COM1027 Programming Fundamentals

Lab 7

Exploring the Collections Framework

Purpose

The purpose of this lab is to develop your skills and ability to:

- Defining your own lists and array lists (implementation of List interface).
- Defining your own maps and hash maps (implementation of Map interface).
- Using t as arguments to methods and relating the arguments to underlying data structures.

Lab Structure

Labs are a mixture of step by step instructions that enable you to learn new skills, and exercises so that you can define your own examples.

We will use **Concept Reminders** in front of text when we are reminding you of concepts you have already learnt previously in labs or lectures.

In using this document it is possible to paste text directly into Eclipse. That means when defining pieces of code contained in lab sheets in Eclipse their value can be copied directly from here, which will remove the possibility of mistyping values. If you are reading a PDF version of the document, then clicking on the 'Select' icon activates text selection. Before running the code, ensure that the code has been copied correctly, and make the relevant changes to make the code fully-functional.

These lab exercises will be **assessed**. They are marked, and contribute to your final grade. This lab exercise has 10 marks to earn which will contribute towards your final grade. Every place you have to add code is indicated as an 'Exercise' with instructions.

A total of 10 labs will be assessed, that will correspond to the first 20% of your final grade. Please submit your completed work using GitLab before 4pm on Friday Xth Month 2021.

You must comply with the University's academic misconduct procedures: `https://www.surrey.ac.uk/office-student-complaints-appeals-and-regulation/academic-misconduct`

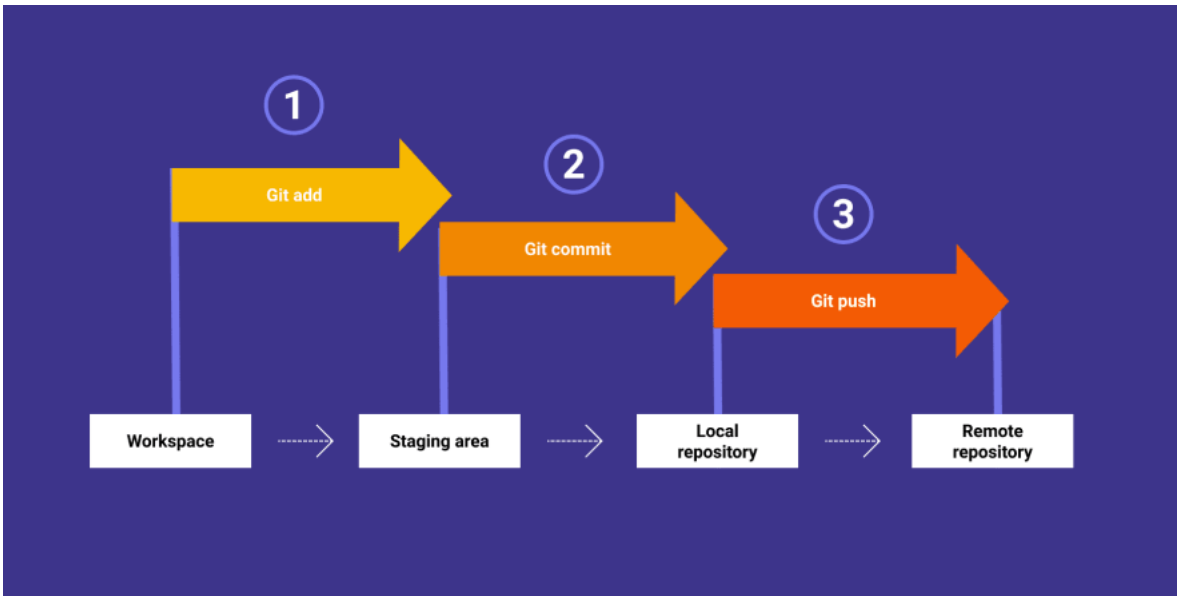
Please ask us lots of questions during the labs, or use the discussion forums on SurreyLearn. All the demonstrators are here to help you get a better understanding of the Java programming concepts.

Instructions

For this module, we will be using automated assessment tools for all lab activities and coursework. To achieve this, we will be using the Faculty of Engineering and Physical Sciences (FEPS) GitLab platform to upload completed code.

