Programming 2:
Tutorial 7

Set by: Luke Dickens

18th – 22nd Nov, 2019

Reminder about the tutorial sheets

Remember that the best way to learn a programming language and understand the concepts is to do lots of programming. This involves a good deal of problem solving, and that requires you to think, experiment and test things. Please try all the standard questions (those not [hard] or [harder]), and spend some time thinking carefully about them, before asking for help. If you are still stuck:

- ask me, or the lab helpers, for help at the lab sessions
- or post a question on the moodle course page

Questions marked as *harder* are there to make you think. You only need to *sketch* a solution to these, and model solutions may not be provided. Do not worry if you cannot complete the harder questions without help.

1 A Cyclable Interface

Look at the `Cyclable` interface in the subfolder `cycling`, and the test programme `TestCyclable`. The `Cyclable` interface is a toy example, to practice with interfaces. You are to complete two classes, `ExerciseBike` and `Bicycle`, that `implement` the `Cyclable` interface. Therefore, each class **must** define the three methods declared in the `Cyclable` interface: `changeUp`, `changeDown` and `cycle`. The two classes are described below.

ExerciseBike

The class `ExerciseBike` represents an exercise bicycle. `ExerciseBike` has one field `frictionLevel` which controls how many calories you burn when you cycle at a certain speed. When called, the `changeUp` method will increase `frictionLevel` by 1, up to a maximum of 20 (at which point it will do nothing). The `changeDown` method will decrease `frictionLevel` by 1, down to a minimum of 0 (at which point it will do nothing). When `cycle` is called, `ExerciseBike` objects will output how many calories are being burned per hour, using the forumla:

$$\texttt{caloriesPerHour} = 30 \times \texttt{pedalSpeed} \times (\texttt{frictionLevel} + 5)$$

Bicycle

The class `Bicycle` represents a true road bicycle, and has two fields `gear` (the gear the `Bicycle` is in) and `direction` (the direction in degrees, in which the `Bicycle` is pointing). When called, the `changeUp` method will increase `gear` by 1, up to a maximum of 5 (at which point it will do nothing). The `changeDown` method will decrease `gear` by 1, down to a minimum of 1 (at which point it will do nothing). When `cycle` is called, `Bicycle` objects will output the speed and direction. The speed of the bike is determined by the `gear` and the input `pedalSpeed` in the following way:

$$\texttt{cycleSpeed} = \texttt{pedalSpeed} \times \texttt{gear}$$

Initially `direction` is set to 0, but `Bicycle` objects have an additional `public` method `steer`, that takes a single `int` argument called `angle`, and returns nothing. When called, `steer` will update the `direction`, by adding the `angle` variable to it (remember that `direction` is in degrees, so it cannot be larger than 360 or smaller than 0).

Finally...

a) Can you predict what will be output if the programme is compiled and run? Try compiling and running the `TestCyclable` programme. Can you explain what is happening in the programme, and why?

b) Now look at the commented code marked `// Commented code A`. Which of these lines will compile, and which will not. Can you explain why?

c) Finally, look at the commented code marked `// Commented code B`. Which of these lines will compile, and which will not. Can you explain why?

2 Shopping Lists

Look at the `BrandedItem` class and `TestShoppingList` programme in the code provided. This shows a very simple example of how a collection of supermarket items might be stored together and printed out.

a) Can you predict what the programme does? Compile and run the code to see if your predictions are correct.

b) Edit the `showShoppingList` method in the `TestShoppingList` programme so that the total cost of all the items are printed out after the items are listed.

c) Imagine that you want to include other kinds of item on your shopping list. To do this, you are going to first define an interface called `ProductItem`, that declares two methods: `getPrice` and `getShortDescription`. Begin with the `ProductItem.java` file that holds the skeleton of the interface.

   Next edit `BrandedItem` so that it implements `ProductItem`, and rewrite `TestShoppingList` so that the `shoppingList` is an array of `ProductItems`. Your code should now compile, and produce the same output as before.

