OCP Java SE 8

Lambda

Lambda-Ausdruck

- sieht ähnlich wie eine Deklaration einer Methode ohne Name und Ergebnistyp aus
  - `(a, b) -> b.compareTo(a)`

Lambda

- Vor Java 8
  - Ausführbarer Programmcode („Verhalten“) nur innerhalb einer Klasse in Form einer Methode definierbar

Lambda

- Ab Java 8
  - Lambda ist ein Behälter für Sourcecode ähnlich einer Methode, allerdings ohne Namen und ohne die explizite Angabe eines Rückgabetyps oder ausgelöster Exceptions.

    Vereinfacht ausgedrückt kann man einen Lambda am ehesten als anonyme Methode mit folgender Syntax und spezieller Kurzschreibweise auffassen

    (Parameter-Liste) -> { Ausdruck oder Anweisungen }

Lambda

- Auf die Angabe von Datentypen für die Parameter kann verzichtet werden. Lambdas nutzen die sogenannte Type Inference : Ähnlich wie beim Diamond Operator bei der Definition generischer Klassen. Dazu ermittelt der Compiler die passenden Typen aus dem Einsatzkontext

- Falls das auszuführende Stück Sourcecode ein Ausdruck ist, können die geschweiften Klammern um die Anweisungen entfallen.

- Ebenfalls kann dann das Schlüsselwort return weggelassen werden und der Rückgabewert entspricht dem Ergebnis des Ausdrucks

- Existiert lediglich ein Eingabeparameter, so sind die runden Klammern um den Parameter optional

Lambda

- Lambda-Ausdrücke lassen sich nicht auf ein Object abbilden. Folgende Code führt zum Compilerfehler:

```
Object o = () ->{
    System.out.println("Moin Moin");
};
```

Binding

- kann auf Klassen- und Instanzvariablen der umgebenden Klasse zugreifen (lesend und schreibend)
- lokale Variablen der umgebenden Methode müssen final oder effektiv final, (d.h. es gibt keine Zuweisung, die sie ändert) sein
- Lambdas repräsentieren ein Stück Funktionalität und haben keine Bindung zu einem Objekt. this innerhalb eines Lambdas hat die gleiche Bedeutung wie in den Zeilen direkt außerhalb davon

FunctionalInterface

- Grundlage jedes Lambda-Ausdrucks ist ein FunctionalInterface, SAM-Typ genannt
  (SAM = Single Abstract Method)
  - es hat nur eine abstrakte Methode
  - es kann statische Methoden enthalten
  - es kann default Implementierungen enthalten
  - Alle im Typ Object definierten Methoden können zusätzlich zu der abstrakten Methode in einem FunctionalInterface angegeben werden
  - es kann mit der Annotation @FunctionalInterface als solchen gekennzeichnet werden

FunctionalInterface

- Zum Einsatz von Lambdas sowie zur Ergänzung und Flexibilisierung wurde das JDK um diverse Functional Interfaces (also SAM-Typen) erweitert.

- Package: java.util.function

FunctionalInterface

- Consumer<T>
- Beschreibt eine Aktion auf einem Element vom Typ T
- Methode: void accept(T t)

FunctionalInterface

- Function<T,R>
- Beschreibt ein allgemeines Konzept von Transformationen
- Methode: R apply (T t)

FunctionalInterface

- BiFunction<T,U,R>
- Wie Function<T,R> bzw. Consumer<T>, jedoch mit jeweils zwei Eingabewerten vom Typ T und U
- Methode: R apply (T t, U u)

FunctionalInterface

- Supplier<R>
- Stellt ein Ergebnis vom Typ R bereit. Im Gegensatz zu Function<T,R> erhält ein Supplier<R> keine Eingabe. Ist eine Fabrik.
- Methode: R get ()

FunctionalInterface

- Predicate<T>
- Führt einen Test auf T durch und liefert true, wenn das Kriterium erfüllt ist, sonst false.
- Ein Prädikat ist eine Aussage über einen Gegenstand, die wahr oder falsch ist.
- Methode: boolean test(T t)

FunctionalInterface

- UnaryOperator<T>
- Ist eine spezielle Funktion bei der die Typen für „Eingang“ und „Ausgang“ gleich sind
- Erweitert Function
- Methode: T apply (T t)

FunctionalInterface

- BinaryOperator<T>
- Wie UnaryOperator aber mit zwei Parametern gleichen Typs
- Methode: T apply (T t1, T t2)

FunctionalInterface

- Es gibt Varianten, die auf die Verarbeitung primitiver Typen spezialisiert sind (int, long und double)
  - IntPradicate
  - DoubleFunction
  - LongSupplier
  - etc.