In last week's lab activity, you completed the following:

- Downloaded a project from SurreyLearn and imported it into your workspace
- Made changes to the project by creating new classes
- Used the UML diagrams to convert the structured English language to Java code
- Commited the changes to your local repository and
- Pushed those changes onto your remote repository on GitLab



Screenshot taken from `https://docs.gitlab.com/`

In this lab, you will download a new project from SurreyLearn, make substantial changes to the project, commit and push those changes to your remote repository, just like in previous weeks.

Important Note: Do not push any buggy code to the repository unless you have to. Use the `'Run As > Java Application'` or `'Run As > JUnit Test'` as indicated in the instructions below.

Reminder: You can access your GitLab repository via `https://gitlab.eps.surrey.ac.uk/`, using your username and password.

Example

In this example, we will use the debugging tool to inspect instance variables and see in detail how the values change in each line of code.

1. Import an existing Eclipse project and set your workspace
From SurreyLearn, download the project called `COM1027_Lab07.zip`. Extract the contents of the project in the Downloads folder. Copy the uncompressed (unzipped) `Lab07` folder to your local repository. **Note:** Do not copy the `COM1027_Lab07` folder. Navigate in the folder, and only copy `Lab07` folder.

If you are using the computer labs, then your local git directory would look like this:
`/user/HS223/[username]/git/com1027[username]/`
For example:
`/user/HS223/sk0041/git/com1027sk0041/`
where sk0041 shows a sample username.

In your local directory you should have a file called `gitlab-ci.yml`. If you cannot see it, then use the icon (three parallel horizontal lines) on the top right-hand side. From the dropdown menu, select `Show Hidden Files`. This options should remain activated to ensure that you can see all the hidden files in your repository. Now, open the `gitlab-ci.yml` document and change the following line:

```
include: '/Lab06/.project_cicd_config.yml'
```

To:

```
include: '/Lab07/.project_cicd_config.yml'
```

Start Eclipse and open the same lab workspace as last time. Import the project in Eclipse, by clicking on `File > Import > General > Project from Folder or Archive`. Select `Directory` to locate `Lab07` from your local git repository. Once the files are loaded, select the Eclipse project and click `Finish`.

2. ArrayList Walkthrough

Once imported, look at the `Example1` class in the `lab7` package.

Explanation: This code is almost identical to an array example from Lab 4 except that we have replaced the:

```
String[] names = new String[4];
```

with:

```
List<String> names = new ArrayList<String>();
```

We have also removed the index `i` and we add names to the list using:

```
names.add(line);
```

We define an ArrayList to hold a particular type of object. The `<String>` tells Java that the names list can only hold String objects.

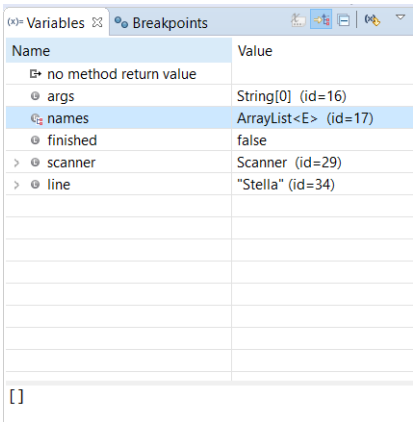
This is called a generic:
`http://docs.oracle.com/javase/tutorial/java/generics/`

Now run the code and enter `Helen`, `Stella`, `Matthew`, `Sid`, `Bobby`, `George` and `exit`. You should not get an exception (despite not specifying the size of the arraylist) and instead you should get the following:

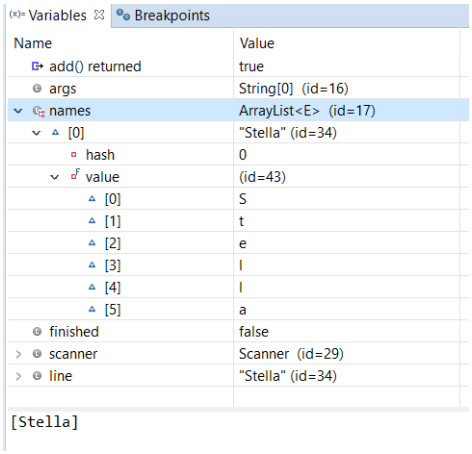
```
Enter names to add to the array list, one per line
Helen
Stella
Matthew
Sid
Bobby
George
exit
Name is: Helen
Name is: Stella
Name is: Matthew
Name is: Sid
Name is: Bobby
Name is: George
```

In fact you could add as many names as you like (up to the limit of memory available for the Java Virtual Machine) without causing an exception. Let us look inside to see how this is achieved.

