COM1027 Programming Fundamentals

Lab 3

Experimenting with variables and complex objects

Purpose

The purpose of this lab is to further develop your ability to:

- Define several classes,
- Create objects which themselves having complex objects as fields,
- Define and initialise variables of Primitive and Non-primitive data types

This lab example is based on Chapter 2 of Helbert (2018).
- Herbert, S. (2018). Java. A Beginner's Guide. 8th Edition. McGraw-Hill Education, Inc.

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 values 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 please click on the 'Select' icon that 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. Sometimes, some hidden or incorrect characters could be copied to Eclipse.

The lab exercise at the end of this document 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 (2% of each lab). Please submit your completed work using GitLab by Week 12.

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

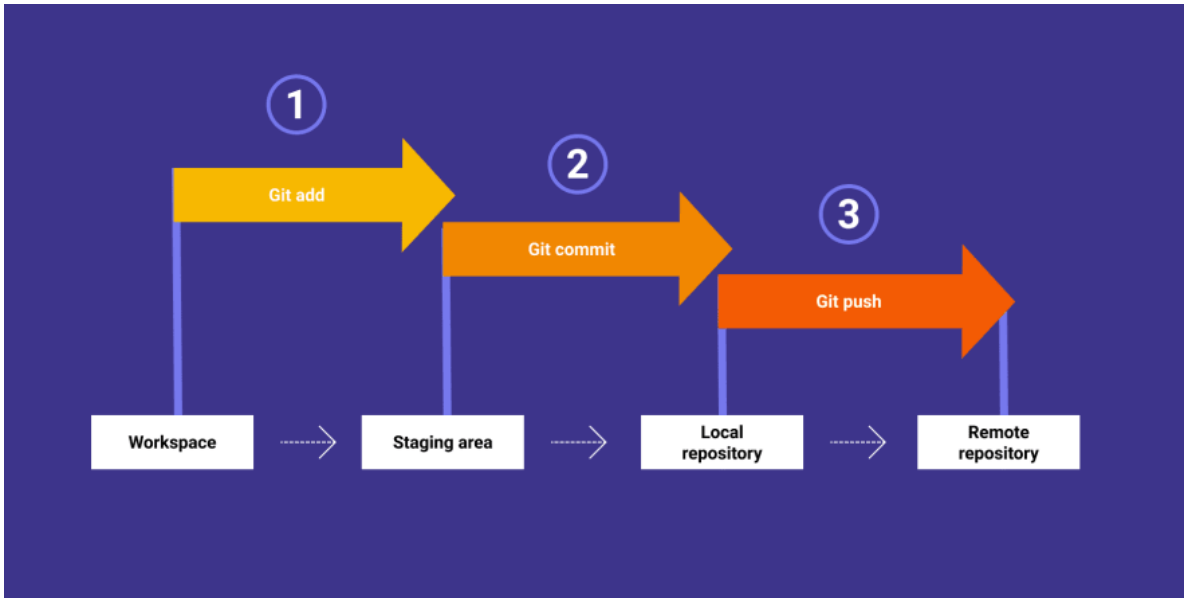
Please ask us lots of questions during the labs, or use the MS Teams group of COM1027. All the demonstrators are here to help you get a better understanding of the Java programming concepts.

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 (https://gitlab.surrey.ac.uk) to upload your 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 and objects
- 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.

Important Note: Do not push any buggy code to the repository. Use the `'Run As > Java Application'` or `'Run As > JUnit Test'` as indicated in the instructions below. You should only 'Commit and Push' code when it works.

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

Example

In this example we will use a simple mail class and modify it in order to practice the use of complex objects.

1. Import an existing Eclipse project and set your workspace

From SurreyLearn, download the project called `COM1027_Lab03.zip` from Week 3's lab section. Extract the contents of the project in the Downloads folder. Copy the uncompressed (unzipped) `Lab03` folder to your local repository. **Note:** Do not copy the `COM1027_Lab03` folder. Navigate into the folder, and only copy the `Lab03` folder to your local repo.

