Funktionale Interfaces und Lambda-Ausdrücke – Mit Bezug zu anonymen Klassen

Hintergrundwissen: Anonyme Klassen als Ausgangspunkt

In der objektorientierten Programmierung (OOP) in Java sind **anonyme Klassen** eine praktische Möglichkeit, um kleine Implementierungen von Interfaces oder abstrakten Klassen direkt im Code zu erstellen, ohne dafür eine eigene separate Klassendatei anlegen zu müssen.

Beispiel mit einer anonymen Klasse:

```
Rechner addierer = new Rechner() {
    @Override
    public int berechne(int a, int b) {
        return a + b;
    }
};
```

Diese Schreibweise wurde vor allem bei **Callback-Mechanismen** oder kleinen Helfer-Implementierungen genutzt. Jedoch hat sie einen Nachteil: Die Syntax ist relativ **lang und umständlich**.

Mit **Java 8** wurde dieses Problem durch **funktionale Interfaces** und **Lambda-Ausdrücke** eleganter gelöst.

---

1. Funktionales Interface und Lambda

1.1 Begriffe

1.1.1 Was ist ein Funktionales Interface?

Ein **funktionales Interface** ist ein Interface, das genau **eine abstrakte Methode** definiert. Es repräsentiert ein einziges „Verhalten“ und bildet die Grundlage für Lambda-Ausdrücke.

Merkmale:

- Darf beliebig viele `default` und `static` Methoden enthalten.
- Kann andere Interfaces **erweitern**, solange keine zusätzlichen abstrakten Methoden dazu kommen.
- Wird häufig mit der Annotation `@FunctionalInterface` gekennzeichnet, damit der Compiler sicherstellt, dass nur eine abstrakte Methode existiert.

Beispiel:

```
@FunctionalInterface
interface Rechner {
    int berechne(int a, int b);
}
```

Erklärung: Funktionales Interface und abstrakte Methoden in Java

Warum ist `berechne()` abstrakt, auch ohne das Schlüsselwort `abstract`?

In **Java** gilt:

- Alle Methoden in einem **Interface** ohne das Schlüsselwort `default` oder `static` sind **automatisch** `public abstract`.
- Man **muss** `abstract` **nicht hinschreiben** – der Compiler behandelt sie trotzdem als **abstrakte Methoden**.

Beispiele

Vollständig ausgeschrieben:

```
@FunctionalInterface
interface Rechner {
    public abstract int berechne(int a, int b);
}
```

Gekürzt (gleiche Bedeutung):

```
@FunctionalInterface
interface Rechner {
    int berechne(int a, int b);
}
```

Beide Varianten sind **gleichwertig**.

Erklärung der Merkmale

☑ `default` **Methoden**: Methoden mit einer Standardimplementierung.

```
default void info() {
    System.out.println("Dies ist eine Default-Methode.");
}
```

☑ `static` **Methoden**: Statische Methoden im Interface, die direkt über den Interface-Namen aufgerufen werden.

☑ **Wichtig**:

- Ein funktionales Interface **darf beliebig viele** `default` **und** `static` **Methoden enthalten**,
- Aber **nur eine einzige abstrakte Methode**.

Übersicht der Regeln

Methodentyp	Zählt für `@FunctionalInterface`?	Muss implementiert werden?
abstrakt	☑ Ja, maximal **eine** erlaubt	☑ Ja (durch Lambda oder anonyme Klasse)
default	❌ Nein	❌ Nein (hat eigene Implementierung)
static	❌ Nein	❌ Nein (direkt im Interface aufrufbar)

Kombiniertes Beispiel

```
@FunctionalInterface
interface Rechner {
    // Abstrakte Methode (muss implementiert werden)
    int berechne(int a, int b);

    // Default-Methode (optional, hat Implementierung)
    default void info() {
        System.out.println("Das ist ein Rechner Interface.");
    }

    // Statische Methode (direkt über Interface nutzbar)
    static void hilfe() {
        System.out.println("Rechner Interface Hilfemethode.");
    }
}
```

☑ **Korrekt**: Dieses Interface bleibt ein funktionales Interface, weil es nur **eine abstrakte Methode** enthält.

Zusammenfassung

- Die **eine abstrakte Methode** macht das Interface **funktional**.
- `default` und `static` Methoden sind erlaubt, da sie **keine abstrakten Methoden sind**.
- Lambda-Ausdrücke implementieren **nur die abstrakte Methode** – darum funktioniert es so kompakt.

1.1.2 Was ist ein Lambda-Ausdruck?

Ein **Lambda-Ausdruck** ist eine **verkürzte, anonyme Methode**, die **ein funktionales Interface implementiert**, ohne dass man den ganzen „klassischen“ Code einer anonymen Klasse schreiben muss.

Kurz gesagt:

> Lambda = Kurzform von anonymer Klasse für funktionale Interfaces.

Beispiel Vergleich:

Anonyme Klasse:

```
Rechner addierer = new Rechner() {
    @Override
    public int berechne(int a, int b) {
        return a + b;
    }
};
```

Lambda-Ausdruck:

```
Rechner addierer = (a, b) -> a + b;
```

Der Compiler erkennt durch den **Kontext**, dass `Rechner` ein funktionales Interface ist, daher weiß er, welche Methode implementiert werden muss.

