COM1027 Programming Fundamentals

Lab 2

Experimenting with several classes in Eclipse

Purpose

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

- Define several classes,
- Define objects and complex objects,
- Implement conditional statements (if statements) and Strings,

This lab example is based on Chapter 3 of Barnes & Kolling (2012) and Chapter 4 of Helbert (2018), re-written for Eclipse.

- *Barnes, D. & Kolling, M. (2012). Objects First with Java: A Practical Introduction using BlueJ. 5th Edition. Upper Saddle River, NJ.: Pearson Education, Inc.*

- *Herbert, S. (2018). Java. A Beginner's Guide. 8th Edition. McGraw-Hill Education, Inc.*

Lab Structure

The COM1027 labs are a mixture of activities with step-by-step instructions (called **examples**) as well as independent **exercises** that enable you to learn new skills.

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, but it is **not recommended** as hidden characters could be added into the class file that would cause the Git pipeline to fail. This means that when defining pieces of code contained in lab sheets into Eclipse, their values should not be copied directly from here.

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 (that would be assessed) 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 the deadline (Week 12). The exact date of the deadline will be released later. You can submit as many times as you wish before the deadline, and only the final submission will be assessed.

You must comply with the University academic misconduct procedures for completing your lab assessments independently: https://www.surrey.ac.uk/office-student-complaints-appeals-and-regulation/academic-misconduct

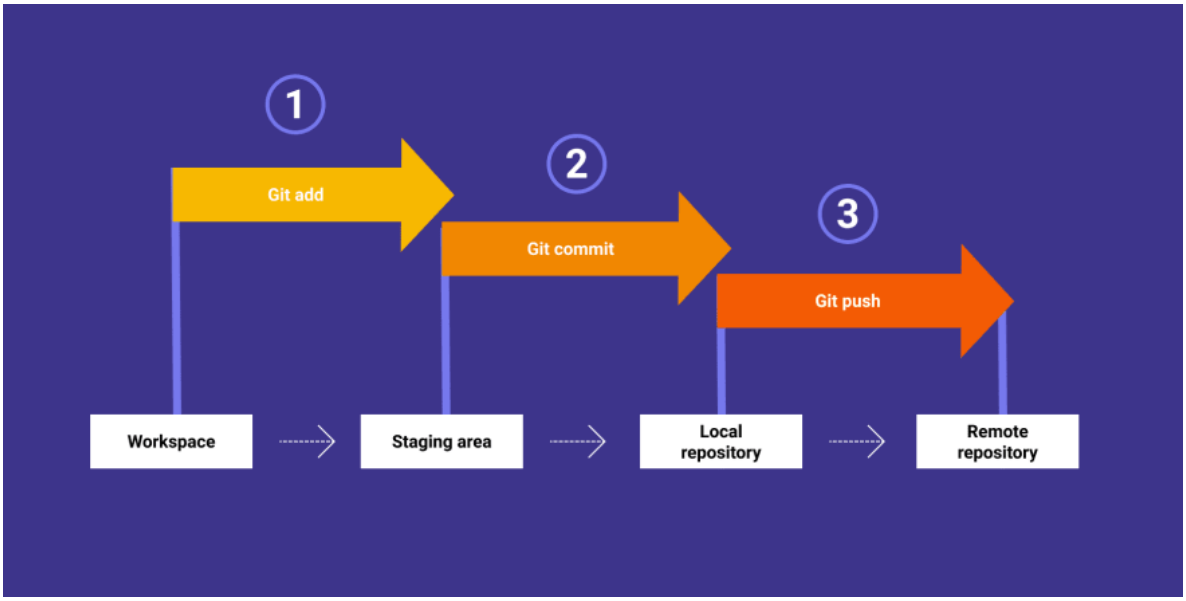
Please ask us lots of questions during the labs, or use the discussion forum on SurreyLearn and MS Teams channel for your questions.
All the demonstrators are here to help you get a better understanding of the Java programming concepts.

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 activated your GitLab account and completed the following:

- Cloned a remote repository (from GitLab)
- Downloaded a project from SurreyLearn and imported it into your workspace
- Made changes to the project by writing a simple Java command (return "Hello World!")
- 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 university IT username and password.