If you are using the IFH lab PCs, then your local git directory would look like this (unless you changed this location then cloning the repo):

`/user/HS***/[username]/git/com1027[username]/`

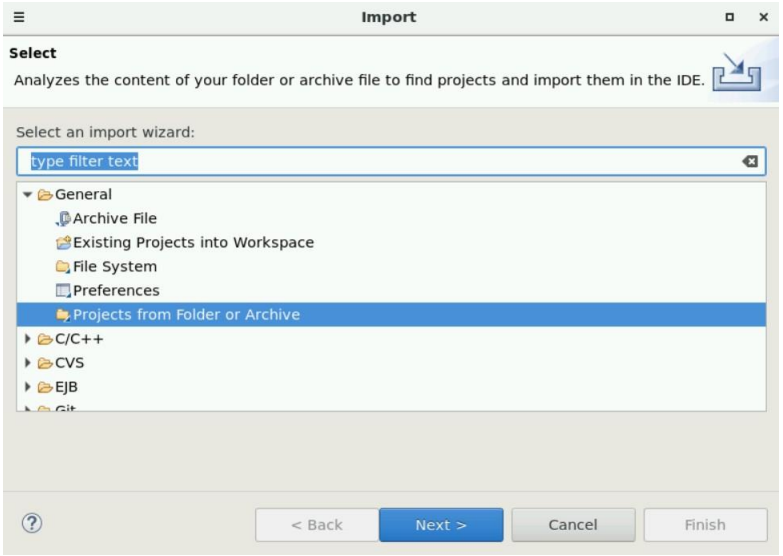
For example:

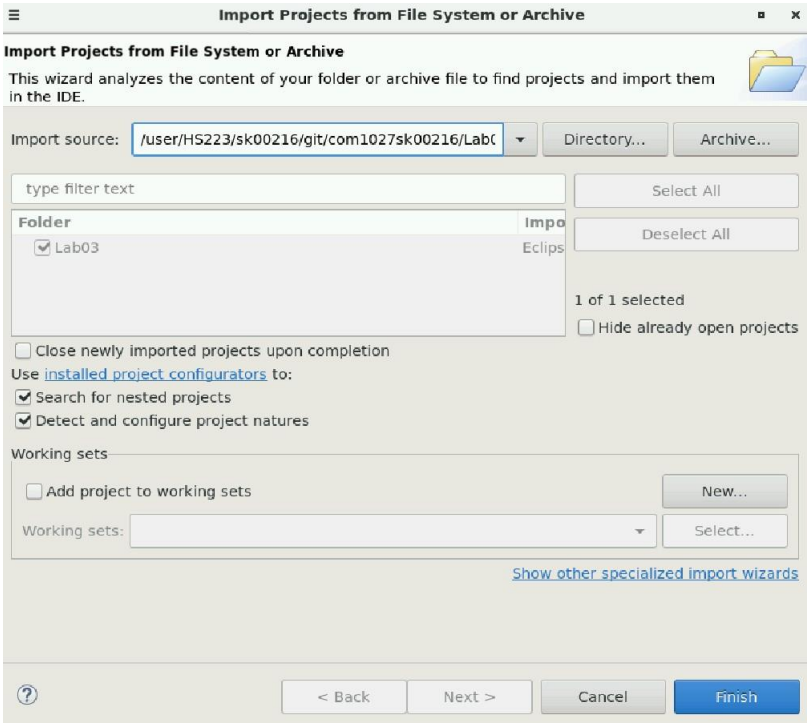
`/user/HS223/ab1234/git/com1027ab1234/`

where ab1234 is 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 option should remain activated to ensure that you can see all the hidden files in your repository. Note that you should **not** change the contents of the `gitlab-ci.yml` file under your repository.

