Übungen – Funktionale Interfaces, Lambdas, Stream-API und IO

Aufgabe 1 – Warenverwaltung und Lambdas

(a) Sortierung eines Warenkatalogs mittels Lambda-Ausdrücken

Für die Sortierung wurde in der vorherigen Übung für jede Art ein eigener `Comparator` als statisch eingebettete private Klasse implementiert. Mit der Einführung von Lambdas kann darauf verzichtet werden und der `Comparator` als ein Lambda-Ausdruck direkt einer Variablen zugewiesen oder einer Methode als Parameter mitgegeben werden.
Ersetzen Sie in Ihren bisherigen Sortier-Funktionen die Sortier-Logik durch Lambda-Ausdrücke. Testen Sie Ihre Implementierung.

(b) Einsatz eines Consumers

Es soll nun im Warenkatalog die Möglichkeit geschaffen werden, eine prozentuale Preiserhöhung auf die Waren anzuwenden. Implementieren Sie im Warenkatalog eine neue Funktion `void erhoehePreise(double erhoehung)`. Iterieren Sie in dieser Methode mit Hilfe der in Java 8 neu eingeführten Methode `forEach(Consumer<Ware> action)` über die `Ware`-Elemente und berechnen Sie den neuen Preis. Ersetzen Sie den alten Preis durch den neu berechneten.
Testen Sie die Funktion, indem alle Waren im Warenkatalog einer Preiserhöhung von z.B. 5 % unterziehen.

(c) Einsatz eines Predicates

In der Praxis wird eine Preiserhöhung nicht unbedingt für den gesamten Warenkatalog vorgenommen werden. Erweitern Sie Ihre Methode `erhoehePreise()` um einen Parameter vom Typ des in der Schulung vorgestellten funktionalen Interfaces `Predicate`. Es soll nun nur bei den Waren, die dem Predicate entsprechen, der Preis erhöht werden.
Testen Sie die Funktion, indem Sie einen Teil Waren (z.B. alle Waren die eine bestimmte Zeichenkette in der Bezeichnung haben) im Warenkatalog einer Preiserhöhung von 5 % unterziehen.

Hinweis: es soll hier noch ohne das Stream-API gearbeitet werden!

(d) Verkettung von Comparatoren

Sollten Sie aus der letzten Übung die gestufte Sortierung mit Hilfe des stabilen Sortierverfahrens implementiert haben, das mehrfach hintereinander aufgerufen wird:

Das Interface `Comparator` stellt die Methode `thenComparing` zur Verfügung, mit Hilfe derer eine Verkettung von `Comparator`-Objekten möglich ist. Schauen Sie sich die Dokumentation des Interfaces an. Setzen Sie diese Variante für die mehrstufige Sortierung ein.

Aufgabe 2 – Warenverwaltung und das Stream-API

Die Klasse Ware wird um ein enum-Attribut `warengruppe` erweitert. Eine Warengruppe besitzt ein Attribut `bezeichnung`. Mit Hilfe des Konstruktors wird dieses Attribut befüllt. Die `toString`-Methode gibt die `bezeichnung` zurück.

(a) Erweiterung der `Ware`

Implementieren Sie den enum-Typ `Warengruppe` (Warengruppen könnten z.B. Bekleidung, Werkzeug oder Schuhe sein) und erweitern Sie die Klasse `Ware`, so dass diese ihre zugehörige Warengruppe kennt. Erweitern Sie dafür auch den Konstruktor.

(b) Filtern des Warenkatalogs

Es soll die Möglichkeit geschaffen werden, die Waren einer vorgegebenen Warengruppe herausfiltert. Implementieren Sie in der Klasse `Warenkatalog` eine Methode
`List<Ware> filtern(Warengruppe wg)`
in welcher Sie mit Hilfe eines `Predicate` die entsprechenden Waren herausfiltern. Geben Sie die gefilterten Elemente in einer neuen Liste zurück.