Set a break point on line 47 of code which adds the name: `names.add(line);` Now run the code again in debug mode and enter one name. When the execution stops on line 47 expand the `names` array list in the Variables view:



Step over line 47 and you should see:



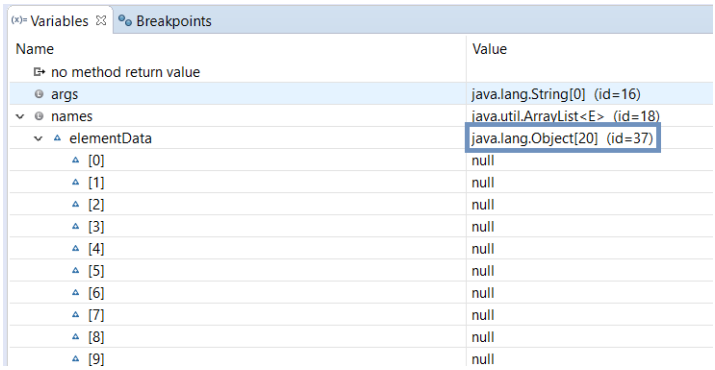
The field `size` keeps a count of the items that have been added to the list. To see the size you might have to press on the right hand side of the variables pane so that you can see the size value.

Concept Reminder:
An `ArrayList` uses an array (hence the name) internally to store the values in the list. What is clever about `ArrayList` is that it keeps track of the number of items added and when the limit of the internal array is reached, it creates a new array big enough and copies over the values from the original array into the new array.
The copy is an expensive operation so each time a new internal array is created, it is created with more than enough space than is currently needed to make sure the copy happens only occasionally. If we wanted to, we could tell `ArrayList` what initial storage capacity it should use when we construct the object.

Let us do this. Change the `ArrayList` declaration to add in the `20` as below:

```
List<String> names = new ArrayList<String>(20);
```

Now run the code again in debug mode and enter one name. When the execution stops on line 47 expand the names array list in the Variables view and look at the `elementData`. You should see that the internal array now has 20 elements:



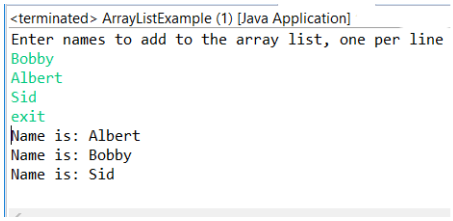
The `20` will tell the ArrayList to create an array with 20 elements to start with but it will not stop the ArrayList creating a bigger array if needed. Just after the line:

```
    scanner.close();
```

Add the following code:

```
    // Sort the names into alphabetical order.
    Collections.sort(names);
```

Run the code again (without debug) and enter names `Bobby`, `Albert`, `Sid` and then `exit`. You should get:



Concept Reminder:
Using an ArrayList not only gives us flexibility on how many entries we add to the list, but it also allows us to use a host of methods that work with Collections.

The Collections framework includes other useful classes, such as Set and Map.

Methods from the Collections framework that we can use on a list include sorting the list. This one line of code sorts the list into its default order, which for strings is alphabetic order. Hence the names in the list are presented alphabetically.

3. Working with ArrayLists

Look at the `Example2` class in the `lab7` package. This is an incomplete concrete class that consists of a set of user-defined methods that enable you to use some of the built-in methods of the List interface and ArrayList class.

The class consists of the following methods:

- `addElement`: This method adds a specified value in the list
- `removeElement`: This method uses the private method `checkIfElementExists` to ensure that the specified value exists in the list prior to removing it.
- `checkifElementExists`: This method is responsible for checking whether the specified `double` value exists in the list. If it does, then the method should return `true` otherwise, `false`.
- `retrieveElement`: This method returns the `double` value at the specified index. It first checks if the index is within the acceptable range of numbers (*note: this could be done by checking the size of the list*), and if it is then it retrieves the value. If the index specified is invalid, then it should display "Invalid Index" on the console and return 0.
- `displayList`: This method return a formatted `String` that represent all the values of the list (separated by spaces)
- `displayOrderedList`: This method is similar to `displayList` method but it displays the values in an ordered list (each value in a separate line) Add the following heading to your `String`:

```
..... = buffer.append("Ordered List\n");
```

  where `buffer` is a `StringBuffer` object.

