Übungen – Generics, Enums, Innere Klassen

Aufgabe 1 – Generics im Zoo

Da Stuttgart recht bekannt für seine Wilhelma ist, muss jede ernsthafte Vorlesung sich natürlich irgendwann diesem Thema widmen.

Natürlich dürfen Sie, je nach Vorliebe und Landsmannschaft, auch andere Hype-Eisbären wie Flocke oder Wilbär verwenden. Es müssen auch keine Eisbären sein, Elche wären ebenfalls gestattet. Oder Tapire.

Teil 1 – Knut

Schreiben Sie eine allgemeine Klasse `Tier` mit einem Attribut für den Namen des Tiers, die `toString()`-Methode und einen passenden Konstruktor.

Schreiben Sie die Klasse `Eisbär`, die von `Tier` abgeleitet ist und vermutlich nur den Konstruktor implementieren muss.

Teil 2 – Der Zoo

Unser Zoo wird eine Spezialvariante eines Zoos, denn er kann alle Tiere oder auch nur Unterarten aufnehmen. D.h. es muss möglich sein, einen Zoo nur für Eisbären anzulegen, in den dann auch kein Erdferkel darf.

Folgende Methoden enthält der Zoo:

1. Konstruktor
2. Methode `fuegeHinzu(...)`, damit Tiere in den Zoo aufgenommen werden können.

Testen Sie, ob folgender Code fehlerfrei läuft:

```
Eisbaer knut = new Eisbaer("Knuuut");
Eisbaer lars = new Eisbaer("Lars"); // Knuts Vater

Zoo<Eisbaer> zoo = new Zoo<Eisbaer>();

zoo.fuegeHinzu(knut);
zoo.fuegeHinzu(lars);
```

Teil 3 – Listen für den Tierarzt

Damit der Tierarzt alle Tiere im Zoo untersuchen kann, lässt er vom Zoo zwei von ihm vorbereitete Liste ausfüllen. Implementieren Sie im Zoo eine Methode `schreibeAufListe(...)`, so dass der folgende Code funktioniert:

```
Eisbaer knut = new Eisbaer("Knuuut");
Eisbaer lars = new Eisbaer("Lars"); // Knuts Vater

Zoo<Eisbaer> zoo = new Zoo<Eisbaer>();

zoo.fuegeHinzu(knut);
zoo.fuegeHinzu(lars);

// Achtung: der Tierarzt hat 2 Listen, eine nur für Eisbären
// (von mehreren Zoos) und eine für alle Tiere dieses Zoos.
// Beides soll funktionieren
List<Eisbaer> eisbaerenListe = new ArrayList<>();
zoo.schreibeAufListe(eisbaerenListe);

List<Tier> allgemeineListe = new ArrayList<>();
zoo.schreibeAufListe(allgemeineListe);

for (Tier tier : allgemeineListe)
{
    System.out.println(tier);
}
```

Aufgabe 2 – Generics im Warenkatalog

Wir stellen nun den Warenkatalog auf typisierte Collections um. Verändern Sie auch den Rückgabetyp der `alle...()`-Methoden, damit folgendes funktioniert:

```
for (Ware ware : this.alleWarenNachBezeichnung())
{
  System.out.println(ware);
}
```

Auch die Comparator-Implementierungen erhalten Typ-Parameter.

Die obige Schleife setzt eine passende Implementierung der `toString()`-Methode von Ware voraus. Falls Sie diese also bislang noch nicht genutzt haben: jetzt ist der Zeitpunkt gekommen!

Aufgabe 3 – Umbau der Sortierung

(a) Veränderung der Nutzung

Erklären Sie alle zeigeKatalog...-Methoden als deprecated, da wir für Sortierungen gleich eine etwas andere Struktur wählen werden.

(b) Iterable

Nachdem die zeige-Methoden deprecated sind, soll der Warenkatalog selbst für Nutzer der Klasse einfach iterierbar sein. D.h. die Darstellungsfunktionalität wird aus der Klasse herausgelöst. Die Klasse dient nur noch als reine Datenverwaltungseinheit. Dies ist ein sinnvoller Schritt, weil in „guter" Software die Datenverwaltung immer von der Datendarstellung getrennt sein sollte. Dadurch bleibt das Datenmodell unabhängig von einer eventuellen Darstellung auf Drucker, Bildschirm, HTML-Seite, graphisch aufwendiger GUI etc.

