

# Java Functional Interfaces

## Einleitung:

---

Java ist von Grund auf eine *Object-Oriented Programming language*.  
Bei Java dreht sich alles um *Klassen* und *Objekte*.  
Ausnahme sind die *primitive Datatypes*.



Eine Methode kann bei Java nicht für sich alleine stehen, sie ist immer Teil einer Klasse.

Mit der Einführung *Functional-Interfaces* und der Möglichkeit von *lambda Expression* in Java, besteht nun die Möglichkeit mit Java das *Konzept der funktionalen Programmierung* anzuwenden.

*Funktionale Interfaces* werden auch als *Single Abstract Method Interfaces SAM* bezeichnet.

Die Methode als Funktion tritt in den Vordergrund. Als Klasse dient eine namenlose, anonyme Klasse, die die *abstrakte Methode* eines *Functional Interface* implementiert. Und die Implementierung der Klasse mit einer Methode kann durch eine *lambda Expression* im Umfang stark vereinfacht ausgedrückt werden.

Die Klasse tritt in den Hintergrund, während die eigentlichen Funktion in den Vordergrund tritt.

*Lambda Expressions* und *Methodenreferenzen* machen den Code lesbarer, sauberer und einfacher.

In diesem Dokument geht es um die verschiedenen *Functional Interfaces* von Java. Worin sie sich unterscheiden und welche spezifische Verwendung sich durch sie ergibt.

## Ziel

---

- ⇒ Sie können die Unterschiede verschiedener Functional Interfaces von Java erklären.
- ⇒ Sie können Functional Interfaces von Java spezifisch anwenden.

## Inhalte

---

Einleitung: .....	1
Ziel .....	1
Inhalte .....	1
Warming Up mit bekannten Beispielen .....	2
Zu implementierende Methode des Functional Interfaces .....	3
Arten von lambda Expressions.....	4

## Warming Up mit bekannten Beispielen

---

Um ein Gefühl zu bekommen, wo und wie Functional Interfaces angewandt werden zwei Beispiele aus zwei unterschiedlichen Java packages.

### java.util.function

#### @FunctionalInterface

#### public interface Consumer<T>

- ⇒ Represents the operation **void accept(T t)** that accepts a **single input argument** and returns **no result**.

```
//Kompakt mit Hilfe einer lambda expression  
List<String> list = Arrays.asList("A", "C", "D", "B");  
list.forEach(s -> System.out.println(s));
```

```
//Mit Hilfe einer Methodenreferenz wird das Functional Interface sichtbar  
Consumer<String> consumer2 = s -> { System.out.println(s); };  
list.forEach(consumer1);
```

### java.util

#### @FunctionalInterface

#### public interface Comparator<T>

- ⇒ Represents the operation int **compare(T o1, T o2)** that accepts **two input argument** and returns **as result an int value**.

```
//Kompakt mit Hilfe einer lambda expression  
List<Integer> list = Arrays.asList(1, 3, 4, 2);  
list.sort((i1, i2) -> i1-i2);
```

```
//Mit Hilfe einer Methodenreferenz wird das Functional interface sichtbar  
Comparator<Integer> comparator = (i1, i2) -> {return i1-i2;};  
list.sort(comparator);  
System.out.println(list);
```

## Ergänzung:

- <T>                    <T> steht für einen generischen Typ.  
Generische Typen ermöglichen es, Klassen, Interfaces und Methoden zu schreiben, die mit unterschiedlichen Datentypen arbeiten können, ohne den Datentyp vorab festzulegen.  
Der Buchstabe "T" steht dabei für "Type" und ist der Platzhalter für den tatsächlichen Datentyp.

## Zu implementierende Methode des Functional Interfaces

Bei der Implementierung mit Hilfe der lambda expression taucht der Name der zu überschreibenden, abstrakten Methode von *Consumer* → *accept* oder *Comparator* → *compareTo* nicht mehr auf.

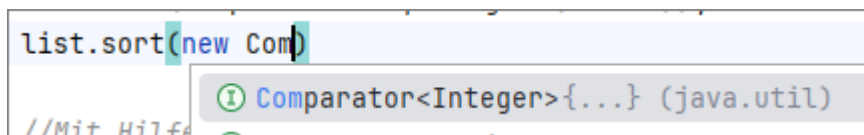
Parameter und Returnwert der abstrakten Methode sind aus dem Namen des Functionalinterface nicht erkennbar.

Wie findet man also heraus, welche Methode überschrieben werden muss und welche Parameter und welche Art von Return-Wert verlangt sind.

⇒ Entweder Sie schlagen in den Erläuterungen zum Funtional Interface nach.

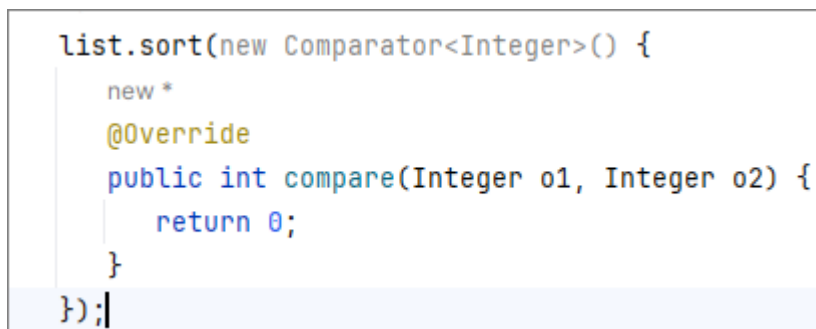
Oder

⇒ Sie lassen die IDE die Klasse generieren und können so die Methode untersuchen.



```
list.sort(new Comp  
//Mit Hilfe
```

generiert



```
list.sort(new Comparator<Integer>() {  
    new *  
    @Override  
    public int compare(Integer o1, Integer o2) {  
        return 0;  
    }  
});
```

Wenn Sie es verstanden haben, durch eine kompakte lambda expression implementieren.

```
list.sort((i1, i2) -> i1-i2);
```

## Arten von lambda Expressions

Wieviele Parameter eine lambda Expression hat und ob sie einen Return Wert liefern muss, das bestimmt das Functional Interface und dessen Methode, die durch die lambda Expression implementiert wird.

### Beispiele:

```
// kein Parameter
// kein Return-Wert
() -> System.out.println("Hello")

// ein Parameter
// Return-Wert
s -> System.out.println(s)

// zwei Parameter
// Return-Wert
(x, y) -> x+y

// zwei Parameter mit explizitem Type
// Return-Wert
(Integer x, Integer y) -> x+y

// zwei Parameter
// Return-Wert
// mehrzeilige Implementierung
(x, y) -> {
    System.out.println(x);
    System.out.println(y);
    return (x+y);
}
```

## Lambda Syntax

- No arguments: `() -> System.out.println("Hello")`
- One argument: `s -> System.out.println(s)`
- Two arguments: `(x, y) -> x + y`
- With explicit argument types:
 

```
(Integer x, Integer y) -> x + y
(x, y) -> {
    System.out.println(x);
    System.out.println(y);
    return (x + y);
}
```
- Multiple statements:

6