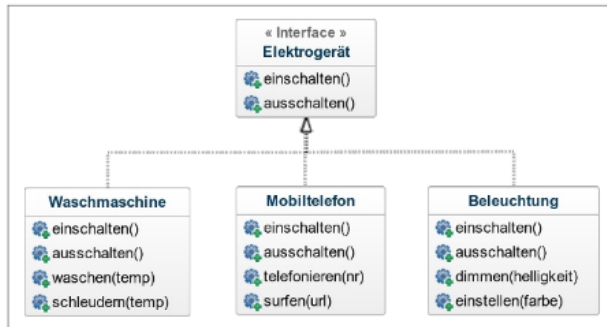


# Interfaces

## ► Einführung

- Interface → *gemeinsamer* Datentyp für Objekte *unterschiedlicher* Klassen
  - Gemeinsame Verhaltensformen → gemeinsame Methoden
  - Legt fest, *welche* gemeinsamen Methoden es gibt
  - *Was* diese Methoden machen, legt jede Klasse für sich fest



## ► Interfaces definieren

- Analog zu Klassendefinition, aber:
  - **interface** statt **class**
  - Methoden werden nur aufgezählt, aber nicht ausprogrammiert
  - Methoden sind immer **public** → **public** kann entfallen

```
public interface Elektrogeraet {
    void einschalten();
    void ausschalten();
}
```

- Interface-Name wird zu einem Datentyp
  - Für Variable, Methoden- und Typparameter verwendbar

```
private Elektrogeraet verbraucher;
...
private Collection<Elektrogeraet> pcs;
...
void entsorgen(Elektrogeraet alt) { ... }
```

## ▸ Interfaces implementieren

- Einer Klasse die Verhaltensformen von einem oder mehreren Interfaces geben
  - Interface angeben
  - Interface-Methode in der Klasse ausprogrammieren

```
public class Mobiltelefon implements Elektrogeraet {  
    public einschalten() {  
        // Booten, PIN-Abfrage, Homescreen, ...  
    }  
  
    public ausschalten() {  
        // Bestätigungsdiallog, Dienste stoppen, ...  
    }  
  
    public void telefonieren(String nr) { ... }  
    public void surfen(String url) { ... }  
}
```

## ▸ Ergänzungen

- Lies **X implements Y** als „X verhält sich wie Y“,  
z.B. „ein Mobiltelefon verhält sich wie ein Elektrogerät“
- Eine Klasse kann auch mehrere Interfaces implementieren

```
public class Mobiltelefon implements Elektrogeraet, Display { ... }
```

- Auch abgeleitete Klassen können Interfaces implementieren
  - Erst **extends**, dann **implements** (alphabetische Reihenfolge)

```
public class Mobiltelefon extends Telefon implements Elektrogeraet, Display { ... }
```

- Von einem oder mehreren Interfaces kann man ein neues Interface ableiten

```
public interface Batteriegeraet extends Elektrogeraet {  
    public void aufladen();  
}
```

- Aus Interfaces kann man keine Objekte erzeugen → kein Konstruktor!

```
Elektrogeraet dings = new Elektrogeraet(); // Syntaxfehler
```

## ▸ default-Methoden

- Werden *schon im Interface* implementiert
- Werden an Subinterfaces und implementierende Klassen vererbt
- Können überschrieben werden
- Machen Interfaces erweiterbar, die bereits implementiert wurden, z.B.

```
public interface Comparator<T> {  
    int compare(T o1, T o2);           // abstrakt  
  
    default Comparator<T> reversed() { // implementiert  
        return Collections.reverseOrder(this);  
    }  
    :  
}
```

## ▸ static-Methoden

- Werden *schon im Interface* implementiert

Statische Methoden in Interfaces verhalten sich im Wesentlichen so wie statische Methoden in Klassen, mit einem Unterschied: Sie vererben sich nicht an abgeleitete Typen.