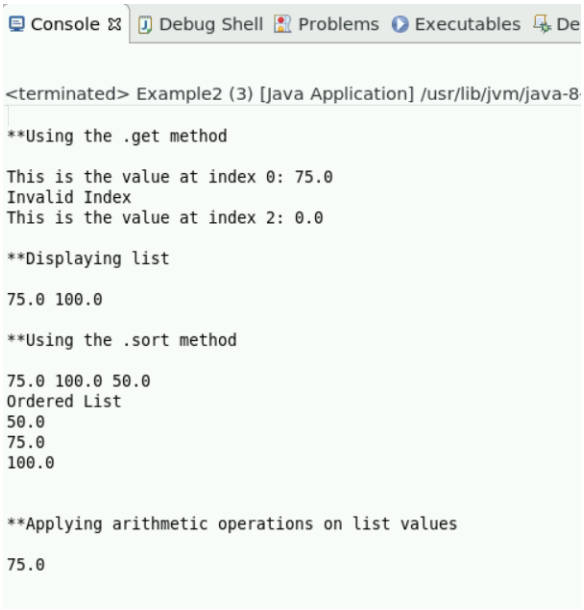
- `calculateAverage`: This method calculates and return the average of all the values in the list. For example, if a list consists of the following values `<0, 2>` it would return 1, and if a list consists of `<20.2, 55.4, 60>` it would return 45.2.

Using the above specification, complete the functionality of the `Example2` class. Make sure that the `displayList` and `displayOrderedList` do not make use of the **System.out.print** statements.

To test the functionality of your methods, first look at the `main` method. Right-click the `Example2` class from your Project Explorer in Eclipse and select `Run As` > `Java Application`.

The lab7 package is not assessed with JUnit files to give you more flexibility. Try different built-in methods from the List interface, as well as repetition**/**conditional statements to achieve the required functionality.

Your output should look as follows:



If the output is not the same, go back and check each and every method. Most common errors occur in control flow statements that change or break the flow of the program's execution. More specifically, check the conditional statements in the `removeElement` and `retrieveElement` methods.

Go to the next page

4. HashMap Walkthrough

It is now time to turn our attention to Maps. Look at the `Example3` class in the `lab7` package.

Explanation: This code is almost identical to `Example1` and `Example2` classes combined. We have replaced the:

```
List<String> names = new ArrayList<String>();
List<Double> values = new ArrayList<Double>();
```

with the following:

```
Map<String, Double> values;
 this.values = new HashMap<String, Double>();
```

where `String` represents the unique keys in the map (module names instead of student names), and `Double` represents the values in the map (module grades in a numeric format).

Now run the code and enter `com1027`, `com1025`, `com1026`, and `exit`. You should not get an exception and instead you should get the following prompt:

```
Enter module names, one per line
com1027
com1026
com1025
exit
You have defined 3 modules.
Enter the grade for COM1027 :
```

In fact you could add as many module names as you like (up to the limit of memory available for the Java Virtual Machine) without causing an exception. All this without specifying the size of the map.

The same example continues on the next page

Now enter, the following `80`, `75` and `91`. You should get the following instead of the correct values for the specified methods:

```
You have defined 3 modules.
Enter the grade for COM1027 :
80
Enter the grade for COM1026 :
75
Enter the grade for COM1025 :
91

**Using the .get method

Invalid Index
This is the value associated to the key 'COM1025': 0.0
Invalid Index
This is the value associated to the key 'COM1026': 0.0

**Displaying Map



**Using the .sort method


Ordered Map


**Applying arithmetic operations on list values

NaN
```