(c) Filter des Warenkatalogs erweitern

Für jeden Filter eine eigene Methode zu implementieren kann u.U. zu sehr viel Code führen. Erweitern Sie die in der vorherigen Aufgabe erstellte Methode `filtern(Warengruppe wg)`, dass dieser Methode eine beliebige Bedingung (`Predicate`) als Parameter übergeben werden kann.
Testen Sie die Methode, in dem Sie nach Waren, die einer bestimmten Warengruppe angehören und die eine bestimmte Preisgrenze überschreiten, suchen.

(d) Mappen des Warenkatalogs

Erweitern Sie die Klasse `Warenkatalog` um eine Methode, welche ein Mapping durchführt. Es soll mit Hilfe der Stream-Methode `map(Function<Ware, String>)` eine Liste zurückgegeben werden, welche die Warenbezeichnungen enthält. Duplikate von Bezeichnungen sollen dabei herausgelöscht werden.

(e) Existenz von Waren prüfen

Fügen Sie der Klasse `Warenkatalog` eine weitere Methode hinzu:
`boolean warenVorhanden(Predicate<Ware> bedingung)`
Dabei soll abgeprüft werden, ob der `Warenkatalog` Waren enthält, die der gegebenen Bedingung entsprechen.
Testen Sie die neue Methode, indem Sie z.B. nach danach suchen, ob es Artikel gibt, deren Preis über 18,00 EUR liegt.

(f) Komplexes Streaming des Warenkatalogs

Implementieren Sie im `Warenkatalog` eine Funktion
`Map<Warengruppe, Integer> anzahlWarenJeWarengruppe()`. Geben Sie eine Map zurück, welche als Schlüsselwerte Warengruppen besitzt und deren Values die Anzahl der Artikel der jeweiligen Warengruppe enthält.

Implementieren Sie zunächst diese Funktionalität „klassisch“ und danach mit Hilfe von Stream-API und Lambdas. Bewerten Sie die Lösungen im Hinblick auf Les- und Wartbarkeit und intuitives Code-Verständnis. Wo müssen Sie ggf. mehr interne Dokumentation anbieten?

Hinweis: Für die Stream-Lösung dieser Aufgabe wird die `Stream`-Methode

```
collect(Collector<? super T, A, R> collector)
```

benötigt. Schauen Sie sich außerdem die beiden statischen **Collectors**-Methoden

```
groupingBy(Function<? super T, ? extends K> classifier, Collector<? super T, A, D> downstream)
```

und

```
summingInt(toIntFunction<? super t> mapper)
```

an. Mit Hilfe von `classifier` werden die Schlüsselwerte der zu entstehenden Map definiert. `downstream` ist ein weiterer `Collector`, mit Hilfe dessen z.B. Objekte auf andere Werte (z.B. `int`) gemappt werden können.

Aufgabe 3 – Der Warenkatalog wird persistiert

(a) Schreiben

Erstellen Sie für den Warenkatalog eine Methode `save()`, die in eine Datei mit vordefiniertem Namen ins User-Verzeichnis den Inhalt des Warenkatalogs schreibt.

(b) Formatiertes Schreiben

Überlegen Sie sich ein sinnvolles Datei-Inhaltsformat, so dass im nächsten Schritt auch eine Einleseoperation implementiert werden kann.

(c) Lesen

Der Warenkatalog soll nun ebenfalls wieder eingelesen werden, d.h. beim Einlesen ("`load()`") wird erst der Katalog geleert, die Datei ausgewertet und die Waren-Objekte im Katalog neu angelegt.

Aufgabe 4 – Dateisystemoperationen

(a) Suche nach Dateien

Schreiben Sie ein Java-Programm, das ausgehend von einem in der Kommandozeile übergebenen Startverzeichnis rekursiv alle Unterverzeichnisse besucht und nach Dateien mit einer ebenfalls in der Kommandozeile übergebenen Endung sucht und diese mit vollem Pfad ausgibt.
Verwenden Sie in der ersten Version die Klasse `java.util.File`.

(b) Streaming

Implementieren Sie die gleiche Funktionalität wie unter (a), nur diesmal mit Hilfe der Methode `java.nio.Files.list()`.

(c) Luke FileWalker

Implementieren Sie die gleiche Funktionalität wie unter (a), nur diesmal mit den Mechanismen von `java.nio.Files.walkFileTree(…)`.