PGCert IT: Programming for Industry

Lab 09: Generics, Collections & Enums

Exercise One: Simple Generic Collections

a. What is the output of the following code?

```
ArrayList list = new ArrayList();
Character letter = new Character('a');
list.add(letter);

if (list.get(0).equals("a")) {
	System.out.println("funny");
} else {
	System.out.println("Not funny");
}
```

b. What is the output of the following code?

```
ArrayList<Point> list = new ArrayList<Point>();
Point pt1 = new Point(3, 4);
list.add(pt1);
Point pt2 = list.get(0);
pt2.x = 23;
if (pt2 == pt1) {
	System.out.println("Same object");
} else {
	System.out.println("Different object");
}
```

c. What is the error of the following code?

```
ArrayList list = new ArrayList();
list.add('a');
list.add(0, 34);
String c1 = (String) list.get(1);
```

Exercise Two: An Array of Strings

```
String[] array = {"ONE", "TWO", "THREE", "FOUR", "FIVE", "SIX", "SEVEN"};
```

Given the array of strings above, convert the array of strings to an ArrayList.

Then, write methods that make each string in the array list all lowercase. Implement the method in three different ways:

a. Using a loop over the index values
b. Using an enhanced for loop
c. Using an iterator

Exercise Three: Arrays to Lists

Examine the code in the `ex03` package. The `ExerciseThree_Arrays` class contains a complete program which has three methods:

- **`union`** – takes two arrays and returns an array containing all elements in the first array plus all elements in the second array.
- **`intersection`** – takes two arrays and returns an array containing all elements in the first array which are also in the second array.
- **`difference`** – takes two arrays and returns an array containing all elements in the first array which are *not* in the second array.

The class also contains a start method which creates some test arrays and runs them through each of these methods for testing purposes, and prints out the result.
As you can see from the implementations of some of those methods, arrays perhaps aren't the most ideal data structures for doing this kind of thing! In two of the methods we even have to loop twice – once just to figure out how big the result array should be.

In the `ex03` package, there is also an `ExerciseThree_Lists` class which is intended to implement identical functionality to the first class, but do so using Lists instead of Arrays. The start method in this class is almost complete, except that two lines are commented out, as you've not implemented the corresponding methods yet.

For this exercise, implement the `union`, `intersection` and `difference` methods in the `ExerciseThree_Lists` class. A skeleton for the `union` method has already been provided.

Once that's done, uncomment the two commented lines of code in the `start` method to run and test your program.

Hint: You should be able to write a lot less code in your three implementations compared to the Array implementations, by making use of appropriate `List` and `Collections` methods.

Exercise Four: Pancakes

The restaurant, **PancakeTopia**, is a very unusual, yet popular restaurant. Every day for lunch, they make a certain number of pancakes at random and put them in a large **stack**. Customers who want some delicious pancake-y goodness for lunch form a large **queue** outside. In turn, the restaurant lets one customer into the restaurant, who sits down at the table and eats as many pancakes as they want from the top of the stack. One that person is done, the next person sits down, and so on, until PancakeTopia runs out of either pancakes or customers for the day. Sometimes, customers at the end of the queue might not get fed – but PancakeTopia remains in business since their pancakes are the best in the land – the reward is worth the risk!

In this exercise, we'll complete a program which models a typical lunchtime at PancakeTopia. The nearly-complete application is located in the `ex04` package, and example outputs for the *complete* program (once you've made the necessary changes) can be found in the files `PancakeTopia-ExampleOutput-01.txt` and `PancakeTopia-ExampleOutput-02.txt`, which are located directly in the project directory. Here are the steps to go through to complete this exercise.

Step 1: Understanding

Have a look at the code and see what's there. Try to get an idea of how everything fits together. Perhaps try to draw some quick UML diagrams to assist you. This is often a good first step when trying to learn any new system.

Step 2: Getting the customers to form a queue

In the PancakeApp class, you'll have noticed a method called `createCustomerQueue()` which creates a random number of customers, and should add those customers to a queue. For this step, complete this method. Firstly, initialize the `queue` variable to something other than `null`. Secondly, call one of `queue's` methods at the marked location to add the generated `customer` to that queue. Remember that queues are **First-In-First-Out (FIFO)** – meaning, the first customer to line up will be the first to get served.
Next, in `PancakeShop`'s `serveLunch()` method, there's a loop where we want to continually get the customer at the front of the queue. Complete that line by using an appropriate `poll` statement. Remember that we should be de-queuing customers in the same order that we queue them.

Step 3: Stacking those pancakes

In `PancakeShop`'s `createPancakes()` method, we are creating a random number of `Pancakes`. Complete the method so that created pancakes get added to the top of the `pancakes` stack. Remember that stacks are **Last-In-First-Out (LIFO)** – meaning, the last pancake to be added to the stack will be the first one that's eaten by a customer.

Step 4: Eat pancakes!

We now have a queue of customers and a stack of pancakes. It's time to teach the customers how to eat! This is handled in the `Customer`'s `eat()` method, which is largely unimplemented at this point. In this method, customers are handed a stack of pancakes and should try to eat as many pancakes from the top of the stack as will fit in their belly. If there's not enough pancakes for them, they should complain (by throwing a tantrum / `Exception`). Further details about what the method should do can be seen in the provided comments and by examining the provided example outputs.

