Programming 2: Tutorial 1

Set by: Luke Dickens

30th Sep – 4th Oct, 2019

About the Tutorial Sheets

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.

The aim of these tutorial sheets is to provide a wealth of reasonably small programs for you to write and inspect. You are strongly encouraged to **attempt all of the questions**, and to study the model answers that will be distributed after any appropriate in lab assessment has taken place.

Assessed Tutorial Questions: Some questions in the early labs (typically the first 3 questions) 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. For most of you, this means you will be assessed on:

- Tutorial sheet 1 in Lab 2
- Tutorial sheet 2 in Lab 3
- Tutorial sheet 3 in Lab 4
- Tutorial sheet 4 in Lab 5

Once you have completed the assessed questions you should upload the specified files to the Lab Exercises tab on moodle under the appropriate submission link. You can submit at any time before the end of the day before your next lab session.

The TAs will move round the room and assess you in a prespecified, but randomised, order. If you are not present when the TA comes to assess you, you will be marked zero on that week's assessment (even if you turn up later to the lab). If for any reason you cannot make (or will be late for) a particular lab on a particular week then you should submit an extenuating circumstances (EC) form with your department. Those with EC forms for lab assessments will be assessed later in the term in labs in a similar way.

For non-assessed questions, you are still encouraged to attempt them. Some questions may be marked with [!] or [!!], and model solutions may not be provided for these. Questions with [!] or [!!] are more challenging, so do not worry if you cannot complete them on your own. You only need to *sketch* a solution to these. However, these questions are designed to stretch you, and may help you understand concepts earlier.

I also encourage you to explore your own variations of and extensions to the tutorial questions. Think of extra features you could add to some of the programs, and try to add them.

For everyone's benefit, please email me if you find errors in these sheets or the model answers. If you get stuck:

- come to the lab sessions and ask me, or the TAs, for help
- or post a question on the moodle course page (so that the answer can be shared by others)

However before you ask for help, do make sure you have spent a significant amount of time scratching your head and thinking about your problem, and looking for solutions in the lecture notes and other available sources. Also, try implementing things, and testing how the compiler responds and whether the behaviour is as you expect. You will likely learn a lot by really trying to figure things out for yourself.

If you are still stuck after this then I, and the others involved in the course, will be very happy to help. Note that TAs or I may not provide a solution to your precise problem, but may want to discuss a related piece of code so that you can understand the concept, then solve the problem yourself.

Getting Started

For these tutorial questions there are some supporting files that can be found on moodle. Download the zip file on moodle and place in a folder on your machine: these notes will refer to this folder as `<root>`. Each question will tell you which *subfolder* your files are in. For example, if you unzip the file in folder `<root>`, then any files for a question in subfolder `example` can be found at location: `<root>\tutorial1\example\`.

You must also learn how to navigate the command line and compile and run your programmes. Please look at Lecture 1 materials on the Taught Content tab of the Moodle page for some videos to help you get started with this.

A reminder on how to compile a programme

This is just a quick reminder of how to compile a simple Java program.

- Make sure all your files are in the same folder and their filenames end with `.java`
- At the command line, make sure that your *present/current working directory* is the same as the directory that contains your files.

  For instance, if your files are in directory: `N:\some\directory\tutorial1\coins\`, you should also have this as your working directory. If you are already in device `N:` type:

  `cd \some\directory\tutorial1\coins\` to go to that directory.

You should already by within device `N:`. If not, make sure you have started the java command-line from Windows. Ask a friend or TA, if you are not sure how to do this.

To check your current directory on Windows type `cd` and press return.

- To compile use the `javac` command. For instance, compile a file called `HelloWorld.java` with `javac HelloWorld.java`
- You run your compiled programme with the `java` command. For instance, to run the `HelloWorld` programme, type `java HelloWorld`

Again, the videos provided for your first lab session should guide you through this.

1 Counting Heads [*]

Look at the `Coin` class in subfolder `coins`, this is simliar to the class that appeared in the lectures. There is a simple program in `CoinProg.java`, that takes user input and flips the coin when requested. Try compiling and running `CoinProg.java`.

There is another java file in the same folder called `CountingHeadsProg.java`. This contains a class `CountingHeadsProg` with a single `main` function, which is incomplete. When complete the programme should:

a) request a number of tosses from the user and assign to variable `numTosses`

b) initalise a variable called `headCount` to zero

c) create a new object of type `Coin`

d) toss this coin `numTosses` and increment `headCount` by one each time the coin shows heads

e) output the number of tosses

Steps a) and e) are already completed for you. You will need to implement steps b), c) and d).

Hint: *To create a new object you will need the* `new` *operator (see slides). You will also need to call one or more methods on the coin object.*

Submission: You should submit `Coin.java` and `CountingHeadsProg.java`

2 Animal Guesser [*]

Look at the `AnimalGuesser` and `Animal` classes in folder `animal_guesser`. The `AnimalGuesser` class is complete and you do not need to edit it. When complete, this programme will be game similar to the `NumberGuesser` game we saw in class. Edit the `Animal` class in the following way:

- The `Animal` class has two attributes defined for you, a species (type `String`) and a size (type `int`). You do not need to edit these.
- A constructor has been partly written for you. The declaration is correct, but the body is incomplete. Complete the body. This should takes two arguments: a species (type `String`) and a size (type `int`). The constructor should assign the inputs to the corresponding attributes.

