ECS510 Algorithms and Data Structures in an Object-Oriented Framework

Exercise Sheet 8: Functional Programming in Java

This is a set of exercises to support the "Interface Types and Generics" section

This set of exercises is called "Functional Programming in Java" because it is about a style of programming like that used in programming languages called "Functional languages". The language Lisp is the originator of this sort of language, the language Haskell is a modern example. Important early work in developing the idea of functional programming was done by Peter Landin, who was a professor at Queen Mary. The languages Scala, Clojure and Erlang, which have recently received attention in the programming world as possible future directions in programming languages, are strongly influenced by functional programming.

The latest version of Java, called Java 8, introduces some new features called "Lambda Expressions" which enable Java to be used in a way that is more directly like functional programming. As this is such a big development, there is not time to cover it here, but some experts have predicted that it will eventually be seen as a standard way to program in Java, which will need to be covered in introductory programming courses. However, this set of exercises is about using generics and the concept of a "function object" to provide a functional style of programming. It is a simple way of introducing the techniques of generalisation of code through generics and delegating tasks to objects passed through parameters, which have widespread applicability in programming.

1) Look at the code folder for this section. You will find a file, `Transformer.java`, which defines a simple generic type. It has a single type parameter `T`, and is the type of objects which have a method `transform` which takes an argument of type `T` and returns a value of type `T`. Then there is the file `Transformers.java`, which has a class with a single generic static method, `applyConst`, which takes a `Transformer` and an `ArrayList` of its base type and returns an `ArrayList` of its base type. It produces a new `ArrayList` by applying the `transform` operation of the `Transformer` object to each of the items in the argument `ArrayList`.

   The file `TenTimes.java` contains an example of a class which implements the interface `Transformer<Integer>` (that is, `Transformer` with its type argument set to `Integer`). Its `transform` method returns its `Integer` argument multiplied by 10. The code in the file `UseTransformers1.java` demonstrates a `TenTimes` object being passed as an argument to the `applyConst` method to multiply all the integers in an `ArrayList` of integers by 10.

   For another example, in `HelloAdder.java` there is a class which implements `Transformer<String>` with a `transform` method which adds `"Hello"` to its `String` argument. The demonstration which shows a use of this is in file `UseTransformers2.java`.

   Download these files and run them. Make sure you understand how they fit together and operate.

2) The static method `applyConst` in the class `Transformers` works constructively. Add a static method `applyDest` to the class `Transformers` which works similarly to `applyConst` but destructively rather than constructively (that is, it changes its `ArrayList` argument rather than constructing and returning a new one).

3) The file `Joiner.java` defines a generic type, which has a single type parameter `T`, and a single method `join` which takes two objects of type `T` and returns an object of type `T`. In the file `JoinByAdding.java`, there is a class which implements interface `Joiner<Integer>` with a `join` method which adds its two integer arguments. In the file `Joiners.java`, there is a class with a single generic static method, `zipLists`. This method takes a `Joiner` object and two `LispLists` of its base type and returns a `LispList` whose first item is obtained by joining the first two items of the argument lists, second item is obtained by joining the second two items of the argument lists, and so on using the `join` operation of the `Joiner` argument. A demonstration of this is given in the file `UseJoiners1.java`. Download these files, and the files `LispList.class` and `LispList$Cell.class`, which are also in this directory, run the demonstration, and make sure you understand how the files all fit together. 4) Write a static method `zipArrayLists` to go in class `Joiners` which takes two `ArrayLists` and produces a third one joining their contents according to a `Joiner` argument, similar to the way `zipLists` works. Write some code to demonstrate it working.

4) Write a static method `zipArrayLists` to go in class `Joiners` which takes two `ArrayLists` and produces a third one joining their contents according to a `Joiner` argument, similar to the way `zipLists` works. Write some code to demonstrate it working.

5) Write a static method `transformList` to go in class `Transformers` that takes a `Transformer` object and a `LispList` and returns the result of applying the `transform` method from the `Transformer` object to each of the items in the `LispList`. Write some code to demonstrate it working.

6) In file `Multiplier.java`, there is an example of a class of objects which implement `Transformer<Integer>` but which require an argument when they are constructed. The class `Multiplier` is a generalisation of the class `TenTimes`, in which the number the `transform` method multiplies by is not fixed to 10, but is given by the argument to the constructor. A demonstration of this class can be run from the file `UseTransformers3.java`. Download these files, run them, and make sure you understand how they fit together.

   Now write a class which generalises the `HelloAdder` class by allowing objects of the class to add any greeting to a string, with the greeting specified when the objects are created. Write two test methods demonstrates this working, firstly passing objects of this new class to the method you wrote in answer to part 2), secondly to the method you wrote in part 5).

7) Write a generic interface which defines the class of objects which have the method `check` in them which takes an object of the class's type argument and returns a `boolean`. Then write some classes which implement this checking interface. For example, a class whose type argument is `Integer`, and whose `check` method returns `true` if its `Integer` argument is odd, `false` otherwise. Or a class whose type argument is `String` and whose `check` method checks whether its `String` argument is less than a particular length. Then write a generic static method in a separate class which takes one of these checking objects and an `ArrayList`, and destructively removes from the `ArrayList` all those items which fail the checking object's `check` test.

8) Write a class which implements interface `Joiner<String>` with a `join` method which takes two strings and joins them together with a space in between. So if the strings are `"Hello"` and `"world"`, the resulting string will be `"Hello world"`.

   Now write a generic static method called `fold` to go inside class `Joiners`. This method should take an `ArrayList` and a `Joiner` object and join all the items in the `ArrayList` into one item using the `join` method of the `Joiner` object. For example, if it takes as arguments a `JoinByAdding` object and an `ArrayList` of integers, a call to the method `fold` will return the sum of all the integers in the `ArrayList`. If its arguments are an object of the `Joiner` subclass mentioned in the first part of this question, and an `ArrayList` of strings, it will return a single string consisting of the strings from the `ArrayList` joined into this one string with spaces separating them.

Matthew Huntbach