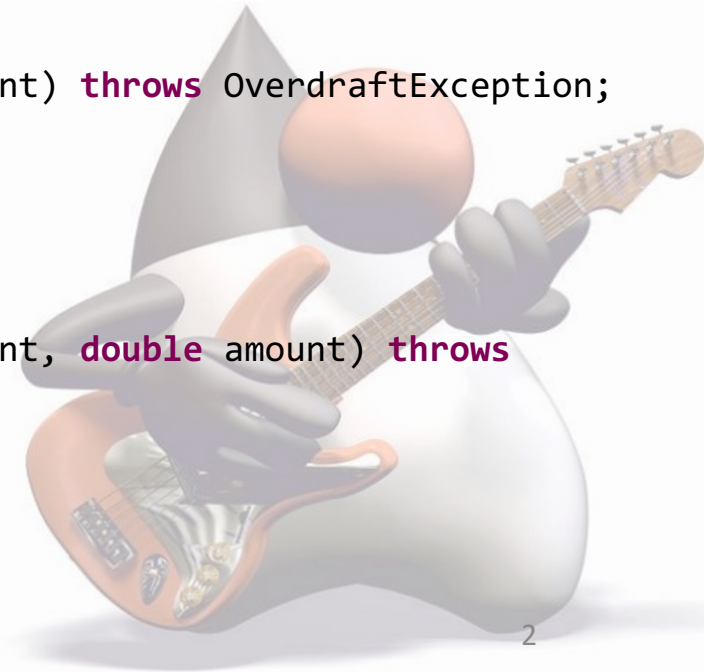
Abstract Classes & Interfaces



Abstrakte Klassen

- Eine abstrakte Methode definiert eine Signatur wie eine Methode aussehen soll, aber ohne Implementierung
- Eine Klasse muss als abstrakt definiert sein, sobald sie eine abstrakte Methode besitzt

```
public abstract class Account {  
    private String accountNo;  
    private double balance;  
    private double overdraftLimit;  
  
    public Account(String accountNo, double balance, double overdraftLimit) {  
        super();  
        this.accountNo = accountNo;  
        this.balance = balance;  
        this.overdraftLimit = overdraftLimit;  
    }  
  
    public abstract void withdraw(double amount) throws OverdraftException;  
  
    public void deposit(double amount) {  
        balance += amount;  
    }  
  
    public void transfer(Account anotherAccount, double amount) throws  
    OverdraftException{  
        withdraw(amount);  
        anotherAccount.deposit(amount);  
    }  
}
```



Abstrakte Klassen

- Jede konkrete Klasse, die eine Subklasse von Account ist, muss die Methode withdraw() implementieren.

```
public class CheckingAccount extends Account {  
    public CheckingAccount(String accountNo, double balance,  
                           double overdraftLimit) {  
        super(accountNo, balance, overdraftLimit);  
    }  
}
```

Compile error: The type CheckingAccount must implement the
inherited abstract method Account.withdraw(double)