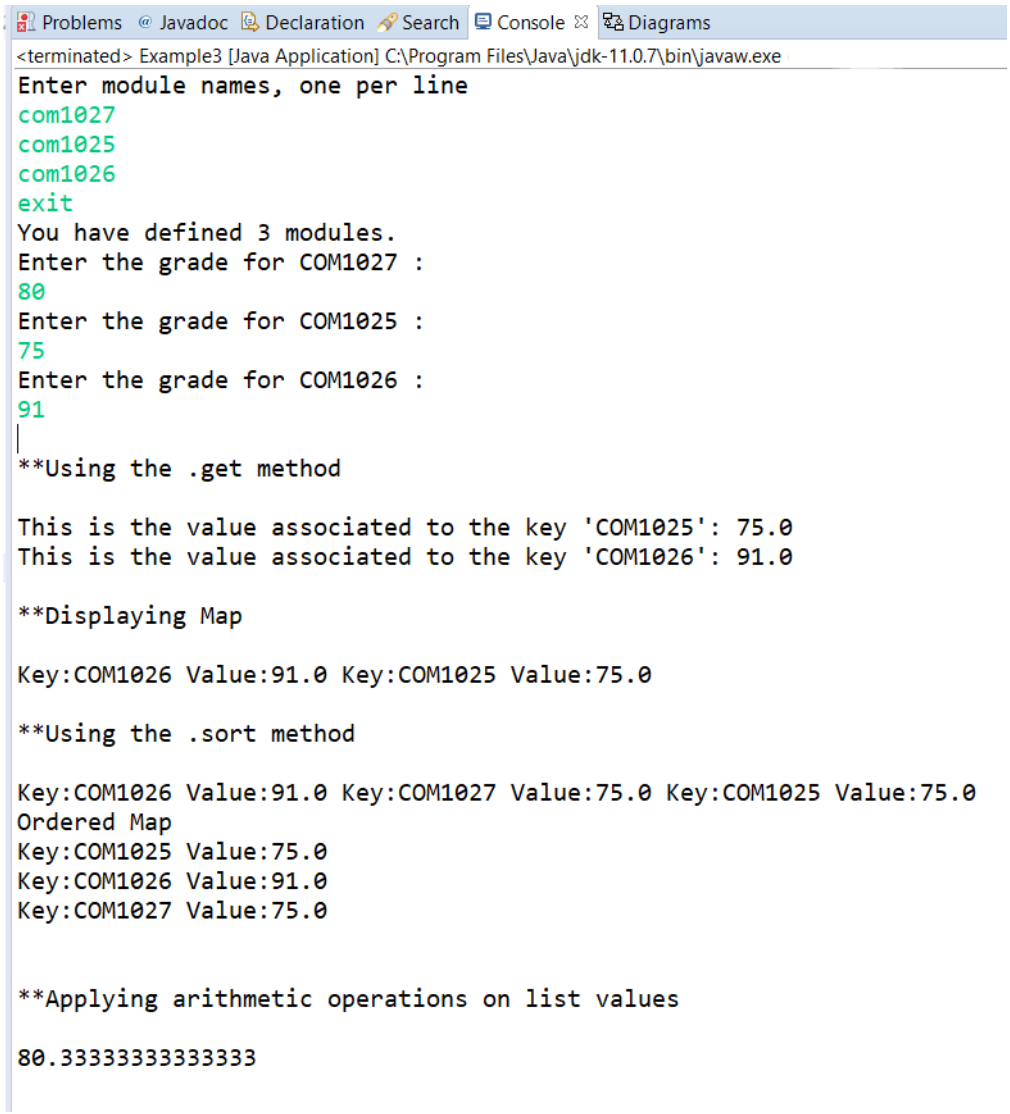
Let us look inside the various methods to see why this is the case.

Upon inspection of the code, you will see that some methods are missing key information. Prior to updating the methods, check the `main` method. This has been defined for you. It uses two separate repetition statements in order to get the user's input. The first repetition statement (`while` loop) uses the `!finished` condition which will continue reading the console for user input, until the word 'exit' is entered on the console. Whereas, the second repetition statement, uses the size of the temporary list in order to determine the number of iterations.