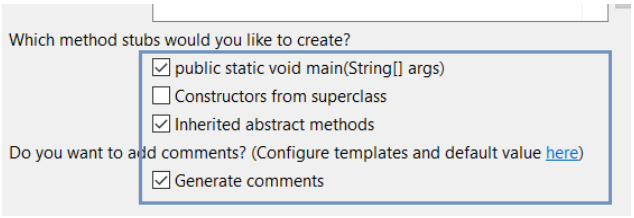
Start Eclipse and open the same lab workspace as last time. Import the project in Eclipse, by clicking on `File > Import > General > Project from Folder or Archive`. Select `Directory` to locate `Lab03` from your local git repository. Once the files are loaded, select the Eclipse project and click `Finish`, as shown in the figures below.





Once added in your workspace, inspect the code in the `MailSimple` class under the `src/main/java/lab3` package. This class is used to model a simple electronic mail (email). It consists of three attributes, a parameterised constructor, three getter methods (also known as accessors) and a method that displays the values on the console.

Define a new class called `MailTest` in the corresponding package (i.e. `src/main/java/lab3`. This is only a temporary class that we will define to help us during the development of our example (this is a common thing to do). Make sure that you create a main method for the class by ticking the following box:



In the `main` method, create a `MailSimple` object with the values `sender = sender@surrey.ac.uk`, `recipient = recipient@surrey.ac.uk` and `message = "Hello"`. Display its values to test the functionality of the print method. Use the following code to achieve this:

```
ms.print();
```

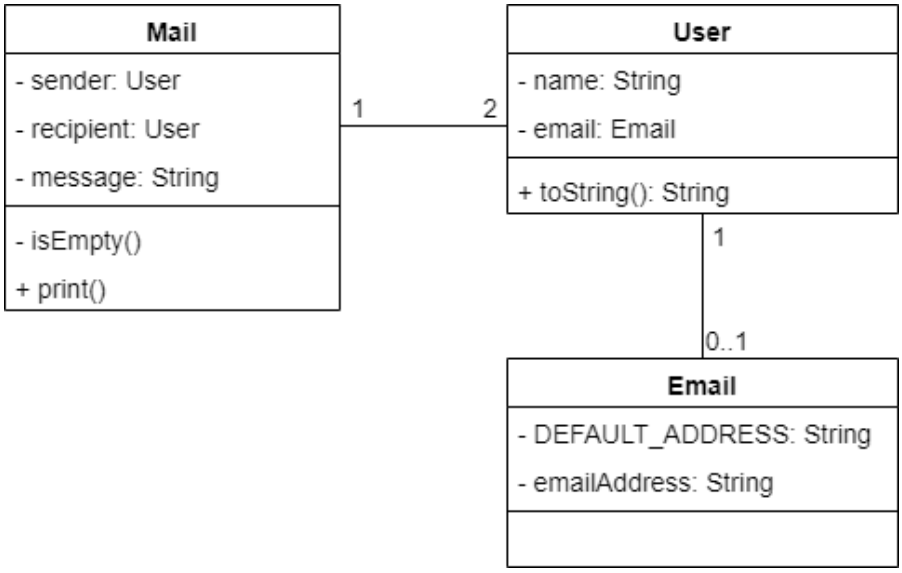
where `ms` is the `MailSimple` object.

Run the `MailTest` program, by right-clicking on the `MailTest` class and then `Run As > Java Application`.

You will see that the values are successfully displayed on the console. It is now time to modify the code and introduce additional classes: `Mail`, `User` and `Email`.

2. Implement classes and objects using class diagrams

The following class diagram defines the `Mail`, `User` and `Email` classes showing their attributes, methods and relationship.



We will first define the `Email` class to hold the email address of a user. This is the only class that does not depend on any other classes being defined first.

Define a class called `Email` within the `lab3` package (*Note: This is where the `MailSimple` class is defined*). This class **does not need** a `main` method. Define two fields, `DEFAULT_ADDRESS` and `emailAddress` to be of type `String` as follows:

```
private static final String DEFAULT_ADDRESS = "unknown@unknown";
private String emailAddress = DEFAULT_ADDRESS;
```

The `DEFAULT_ADDRESS` is defined with the keywords **`static`** and **`final`**, and all its characters are in uppercase. This demonstrates an example of a **constant**!