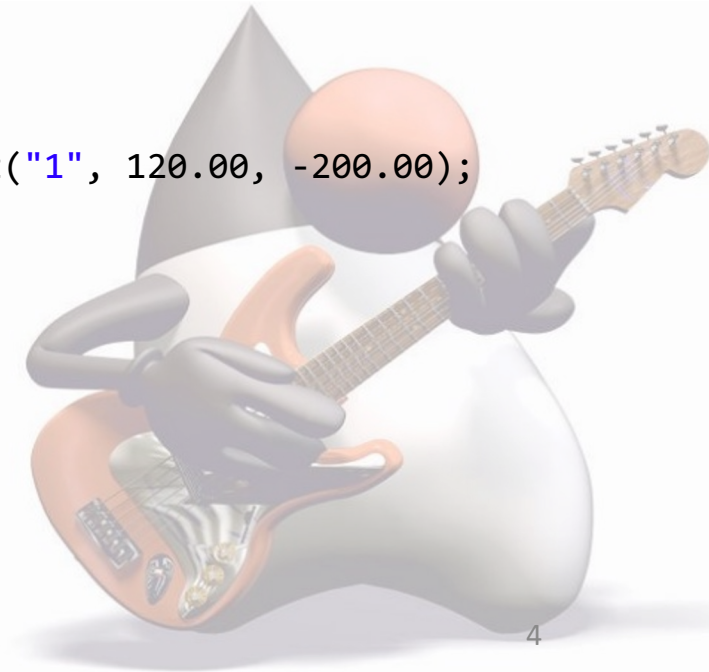


Abstrakte Klassen

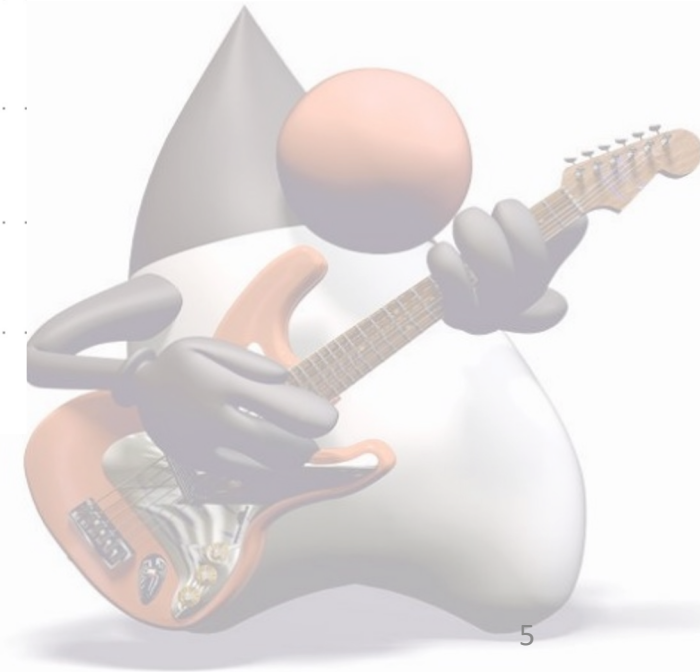
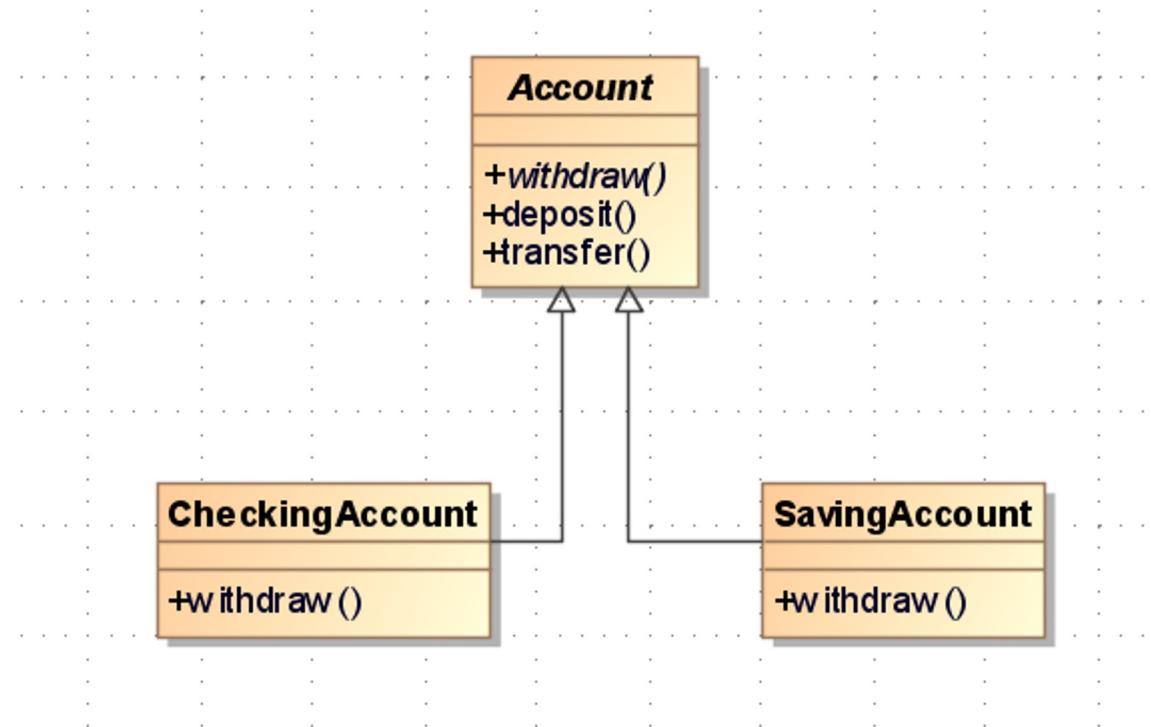
Abstrakte Klassen können nicht instanziiert werden

