Praktikum Functional Programming

Einleitung

Ziele dieses Praktikums sind:

- Sie können Lambda Expressions schreiben
- Sie können eigene funktionale Interfaces schreiben und verwenden
- Sie können `Optional` sinnvoll anwenden
- Sie kennen Methoden-Referenzen und können diese einsetzen
- Sie kennen die wichtigsten Klassen und Methoden aus `java.util.stream` und `java.util.function` und können diese anwenden

Voraussetzungen

- Vorlesung Programmieren 2 – Functional Programming

Tooling

- Installiertes JDK 17+
- Gradle 7.6+

Struktur

Ein Praktikum kann verschiedene Arten von Aufgaben enthalten, die wie folgt gekennzeichnet sind:

[TU] – Theoretische Übung

Dient der Repetition bzw. Vertiefung des Stoffes aus der Vorlesung und als Vorbereitung für die nachfolgenden Übungen.

[PU] – Praktische Übung

Übungsaufgaben zur praktischen Vertiefung von Teilaspekten des behandelten Themas.

[PA] – Pflichtaufgabe

Übergreifende Aufgabe zum Abschluss. Das Lösen dieser Aufgaben ist Pflicht. Sie muss bis zum definierten Zeitpunkt abgegeben werden, wird bewertet und ist Teil der Vornote.

Zeit und Bewertung

Für das Praktikum stehen die Wochen gemäss den Angaben in Moodle zur Verfügung.
Je nach Kenntnis- und Erfahrungsstufe benötigen Sie mehr oder weniger Zeit. Nutzen Sie die Gelegenheit den Stoff zu vertiefen, auszuprobieren, Fragen zu stellen und Lösungen zu diskutieren (Intensive-Track).
Falls Sie das Thema schon beherrschen, müssen Sie nur die Pflichtaufgaben lösen und bis zum angegebenen Zeitpunkt abgeben (Fast-Track).

Die Pflichtaufgaben werden mit 0 bis 2 Punkten bewertet (siehe *Leistungsnachweise* auf Moodle).

Auch wenn Sie das Thema schon beherrschen, prüfen Sie bitte Ihr Wissen über das Design Pattern *Chain of responsibility*.

1. Functional Interfaces [TU]

Sie können Textantworten in der Datei `solutions-sheet.adoc` (eine Muster-Datei ist im Code-Verzeichnis) oder `solutions-sheet.md` im Root-Verzeichnis der Übung sammeln.

Java bietet für viele Zwecke im Package java.util.functional Functional Interfaces.

a. Welche Interfaces aus dem Package `java.util.function` können Sie alles nutzen, um
   - die mathematische Funktion f(x) = x ^ 2 - 3 für Zahlen des Typs `long` abzubilden?
   - um den Zinsfaktor (double) für n (int) Jahre bei einem Zinssatz von p Prozent (float) zu berechnen mit der Formel zf = (1 + p / 100)^n ?
   - ein Objekt vom Typ `Person` (ohne Parameter) zu generieren?

b. Welche Eigenschaft muss eine Funktion haben, damit Sie ein eigenes Interface schreiben müssen, also keines der in `java.util.function` vorhandenen Interfaces verwenden können?

c. Welche der Aussagen stimmen für ein funktionales Interface?
   - ☐ Es ist ein Java-Interface (Schlüsselwort `interface` im Code)
   - ☐ Es hat **genau eine** abstrakte Methode
   - ☐ Das Interface **muss** mit `@FunctionalInterface` markiert sein
   - ☐ Es hat **keine** default-Methoden (Schlüsselwort `default`)

d. Welche Aussagen stimmen?
   - ☐ Zu **jedem** funktionalen Interface können Lambda-Ausdrücke (*lambda expressions*) geschrieben werden
   - ☐ Ein Lambda-Ausdruck kann **ohne** passendes funktionales Interface erstellt werden
   - ☐ Eine Variable vom Typ `Optional` kann nie `null` sein.

2. Übungen auf der Stepik-Plattform [PU]

Starten Sie den Kurs Java. Functional programming. Dazu müssen Sie dort ein Konto anlegen. Die Plattform ist von der ZHAW unabhängig.

Sie können dort alle Aufgaben direkt im Browser lösen. Oft ist es aber zweckmässig, den Code in die IDE zu übernehmen und die Lösung dort zu entwickeln.

Auf dieser Plattform wird Ihre Lösung online geprüft und Sie erhalten Feedback, ob Ihre Lösung alle Tests erfüllt.

