Inhaltsverzeichnis

1. Kurstag 3 - Generics, Streaming & funktionale Programmierung

Allgemeine Inhalte

- ▢ Generics
- ▢ Streaming API
- ▢ Funktionale Programmierung mit Lambda-Ausdrücken

Fachlicher Kontext

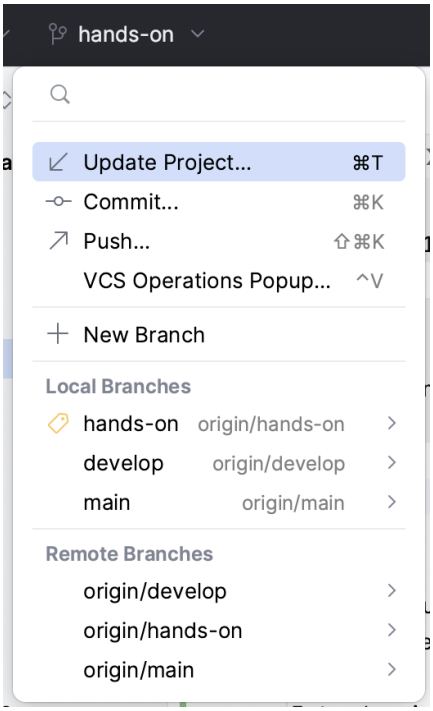
```
+ eher Theorie
+ Standard-Beispiele/-Übungsaufgaben
```

1.1. UPDATE

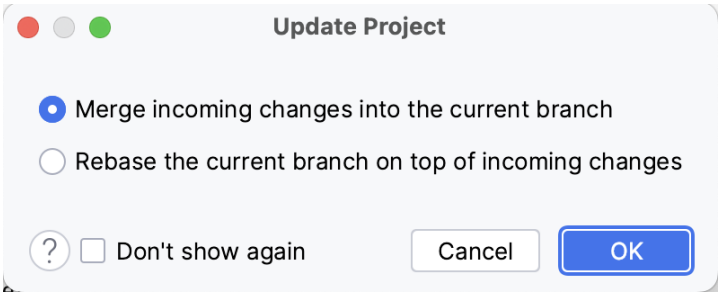
Das **Projekt** bzw. der "*lokale Workspace*", d.h. euer lokales Arbeitsverzeichnis, in dem alle Sourcen liegen, muss als allererstes zum Start in den Tag aktualisiert werden, d.h. ...

→ *"Update Project"*

Das geht am besten mithilfe der IDE im Menü oder über das GIT Icon:



Danach muss - im sich öffnenden Dialog - noch folgendes bestätigt werden: *merge incoming changes into the current branch*



Der Vorgang sollte mit einer Erfolgsmeldung abschließen.

1.2. Recap

Was bisher geschah ...

- ☐ Methoden überladen, überschreiben/übersteuern
- ☐ Collections API
- ☐ Beziehungsarten (Aufruf, Vererbung, Assoziation, ...)
- ☐ Assoziationen (Aggregation vs. Komposition im Code)

1.3. Generics

Generische Programmierung in Java ist durch Generics seit langem möglich. Der Begriff steht synonym für "parametrisierte Typen". Die Idee ist, zusätzliche Variablen für Typen einzuführen. Diese Typ-Variablen repräsentieren zum Zeitpunkt der Implementierung unbekannte Typen. Erst bei der Verwendung der Klassen, Schnittstellen und Methoden werden diese Typ-Variablen durch konkrete Typen ersetzt. Damit kann typsichere Programmierung meistens gewährleistet werden. In der Regel wird die Codemenge durch Generics reduziert (Prinzip: `DRY`), manchmal wird er allerdings auch schwerer wartbar und abnehmende Lesbarkeit. Die folgenden zwei Varianten finden sich in der Praxis am häufigsten:

- Java Generics `Klasse`
- Java Generics `Methode`

> *Viele Beispiele finden sich auch im Collections Framework, etwa die Interfaces `List<T>` oder `Map<K,V>`. Siehe dazu z.B. → Java 17 Package Documentation für `java.util` (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/package-summary.html)!*

Beispiel einer generischen Klasse

JAVA

```
1  public class Joiner<T> {
2
3      public String join(Collection<T> collection) {
4          StringBuilder builder = new StringBuilder();
5          Iterator<T> iterator = collection.stream().iterator();
6
7          builder.append("[");
8          while (iterator.hasNext()) {
9              T item = iterator.next();
10             final String formattedItem =
11                     iterator.hasNext()
12                             ? String.format("%s, ", item)
13                             : String.format("%s", item);
14             builder.append(formattedItem);
15         }
16         builder.append("]");
17
18         return builder.toString();
19     }
20 }
```

Zeile 1 macht die Klasse generisch, in Zeile 9 wird der unbekannte Typ genutzt.