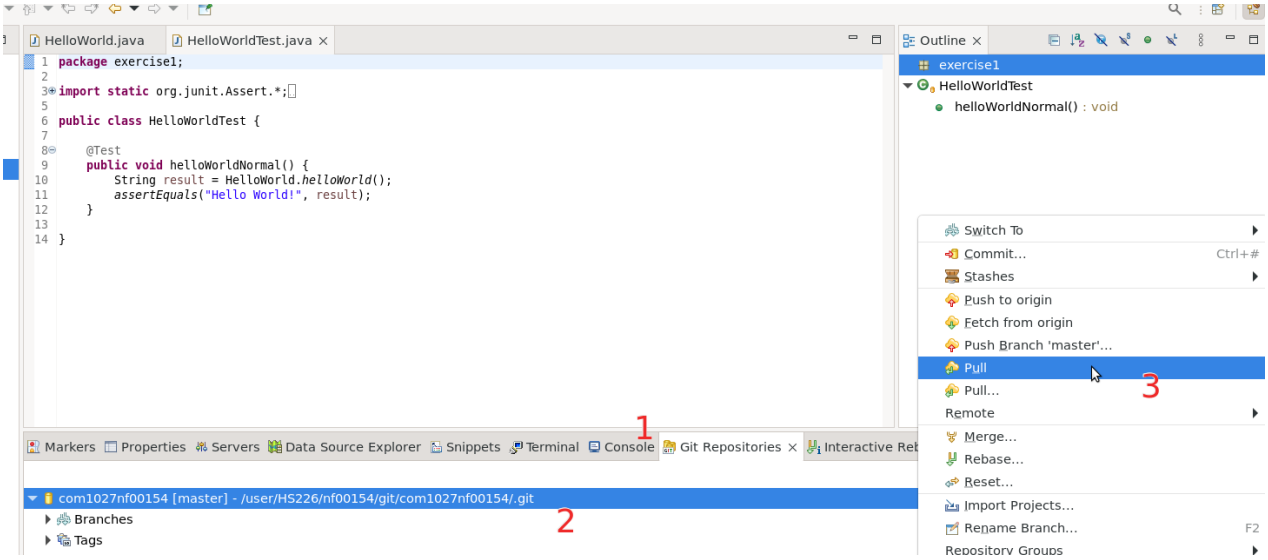
Example

In this example we will define a clock display, initially to be set to 00:00. The display is made up of hours and minutes, and these concepts will be captured as objects of a NumberInfo class within the ClockDisplay class. The hours and minutes are incremented with the execution of a single timeTick method. So we are ultimately looking for an output which looks like the following:

```
00:01
00:02
00:03
00:04
...
```

1. Update your repository

We have updated your repositories since last week. To update your local copy, open Eclipse. Go to Window > Show View > Other > Git > Git Repositories. Right-click on your repository in the Repository view and click Pull. You may be asked for your Surrey credentials.



2. Create the project structure

From SurreyLearn, download the project called 'COM1027_Lab02.zip'. Extract the contents of the project in the Downloads folder. Copy the uncompressed (unzipped) 'Lab02' folder to your local repository. **Note: Do not copy the 'COM1027_Lab02' folder. Navigate in the folder, and only copy the 'Lab02' folder to your local git repository.**

If you are using the computer labs, then this would look like this:

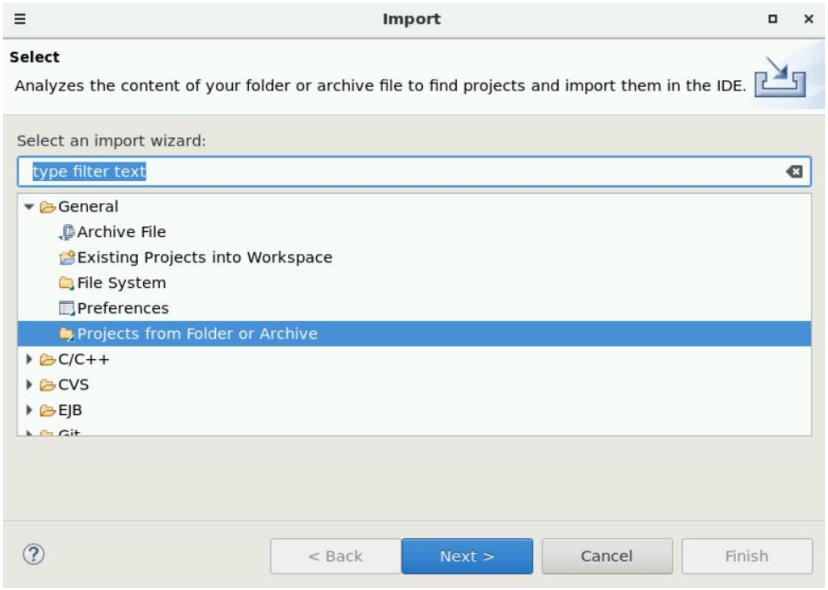
`/user/HS223/[username]/git/com1027_2022[username]/`