Abstrakte Klassen können aber als Variablentypen verwendet werden (Polymorphie)

```
public class CheckingAccount extends Account {  
  
    public CheckingAccount(String accountNo, double balance,  
                           double overdraftLimit) {  
        super(accountNo, balance, overdraftLimit);  
    }  
  
    @Override  
    public void withdraw(double amount) throws OverdraftException  
    {  
        // Implementation here  
    }  
}  
  
Account account = new CheckingAccount("1", 120.00, -200.00);
```



Wir könnten unsere Account Hierarchie wie folge ändern



Variablen und Konstruktoren

- Abstrakte Klassen können Variablen haben
- Inklusive Getter und Setter
- Abstrakte Klassen müssen einen Konstruktor haben

```
public abstract class AbstractTestDefinition {  
  
    public String name;  
  
    public AbstractTestDefinition(String name) {  
        this.name = name;  
    }  
  
    public abstract void defineTest();  
  
    public abstract void removeTest();  
  
    public void checkTest() {  
        defineTest();  
        removeTest();  
    }  
}
```

```
public class AbstractMain {  
    public static void main(String[] args) {  
        TestDefinition def = new TestDefinition("Name of Testdefinition");  
        def.checkTest();  
        System.out.println(def.name);  
    }  
}
```

Output:

A Test has been defined
A Test has been removed
Name of Testdefinition

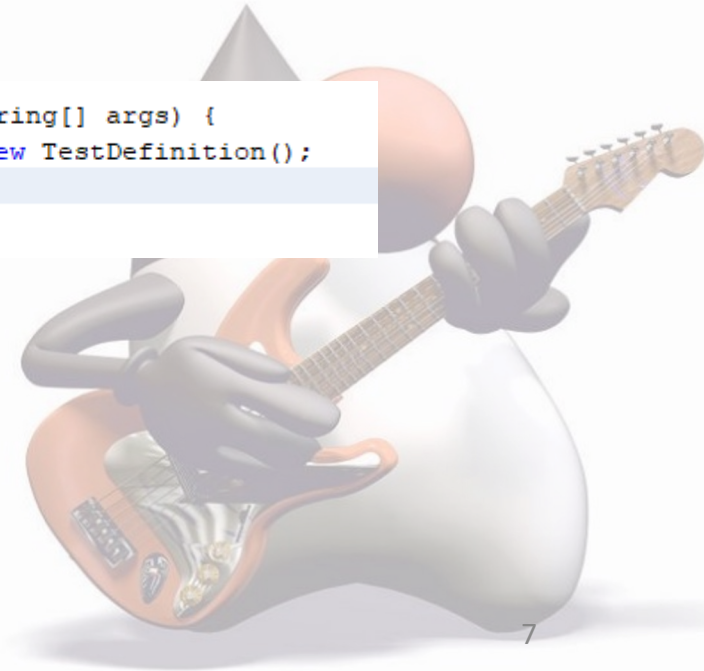


Erweiterte Verwendung

- Konkrete Methoden dürfen abstrakte Methoden aufrufen, sie werden später durch die konkrete Subklasse definiert werden

```
public abstract class AbstractTestDefinition {  
  
    public abstract void defineTest();  
  
    public abstract void removeTest();  
  
    public void checkTest() {  
        defineTest();  
        removeTest();  
    }  
  
    public class TestDefinition extends AbstractTestDefinition {  
  
        @Override  
        public void defineTest() {  
            System.out.println("A Test has been defined");  
        }  
  
        @Override  
        public void removeTest() {  
            System.out.println("A Test has been removed");  
        }  
    }  
}
```

```
public static void main(String[] args) {  
    TestDefinition def = new TestDefinition();  
    def.checkTest();  
}
```



