Inhaltsverzeichnis

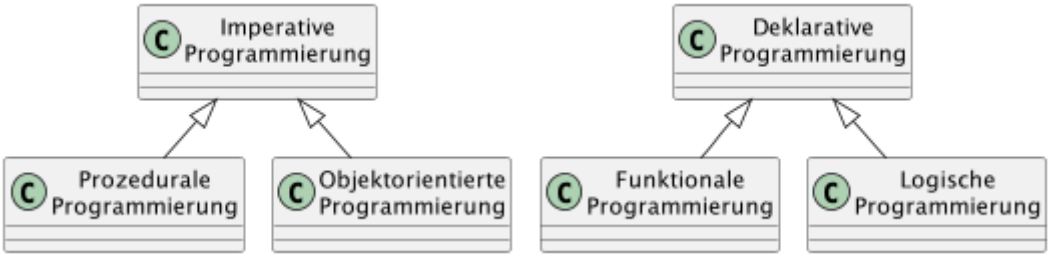
1. Funktionale Programmierung in Java . . . 1
1.1. Imperative Programmierung . . . 1
1.2. Deklarative Programmierung . . . 2
1.3. Lambda Ausdrücke . . . 2
1.4. Streaming (API) . . . 4
1.5. Demonstrationen . . . 7
1.6. Übungen . . . 7
1.7. Tipps, Patterns & Best Practices . . . 9

1. Funktionale Programmierung in Java

Im Wesentlichen stehen sich 2 **Programmierparadigmen** gegenüber:

- das **imperative** Paradigma, aus dem sich OOP (objektorientierte Programmierung) entwickelte, und
- das **deklarative** Paradigma, das die Grundlage für die *funktionale Programmierung* bildet.

In der Übersicht:



1.1. Imperative Programmierung

Bei einem **imperativen** Ansatz ist ein Programm eine *Abfolge von Schritten*, die ihren Zustand ändern, bis das Zielergebnis erreicht wird. Dieser Ansatz steht in engem Zusammenhang mit der Computerarchitektur (Neumann), bei der Anweisungen im Maschinencode einen globalen Zustand ändern. Der Maschinenzustand besteht aus Speicherinhalten und Prozessorregistern. Die meisten Computer verwenden dieses Modell.

Imperative **Programme** funktionieren ähnlich. Zuweisungsanweisungen werden verwendet, um im Speicher gespeicherte Daten zu manipulieren. Anweisungen werden in einer bestimmten Reihenfolge ausgeführt und verwenden diese Daten, um das gewünschte Ergebnis zu berechnen. Imperative Sprachen verwenden häufig Elemente wie Variablen, Schleifen und bedingte Anweisungen.

Zusammenfassend sind die wichtigsten Eigenschaften imperativer Programmierungen:

- veränderlicher Zustand

- Schritt-für-Schritt-Ausführung von Anweisungen und
- die Reihenfolge der Anweisungen ist wichtig

1.2. Deklarative Programmierung

Der **deklarative** Ansatz konzentriert sich auf die Endbedingungen eines gewünschten Ergebnisses und nicht auf die Abfolge von Schritten, die erforderlich sind, um dieses zu erreichen. Ein deklarativ geschriebenes **Programm** ist also keine Folge von Anweisungen, sondern eine Reihe von Eigenschaftsdeklarationen, die das resultierende Objekt haben sollte.

Die Charakteristiken von deklarativer Software sind:

- Das Endergebnis hängt nicht von einem externen Zustand ab
- Fehlen eines inneren Zustands zwischen den Ausführungen
- Determinismus – für die gleichen Eingabeargumente liefert das Programm immer das gleiche Ergebnis
- Die Ausführungsreihenfolge ist nicht immer wichtig – sie kann asynchron sein

Die **funktionale Programmierung** ist eine *Teilmenge* der deklarativen Programmierung. Sie stellt ein *eigenständiges Programmierparadigma* dar.

1.3. Lambda Ausdrücke

Bei der **funktionalen Programmierung** stehen also Funktionen im Mittelpunkt, sie werden nicht nur

- definiert und
- angewendet

sondern auch

- wie Daten miteinander verknüpft,
- als Parameter eingesetzt verwendet oder als
- Funktionsergebnisse genutzt.