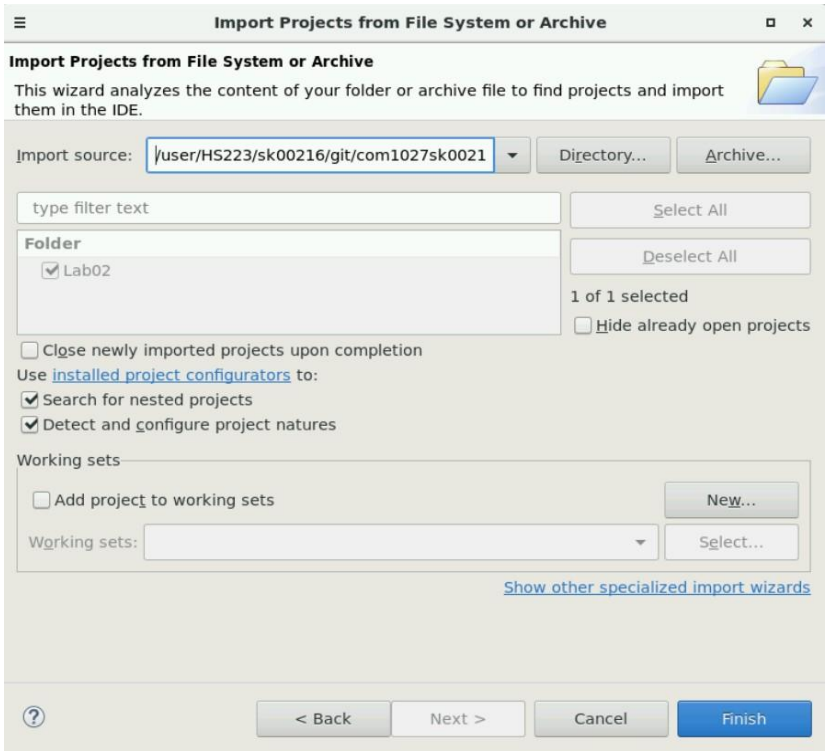
For example:

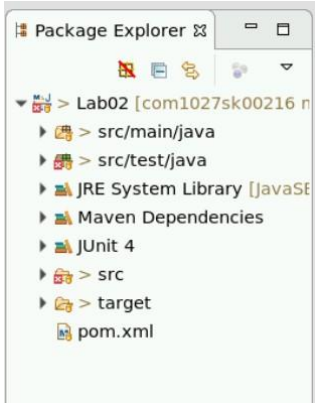
`/user/HS223/ab1234/git/com1027_2022ab1234/`

where ab1234 shows a sample username.

Import the project in Eclipse, by clicking on "File > Import > General > Project from Folder or Archive". Select Directory to locate Lab02 from your local git repository. Once the files are loaded, select the Eclipse project and click Finish.

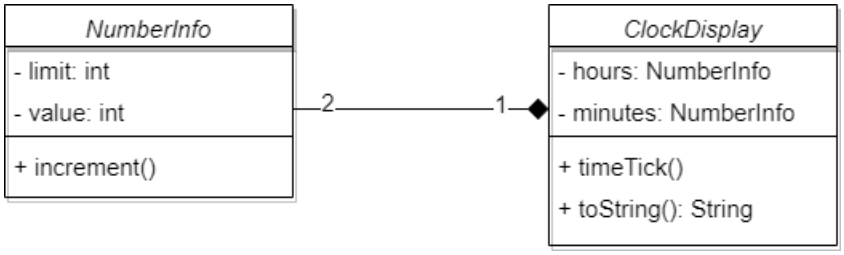

Import
Select
Analyzes the content of your folder or archive file to find projects and import them in the IDE.
Select an import wizard:
type filter text
General
Archive File
Existing Projects into Workspace
File System
Preferences
Projects from Folder or Archive
C/C++
CVS
EJB
< Back
Next >
Cancel
Finish


Import Projects from File System or Archive
Import Projects from File System or Archive
This wizard analyzes the content of your folder or archive file to find projects and import them in the IDE.
Import source:
/user/HS223/sk00216/git/com1027sk0021
Directory...
Archive...
type filter text
Select All
Folder
Lab02
Deselect All
1 of 1 selected
Hide already open projects
Close newly imported projects upon completion
Use installed project configurators to:
Search for nested projects
Detect and configure project natures
Working sets
Add project to working sets
New...
Working sets:
Select...
Show other specialized import wizards
< Back
Next >
Cancel
Finish


Package Explorer
Lab02 [com1027sk00216
src/main/java
src/test/java
JRE System Library
Maven Dependencies
JUnit 4
src
target
pom.xml

3. Implement classes and objects using class diagrams

The following class diagram defines the `NumberInfo` and `ClockDisplay` classes showing their attributes and methods.