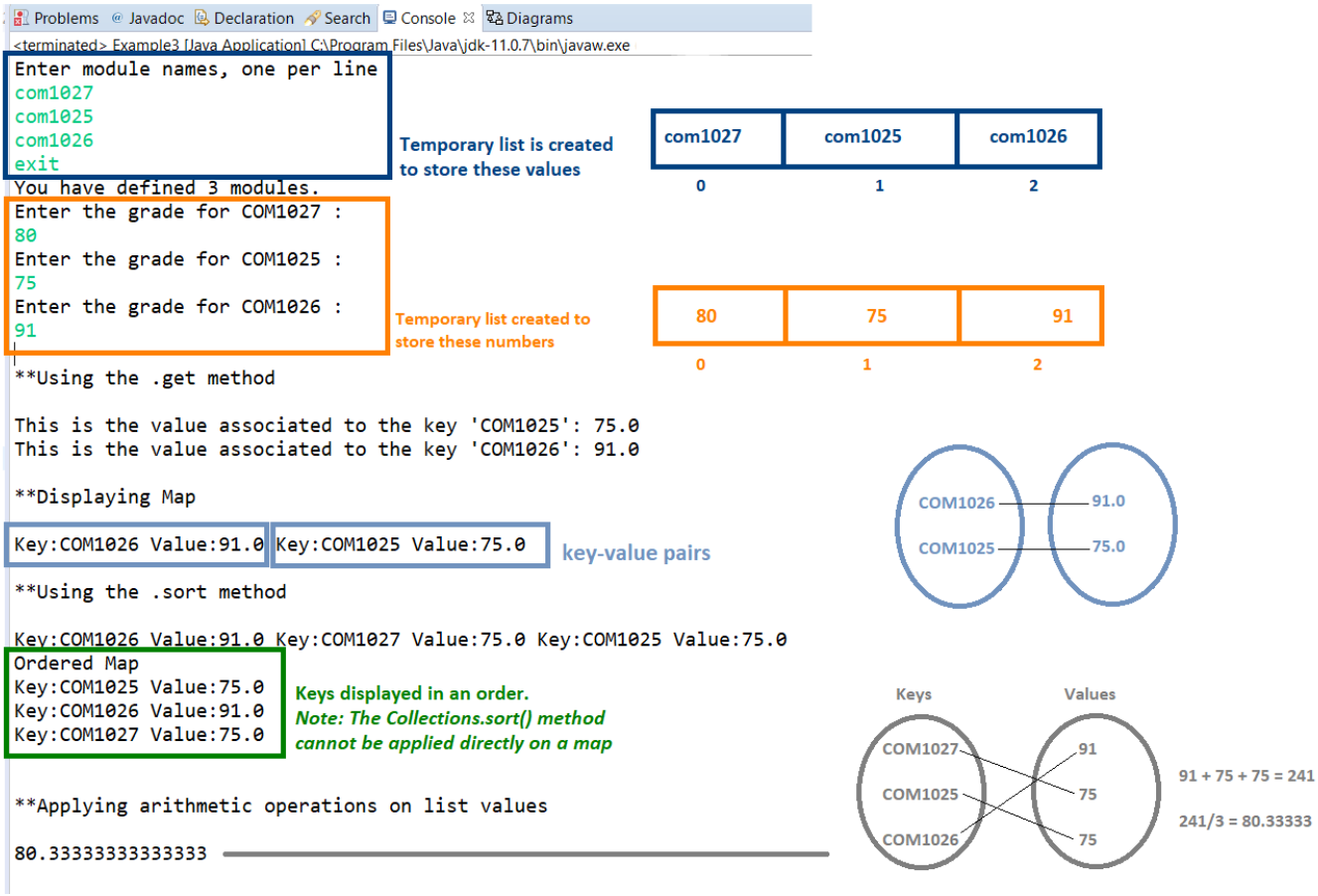
You can now complete the following methods:

- `addElement`: This method adds a specified value in the map. A line of code is missing within the body of the method.
- `removeElement`: This method uses the private method`checkIfElementExists` to ensure that the specified value exists in the map prior to removing it. The method has correctly been called in the conditional statement but a line of code is missing.
- `retrieveElement`: This method returns the `double` value at the specified key. *Reminder: A map holds key-value pairs. Each key object is unique.* It first checks if the key exists in the map, and if it is then it retrieves the value associated to that key. A line of code is missing from the conditional statement (`if` block of code).

Run your program as a Java Application, to check your work. The output should look as follows:



Annotated output:

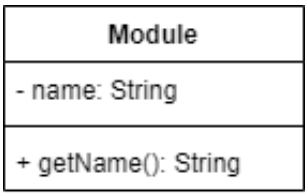


Feel free to modify the `main` method and experiment with the built-in methods (from the `Map` interface and `HashMap` class). This will not affect the Maven build!