Interfaces

- Interfaces definieren eine Schnittstelle
 - Ursprünglich hatten Interfaces nur abstrakte Methoden
 - sie können static final Variablen haben
 - sie können keine Instanzvariablen haben
 - Interfaces geben Vertrag vor, schränken Implementierungs-Klassen aber nicht ein!
 - Ein Interface kann von einem oder mehreren Interfaces erben (Mehrfachvererbung)
- Was bedeutet das?
 - Eine Klasse kann jedes Interface implementieren
 - Eine Klasse kann mehrere Interfaces implementieren!



Beispiel: Java.lang.Comparable

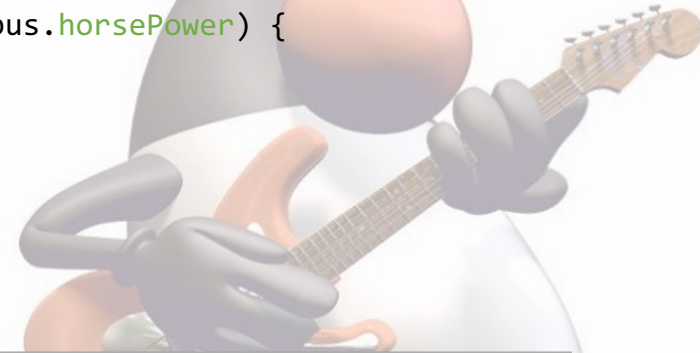
```
public static void main(String[] args) {  
    ComparableBus bigbus = new ComparableBus(200);  
    ComparableBus smallBus = new ComparableBus(100);  
    ComparableBus middlebus = new ComparableBus(150);  
  
    List<ComparableBus> busList = new ArrayList<>();  
    busList.add(bigbus);  
    busList.add(smallBus);  
    busList.add(middlebus);  
    Collections.sort(busList);  
    System.out.println(busList);  
}
```

```
public interface Comparable {  
    public int compareTo(Object o);  
}
```

```
public class ComparableBus extends Vehicle implements Comparable {  
  
    private int seats = 32;  
    private int horsePower;  
    private String busName;  
  
    public ComparableBus(int horsePower){  
        this.horsePower = horsePower;  
    }  
  
    @Override  
    public int compareTo(Object o) { //CONTRACT METHOD!  
        ComparableBus bus = (ComparableBus) o;  
        if (horsePower > bus.horsePower) {  
            return -1;  
        } else if (horsePower < bus.horsePower) {  
            return 1;  
        } else {  
            return 0;  
        }  
    }  
    //toString method etc.  
}
```

Output:

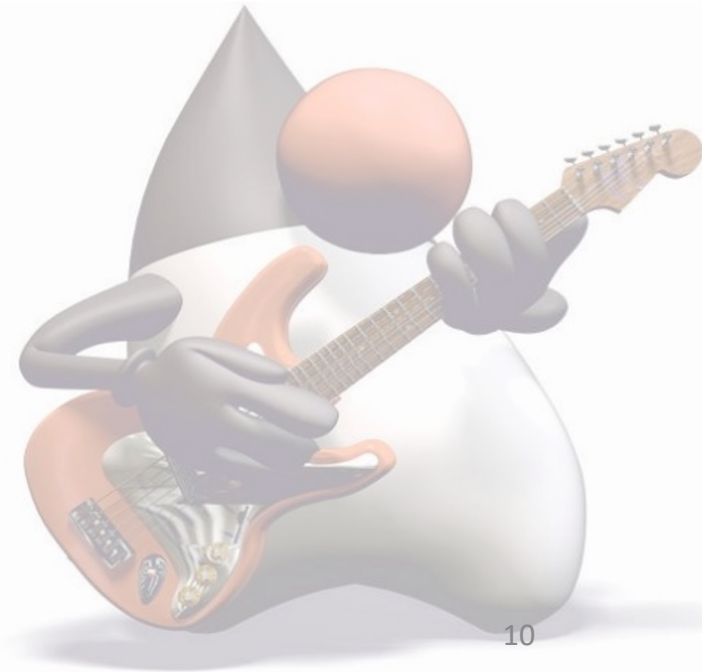
```
[ComparableBus{seats=32, horsePower=200, busName=null},  
ComparableBus{seats=32, horsePower=150, busName=null},  
ComparableBus{seats=32, horsePower=100, busName=null}]
```