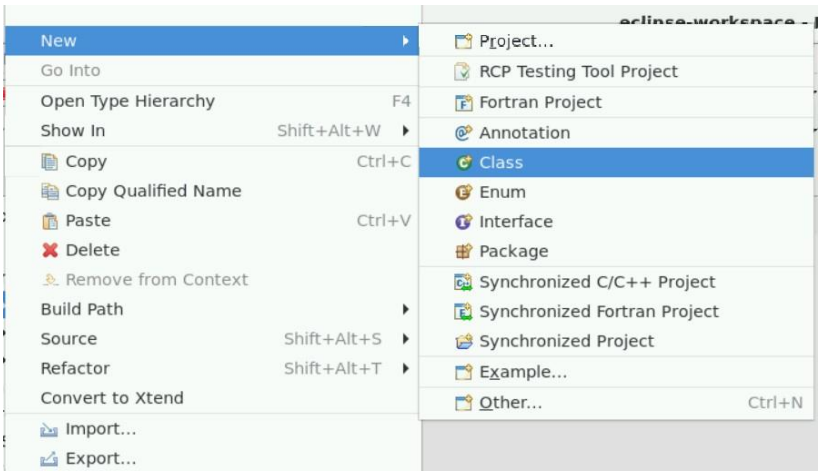


Concept Reminder:
Association is the relationship between two separate classes. This can be a one-to-one, one-to-many relationship, etc. In this case, the above UML diagram, shows an example of Composition which states that the objects of those classes have a part-of relationship. In other words, both are dependent on each other and cannot exist independently.

We will first define the `NumberInfo` class to hold a number which changes incrementally up to a limit and then goes back to 0, e.g., 0, 1, 2, 3, 0, 1, 2, 3, etc.

Define a class called `NumberInfo` within the `lab2` package.

This class **does not need** a main method. To create a new class, right-click on the `lab2` package and select `New>Class` from the sub-menu.



Define two fields, `limit` and `value` to be both of type `int` as follows:

```
private int limit;
private int value;
```

Define a `parameterised constructor` which takes in as a parameter an appropriate `limit` parameter.

```
public NumberInfo(int limit){
      super();
}
```

The `value` field is set to its default value of 0.

```
public NumberInfo(int limit){
      super();
      //add code here
}
```

The purpose of the `limit` field is to control how the `value` field changes. So we do not need to provide getters and setters for this field. However, the class that we will write shortly is going to use objects of `NumberInfo` and will need to access `value`. Thus, we will need to define a getter and a setter for the `value` field.

Concept Reminder:

You can use Eclipse to help you create getters and setters via the Source menu.

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.

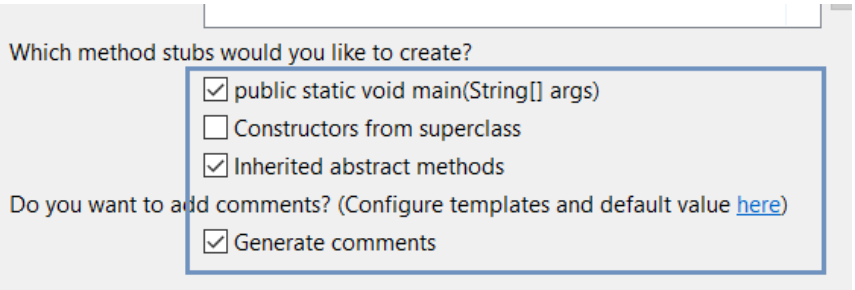
Setters, unlike getter methods, contain a local variable. The local variable is said to shadow the instance variable in the method's body, and the keyword this is used to refer to that shadowed instance variable. Setters are used to enable any modification to the instance variable and they are also commonly called mutator methods.

To generate selected getter and setter methods, go to `Source > Generate Getters and Setters...`

and then select those required.

Now let us check whether what we have done so far makes sense by trying out the code. To do this, we will define a new class.

Define a new class called `NumberTest`. This is only a temporary class that we define to help us during the development of our example (this is a common thing to do). This time, make sure that you create a main method for the class by ticking the following box:



In the main method, create a `NumberInfo` object with `limit=3` and display its value to test the getter, and then change the value to 4 to test the setter and display it again.

Concept Reminder:
`System.out.println` is a Java statement that prints the argument passed, into the `System.out`. In other words, this is a command that we use to display information on our output screen (or console).