> **Concept Reminder:**
> A constant is used to store information. Unlike a variable though, it has a fixed value that cannot be changed once it has been assigned. Its value can be set only at declaration and cannot be re-assigned. In Java, to define a variable as a constant we use the keywords `static final` or `final`.

Define a `parameterised constructor` which takes in as a parameter an appropriate `emailAddress` parameter. The class that we will define shortly is going to use objects of `Email` and will need to access `emailAddress`. Thus, we will need to define a getter and a setter for the `emailAddress` field.

> **Concept Reminder:**
> Getters have an empty parameter list, so they do not require additional information to return the variable's value. Getter methods, are also commonly called accessor methods.
>
> Setters, unlike getter methods, contain a local variable. The local variable is said to shadow the instance variable in the method's body, and the keyword `this` is used to refer to that shadowed instance variable. Setters are used to enable any modification to the instance variable and they are also commonly called mutator methods.

To check that the code functions as expected, we will create a temporary `Email` object in the `MailTest` class called `email`. Create the object with the value `email@surrey.ac.uk` and display its value to test the getter. Then change the value to `test@surrey.ac.uk` to test the setter, and display it again.

The following output should be displayed on your console:

Note: This includes the output from the `ms.print()` command too.
`ms.print()` returns:

```
From: s.kazamia@surrey.ac.uk
To: unkown@surrey.ac.uk
Message: Hello!
```

`System.out.println(...)` statements return:

```
email@surrey.ac.uk
test@surrey.ac.uk
```

Continue to next page

3. Create objects with complex objects as fields

Next we turn our attention to defining the `User` and `Mail` classes.

Define a class called `User`. This is where we will use the `Email` class, in order to create our first complex object. Define two fields, `name` and `email` with the data types `String` and `Email`, respectively. Initialise both fields with default values.

> **Concept Reminder:**
> A field, or an instance variable, can also be a variable of `object` type. For such variables, the default initial value is `null`. This is also the case for Strings.

Define a `paramterised constructor` which takes in appropriate `fullName` and `emailAddress` paramters. (intentionally different names from the fields)

We would like the user to display the name of the user, followed by the email address, but so far we have not written the code to do this. Generate a `toString()` method using the steps that we took in the last lab session. (*See Example - part 4b*).

Quickly go back to the `MailTest` class and add the following code in the `main` method to test your code:

```
// reusing the email object that was defined earlier
User user = new User("Bugs Bunny", email);
System.out.println(user.toString());
```

Run the `MailTest` again by right-clicking on the `MailTest` class and then `Run As > Java Application`. This time you will see an output which is not what we expect.

```
User [name=Bugs Bunny, email=lab3.Email@41a4555e]
```

Despite displaying the name correctly, the email address is shown as `lab3.Email@41a4555e]`. This is the object reference of the `Email` object.

Replace the body of the `toString()` method with our own code as follows:

```
return this.name + " <" + this.email.getEmailAddress() + ">";
```

And, run the code again using the `Run As > Java Application`.

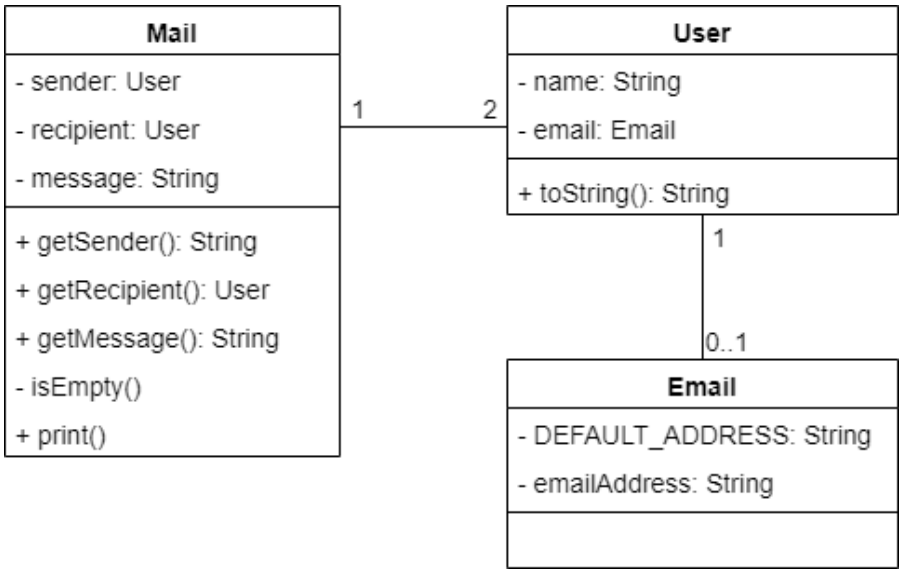
4. Use private methods to support the implementation of the Mail class

