Programming 2:
Tutorial 4

Set by: Luke Dickens

21st – 25th Oct, 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 look at all the questions spend some time thinking carefully about them, before asking for help. If you are still stuck:

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

Some questions in the early labs (typically the first 3) are marked with a [*] symbol. **These questions are compulsory** and **you will be assessed** on one or more of these in the following week's lab.

Questions marked as [!] or [!!] 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 these harder questions without help.

1 Lockable Door [*]

Look at the files `LockableDoor` and `LockableDoorProg` in folder `lockable_door`. Look first at class `LockableDoor`. Note that a `LockableDoor` object has two `boolean` attributes: `open` which indicates whether the door is open (`true`) or closed (`false`) and `locked` which indicates whether the door is locked (`true`) or unlocked (`false`). Two methods have already been written for you too: `close` which closes the door and `unlock` which unlocks the door, both method report what they are doing.

a) Add a method `attemptOpen` which takes no input and returns no value. When called it should:

- report `"Attempting to open..."`
- if the door is unlocked then set `open` to `true` and report `"Success!"`
- if the door is locked, leave the attributes unchanged but report `"Cannot open, door is locked!"`

b) Add a method `attemptLock` which takes no input and returns no value. When called it should:

- report `"Attempting to lock..."`

- if the door is closed then set `locked` to `true` and report `"Success!"`
- if the door is open, leave the attributes unchanged but report `"Cannot lock, door is open!"`

c) Now look at `LockableDoorProg`. Which lines violate encapsulation? Comment out these lines.

d) Why is encapsulation particularly important for `LockableDoors`?

e) What changes should you make to `LockableDoor` to ensure encapsulation? Make those changes.

Hint: *Think about the visibility of the attributes.*

f) What visibility should the methods be to ensure `LockableDoor` methods are accessible from anywhere? Make these visibility changes.

Submission: You should submit `LockableDoor.java` and `LockableDoorProg.java`

2 International Oven [*]

Look at the files `InternationalOven.java` and `TestInternationalOven.java` in folder `international_oven`. Ultimately, this will represent an oven that can be set to temperatures either using the UK region standard Celcius, or the US region standard Fahrenheit. Note that conversion from a temperature c in Celcius to f in Fahrenheit can be achieved with:

$$f = \frac{9c}{5} + 32 \tag{1}$$

Likewise a conversion from f in Fahrenheit to c in Celcius can be achieved with:

$$c = \frac{5(f - 32)}{9} \tag{2}$$

a) Add two **class methods** to `InternationalOven`: `celciusToFahrenheit` and `fahrenheitToCelcius`. Both methods should take one `double` as input and return a `double`. `celciusToFahrenheit` should take a temperature in Celcius and return the corresponding temperature in Fahrenheit, and `fahrenheitToCelcius` should take a temperature in Fahrenheit and return the corresponding temperature in Celcius. Compile and run `TestInternationalOven` to test your changes.

b) Uncomment Block B in `TestInternationalOven`. For this code to compile you will need to make the following changes:

- Add a `double` attribute called `tempCelcius`. This should enforce encapsulation.
- The existing constructor should set `tempCelcius` to `100.0`.
- Add method `getTempCelcius` – a standard getter method.
- Add method `getTempFahrenheit`, which should take no input arguments and return a double, equal to the oven temperature in Fahrenheit.

Compile and run `TestInternationalOven` to test your changes.

Hint: *You should reuse your predefined class methods where possible.*

Comment: *Notice that line 31 concatenates a literal String with* `oven.toString()` *using operator* `+`. *When using the concatenation operator you do not need to explicitly call the* `toString()`

method. Replace `oven.toString()` *with* `oven` *and recompile. You should have exactly the same result!*

c) Uncomment Block C in `TestInternationalOven`. For this code to compile you will need to make the following changes:

- Add method `setTempCelcius` – a standard setter method.
- Add method `setTempFahrenheit`, which should take one `double` as input but return nothing. This should allows the calling code to pass a desired temperature in Fahrenheit for the oven. Remember that the oven stores its temperature in Celcius. You **should not add** any other attributes.

Compile and run `TestInternationalOven` to test your changes.

d) Why is it a good idea to only store the oven temperature in a single attribute?

Submission: You should submit `InternationalOven.java` and `TestInternationalOven.java`. Notice that there is a Block D in `TestInternationalOven.java`. This relates to unassessed material and you can leave it commented out for the submission.