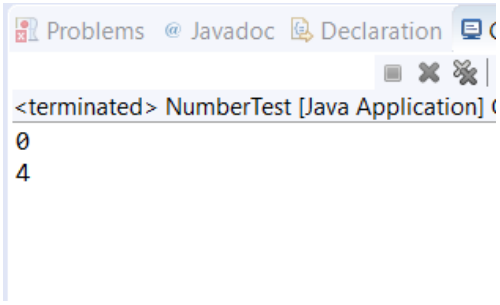
Use the following code to achieve this:

```
NumberInfo number = new NumberInfo(3);
System.out.println(number.getValue());

number.setValue(4);
System.out.println(number.getValue());
```

Run the `NumberTest` program. To do this, right-click on the project name, and select `Run As > Java Application`. Check the output on your console.

Go to next page

The output should be:



You will see that the value can easily be set to 4. This means that the limit field is having no impact on how the value changes, which does not make sense. Therefore, a simply defined setter method `setValue()` is not correct for the behaviour we want. Remove the definition from your `NumberInfo` and `NumberTest` code as we need to be more careful and we should always be selective about what setters we create.

Instead, we will define a new public instance method called `increment()` in the `NumberInfo` class. It has the return type `void` because it does not need to return anything. So it starts off with:

```
public void increment() {
}
```

Enter the body of the code as follows inside the increment method (there is an error in it but this is deliberate for the minute we will get you to fix it below):

```
this.value = (this.value) % this.limit;
```

In the `NumberTest` class, make the appropriate changes to ensure that your code reflects the following:

```
System.out.println(number.getValue());
number.increment();
System.out.println(number.getValue());
```

Run the code again using the `Run As > Java Application`. What went wrong?

4. Introduction to Modular Arithmetic in Java

Slight deviation from the exercise to introduce a useful operator: the % operator is called a modulus operator. If you want to see how the % operator behaves enter the following code underneath the code you already have in the `NumberTest` class and run it again.

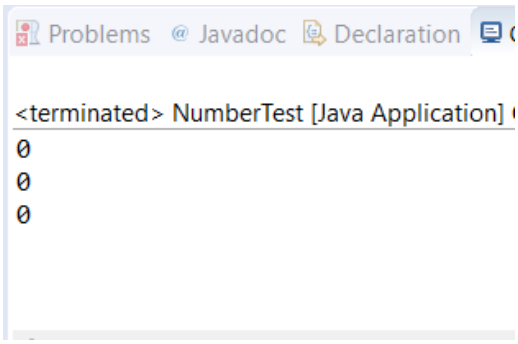
```
System.out.println("25 % 4 is " +(25%4));
System.out.println("25 divide by 4 is " +(25/4));

System.out.println("17 % 3 is " +(17%3));
System.out.println("16+1 % 3 is " +((16+1)%3));

System.out.println("17 divide by 3 is " +(17/3));
System.out.println("16+1 divide by 3 is " +((16+1)/3));
```

Inspect the examples with the % operator and ensure that you understand how it works, and why 25 % 4 returns 1, and 17 % 3 returns 2.

Now remove (or comment out) the above code, so that the main method in the `NumberTest` class only has the following:

```
public static void main(String[] args){
    NumberInfo number = new NumberInfo(3);
    number.increment();
    System.out.println(number.getValue());
    number.increment();
    System.out.println(number.getValue());
    number.increment();
    System.out.println(number.getValue());
}
```

Run the code again using the `Run As > Java Application`. What went wrong?



Despite calling `increment()`, the number is not increasing by 1 each time we call the method. This means that we are missing something in the body of the `increment()` method. Change the method to increment the value by 1 and try it again. Your code should look like this:

```
this.value++;
this.value = (this.value) % this.limit;
```

Concept Reminder:

In many programming languages, including Java, the increment operator ++, i.e. `value++` increases the value of the variable value by 1. This is the same as writing `this.value = this.value + 1;`

Note that we are incrementing the value before checking whether it is above the limit or not. If we did this the wrong way round, the limit would be exceeded.

Go to next page

4a. Create objects with complex objects as fields

Now we are ready to define the `ClockDisplay` class. Define the class (without a main method) to have two fields as follows so that a `ClockDisplay` object contains two `NumberInfo` objects:

```
private NumberInfo hours;
private NumberInfo minutes;
```

Define a default constructor. Within the body of the constructor create two `NumberInfo` objects with the limits of 24 for the `hours` object and 60 for the `minutes` object. These new statements go inside the constructor. The first object is given to you in the code below - do the same for the second object.

```
this.hours = new NumberInfo(24);
```

This is the first time that we will have created objects inside classes. So this shows that main is not the only place where objects can be created.

Concept Reminder:

The keyword newis used to create and instantiate new objects. For example,

```
ClassName objectName = new ClassName([parameters if needed]);
```

We will not define getters and setters for the attributes in `ClockDisplay` because we want to keep these protected and only use them to display the clock.