Create a `Mail` class, and copy the code from the `MailSimple` class that was provided.

Modify the code to match the class diagram below. This includes changing:

- the data types of the fields
- the return statement of some of the getters
- and introducing an empty method called `isEmpty` that has the visibility `private`.

The class diagram is as follows:



The `isEmpty` method, is used to support the implementation of the `print` method and needs to be defined with the visibility `private`.

> **Concept Reminder:**
> Private methods are only accessible inside the class they are declared in. They are particularly useful for preventing duplication of code (often needed by one or more methods) or for validation of fields, but should not be called outside of that class.

Add the following code in the `isEmpty` method, to check if the value of the `message` field is empty (null or ""). Part of the IF block has been provided for your, but some changes need to be made. Make the appropriate changes so that if the `message` is empty, then the value of the field is changed to `** Something went wrong! **`:

```
    if (this.getMessage() == null || this.getMessage() == "") {
        //add your code here
    } else {
    }
```

In order to use the `isEmpty` method, it needs to be called somewhere in your program. Make the following changes to the `print` method so that the output displayed on the console reflect the changes that we have made in the `isEmpty` method:

```
    //call isEmpty method
    this.isEmpty();

    //this line already exists
    System.out.println("Message: " + this.message);
```

This will enable us to display the `sender`, the `recipient`, and then call the `isEmpty` method which will execute the lines of code within the body of the `isEmpty` method, including the IF statement and the code that we added.

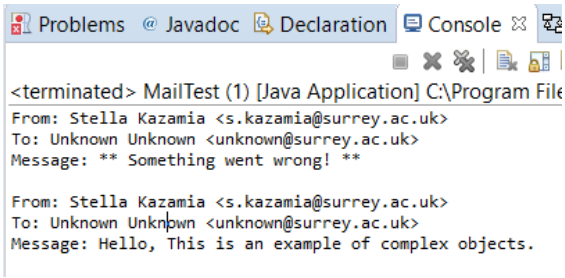
If `this.getMessage()` returns the value `null` OR `this.getMessage()` returns an empty String, then the value of the `message` will change.
It will then continue with the next line, and display the `message`.

We can now test our code. To do this, we need to create two `Email` and two `User` objects, representing the sender and the recipient. And, two `Mail` objects; one with a message and one without (empty) in order to check whether our private `isEmpty` works as expected.

We have added the code for this, below:

```
    System.out.println();
    Email senderEmail = new Email("s.kazamia@surrey.ac.uk");
    User sender = new User("Stella Kazamia", senderEmail);

    Email recipientEmail = new Email("unknown@surrey.ac.uk");
    User recipient = new User("Unknown Unknown", recipientEmail);

    String message = "Hello, This is an example of complex objects.";

    Mail emptyEmail = new Mail(sender, recipient, "");
    emptyEmail.print();

    System.out.println();

    Mail finalEmail = new Mail(sender, recipient, message);
    finalEmail.print();
```

Enter the code in your `MailTest` class, and run the program using the `Run As > Java Application`.
The output should be:

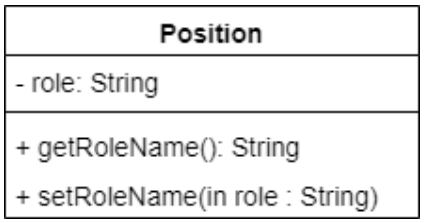


If this is not the case, check your code in the `Mail` class.
Prior to moving to the exercises, right-click your project name and `Run As > JUnit Test`.