Exercise 1 (1 Marks)

Note:
Each exercise builds on the previous one. Make sure that you start from Exercise 1.

This exercise requires you to create a simple class called `Module` as follows:



For this exercise, you will be using the `lab7_exercise1` package and you are expected to demonstrate your understanding of constructors, getters and regular expressions. Create a new class with the name `Module` and add the required field and methods as specified in the UML diagram above. *Note: The `name` field should only hold the module's code, for example COM1027 which consists of three capital letters followed by 4 numbers.*

- Define the field
- Define a constructor for the class. Does this have to be a default constructor or a parameterised one?
- Add the `getName()` method which should return the value of the `name` field.
- Test your code by right-clicking the project name and then `Run As > JUnit Test`. JUnit tests have been predefined in the `src/test/java` source folder. Do not modify them as this would cause your Maven Test to fail!

Go to next page

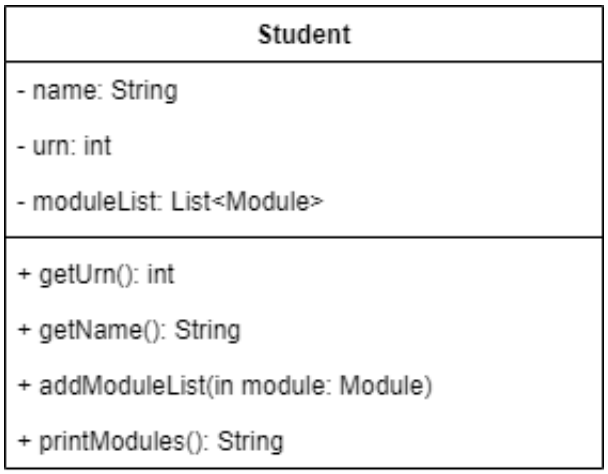
Exercise 2 (3 Marks)

This exercise requires you to create a new class using your understanding of conditional statements, repetition statements and lists.

For this exercise, you will be using the `lab3_exercise2` package. Copy the `Module` class from the previous exercise. Then create a new class called `Student`.

The aim of this exercise is to assign a list of modules to a `Student` object. In order for a valid `Student` object to be created, the `name` field should consist of the student's name and surname in the correct format. For example "Stella Kazamia" instead of "stella kazamia". Individual modules can be assigned to each student, and each module only demonstrates the module code.

For simplicity, our program will NOT include any further information for each student (for example email address, course name, etc.) The following class diagram shows the structure of the `Student` class. Convert the class diagram into Java code:



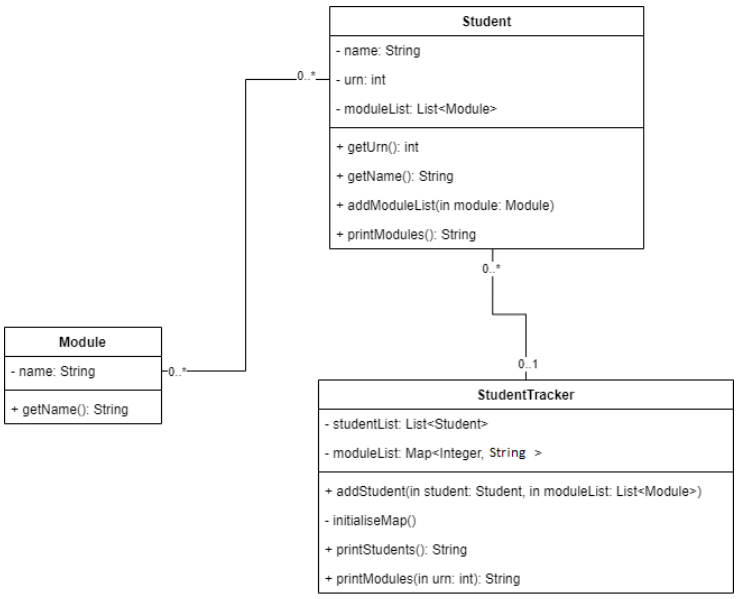
- Define a `Student` class. Each `Student` object comprises of a name, a urn and a list of modules.
- Define a constructor for the class and define the `moduleList` in the body of the constructor
- Add the following getter methods:
  - `getUrn`
  - `getName`
- Create a method called `addModuleList()`. This method should enable a `Module` object to be parsed. Modules can only be added in the list if they do not already exist in the list. How can this be achieved? You can add this functionality by introducing a private method, or by adding it directly in the `addModuleList` method.
- Create a `printModules` method that returns a String representation of the values of the list in the following format:
  `COM1025, COM1026, COM1027, COM1028`
  Note: The returned String does not end with a comma - How can the built-in methods in the `String` or `StringBuffer` classes be used to achieve this?
- Before moving to Exercise 3, test your code by right-clicking the project name and then `Run As > JUnit Test`. Once all the tests for the `Module` and `Student` classes pass, you can start working on Exercise 3.