Hint: You will need to change a few things in ***TestShoppingList****, such as the type of* ***shoppingList*** *and the input argument type for* ***showShoppingList****.*

d) To take advantage of this interface, you should now write a new class called `GroceryItem`, with the following properties:

- `GroceryItem` should implement the `ProductItem` interface.
- A `GroceryItem` has four fields: an `itemType`, of type `String`, e.g. `"Tomatoes"`; a `variety`, of type `String`, e.g. `"Roma"`; a `pricePerKilo`, of type `int`; and a `weight` of type `double`.
- This class should be immutable.
- The price of a `GroceryItem` is the `pricePerKilo` times the `weight`.
- You should provide **all** basic getter methods, and **any** setter methods that you think are appropriate.
- You should also provide any methods needed to implement the `ProductItem` interface.

e) Finally, add some `GroceryItems` to your `shoppingList` in `TestShoppingList`. Compile and run the code to test it.

f) **[harder]** Design and implement another kind of item for your shopping list that also implements `ProductItem`. Add some of these items in `TestShoppingList`. What changes do you have to make to the method `showShoppingList`?

3 ArrayLists and HashSets [!!]

This question is designed to introduce you to the `ArrayList` and `HashSet`. Understanding these objects can save you a lot of time!

Introducing ArrayLists

An `ArrayList` is a flexible version of an array – you do not need to tell it in advance how many objects it will contain. It is also a generic class (which we learn more about later in the course). You can have an `ArrayList` of any type of object, but the constructor needs to know what type that is. To do this you mention the type in angled brackets, as we shall see.

For instance, instead of using a `String` array (`String[]`), you may choose to replace this with an `ArrayList<String>`. Similarly an array `Integer[]`, can be replaced with an `ArrayList<Integer>`. You would create an `ArrayList<String>` and assign it to the `myList` reference variable with:

```
1  ArrayList<String> myList = new ArrayList<String>();
```

Assuming that you have a `String` called `s`, you can add this to the end of the current array with:

```
1  myList.add(s);
```

And you can iterate over each String with the *for-each syntax*, e.g.

```
for (String s: myList) {
  // for each iteration of the loop s will
  // refer to the String at that position.
}
```

You should look at the `ArrayList` in the Java docs. Other useful methods, include `get` (get an element using its index), `size` (the number of objects in the list), and `set` (replace an existing object with another).

Introducing `HashSets`

A `HashSet` is similar to an `ArrayList`, but will only hold one copy of each identical object. Also, it does not preserve the order of items, meaning that you may get objects out in a different order than you put them in.

The `HashSet` is another generic, and again you must specify a type argument in angled brackets for the constructor. For instance, to create a `HashSet` of `String`s, called `mySet` you would use:

```
HashSet<String> mySet = new HashSet<String>();
```

Assuming that you have a `String` called `s`, you can add this to the end of the current array with:

```
mySet.add(s);
```

Again, you can iterate over each String with the *for-each syntax*, e.g.

```
for (String s: mySet) {
  // For each iteration of the loop, s will
  // refer to a new member of the set.
}
```

This exercise is designed to help you to feel more comfortable with these containers. Feel free to also read the Java tutorials, look at the Java APIs, and search online for examples of how they can be used.

Your Task

Look at the programme `ContainersExample` in `<root>\tutorial7\containers`. Before you compile and run this, try to predict what the output would be.

1. Compile and run the code to see if your predictions were correct.
2. Now consider adding another two additional `String`s, the `String` `"Chips"` and another `"Spam"`. Modify the code so that after the output, you add both of these `String`s to each

container (if you can), then output the contents of each container again. Before you compile and run the programme can you predict what this new output will be.

3. The `ArrayList` generic class has an associated generic interface `List`. This means that the class `ArrayList<String>` implements the interface `List<String>`; and similarly the class `ArrayList<Car>` implements the interface `List<Car>`. Look `List` up in the documentation. Can you update the code, so that the reference variable is of type `List<String>`?

   [hint] You will need an additional import statement

4. Likewise, the `HashSet` `implements` a `Set`. Can you make similar changes to the code so that the reference variable has the same type as the interface?

5. Now return to the shopping example from earlier in this tutorial. Can you use one of the generic containers `ArrayList` or `HashSet` to contain your shoppping list? Which of the two should you use?

6. [harder] Another implementation of `List` is `LinkedList`. How would you replace your use of `ArrayList`s with `LinkedList`s?

   After you have attempted this question you may look up `LinkedList` in the documentation, but you do not need to do so beforehand. Why Not?