Exercise 1 (1 Mark)

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 `Position` as follows:



For this exercise, you will be using the `lab3_exercise1` package. Create a new class with the name `Position` and add the required field and methods as specified in the UML diagram above.

- Define the field
- Define a default constructor for the class. Make sure that you do not define any unnecessary constructors (i.e. both default and parameterised); define only the one required!
- Add the following methods:
  - The `getRoleName()` method is known as a getter or sometimes as an accessor. It just returns the value of a private field in the object, which in this case is called `role`.
  - The `setRoleName()` method is known as a setter or as a mutator. It allows you to pass in the new value of a private 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 getters, setters, and the use of arithmetic operations.

For this exercise, you will be using the `lab3_exercise2` package. Create a new class called `AnnualSalary`.

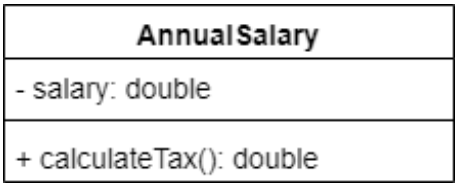
The aim of this exercise is to calculate the amount of tax that needs to be deducted annually from an Employee's salary. Let's define a simple tax rule that provides a personal allowance of £12500.00 which is not taxed. Once the personal allowance is removed from the annual gross salary, then it is taxed 20% for the amount left up to £50000 and 40% for anything beyond that threshold.

- **20% (Basic Rate)** - Between Personal Allowance, and Personal Allowance + £37,500.
  (Over £12,500 to £50,000)
- **40% (higher Rate)** - Above Personal Allowance + £37,500
  (Over £50,000)

Example 1:
Annual Salary: 16475
Personal Allowance: 12500
Taxable Salary: 3975
20% tax on 3975 is 795
Total income: 16475 – 795 = 15680

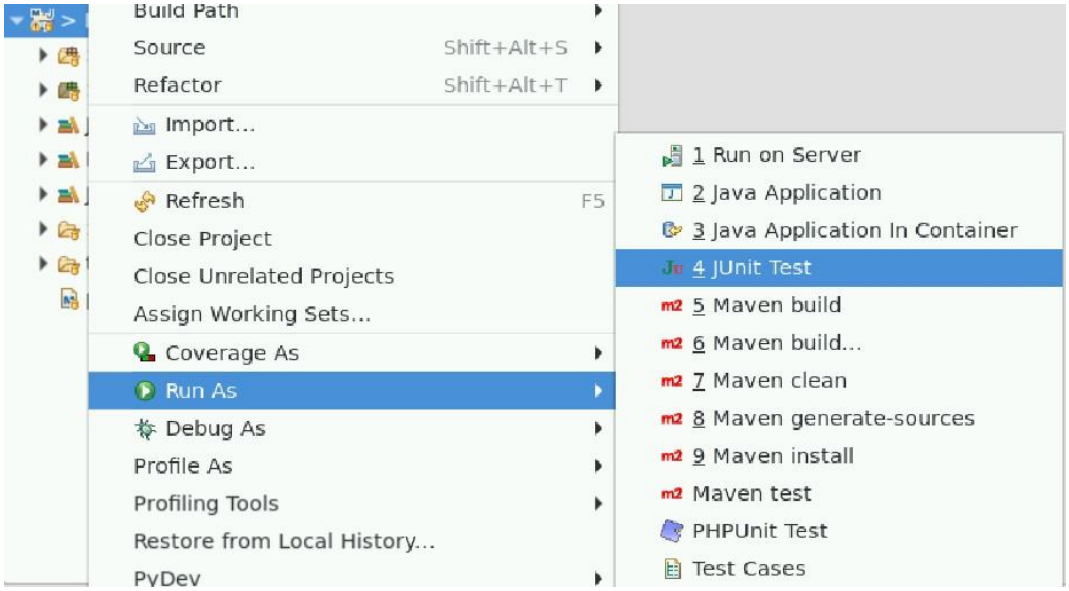
Example 2:
Annual Salary: 53875
Personal Allowance: 12500
Taxable Salary: 53875 - 12500
20% tax on 37500 (50000-12500): 7500
40% tax on 3875 (53875-50000): 1550
Total income: 53875 – 7500 - 1550 = 44825