3 Unassessed additions to `InternationalOven`

This question extends the `InternationalOven` class, but these changes are unassessed.

1. Add a class attribute to `InternationalOven` called `region` of type `int`. This tells us whether the oven class should output information for the UK or US market. Add two additional immutable class attributes `UK` and `US`. These specify the two valid values that `region` can take. Make sure that `region` is initialised with the `UK` value.
2. Add an instance method `getTemp` which will get the region specific value of the temperature. E.g. if `region` takes value `UK` this should return a temperature in Celcius.
3. Edit the `toString` method so that `regionStr` takes values `"UK"` or `"US"` depending on the current value of `region`. The temperature shown should be region specific. How should you do that to ensure good code reuse?
4. Add a **class method** `switchRegion`, which will change the value of `region` from `UK` to `US` and vice versa. This method takes no input and returns no value.
5. Uncomment Block D in `TestInternationalOven.java` and test your changes. Can you explain which attributes are immutable and which are not? Can you explain their visibility? Can you explain which methods are instance methods and which are class methods?

4 Simple Vectors [*]

Your job is to write a class representing simple vectors in the class `SimpleVector` and test it with `VectorArithmetic`. Look in the subfolder `simple_vectors`. You will write your classes in this folder, the two files `SimpleVector.java` and `VectorArithmetic.java` have been created for you.

Your job is to write a rudimentary implementation of 2-dimensional integer vectors. This will include writing some mathematical properties such as addition and subtraction. However, we will write this class in stages, compiling and testing our changes at each step.

a) We want each `SimpleVector` to be a pair of `int`s, e.g. (x, y). It should have two **immutable** `int` attributes `xVal` and `yVal`. Initially, we would like to be able to create new vector from two `int`s, and we would like to output our vector to the screen. For example, the following code:

```
SimpleVector v1 = new SimpleVector(1,1);
SimpleVector v2 = new SimpleVector(2,3);
System.out.println(
  "We have created two SimpleVectors and they look like this:");
System.out.println("  v1 = " + v1.toString());
System.out.println("  v2 = " + v2.toString());
```

should give output like this:

```
We have created two SimpleVectors and they look like this:
  v1 = (1,1)
  v2 = (2,3)
```

You should write the `SimpleVector` code in the corresponding file, and the test code is written for you in `VectorArithmetic.java`. Compile and run the `VectorArithmetic` to test your changes.

Hint: *Your `SimpleVector` objects should be immutable, and they should follow good practice for well behaved classes, e.g. encapsulation, so you may need to write getter methods for your class. Should you also write setter methods?*

b) We would also like to compare two `SimpleVector`s with an `equals` method. Write an appropriate method and test it in the `VectorArithmetic` programme. For instance, the following code:

```
SimpleVector v1 = new SimpleVector(8,7);
SimpleVector v2 = new SimpleVector(8,7);
SimpleVector v3 = new SimpleVector(7,8);
SimpleVector v4 = null; // a null variable

System.out.println("Testing equals method:");
System.out.println(
  "  v1.equals(v2) evaluates to "
  + (v1.equals(v2)? "true": "false"));
System.out.println(
  "  v1.equals(v3) evaluates to "
  + (v1.equals(v3)? "true": "false"));
System.out.println(
  "  v1.equals(v4) evaluates to "
  + (v1.equals(v4)? "true": "false"));
```

Should produce the following output:

```
Testing equals method:
  v1.equals(v2) evaluates to true
  v1.equals(v3) evaluates to false
  v1.equals(v4) evaluates to false
```

Hint: *What does the signature to* `equals` *look like? You need to determine the visibility, return type and input arguments for the method. This method should return* `true` *if the two vectors are equal, and* `false` *if the input has different elements or is a* `null` *variable. How do you test if a variable is* `null`*?*

c) You should now write four methods:

- `negation` – this method takes no inputs and returns a *negation* of the `this` `SimpleVector`, i.e. the negation of vector (x, y) is the vector $(-x, -y)$.
- `add` – takes another `SimpleVector` `other` as argument and returns a new `SimpleVector` equal to `this` plus `other`. So if `this` vector is (x, y) and `other` is (u, v), then `add` should return $(x + u, y + v)$.
- `subtract` – takes another `SimpleVector` `other` as argument returns a new `SimpleVector` equal to `this` minus `other`. So if `this` vector is (x, y) and `other` is (u, v), then `subtract` should return $(x - u, y - v)$.
- `multiply` – takes an `int` as argument, and returns a `SimpleVector` equal to the argument times `this`. So, if `this` vector is (x, y) and the input is c then the output should be $(c \cdot x, c \cdot y)$.