Unternehmen Sie die notwendigen Schritte, dass der Warenkatalog wie folgt genutzt werden kann:

```
Warenkatalog katalog = ...;
for (Ware ware : katalog)
{
  System.out.println(ware);
}
```

(c) Ein Enum für die Sortierung

In Aufgabe (b) ist nicht gleich ersichtlich, in welcher Sortierung bzw. Reihenfolge über den Warenkatalog iteriert wird. Erweitern Sie die Klasse um eine Methode

`public void setSortierkriterium(Sortierkriterium k)`
Als Sortierkriterien soll es `NACH_NUMMER`, `NACH_BEZEICHNUNG`, `NACH_PREIS` geben.

Erzeugen Sie das Enum entsprechend und bauen Sie die Klasse so um, dass nach Setzen eines Sortierkriteriums die unter (b) gezeigte Schleife die Waren in der geforderten Reihenfolge abarbeitet. Falls die Methode nicht aufgerufen wird, soll die Sortierung nach Warennummern erfolgen.

(d) Mehrere Sortierkriterien

Was muß im aufrufenden Code und was in der Katalog-Implementierung getan werden, um folgendes zu erreichen:

```
katalog.setSortierkriterium(NACH_BEZEICHNUNG, NACH_PREIS);
// Bei einer Ausgabe wird zunächst nach Bezeichnung
// sortiert. Wenn die Bezeichnung gleich sein sollte, wird
// zusätzlich noch nach Preis sortiert.
```
Es existieren hier natürlich – aufgrund auch der wenig gemachten Vorgaben – viele Implementierungsmöglichkeiten. Eventuell können Sie es sich zu nutze machen, dass die Methode

`Collections.sort()`
einen stabilen Sortiermechanismus implementiert, d.h. gleiche Elemente werden nie umsortiert. Dann könnte man mit einer geschickt gewählten Reihenfolge der Sortierungen... Oder ganz anders…

Aufgabe 4 – Innere Klassen

In der Anwendung gibt es die drei verschiedenen Comparator-Klassen, die für die unterschiedlichen Sortierungen benötigt werden.
Diese Klassen sind in ihrer Verwendung so eng mit der Klasse `Warenkatalog` verbunden, dass es keinen Grund gibt, sie anderen Klassen anzubieten. Sie sind Kandidaten für eingebettete Klassen der Klasse `Warenkatalog`.

(a) Statisch eingebettete Klassen

Da die Comparator-Klassen nur im Zusammenhang mit der Katalog-Klasse gebraucht werden, kann man sie zu statisch eingebetteten privaten Klassen machen.

(b) Anonyme Klassen

Es wird von diesen Klassen eigentlich jeweils nur ein einziges Objekt gebraucht. Eventuell haben Sie dies bereits in der Aufgabe vorher schon eingeführt. Nun versuchen wir, die statische eingebetteten Klassen zu anonymen Klassen zu machen, indem wir die Stellen mit dem `new` durch Implementierungen der anonymen Klasse ersetzen.
Betrachten Sie hinterher den Code hinsichtlich Übersichtlichkeit und Lesbarkeit.

(c) Refactoring

Vielleicht sind die Umbauten in (b) doch nicht so schön. Die Warenkatalog-Implementierung wird zu einem sehr großen Brocken. Man fragt sich, ob die Modularisierung innerhalb eines Pakets nicht doch die bessere ist, anstatt innerhalb der Klasse zu bündeln.
Um die Umbauten von (b) rückgängig zu machen, können wir uns gleich die netten Refactoring-Features unserer IDE anschauen.

Hinweise für Eclipse:
Klicken Sie auf dem inneren Enum-Typ rechts und wählen Sie *Refactor→Move Type to New File...* Betrachten Sie die Änderungen und führen Sie diese durch. Es wird wieder ein externer Typ erstellt und alle Vorkommen des Enums werden syntaktisch angepasst.
Klicken Sie nun nacheinander auf die Implementierungen der anonymen Klassen rechts und wählen Sie *Refactor→Convert Anonymous Class to Nested...* Wählen Sie sowohl final als auch static aus. Sie müssen an dieser Stelle natürlich einen Typnamen angeben, denn die anonyme Klasse hat sich ja gerade dadurch ausgezeichnet, keinen zu besitzen.

Testen Sie, ob noch alles funktioniert.