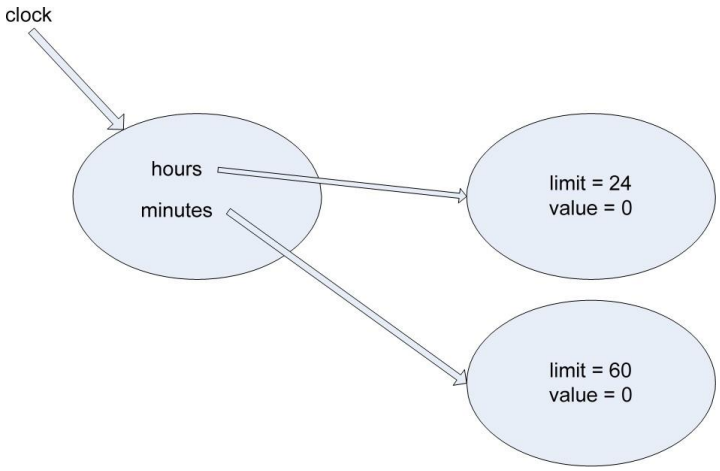
Now we want to define a method which is going to increment the minutes in the clock which also handles the changes to hours. We have got individual increment methods for hours and minutes as defined by `NumberInfo` , but we obviously want to keep them synchronised so that when the minutes have reached 60 the hour also increments. So define a method called `timeTick()` which returns nothing and takes no arguments. Complete the following by entering code on the line below the comments "//". (Hint: you will need to call the increment method on each object.)

```
public void timeTick() {
   // Increment the minutes.

   if (minutes.getValue() == 0) {
      // The minutes just rolled over so we must increment the hours.
   }
}
```

Next we turn our attention to testing the `ClockDisplay` object.

Define a `class` called `ClockDisplayTest`. This is where we will create a `ClockDisplayTest` object. Make sure the class has a main method. This gives us a way of experimenting with the `ClockDisplay` objects.

Inside the main method define a `ClockDisplay` object called `clock`. Therefore, what we visually have is:



Now add the following code after the definition of the `clock` object. The code contains a for loop, we will see these in detail later in the module but for now just concentrate on the fact that the two lines inside are being executed many times.

```
for (int i = 0; i < 100; i++) {
   // The next two lines of code will be executed 100 times.
   clock.timeTick();
   System.out.println(clock.toString());
}
```

Run the `ClockDisplayTest` program using the `Run As > Java Application`.

Explanation: The output is very strange, in fact the output is a hexadecimal representation of the `ClockDisplay` object like this:

```
org.com1027.lab2.ClockDisplay@1e63e3d
org.com1027.lab2.ClockDisplay@1e63e3d
org.com1027.lab2.ClockDisplay@1e63e3d
org.com1027.lab2.ClockDisplay@1e63e3d
...
```

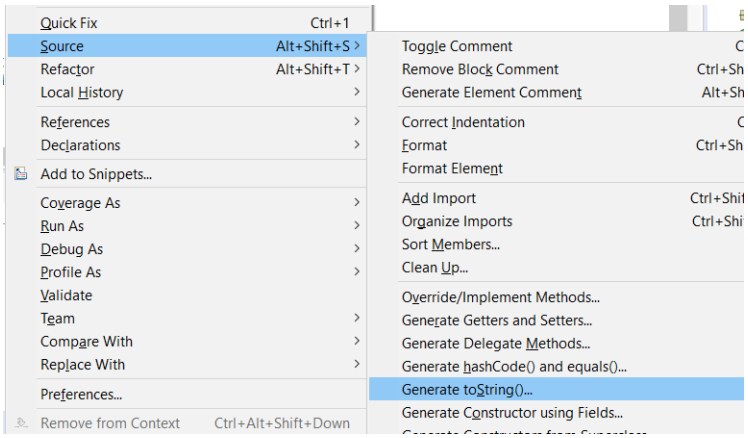
We would like the clock to display the numbers incrementing, but so far we have not written the code to display the clock correctly and hence when we call the built-in method `toString()`, Java automatically provides a basic version for you which just displays the memory reference to the clock object. The method `toString()` is one of a few automatically provided methods which come from the `Object` class.

To provide the required display behaviour we need to override this automatic behaviour by writing our own `toString()` method. - How?

4b. Use the toString() method to format the output of the complex object

We want a string representation of the `ClockDisplay` which looks like this 00:00. Therefore, in Java there is a particular method that is used for this purpose called `toString()`. Any programmer writing Java would know that the returned value of a `toString()` method is a representation of the object so this means that we should not just create a method with a different name to serve the same purpose.

Go back into the `ClockDisplay` class. Define a `toString()` method by right clicking on `Source -> Generate toString()` and a method is automatically generated. Note that the fields hours and minutes should be included in the `toString()` creation. (We will see later that we can change the automatically generated code to suit us.)