Exercise Five: Sorting, Comparable & Comparator

For this exercise, we want to sort a list of `Shapes` in a couple of different ways.

When you've completed steps 1 and 2 below, your program should compile and run without errors, and produce the output shown here:

```
Number of shapes in set: 8
 - RegularPolygon [numSides=6, sideLength=2] has a perimeter of: 12cm
 - RegularPolygon [numSides=8, sideLength=2] has a perimeter of: 16cm
 - Rectangle [width=4, length=5] has a perimeter of: 18cm
 - Circle [radius=3.0] has a perimeter of: 18.85cm
 - RegularPolygon [numSides=4, sideLength=5] has a perimeter of: 20cm
 - Circle [radius=4.0] has a perimeter of: 25.13cm
 - Rectangle [width=3, length=10] has a perimeter of: 26cm
 - Rectangle [width=10, length=9] has a perimeter of: 38cm

Number of shapes in list: 10
 - Circle [radius=4.0] has 1 sides and an area of: 50.27cm^2
 - Circle [radius=4.0] has 1 sides and an area of: 50.27cm^2
 - Circle [radius=3.0] has 1 sides and an area of: 28.27cm^2
 - Rectangle [width=10, length=9] has 4 sides and an area of: 90cm^2
 - Rectangle [width=3, length=10] has 4 sides and an area of: 30cm^2
 - RegularPolygon [numSides=4, sideLength=5] has 4 sides and an area of:
25cm^2
 - Rectangle [width=4, length=5] has 4 sides and an area of: 20cm^2
 - Rectangle [width=4, length=5] has 4 sides and an area of: 20cm^2
 - RegularPolygon [numSides=6, sideLength=2] has 6 sides and an area of:
10.39cm^2
 - RegularPolygon [numSides=8, sideLength=2] has 8 sides and an area of:
19.31cm^2

Process finished with exit code 0
```

Step 1: Implement `Comparable<T>`

If we run the main `ExerciseFive` class now, we'll see that the program throws a `ClassCastException`. This is because we're trying to add Shapes to a `TreeSet` in the `sortedSetExample` method. `TreeSets` automatically sort their contents according to their *natural ordering* – but at this stage, we haven't yet defined a natural ordering for Shapes!

To do this, go to the Shape class and make it implement the `Comparable<Shape>` interface. In the `compareTo` method that you'll have to implement, define a `Shape's` natural ordering such that `Shapes` will be sorted according to their *perimeter*, in *ascending order*.

Once we correctly implement the `Comparable` interface, we should see that the `sortedSetExample` method now prints the shapes out in order of their perimeter. We should also notice something else about the *number* of shapes printed out. What do we notice, and why does it occur?

Step 2: Sort using a `Comparator<T>`

Currently, in `ExerciseFive`, the `comparatorExample` method prints out the shapes in no particular order. We want to write a `Comparator` to define an *external ordering* for the shapes which is different to their natural ordering which we implemented in step 1 above.

We have written the beginnings of a `Comparator` implementation in the method, as an anonymous inner class. You'll see that the implementation currently just returns 0. For this step, use the comparator to define an ordering such that Shapes are sorted first by their number of sides (ascending order) *then* by their area (*descending* order). For example, a shape with 3 sides and an area of 10cm^2 would "come before" a shape with 3 sides and an area of 4cm^2. Both of those would "come before" any shape with more than 3 sides, regardless of its area.

Once you've implemented the `Comparator`, at the marked location in the code, make an appropriate call to `Collections.sort` to sort the `shapeList` using the `shapeComparator`.

Exercise Six: Mapping Character Frequency in a Text Block

The program in the `ex06` package is intended to look at a given piece of text, and examine the number of times each letter (from A-Z) and number (from 0-9) appears in that text. The program should then print out this information in a table similar to the following:

```
Char:   Frequencies:
--------------------
'A'    135
'B'    51
'C'    60
'D'    71
'E'    253

Etc…
```

There are two incomplete methods in the `ExerciseTwo` class which you'll implement in this exercise.

Step 1: Fill a Map

The `getCharacterFrequencies` method is supposed to create and return a `Map`, relating each character to the number of times that character occurs in the text. As you can see from the code, we're only considering characters from '0' – '9' and from 'A' to 'Z' (though that could easily be changed by altering or removing the if statement).

There are a few things we'll want to do to complete this method. Firstly, where we're declaring the `frequencies` `Map`, initialize it to an appropriate implementation class. When deciding which to use, remember that we eventually want to be displaying our frequencies in alphabetical order.

Next, we need to actually add values to our map. For each character in the input string, we want to either *i)* increment the frequency count we already have for that character, or *ii)* add an initial frequency count of 1 for that character if we haven't encountered it before.

Finally, as a bonus, we could add any "missing" keys to the map. For example, if we were examining a string with no 'Z' characters, our current code wouldn't ever add 'Z' to the map. We could, if we wanted to, add 'Z' to the map, with a frequency of 0.

Step 2: Print the Map

In the `printFrequencies` method, print out all characters (keys) and their frequencies (values) in the given `Map`. You may iterate through the map using any technique you like in order to accomplish this.