Mit **Lambda Ausdrücken** wurde ein lang gefordertes Sprachkonstrukt in Java eingeführt, dass es in ähnlicher Form in anderen Programmiersprachen bereits gibt, z.B. in `C#`.

Der Einsatz von Lambdas erfordert zum Teil eine andere Denkweise und führt zu einem anderen Programmierstil, die wie oben gesagt dem Paradigma der *funktionalen Programmierung* folgt.

Insbesondere in den Bereichen der Parallelverarbeitung von Daten sowie im Bereich der Frameworks finden sich viele Implementierungen, ebenso im Bereich des - bereits bekannten - **Collection** Frameworks.

Ein Lambda ist ein **Behälter für Sourcecode**, einer Methode sehr ähnlich, allerdings

- ohne *Namen* und
- ohne expliziten *Rückgabewert* und
- ohne Deklaration von *Exceptions*

Vereinfacht ausgedrückt kann man einen Lambda-Ausdruck am ehesten als eine *anonyme Methode und einer speziellen Syntax*/Kurzform bezeichnen

Seit `Java 8` sind also diese Elemente in der Sprache enthalten.

Die einfachste Form eines **Lambda** Ausdruckes ist:

```
parameter -> expression
```

Um *mehr als einen Parameter* zu nutzen, müssen runde Klammern angegeben werden:

```
(parameter1, parameter2) -> expression
```

Weil die Ausdrücke limitiert sind und direkt einen Wert zurückgeben, kann auch ein Code-Block genutzt werden:

```
(parameter1, parameter2) -> { code block }
```

Zu beachten ist, dass der `{ code block }` ein return statement enthalten muss.

Ein paar einfache Beispiele:

```
(int x, int y) -> { return x + y; }
(long x) -> { return x + 2; }
() -> { String error = "invalid"; System.err.println(error); }
```

Die Zuweisung von Lambda-Ausdrücken zu Variablen geschieht auf folgende Weise. Dabei ist zu beachten, dass sich diese NICHT - wie alle anderen Klassen in Java - einer Referenz von `java.lang.Object` zuweisen lassen, das würde einen Compiler Fehler verursachen.

Stattdessen werden die Ausdrücke grundsätzlich sogenannten **Functional Interfaces** aus dem Paket

```
java.util.function
```

zugewiesen. Diese sind eine Art von Typ und sind ein (normales) Interface, aber mit nur genau einer abstrakten Methode. Dies wird **SAM** = **Single Abstract Method Typ** genannt.

Konkrete Implementierungen dieser funktionalen Interfaces gibt es recht viele neue im Bereich der Lambdas, aber bereits vorher gab es solche auch schon, z.B. im Collections Framework mit dem

Interface `Comparator<T>`.

```
@FunctionalInterface
public interface BinaryOperator<T> {
    public abstract void apply(Object a, Object b);
}
```

Die **wichtigsten** funktionalen interfaces sind:

1. `Consumer<T>` - beschreibt eine Aktion auf einem Element vom Type `T` mittels der Methode `void accept(T)`
2. `Predicate<T>` - definiert eine Methode `boolean test(T)`. Sie berechnet für einen Eingabewert T ein bool'sches Ergebnis. Das lässt sich sehr gut für Filterbedingungen ausnutzen.
3. `Function<T,R>` - Definiert eine Abbildungsmethode durch `R apply(T)`, wodurch Transformationen gut ausgedrückt werden können, gebräuchlich besonders für die Extraktion eines Attributes aus einem komplexen Typ
4. `BiFunction<T,U,R>` und `BiConsumer<T,U>` - wie `Function`, nur mit zwei Eingabewerte `T` und `U`
5. `Supplier<T>` - Stellt ein Ergebnis vom Typ `T` bereit. Im Gegensatz zu `Function<T,R>` erhält ein Supplier keinen Eingabewert mit der Methode `T get()`

Zu jedem dieser funktionalen Interfaces gibt es einen Unit-Test zur **Demonstration** hier:

```
src/test/java/de/dhbw/funcprog/LambdaDemoTests.java
```

1.4. Streaming (API)

Das Streaming API stellt im Wesentlichen Methoden zur *schnellen, leicht lesbaren und bequemen* Verarbeitung von Daten im Sinne einer **Mehrfach-** oder auch **Batch**-Verarbeitung bereit, und zwar in Form sogenannter **Streams**.