The new code looks like this:

```
/**
* @return
*
* @see java.lang.Object#toString()
*/
@Override
public String toString() {
   return "ClockDisplay [hours=" + this.hours + ", minutes=" + this.minutes + "]";
}
```

Notice: `@Override`. This means that the `toString()` method is a method that Java already provides automatically which we are changing in our own class. We are overriding the code that is already provided by Java and making it explicit ourselves.

Run the `ClockDisplayTest` again using the `Run As > Java Application`. This time you will see an output which is slightly more meaningful because we can see that there are two instance variables, but it is still not what we want to see outputted.

```
ClockDisplay [hours=org.com1027.lab2.NumberInfo@1b90b39, minutes=org.com1027.lab2.NumberInfo@18fe7c3]
ClockDisplay [hours=org.com1027.lab2.NumberInfo@1b90b39, minutes=org.com1027.lab2.NumberInfo@18fe7c3]
ClockDisplay [hours=org.com1027.lab2.NumberInfo@1b90b39, minutes=org.com1027.lab2.NumberInfo@18fe7c3]
ClockDisplay [hours=org.com1027.lab2.NumberInfo@1b90b39, minutes=org.com1027.lab2.NumberInfo@18fe7c3]

...
```

Now let us replace the body of the `toString()` method with our own code as follows:

```
DecimalFormat style = new DecimalFormat("00");
return ( style.format(this.hours.getValue()) +
   ":" + style.format(this.minutes.getValue())  );
```

Note: you will get a syntax error initially once you have entered this code. Correct it by importing the appropriate Java class. Put the following line immediately after the package statement at the top of the file. If you hover on the red cross, and choose QuickFix, it will also tell you that this is how to fix the error.

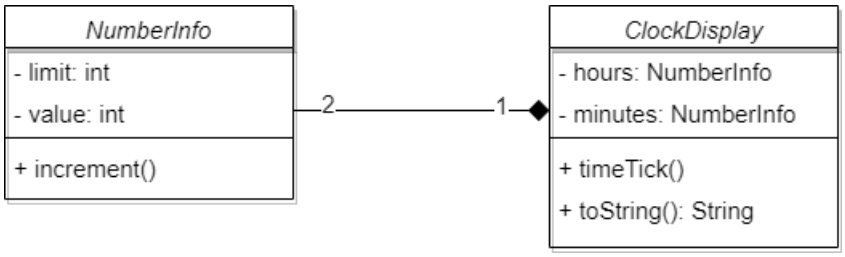
```
import java.text.DecimalFormat;
```

Explanation: This code creates a formatter object, style, and then passes the hours value to the formatter which returns a string padded with an explicit 0 if the hours is not two digits, similarly for minutes. Then the return statement concatenates the formatted hours and minutes together, separated by a colon.

Run `ClockDisplayTest` again using the `Run As > Java Application`.

```
00:01
00:02
00:03
00:04
...
```

You have created a class made up of nested objects and you have seen how to display a representation of the object using a `toString()` method. So we can summarise by saying that `ClockDisplay` "has a" `hours` object and "has a" `minutes` object. So this is why on the class diagram you will see the multiplicity "2" on the line between the classes (*see a copy of the class diagram below, or refer back to page 2*)



Go to next page

Exercise 1 (4 Marks)

This exercise requires you to revisit the `ClockDisplay` and `NumberInfo` classes from this week's lab sheet.

For this exercise, you will be using the `exercise1` package. Copy the classes `ClockDisplay` and `NumberInfo` from the `lab2` package and make relevant changes:

- Modify the `ClockDisplay` class so it can display both 12 hour clock and 24 hour clock. For example 1:41pm and 13:41.

  To do this, the constructor of the clock should be able to decide whether a 12-hr clock or a 24-hr clock should be created by passing a boolean variable to it as a parameteried constructor.

- For a 12-hr clock, change the internal representation of the time so that the value used to hold the hours displays from 0 to 11 for a 12 hour clock.

- Modify the `toString()` method and the `DecimalFormat` object to ensure that the 12-hr clock displays the time with the "am" or "pm" suffix. For example 1:41pm instead of 13:41.

- You may also want to refer to the Junit test under **src/test/java/exercise1** to understand the required functionalities, output format, data types of variables, etc. for this exercise.

To test your code, run the program. To do this, use the `Run As > JUnit Test`. This will enable you to check the functionality of code, and compare the expected with the actual output.

Important note: This is a challenging exercise and you may wish to complete this last.

Tip to complete this activity: A conditional statement may be needed (i.e. IF) to differentiate between the 12-hour and the 24-hour clock. For example, "if (hour >= 12)" where hour is an integer variable that holds the value of hours.