For simplicity, our program will NOT include the National Insurance (which is calculated separately from the income tax), other taxes or allowances. The following class diagram shows the structure of the `AnnualSalary` class. Convert the class diagram into Java code:



- Define an `AnnualSalary` class. Each `AnnualSalary` object comprises of a salary. You need to decide on whether getters and setters are required for this attribute.
- Define a `default constructor` for the class
- Add the `calculateTax()` method as described above. It is also good practice to start to use constants to define values that you use over and over again within a class, such as the tax threshold or the percentages.

- 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 `Position` and `AnnualSalary` classes pass, you can start working on Exercise 3.

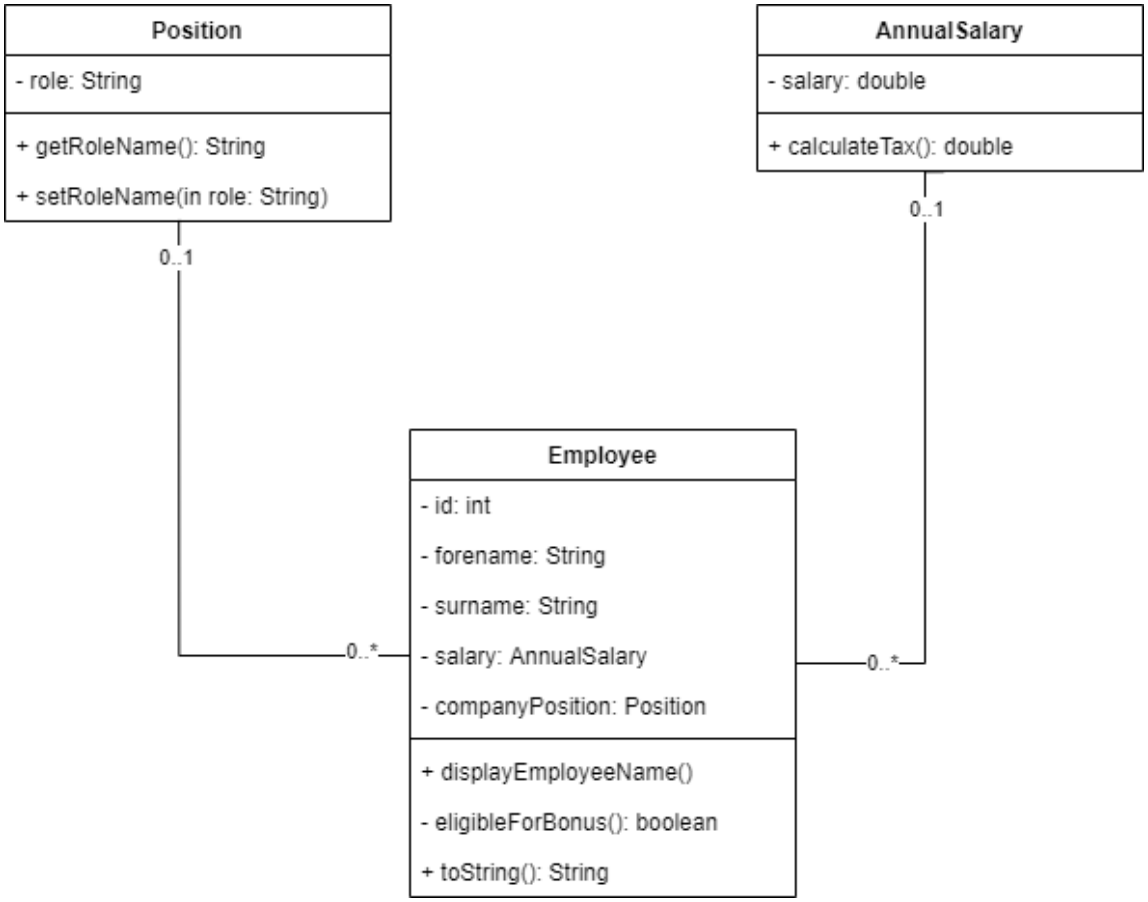
Go to next page

Stretch & Challenge Exercise (6 Marks)

This exercise requires you to use the `Position` and `AnnualSalary` classes with minimum support (i.e. some Getters and Setters are omitted from the class diagram provided).

