Inhaltsverzeichnis

1. Java 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. Dazu wird der sogenannte **Diamond-Operator** `<>` bei Klasse oder Methode genutzt:

```
List<T> list; ①
public Printer<T> { ...}
```

① `T` oft als Kürzel für `Type`

Erst bei der Verwendung der Klassen, Schnittstellen und Methoden im Code werden diese Typ-Variablen durch konkrete Typen durch den Entwickler 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`!

Beispiel einer generischen Klasse

```
public class Joiner<T> {

    public String join(Collection<T> collection) {
        StringBuilder builder = new StringBuilder();
        Iterator<T> iterator = collection.stream().iterator();

        builder.append("[");
        while (iterator.hasNext()) {
            T item = iterator.next();
```

```
            final String formattedItem =
                    iterator.hasNext()
                            ? String.format("%s, ", item)
                            : String.format("%s", item);
            builder.append(formattedItem);
        }
        builder.append("]");

        return builder.toString();
    }
}
```

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

DIe Nutzung einer generischen Klasse sind entsprechend so aus, der Typ `T` wird konkret angegeben:

```
Joiner<String> joiner = new Joiner<>();
```

Auch Methoden können generisch sein, sie werden ähnlich wie Klassen definiert:

Beispiel einer generischen Methode

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

1.1. 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, sodass der generische Typ zwar immer noch unbekannt ist, aber nicht von *jedem* Typ sein kann, sondern nur entsprechend der Einschränkung, z.B.

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

Hier kann z.B. der genutzte Datentyp ausschließlich ein Zahlen-Datentyp sein, der von `Number` erbt.

1.2. Wildcards

Im Rahmen der Generics kann man anstelle der Typvariable - oben z.B. `T` - durchaus auch die sogenannte **Wildcard** `?` nutzen. Das schafft Flexibilität hinsichtlich der spezifizierbaren Typen, verhindert aber die Nutzung der Typvariable selbst innerhalb der Klasse.

1.3. Demonstrationen

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

```
src/test/java/de/dhbw/generics/GenericsTests.java
```

Der zugehörige, in den Tests genutzte **Quellcode** findet sich hier:

```
src/main/java/de/dhbw/generics/demo/*.java
```

1.4. Übungen

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

```
src/test/java/de/dhbw/generics/ExerciseTests.java
```

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

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

Übung 1

Erstelle ein Interface für einen Taschenrechner, der die vier Grundrechenarten in Form von Methoden zur Verfügung stellt, also für…

- addieren,
- subtrahieren,
- multiplizieren und
- dividieren.

Der Taschenrechner sollte mit einem beliebigen Zahlentyp umgehen können.

Zahlentypen in Java haben eine gemeinsame Superklasse `java.lang.Number`.

Optional: Realisiere auch einen konkreten Taschenrechner, der das Interface implementiert, und schreibe dazu einen kleinen Test, der die Funktionsfähigkeit mindestens einer der Rechenarten am Beispiel auch mal testet.

Übung 2

Implementiere eine konkrete Klasse `Workflow`.

Diese Klasse soll eine statische, generische Methode `execute` bekommen, die beliebige Workflow-`Schritte` ausführen kann.

Die Workflow-Schritte sollen von einer Eltern-Klasse namens `Step` erben. Implementiere mindestens 2 konkrete Workflow-Schritte.

Realisiere für die "Ausführung" der `execute` Methode einfach eine Konsolenausgabe, z.B. des Names

des konkreten Workflow-Schrittes (d.h. der Klassenname).

1.5. Tipps, Patterns & Best Practices

- Bei Listen sollte man immer mittels Diamond-Operator `<>` den Datentyp für die Liste angeben
- Benutze den Wildcard-Type `?` (Bsp. `<? extends Number>`) wenn möglich