Wenn Sie eine funktionierende Lösung abgegeben haben, erhalten Sie Zugriff auf Kommentare und Lösungen anderer Personen. Vergleichen Sie Ihre Lösung, Sie können viel von anderen Lösungen lernen.

Übungen zu Functional Interface und Lambda Expression

Lösen Sie die folgenden Übungen:

a. 2.2 Identify the correct lambdas and method references
b. 2.3 Writing simple lambda expressions
c. 2.4 Too many arguments
d. 2.7 Writing closures
e. 3.2 Replacing anonymous classes with lambda expressions
f. 3.3 Matching the functional interfaces
g. 3.5 Your own functional interface

Übungen mit Streams

h. Lösen Sie 2.5 Calculating production of all numbers in the range

   Tipp: Verwenden Sie die passend Methode `.reduce(…)`

i. Lösen Sie 2.6 Getting distinct strings

j. Lösen Sie die Übung 3.7 Composing predicates. Die Aufgabe verlangt, dass Sie ein `IntPredicate` erstellen, das alle `IntPredicate` aus übergebenen Liste `predicates` mit der Oder-Funktion (or) verknüpft. Eine mögliche Lösung ist

```
class Predicate {
  public static IntPredicate disjunctAll(List<IntPredicate> predicates) {
    IntPredicate disjunct = x -> false;
    for(IntPredicate currentPredicate: predicates) {
      disjunct = disjunct.or(currentPredicate);
    }
    return disjunct;
  }
}
```

   Eine Anwendung könnte sein:

```
class Predicate {
  public static void main(String[] args) {
    IntPredicate isEven = x -> x % 2 == 0;
    IntPredicate isDividableBy3 = x -> x % 3 == 0;
    List<IntPredicate> predicateList = List.of(isEven, isDividableBy3);
    IntPredicate disPredicate = disjunctAll(predicateList);
    IntStream.range(1, 10).forEach(i ->
      System.out.printf("%2d -> %s%n", i, disPredicate.test(i)));
  }
}
```

   Suchen Sie jedoch eine Lösung, die mit Streams arbeitet. Sie finden Tests und ein Gerüst für die Aufgabe in `code/Stepik` in der Klasse `ComposingPredicate`.

   Wenn Sie eine Lösung gefunden haben, überlegen Sie sich, wie viele Funktionen (`IntPredicate`) beim Aufruf von `.test()` ausgewertet werden. Lässt sich dies reduzieren?

k. Lösen Sie die folgenden Aufgaben mit Streams:
   - 4.6 Numbers filtering - beachten Sie die Methode IntStream.concat
   - 4.8 Calculating a factorial
   - 4.9 The sum of odd numbers
   - 5.3 Collectors in practice: the product of squares

     In den Folien der Vorlesung sind die `Stream.reduce()`-Methoden aufgeführt. In der Aufgabe wird aber `Stream.collect(`*collector*`)` verwendet und Sie müssen nur den *collector* angeben. Die entsprechenden Funktionen in der Collectors-Klasse heissen `Collectors.reducing()`. Ihre Lösung lautet also `Collectors.reducing(…)`
   - 5.5 Almost like a SQL: the total sum of transactions by each account

     Tipp: Auch wenn steht, dass die Form `Collectors.reducing` verwendet werden kann, ist die Methode `reducing` nicht die Lösung, sie benötigen eine andere Methode aus der Klasse `Collectors`.

3. Design Pattern Chain of responsibility [PU]

Lernen Sie das Pattern Chain of Responsibility kennen.

In der Übung 3.9 The chain of responsibility pattern in the functional style setzen Sie dieses Pattern funktional um.

Das ist eine aufwändige Aufgabe, nehmen Sie sich Zeit dafür.

4. Company Payroll [PA]

Bei dieser Aufgabe geht es darum alles mit Streams zu lösen. Verwenden Sie keine for-, do-, oder while-Schleifen.

Im Package `ch.zhaw.prog2.functional.streaming` finden Sie einige Klassen. Diese ermöglichen einer Firma den Angestellten die Löhne auszubezahlen. Zu den Klassen sind auch passende Tests für die Klassen vorhanden. Für die Tests werden die Objekte mit generierten Daten angereichert.