- An equals method has been partly written for you. The declaration is correct, but the body is incomplete. This method takes another `Animal` as input and returns a `boolean`. Edit the body of the equals method so that it returns `true` if the `other` animal has the same species and size as `this`.
- You must also write a `compareMessage` method, which takes another `Animal` as input and returns a `String`. Again the declaration has been written for you and you should not change it. You must edit the body of the method. When called `compareMessage` should return:
  - `"Wrong Species"` if the `other` animal's species differs with `this` animal
  - `"Too small"` if the `other` animal's size is smaller than `this` animal
  - `"Too large"` if the `other` animal's size is larger than `this` animal
  - `"Perfect match"` otherwise

Hint: *You can use the* `this` *keyword to specify an attribute inside the body of a constructor/method. See the* `Circle` *class in the slides for an example.*

Hint: *You can compare two integers* `a` *and* `b` *with the test* `(a == b)`*. However, to compare two* `String`*s* `c` *and* `d` *you should use* `c.equals(d)`*. Why?*

Submission: You should submit `Animal.java` and `AnimalGuesser.java`

3 Character Counting [*]

This question works on the files in folder `character_counting`. There are two classes in this folder `CharCounter` and `CharCounterProg`. Both classes are only partly written, you must complete both for this question. For `CharCounter` your task is as follows:

- Add an `int` attribute `count`
- Complete the constructor setting `count` to zero
- Complete the `observe` method. This takes a `char` as input called `observed`. This method should check whether `observed` is equal to the attribute `key`. If it is, it should increment `count` by 1. Otherwise it should do nothing.

For `CharCounterProg` your task is as follows:

- Before the while loop, create two `CharCounter` objects, one called `dotCounter` which counts `.` (full-stop/period) characters, and one called star counter which counts `*` (asterisk) characters.
- Inside the (nested) for loop, call the `observe` method first on `dotCounter`, then on `starCounter`, each time passing the `currentChar` variable as a single argument.
- At the end of each iteration of the outer while loop, you should output the counts for the dots and stars (to give the output below).

When finished you should be able to run your programme and interact with it to give the following output:

```
$ javac CharCounterProg.java
$ java CharCounterProg
Input a line of characters: ...
So far I have observed:
3 dots
0 stars
Input a line of characters: *.*
So far I have observed:
4 dots
2 stars
Input a line of characters: ****
So far I have observed:
4 dots
6 stars
Input a line of characters: |
```

Hint: *Note that the constructor takes a single argument of type* `char`. `char` *variables hold a single character (not a string of characters). To assign a character containing the letter q to a* `char` *variable* `myChar` *you can use:*

```
myChar = 'q';
```

If you need to declare the variable and assign at the same time use:

```
char myChar = 'q';
```

Submission: You should submit `CharCounter.java` and `CharCounterProg.java`

4 Equivalent `Coins`

Look again at the files in subfolder `coins`. Now, add an `equals(...)` method to the Coin class, which takes a second Coin as input. `equals` should return `true` if the two coins are showing the same face, and `false` otherwise. Test the extended Coin class in a variation of `CoinProg.java` called `EqualCoinsProg.java`. The program should produce input/output along the following lines:

```
Type:
        f - to flip both coins
        q - to quit

Enter selection: f
The coins are equal (both tails).

Type:
        f - to flip both coins
        q - to quit

Enter selection: f
The coins are different.

Type:
        f - to flip both coins
        q - to quit

Enter selection: f
The coins are equal (both heads).

Type:
        f - to flip both coins
        q - to quit

Enter selection: f
The coins are different.

Type:
        f - to flip both coins
        q - to quit

Enter selection: q
PROGRAM ENDED
```

5 Famine and Feast [!]

This question is much harder than the others and you are not expected to complete it. However, if you have found the other material straight-forward, this should stretch you a little, and improve your understanding of attributes and methods that update them.

Look at the two classes `FamineAndFeast` and `Kingdom` in subfolder `famine_and_feast`. This is a small game that you can play.

a) Look at the code in both classes. Which class runs the main programme? Can you predict what will happen when the code is compiled and run?

b) Compile and run the code. Now, play the game. Does the behaviour match your expectations?

c) The game is quite hard. Can you write some additional code that prompts the player before the main game to select easy or difficult mode? For the easy game, make the target 20 gold and initially give the user only 3 workers.

d) Look at the `newSeason` method in class `Kingdom`. This is quite a long complicated method. Write some helper methods which perform some of the computation in order to neaten things up a bit.

   Restructuring code to make it simpler is called refactoring, and is very common in industry. Refactoring is particularly important when considering code re-use, which we will discuss throughout the course.

   Are there any other parts of the code you think should be refactored?

e) To make the game more interesting, the buy and sell price of grain should be double (2 gold) following a famine season. It will help if you refactor the code a little to add this feature.

f) How would you make this game more interesting?

Hint: *Remember that the* `main` *method is the first to be called in any java programme.*