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

The aim of this exercise is to represent a simple employee record system. In the system, employees are linked to an annual salary and a company position (i.e. their role in the company). The new class will be able to display the employee's record, which includes their identifier, full name, annual salary and the amount of tax deducted from their salary each year. The classes are related as shown in the following UML class diagram:



- Define an `Employee` class. Each `Employee` object comprises of the employee's identifier, forename, surname, salary and position within the company. In previous activities, we made use of getter and/or setter methods. Are they required in this exercise? If so, which ones should be defined?
- Define a parameterised constructor for the class that allows an object to be created with parameters specifying all the fields.
- Add the `displayEmployeeName()` method. This should display the employee's name and surname on the console.
- Add the `eligibleForBonus()` method. An employee who earns £40000 or more is eligible to receive a bonus. Those that earn less that £40000 are not eligible.

- Add the `toString()` method. This should return a String with the required information in the specified format. For employee, 'Sid James', who is a 'Director' on a salary of £53875, the String output should be of the format (precisely):

  James, Sid (1234): Director at £53875.00 (£9050.0 tax) and is eligible for bonus.

  Another example would be, 'Alice Wonderland' with ID '9876', who is a 'Developer' on a salary of £39999.00, and the String output should be of the format (precisely):

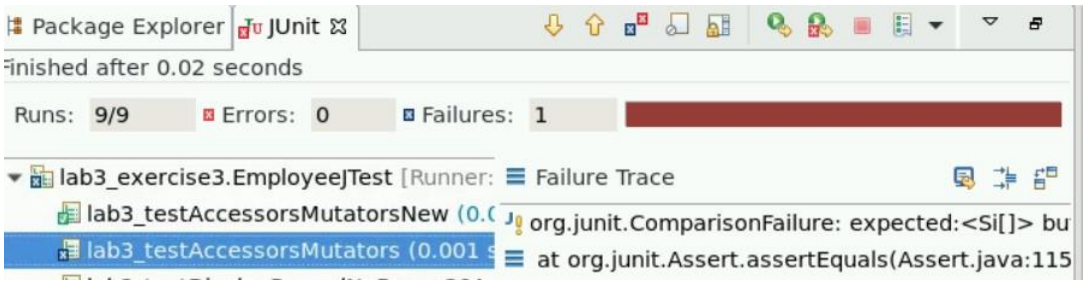
  Wonderland, Alice (9876): Developer at £39999.0 (£5499.8 tax) and is not eligible for bonus..

- You can test your code, by creating a `Test` class with multiple objects.

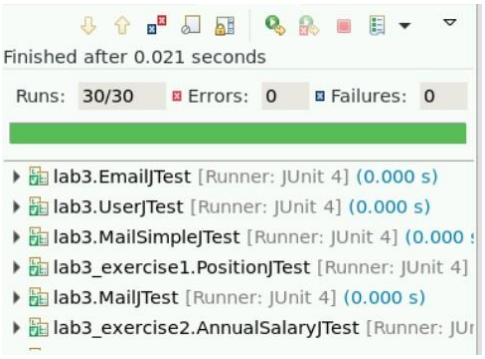
Final Steps

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. In the example below, one of the errors occurred in the `testAccessorsMutators` and on the right hand side it states the expected and actual values.



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:

```
Results :

Tests run: 30, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 3.762 s
[INFO] Finished at: 2020-10-10T15:17:17+01:00
[INFO] ------------------------------------------------------------
```

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.