Take care to specify the visiblity, return types and input arguments appropriately. Then write some test code in `VectorArithmetic` to check that things work okay.

Comment: *Remember that code reuse is a good thing. Think about how you would define one of these methods in terms of two others.*

Submission: You should submit `SimpleVector` and `VectorArithmetic.java`

5 Unassessed additions to `SimpleVectors`

This question extends the `SimpleVector` class, but these changes are unassessed.

1. Write a method, `length`, that returns the length of the vector. To do this, you should use Pythagoras's theorem which gives the length of a vector (x, y) as $\sqrt{x^2 + y^2}$. How many input arguments should the length method take? What is the return type?
2. Write a **class method** `sum` that belongs to `SimpleVector`. This should take an array of `SimpleVectors` as input and sum them all together, returning the result. Write and run some test code for this in `VectorArithmetic`.

6 Dealing Playing Cards

Look at the files in the subfolder `dealing_cards`. There are three files `Card.java`, `Deck.java` and `Dealer.java`. These are as follows:

- `Card` represents individual playing cards
- `Deck` is a collection of cards that can be shuffled and dealt
- `Dealer.java` is a simple programme to test the other two classes

We will start just looking at the `Card` class.

Note that some of the code in `Dealer` and `Deck` is commented out*, so that you can compile and test your answers to earlier questions straight away. You will have to* uncomment *code as you complete successive questions.*

a) Begin by looking at the `Card` class, this represents playing cards to be used in various card games. `Card`s have two attributes `faceValue` and `suit`. Write a constructor that takes a face-value and a suit as input (both `int`s).

b) Write some getter methods for `faceValue` and `suit`, and set the properties of these attributes to forbid direct access outside the class and also to ensure immutability (Why?). When you are done, compile the `Card` class to check for compile errors.

c) Should you add any setter methods (or other mutator methods) to this class?

d) There are a few other methods provided in `Card`, you should leave these as they are. Look at the method input and output types, as well as their properties, e.g. `public` and `static`. Can you predict what they will do?

e) Look at the `Dealer` class, but do not make any changes. What will happen when the code is compiled and run? Compile and run the `Dealer` class to find out.

f) Look at the `Deck` class, but do not make any changes. This is intended to simulate a deck of cards, that can be shuffled and dealt from. There are two instance attributes: What are their names? Can you predict what they represent?

g) The constructor is provided for you, and this creates an ordered deck of cards. There is also a shuffle method, that randomises the order of the cards. Can you see how this works?

h) The `topOfDeck` attribute keeps track of where the top of the deck is (immediately after shuffling the top is zero and counts up as we move down the deck). We do not wish client code to be able to inspect any of the cards in the pack. (Why not?) Therefore, we will not implement standard getter methods. Instead, you should implement the `dealCard` method which takes no input and returns the `Card` pointed to by `topOfDeck`, and change this to *index* the next card in the deck. When you have implemented `dealCard`, uncomment the first commented block in `Dealer` and the first commented helper method. Now compile and run the `Dealer` class.

i) You should also write a method in `Deck` called `isDealtFrom` that takes no arguments, and returns `true` if the deck has been dealt from (since it was created or shuffled) and `false` otherwise. Uncomment the second block of commented code in `Dealer` and the second commented helper method. Then try compiling and running it. Can you explain the output?

j) Finally, you are going to write a method `cutDeck` that takes an `int` called `cutAt` and returns a `boolean`. `cutAt` is an integer index into the deck. `cutDeck` cuts the deck at `cutAt`. Cutting a deck involves splitting the deck at the cut point into a top and a bottom part, then switching the order of the two parts, but it should only be possible if the `Deck` has not been dealt from.

The method should cut the deck and return `true` only if

i) the deck has not been dealt from since last being shuffled

ii) and `cutAt` is a valid index into the deck

Otherwise, you should return `false`. Uncomment the third commented block in `Dealer` and the third commented helper method. Test your changes.