Insbesondere zur verbesserten Verarbeitung von *Listen* aus dem Collections-Framework in Java wurden diese Komponenten mit Java 8 eingeführt

Aus diesem Grund wird auf einer *normalen* Liste auch in aller Regel zuerst die Methode

```
stream()
```

aufgerufen, die aus dem Objekttyp `Liste` einen `Stream` macht. Auf diesem *Stream* lassen sich dann weitere Methoden zur Verarbeitung der Listenelemente aufrufen, und zwar mittels der sogenannten

```
filter-map-reduce
```

Algorithmen genutzt, um diese Möglichkeiten zusammenzufassen.

Ein ausführliches Beispiel für den Gebrauch in **Listen**:

Lamdba + Streaming

```
    @Test
    public void canUseLambdaForLists() {
        // given
        List<String> trains = List.of("ICE 81", "RB 10", "IC 2027", "RE 39", "S8");

        // when - find IC's
        boolean matchesIC = trains.stream().anyMatch(e -> e.startsWith("IC"));

        // when - find regional trains
        String train = trains.stream()
                .filter(e -> e.contains("R"))
                .findFirst()
                .orElse("?");

        // then
        assertTrue(matchesIC);
        assertEquals("RB 10", train);
    }
```

Im Detail:

`Filter`

Filter werden durch die oben bereits genannten `Predicates` implementiert und in eine Methode namens `filter()` als *funktionales Argument* übergeben.

→ *Welchen Datentyp geben die Predicates zurück?*

Bei der Anwendung mehrerer oder komplexer Filteroperationen sollte allerdings die Performanz beachtet werden. Eine Einschätzung:

		Stream Size			
		10.000	100.000	1.000.000	10.000.000
Stream	Multiple Filters	0.2	3.53	30.67	313.98
Stream	Complex Condition	0.01	0.03	6.67	74.78
Parallel Stream	Multiple Filters	0.08	0.79	9.37	96.11
Parallel Stream	Complex Condition	0.01	0.01	2.3	24.8
Old For Loop	Complex Condition	0.01	0.01	1	10.64

Figure 1. Performanz bei komplexen Filteroperationen

`Map`

"Mapping" Operationen auf Listen **transformieren** diese.

Sie werden häufig durch die oben bereits genannten `Functions` implementiert und in eine Methode namens `map()` als *funktionales Argument* übergeben.

Methoden oder Klassen, deren Zweck die **Abbildung/Umwandlung von Klassen** in/auf andere Strukturen implementieren, werden dementsprechend sehr häufig auch *"mapping"* Methoden genannt, daher hier dieser Term hier ebenfalls genutzt.

Besonders die Veränderung der Datentypen von **Listenelementen** ist ein wichtiger Anwendungsfall beim Einsatz von `map(…)`.

Reduce

`Stream.reduce()` Operation **reduzieren** die Ausgangsdatenmenge.

Dies erfolgt grundsätzlich in Teilschritten:

- `Identity` – Ein Element mit einem initialen Wert für die Reduktionsoperation und der "default return value" wenn der Stream leer sein sollte.
- `Accumulator` – Eine Funktion mit zwei Parametern: ein Teilergebnis der Reduktionsoperation und das nächste Element des Streams.
- `Combiner` – Eine Funktion um die Teilergebnisse (der Reduktionsoperation) zu kombinieren wenn `reduce` parallelisiert wird.

Richtig nutzbringend ist oft erst die Kombination `filter`, `map` und `reduce` Operationen, um aus einfachen **Daten** am Ende **Informationen** zu gewinnen.