→ Aufruf nur möglich via `InterfaceName.methodName()`

## ▸ Funktionale Interfaces

- Interfaces mit *einer einzigen* abstrakten Methode → Lambda-Ausdrücke
- Bisher „zufällig“: [Comparator](#), [ChangeListener](#), ...
- Gezielt für häufige Lambda-Anwendungsfälle → [java.util.function](#)
  - [Predicate](#): Bedingung überprüfen, **boolean**-Rückgabewert
  - [Function](#), [BiFunction](#): Funktionswert aus Argument(en) berechnen
  - [UnaryOperator](#), [BinaryOperator](#) ≈ **Function** bzw. **BiFunction**,  
Argumente und Ergebnis jedoch alle vom selben Datentyp
  - [Supplier](#): Rückgabewert erzeugen, kein Argument
  - [Consumer](#): Argument verarbeiten, kein Rückgabewert

# Lambda-Ausdrücke

## ▸ Eine kurze Geschichte der Lambdas (in Java)

- 2006 – [James Gosling](#): „We will never have lambdas in Java“
- 2007 – Drei verschiedene Vorschläge für Lambdas in Java
- 2008 – [Mark Reinhold](#): „We will never have lambdas in Java“
- 2009 – Projekt „Lambda“ beginnt
- 2013 – Java 8 bietet Lambdas
- Zusammengefasst

```
public boolean javaWirdLambdasHaben(int jahr) {  
    return jahr % 2 == 1;  
}
```

## ▸ Motivation

- Funktion („Verhalten“) speichern bzw. als Argument übergeben
  - Einfach möglich z.B. in JavaScript, Python, Ruby, Scala, ...
  - Interface mit einziger Methode implementieren → umständlich:

```
Integer[] zahlen = { 11, 3, 42, 0, 89 };  
Arrays.sort(zahlen, new Comparator<Integer>() {  
    public int compare(Integer z1, Integer z2) {  
        return z2-z1;  
    }  
});
```

- Anonyme innere Klassen
  - Klobige Syntax mit „Nebengeräuschen“
  - Objekt muss erzeugt und entsorgt werden → Laufzeitoverhead
  - Verwendung von **this** verwirrend

## ▸ Vorteile von Lambda-Ausdrücken

- Notation reduziert auf das Wesentliche

- Anonyme innere Klasse:

```
Arrays.sort(zahlen, new Comparator<Integer>() {  
    public int compare(Integer z1, Integer z2) {  
        return z2-z1;  
    }  
});
```

- Lambda-Ausdruck:

```
Arrays.sort(zahlen, (z1, z2) -> z2-z1);
```

- Nicht nur *Daten*, sondern auch *Verhalten* als Argument übergebbar
- Ermöglichen mächtige und ausdrucksstarke APIs, z.B. [Streams](#)

Beispiel: einfaches Layout mit EventHandler für Button.

```
Button button1 = new Button("Hello");
Button button2 = new Button("World");

//      button1.setOnAction();
//      button2.setOnAction();
```

- ➔ Für die Buttons muss die `setOnAction( ... )`-Methode vervollständigt werden ....
- ➔ Der Parameter der `setOnAction()`-Methode ist ein `EventHandler`, dieser ist ein Interface ....
- ➔ `EventHandler<...>` erwartet eine „programmierte“ Methode

```
public void handle(ActionEvent e) { ... }
```

```
Button button1 = new Button("Hello");
Button button2 = new Button("World");

button1.setOnAction(new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent event) {
        System.out.println("Button 1 geklickt");
    }

});
//      button2.setOnAction();
```

(hier wird eine Möglichkeit gezeigt, wie dieses Interface `EventHandler` als Parameter erzeugt werden kann: eine **anonyme innere Klasse** .. die zu implementierende Methode muss gleich geschrieben werden ...

Wenn ein Interface nur genau EINE Methode „vorgibt“ (= Functional Interface), kann ein Lambda Ausdruck verwendet werden, der die Definition der anonymen inneren Klasse „ersetzt“ :

Syntax allgemein :

**(Parameter) -> {Methodenkörper}**

**1) Methodenname wird weggelassen (ist ja bekannt/vorgegeben)**

```
button2.setOnAction( (ActionEvent ev2) -> {System.out.println("Button 2 geklickt");} );
```

**2) Typen der Parameter können weggelassen werden (müssen ohnehin mit der bekannten Methode übereinstimmen)**

```
button2.setOnAction( (ev2) -> {System.out.println("Button 2 geklickt");} );
```

**3) Wenn nur EIN Parameter, können die runden Klammern weggelassen werden.**

```
button2.setOnAction( ev2 -> {System.out.println("Button 2 geklickt");} );
```

**4) Wenn nur EINE Anweisung, können die geschwungenen Klammern und das Semikolon ';' weggelassen werden.**

```
button2.setOnAction( ev2 -> System.out.println("Button 2 geklickt"));
```