Interfaces

- Was kann ich mit Interfaces tun?
 - Einen Vertrag definieren, welche Methoden implementiert werden müssen, sobald das Interfaces mit implements <InterfaceName> verwendet wird
 - Alle abstrakten Methoden des Interfaces sind implizit public

```
public interface TestInterface {  
    void goTest();  
}  
  
public class TestInterfaceImpl implements TestInterface{  
    @Override  
    public void goTest() {  
        throw new UnsupportedOperationException("Not supported yet.");  
    }  
}
```

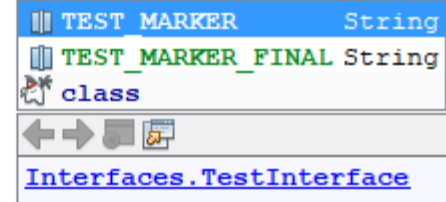


Interfaces

- Was kann ich in Interfaces Speichern?
 - Konstanten
 - Implizit public final und static!

```
public interface TestInterface {  
  
    public final static String TEST_MARKER = "TEST";  
    public String TEST_MARKER_FINAL = "TEST_FINAL";  
  
    void goTest();  
}
```

```
public class TestInterfaceMain {  
    public static void main(String[] args) {  
        TestInterface.  
    }  
}
```



IDE autocomplete popup for `TestInterface.` showing:

- TEST_MARKER String
- TEST_MARKER_FINAL String
- class

Below the popup is a navigation bar with icons for back, forward, search, and other IDE functions.

Interfaces

- Wie können Interfaces vererbt werden?
 - Ein Interface kann ein anderes Interface erben!

```
public interface TestInterface extends Comparable<Object> {  
  
    public final static String TEST_MARKER = "TEST";  
    public String TEST_MARKER_FINAL = "TEST_FINAL";  
  
    void goTest();  
}
```

```
public class TestInterfaceImpl implements TestInterface {  
  
    @Override  
    public void goTest() {  
  
        throw new UnsupportedOperationException("Not supported yet.");  
    }  
  
    @Override  
    public int compareTo(Object o) {  
        throw new UnsupportedOperationException("Not supported yet.");  
    }  
}
```