Beispiel einer generischen Methode

JAVA

```
    public <T> String print(T data)
```

Bounded Generics

Oft kommen sogenannte **bounded generics** zum Einsatz. Dabei wird bei der Definition einfach die Superklasse angegeben, von welcher der generische Typ erben muss. Auf diese Weise wird der ansonsten *beliebige* Typ eingeschränkt, so dass der generische Typ zwar immer noch unbekannt ist, aber nicht von *jedem* Typ sein kann, sondern nur entsprechend der Einschränkung, z.B.

JAVA

```
public <T extends Number> add(T first, T second) { ... }
```

1.4. Funktionale Programmierung mit Lambda Ausdrücken

Funktionale Programmierung ist ein **Programmierparadigma**, in dem Funktionen im Mittelpunkt stehen, sie werden nicht nur definiert und angewendet, sondern auch wie Daten miteinander verknüpft, als Parameter verwendet und als Funktionsergebnisse genutzt.

Seit Java 8 sind Elemente aus der funktionalen Programmierung in der Sprache enthalten. Ermöglicht werden sie durch die **Lambda-Ausdrücke**, sowie - wie bereits gesehen - die Streams.

Die einfachste Form eines Lambda Ausdruckes ist:

```
parameter -> expression
```

Um mehr als einen Parameter zu nutzen, werden runde Klammern erforderlich:

```
(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.

Die Beispiele finden sich in

```
course-3/test/de/dhbw/course3/lambda/LambdaTest.java
```

z.B. für den Gebrauch in **Listen**:

Lamdba Ausdrücke

JAVA

```java
    @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);
    }
```

1.5. Streaming (API)

Zur verbesserten Verarbeitung von "Listen" in Java wurden spezielle Methoden durch "streaming" mit Java 8 eingeführt. Häufig wird der Begriff `filter-map-reduce` genutzt, um diese Möglichkeiten zusammenzufassen.

`Filter`

Bei der Anwendung mehrerer oder komplexer Filteroperationen sollte die Performanz beachtet werden. Hier 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. Methoden oder Klassen, deren Zweck die Abbildung von Klassen in andere Strukturen implementieren, werden häufig *"mapping"* Methoden genannt, daher hier dieser Term hier ebenfalls genutzt.

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

`Reduce`

`Stream.reduce()` Operation reduzieren die Ausgangsdatenmenge. Dies erfolgt 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

JAVA

```java
    @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);
    }
```

Weitere Beispiele finden sich in

```
course-3/test/de/dhbw/course3/streaming/StreamingTest.java
```

sowie zur Demonstration für einen **fachlichen** Nutzen, der "Erkenntnisse" aus den vorhandenen Daten ermittelt:

Fachliches Beispiel (good code?, bad code?)

JAVA

```java
    @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);
    }
```

1.6. Übungen

Wie immer, nutze einfach die Testklasse für die Übungen, hier unter `/course 3`:

```
/src/test/de/dhbw/course3/exercises/ExerciseTests.java
```

Übung 1

Implementiere eine Klasse `Workflow` mit einer statischen & generischen Methode `execute`, die beliebige Workflow-Schritte ausführen kann. Konkrete Workflowschritte erben von einer Klasse namens `Step`. Die "Ausführung" selbst soll hier lediglich die Ausgabe des Names des Workflow-Schrittes sein (der Klassenname)

Übung 2

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

JAVA
```java
    @Test
    @DisplayName("Übung 2: Calculate a total in old fashioned way")
    public void exercise2() {
        // given - a list of at least 10 random Integers

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

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

Übung 3

Für einen `Warenkorb` gilt es, deren enthaltene `Produkte` zu filtern und den gesamten `Preis` zu ermitteln:

Übungsfragen

In der nachstehenden Testklasse finden sich kleine "Quizfragen" für die Inhalte des Kurses 3:

```
<your-repo>/exam/test/de/dhbw/exam/course3/ExamTest.java
```

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

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