1.2 Aufbau von Lambda-Ausdrücken

1.2.1 Grundstruktur

Lambda-Ausdrücke bestehen aus:

- **Parameterliste** (links vom Pfeil `->`)
- **Lambda-Pfeil** `->`
- **Methodenkörper** (rechts vom Pfeil)

Beispiel:

```
int berechne(int a, int b); // Interface-Methode
Rechner addierer = (a, b) -> a + b;
```

Wichtig:

- Der **Rückgabetyp** und die Anzahl der Parameter müssen zur Methode des funktionalen Interfaces passen.
- Der Datentyp der Parameter kann durch **Typinferenz** weggelassen werden, der Compiler leitet ihn aus dem Interface ab.

1.2.2 Komplexere Lambda-Ausdrücke

Wenn der Methodenkörper **mehrere Anweisungen** enthält, verwendet man geschweifte Klammern `{}`. Hier muss das `return`-Keyword explizit angegeben werden.

Beispiel:

```
@FunctionalInterface
public interface QuadratZahl {
    int berechneQuadratZahl(int zahl);
}

QuadratZahl quad = (int zahl) -> {
    System.out.println("Übergebener Wert: " + zahl);
    return zahl * zahl;
};

int ergebnis = quad.berechneQuadratZahl(5);
System.out.println("Das Ergebnis ist: " + ergebnis);
```

1.3 Typische Anwendungsfälle

1.3.1 Lambdas als Argumente an Methoden

Besonders praktisch ist es, wenn man Methoden aufruft, die als Argumente ein funktionales Interface erwarten.

Beispiel:

```
public static int berechnen(int a, int b, Rechner rechner) {
    return rechner.berechne(a, b);
}

int ergebnis = berechnen(3, 4, (x, y) -> x + y);
```

1.3.2 Lambdas beim Iterieren

Lambdas eignen sich perfekt für Methoden wie `forEach()` bei Listen:

```
List<String> namen = Arrays.asList("Anna", "Bob", "Carla");
namen.forEach(name -> System.out.println("Hallo " + name));
```

Beispiel: forEach ohne Lambda (mit anonymer Klasse)

```
import java.util.Arrays;
import java.util.List;
```

```
import java.util.function.Consumer;

public class BeispielForEachOhneLambda {
    public static void main(String[] args) {
        List<String> namen = Arrays.asList("Anna", "Bob", "Carla");

        // Verwendung einer anonymen Klasse statt Lambda
        namen.forEach(new Consumer<String>() {
            @Override
            public void accept(String name) {
                System.out.println("Hallo " + name);
            }
        });
    }
}
```

Beispiel: forEach mit Lambda-Ausdruck

```
import java.util.Arrays;
import java.util.List;

public class BeispielForEachMitLambda {
    public static void main(String[] args) {
        List<String> namen = Arrays.asList("Anna", "Bob", "Carla");

        // Verwendung eines Lambda-Ausdrucks
        namen.forEach(name -> System.out.println("Hallo " + name));
    }
}
```

Erklärung:

- Hier wird ein **Lambda-Ausdruck** verwendet: `(name) -> ...`.
- `forEach` erwartet einen **Consumer**, und das Lambda ersetzt direkt die Implementierung von `accept()`.
- Der Code ist **kürzer und lesbarer** im Vergleich zur anonymen Klasse.

Erklärung:

- Statt `name -> System.out.println()` wird hier eine **anonyme Klasse** verwendet.
- `Consumer<String>` ist ein funktionales Interface mit der Methode `accept()`.
- Der Code ist **deutlich länger**, aber funktioniert ohne Lambda.

1.3.3 Consumer Interface

Ein typisches Beispiel für ein funktionales Interface ist das **Consumer Interface**:

```
Consumer<String> drucker = text -> System.out.println(text);
drucker.accept("Das ist ein Beispiel.");
```

1.3.4 Weitere gebräuchliche Interfaces

Java stellt viele vordefinierte funktionale Interfaces bereit:

- **Predicate** → prüft eine Bedingung und gibt `boolean` zurück
- **Function<T,R>** → überführt einen Typ in einen anderen
- **Supplier** → liefert einen Wert ohne Eingabeparameter
- **Consumer** → führt eine Aktion aus mit einem Parameter

1.4 UML-Diagramm Hinweis

Auch in UML-Diagrammen kann man funktionale Interfaces einfach darstellen:

- Als Interface mit einer einzigen Methode.

Beispiel UML-Skizze für `Rechner`:

```
<<functional interface>> Rechner
+ int berechne(int a, int b)
```

Zusammenfassung

Vorwissen	Neues Wissen
Anonyme Klassen	Lambdas als moderne Kurzschreibweise
Interface mit 1 Methode	Funktionales Interface
Umständliche Syntax	Kompakte, lesbare Syntax
Klassen-Objekte	Ausdrucksbasierte Lösung
Verwendet bei Callbacks, GUI, Streams	Verwendet in Streams, Collections, Events, Threads

Lambda-Ausdrücke sind eine **konsequente Weiterentwicklung** des Konzepts der anonymen Klassen mit dem Ziel, Code **kürzer**, **lesbarer** und **ausdrucksstärker** zu machen.