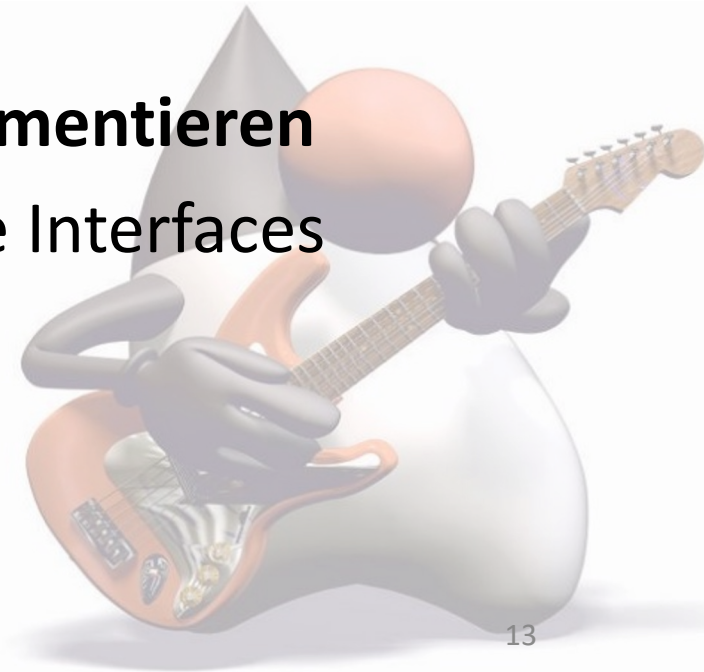


Interfaces / Klassen

- Ein Interface kann **mehrere andere Interfaces erweitern**.

```
public interface TestInterface extends Comparable<Object>, AutoCloseable{  
  
    public final static String TEST_MARKER = "TEST";  
    public String TEST_MARKER_FINAL = "TEST_FINAL";  
  
    void goTest();  
  
}
```

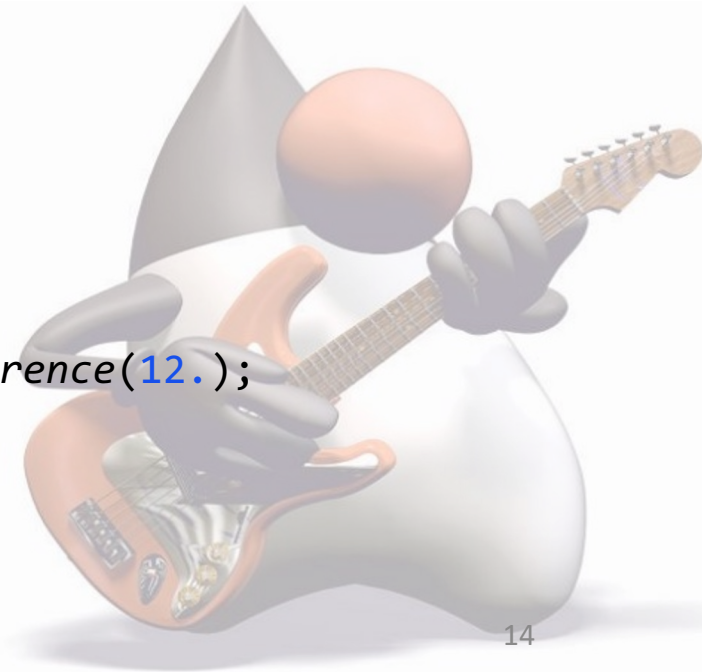
- Eine **Klasse** kann ein oder **mehrere Interfaces implementieren**
- Eine Klasse kann **eine Klasse erweitern** und mehrere Interfaces implementieren.



Statische Methoden in Interfaces

Interfaces können statische Methoden (seit Java 8) haben

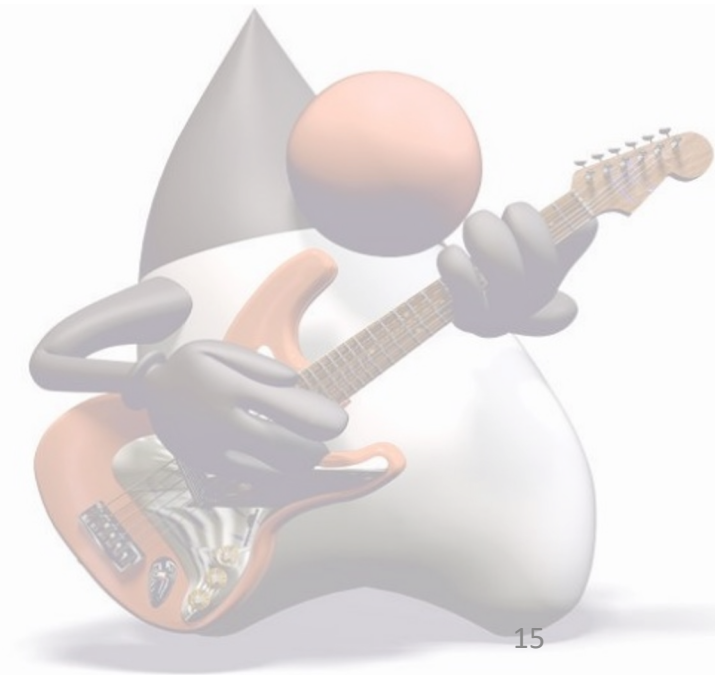
```
public interface GoemetricFigure {  
    static double calculateCircleArea(double radius){  
        return Math.pow(radius, 2.) * Math.PI;  
    }  
  
    static double calculateCircleCircumference(double radius){  
        return 2. * radius * Math.PI;  
    }  
}  
  
class Caller{  
    public static void main(String[] args) {  
        double area = GoemetricFigure.calculateCircleArea(12.);  
        double circumference = GoemetricFigure.calculateCircleCircumference(12.);  
    }  
}
```