Dazu ein Beispiel:

Beispiel einer reduce() Operation

```
    @Test
    public void canCalculateTotalByReducing() {
        // given
        List<Integer> numbers = List.of(1, 4, 78, 3, 54, 19, 234);

        // when - reduce()
        //
        //             "startWert" (Identity)
        //                   |  "naechsteZahl"
        //                   |        |  "ZwischenErgebnis"
        //                   |        |        |
        //   Iteration 1:   (0)  +    1   =    1
        //   Iteration 2:    1   +    4   =    5
        //   Iteration 3:    5   +   78   =   83
        //   Iteration 4:   83   +    3   =   86
        //                    ... usw. ...
        //   Iteration n:  ...   +   243  =  393 ("total")

        // int startWert = 0;
        // Integer total = numbers.stream().reduce(startWert,
        //   (zwischenErgebnis, naechsteZahl) -> zwischenErgebnis + naechsteZahl);

        // vorheriges in kürzerer Form und besser lesbar!
        //   wobei: "Integer::sum" ist hier der "BinaryOperator<T> accumulator"
```

```
        Integer total = numbers.stream().reduce(0, Integer::sum);

        // then
        assertEquals(393, total);
    }
```

sowie als Beispiel für einen **fachlichen** Nutzen, der "Erkenntnisse" aus den fachlichen Daten ermittelt:

Fachliches Beispiel (good code?, bad code?)

```
@Test
public void canUseValidatorPredicates() {
    // given (Timetable shall be created for the year '2023')

    ZonedDateTime departure = DateTimeUtil.from("15.02.2023");
    Schedule schedule = Schedule.of("MA", "DA", departure, 45);

    // when - (a) direct implementation
    Predicate<Schedule> predicate = s -> s.getDeparture().getYear() == 2023;
    boolean testResult1 = predicate.test(schedule);

    // when - (b) concern 'validation' wrapped in a separate class
    boolean testResult2 = Validator.validate(schedule, isScheduledFor2023());

    // then
    assertTrue(testResult1);
    assertTrue(testResult2);
}
```

1.5. Demonstrationen

Die Unit-Tests zur **Demonstration** finden sich hier:

```
src/test/java/de/dhbw/funcprog/LambdaDemoTests.java
src/test/java/de/dhbw/funcprog/StreamingDemoTests.java
```

1.6. Übungen

Nutzt folgendes Package für die **Unit-Tests**:

```
src/test/java/de/dhbw/funcprog/FuncProgExerciseTests.java
```

Die im Test benutzten **Implementierungen** gehören in das Package:

```
src/main/java/de/dhbw/funcprog/exercises/*.java
```

Übung 1:

Implementiere eine Liste mit mehreren `boolean` Werten.

Nutze dann einen Lambda-Ausdruck, durch den die Listenelemente auf der Konsole ausgegeben werden können.

Übung 2a:

Implementiere den *"old fashioned way"* für die Berechnung einer Summe:

```
@Test
@DisplayName("Übung 2a: Calculate a total in old fashioned way")
public void exercise2a() {
    // given - a list of min. 5 arbitrary Integers

    // when - iterate over the list and calculate the total
    // for ( ... : ... ) { ... }

    // then - assert the correct total
}
```

Übung 2b:

Implementiere die Übung 2a mit modernen Mitteln, d.h. ersetze die "alte" `for-each` Loop durch einen **Lambda**-Ausdruck:

```
@Test
@DisplayName("Übung 2b: Calculate a total the modern way")
public void exercise2b() {
    // given - a list of min. 5 arbitrary Doubles

    // when - loop over the list and calculate the total

    // then - assert the correct total
}
```

Übung 3:

Für einen `Warenkorb` gilt es, die darin enthaltenen 3 `Produkte`nach ihrer `Produktkategorie` zu filtern.

Optional: Ermittle den gesamten `Preis` *des Warenkorbs als Summe der produkt-spezifischen Einzel-*`Preise`.

Erstelle einen Unit-Test und (eine oder mehrere) Testmethoden für die Teilaufgaben.

1.7. Tipps, Patterns & Best Practices

Predicates

`Predicates` sollten, wenn möglich, *benannt* werden, d.h. zum Beispiel anstelle von

```
list.stream().filter(i -> i >= 10)
```

besser gekapselt in einer Methode oder mit einer Variable

```
Predicate<Integer> isGreaterOrEqual10 = i -> i >= 10
```

oder

```
Predicate<Integer> isGreaterOrEqualTo(Integer number) {
   return i -> i >= number;
}
```