Sie sollen nur die folgenden Klassen anpassen:

- `Company`
- `PayrollCreator`
- `PayrollCreatorTest` **`Student`** (do not modify `PayrollCreatorTest`)
- `CompanyTest` **`Student`** (do not modify `CompanyTest`)

Lösen Sie mit Hilfe von Streams und basierend auf diesem existierenden Code die folgenden Aufgaben:

a. Mit `Company.allEmployees()` erhalten Sie alle Angestellten.

   Implementieren Sie die Methoden `Company.getDistinctFirstnamesOfEmployees()` und `Company.getDistinctLastnamesOfEmployees()`.

   Die dazugehörigen Tests sind in `CompanyTest` bereits vorhanden.

   

   Die Implementation benötigt keine Hilfsvariablen. Sie können die Implementation mit `return getAllEmployees().stream()` starten.

b. Mit `Employee.isWorkingForCompany` können Sie prüfen, ob der Angestellte noch für die Firma tätig ist. Implementieren Sie `Company.getEmployeesWorkingForCompany()`.
   Der dazugehörige Test ist in `CompanyTest` bereits vorhanden.

c. Als Nächstes sollen alle Angestellten mit dem Attribut `Employee.isFemale` ermittelt werden. Da dies zu ähnlichem Code wie in der vorherigen Aufgabe führt, realisieren Sie eine generischere Methode `Company.getEmployeesByPredicate(Predicate<Employee>)`. Die dazugehörigen Tests schreiben Sie in der Testklasse `CompanyTestStudent`. Als Tests schlage ich vor zu prüfen, ob die Summe der Angestellten mit dem Attribut `isFemale` und ohne dieses Attribut gleich der Summe aller Angestellten ist.

d. Nachdem `Company` uns Methoden für den Zugriff auf die Angestellten bietet, kümmern wir uns um die Lohnzahlungen. Die Klasse `Payroll` sammelt `Payment` in einer Liste. In der Klasse `PayrollCreator` schreiben Sie die dazu nötigen Methoden.

   Implementieren Sie die Methode `PayrollCreator.getPayrollForAll()`, die eine `Payroll` für alle Angestellten erstellt, für die `Employee.isWorkingForCompany` gesetzt ist. Verwenden Sie die Methode `Company.getPayments`.

   Einen passenden Test finden Sie in `PayrollCreatorTest`.

e. Wie hoch ist nun die Lohnsumme? Implementieren Sie `PayrollCreator.payrollValueCHF()`.

   Da verschiedene Währungen verwendet werden, müssen die `Payment` mit der Methode `CurrencyChange.getInNewCurrency` zu CHF gewandelt werden.

f. Nun sollen noch die Summen pro Währung ermittelt werden. Implementieren Sie die Methode `PayrollCreator.payrollAmountByCurrency`.

   Ein Ansatz dazu kann Ihnen das Tutorial über Reduction mit Streams geben.

   Schreiben Sie einen Test dazu in `PayrollCreatorTestStudent`. Verwenden Sie Mocking. Ein Positiv-Test, der prüft, dass die Währungen bei der Summenbildung korrekt berücksichtigt werden, reicht für diese Aufgabe aus.

g. In der Methode `Company.getPayments(Predicate)` ist bisher nicht berücksichtigt, dass der 13. Monatslohn nicht gleichmässig über das Jahr ausbezahlt wird.

   

   Bei einer Anstellung mit einem 13. Monatslohn wird zu den 12 monatlichen Lohnzahlungen ein weiteres Monatsgehalt ausbezahlt. Das monatliche Gehalt ist dann 1/13 des Jahresgehalts. In der Regel wird der 13. Monatslohn im Dezember ausbezahlt.

   Der 13. Monatslohn soll nur im Dezember ausbezahlt werden. Zudem gibt es gelegentlich andere Anpassungen, z.B. 5% firmenweite Sondervergütung. Um dies flexibel definieren zu können, soll die anzuwendende Lohnberechnung in einer Funktion übergeben werden.

   Orientieren Sie sich an der Funktion `Company.getPayments(Predicate)` und implementieren Sie `Company.getPayments(Predicate, Function)`.

   Implementieren Sie auch die dazu passenden Funktionen `Company.paymentForEmployeeDecember` für Zahlungen mit dem 13. Monatslohn im Dezember und `Company.paymentForEmployeeMonthly` für Zahlungen ohne 13. Monatslohn. Die dazu nötigen Deklarationen finden Sie in `Company` am Anfang der Klasse.

Abschluss

Stellen Sie sicher, dass die Tests mit `gradle test` erfolgreich laufen und pushen Sie die Lösung vor der Deadline in Ihr Abgaberepository.