Stretch & Challenge Exercise (6 Marks)

This exercise requires you to use the `Module` and `Student` classes with minimum support (i.e. some Getters and Setters are omitted from the class diagram provided). The new class will also require the use of lists and maps, as well as repetition statements.

For this exercise, you will be using the `lab3_exercise3` package. Copy the classes `Module` and `Student` from the packages `lab3_exercise1` and `lab3_exercise2` to the `lab3_exercise3` package. Create a new class called `StudentTracker`.

The aim of this exercise is to represent a simple student record system. In the system, students are linked to a list of modules. The new class will be able to display the student's record, which includes their full name, urn and all the modules enrolled in. The new class will also be able to display all the modules for a given urn. The classes are related as shown in the following UML class diagram:



- Define a `StudentTracker` class. Each `StudentTracker` object comprises two types of collections; a list and a map.
- Define a default constructor for the class that allows an object to be created. Define the list and map in the body of the constructor.

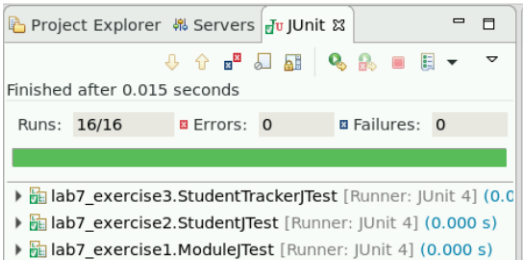
- Add the `printStudents()` method. This method should return a String representing all the students in the `studentList`. The list only gets initialised with values when the `addStudent` method is called.
- Define the `addStudent` method. This is responsible for adding a `Student` object in the `studentList`, and add each module from the `moduleList` parameter in the `Student` object. *Note:Conside Using a repetition statement to read each value of the list. Make use of the `addModuleList` method from the student class to add each module in the specified `Student` object.* This is a tricky method!
- Define a simple `private` method called `initiliseMap` as per the UML diagram. This method is responsible for initialising the `moduleList` map with key-value pairs. The key represent the student's unique urn (`Integer`), and the value represents the list of modules that the student is enrolled in (formatted in a `String`). *Check which existing methods return the list of modules in the following format*:
  `COM1025, COM1026, COM1027, COM1028`
- Add the `printModules()` method. This method is responsible for displaying all the modules listed for a specified urn. This method makes use of the `initialiseMap` method too.
- You can test your code, by creating a `Test` class with multiple objects or run the predefined JUnit tests.

Final Steps

The final step of the lab activity is to check on whether the code is fully functional. To do this, right-click on the project name and select `Run As > JUnit Test`.

If any errors occur, select the relevant file and check the error.

By selecting `Run As > JUnit Test` all the pre-defined JUnit tests will be called. You should get the following in order to proceed:



It is now time to test the structure of your Maven project. To do this, right-click on the project name and select `Run As > Maven Test`.

This should return the following message on the console:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running lab7_exercise3.StudentTrackerJTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.126 sec
Running lab7_exercise1.ModuleJTest
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
Running lab7_exercise2.StudentJTest
Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec

Results :

Tests run: 16, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4.437 s
```

The test passes - now what?
You can now commit and push all the changes made to the remote repository. To do this, select the `Git Staging` view via the menu `Window > Show View > Other...` and then `Git > Git Staging`. Add all the unstaged changes to your local repository, and commit and push the changes to the remote repository.