Default Methoden in Interfaces

Interfaces können schon ausimplementierte „default“ Methoden (seit Java 8) haben

```
public interface GoemetricFigure {  
    double getHeight();  
    double getArea();  
  
    default double getVolume(){  
        return getHeight() * getArea();  
    }  
}
```



Default Methoden in Interfaces

Default Methoden müssen nicht überschrieben werden.

```
public class Cuboid implements GoemetricFigure{
    private double a;
    private double b;
    private double c;

    public Cuboid(double a, double b, double c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    @Override
    public double getHeight() {
        return c;
    }

    @Override
    public double getArea() {
        return a * b;
    }

    // do not have to override getVolume
}
```

```
public interface GoemetricFigure {
    double getHeight();
    double getArea();

    default double getVolume(){
        return getHeight() * getArea();
    }
}
```



Default Methoden in Interfaces

... aber sie können überschrieben werden

```
public class Cuboid implements GoemetricFigure{
    private double a;
    private double b;
    private double c;

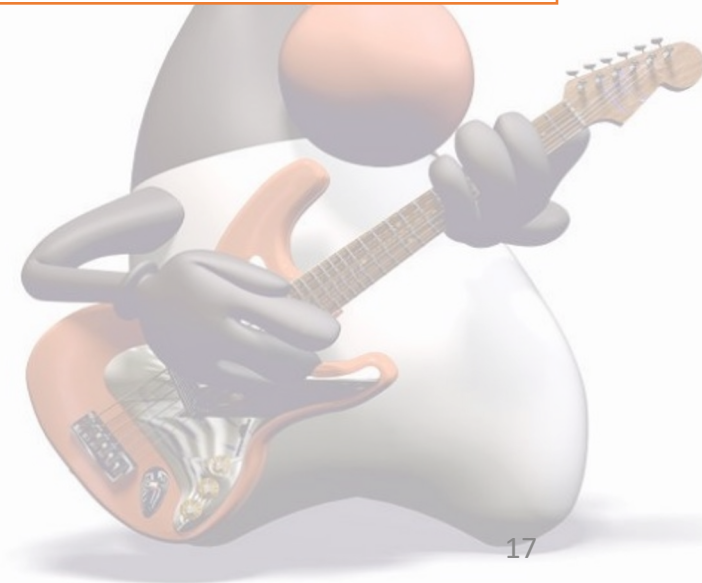
    public Cuboid(double a, double b, double c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    // ...

    @Override
    public double getVolume() {
        return a * b * c;
    }
}
```

```
public interface GoemetricFigure {
    double getHeight();
    double getArea();

    default double getVolume(){
        return getHeight() * getArea();
    }
}
```



Private Methoden in Interfaces

Seit Java 9 können Interfaces auch private Methoden haben

```
public class Cuboid implements GoemetricFigure{
    private double a;
    private double b;
    private double c;

    public Cuboid(double a, double b, double c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    // ...

    @Override
    public double getVolume() {
        return calculateVolume();
    }

    private double calculateVolume() {
        return a * b * c;
    }
}
```

```
public interface GoemetricFigure {
    double getHeight();
    double getArea();

    default double getVolume(){
        return getHeight() * getArea();
    }
}
```



Java Fundamentals

Abstract Classes & Interfaces