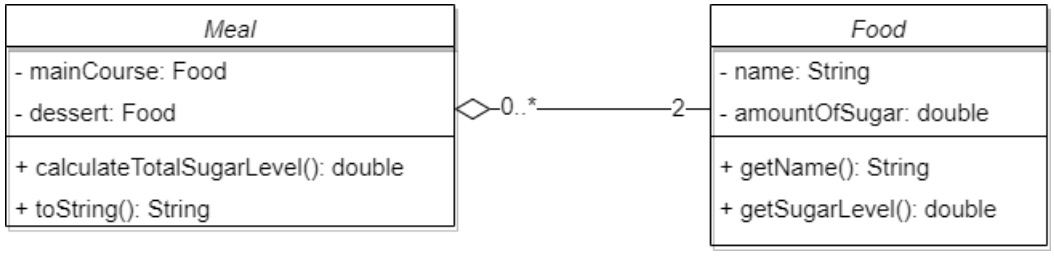
Go to next page

Exercise 2 (3 Marks)

This exercise requires you to create a new program using your understanding of classes, objects and complex objects.

Firstly make sure that you add all your code in the `exercise2` package.

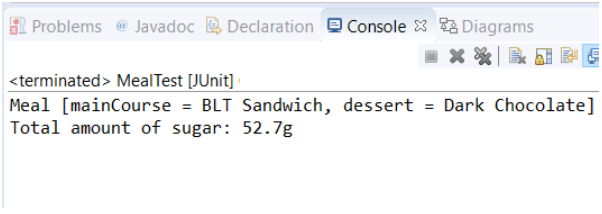
The aim of this exercise is to provide information about the amount of sugar in a meal, which consists of a main course and a dessert. The `Meal` and `Food` classes are related as shown in the following UML class diagram:



Concept Reminder:
Association is the relationship between two separate classes. This can be a one-to-one, one-to-many relationship, etc. In this case, the above UML diagram, shows an example of Aggregation which states that the objects of those classes have a has-a relationship, and can exist independently. In this case, the `mainCourse` and the `dessert` objects can exist without being part of a meal, but they contribute to a meal.

- Define a `Food` class. Each `Food` object comprises of a name and the amount of sugar in it. You need to provide getters for each attribute because in order for the `Meal` class to access the variable details we need these methods.
- Define a `Meal` class. A `Meal` object contains a main course and a dessert, and both can be defined as instance variables. *(**Note:** This is similar to the* `ClockDisplay` *class in the example)*

  As shown in the above UML diagrams, the `Meal` class also has two methods: the public `calculateTotalSugarLevel()` and the public `toString()` methods. The `toString()` method will be used to return the string representation of the object in the required format, and the `calculateTotalSugarLevel()` method which needs to return the total amount of sugar for the meal; from the main course and the dessert.
- Define a new class called `MealTest`. This is a temporary class (***Note:** This is similar to the* `ClockDisplayTest`) which will help us during the development of our code. This time, make sure that you create a main method for the class. Create a meal which consists of a 'BLT Sandwich' with 4.7g of sugar and 'Dark Chocolate' bar with 48g of sugar. Make use of the `System.out.println` command to display the required values on the console. Confirm that you get the same output as the example displayed below:



To test the overall functionality of your code, run the program. To do this, use the `Run As > JUnit Test`. This will enable you to check the functionality of code, and compare the expected output with the actual output.

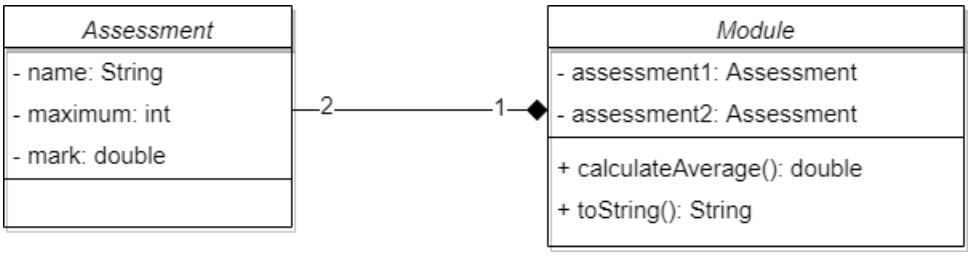
Go to next page

Stretch & Challenge Exercise (3 Marks)

This exercise requires you to create a new program using your understanding of classes, objects and complex objects with minimum support (i.e. Getters and Setters are omitted from the class diagram provided).

First, make sure that you add your code in the `exercise3` package.

The aim of this exercise is to provide a summary of someone's progress in COM1027 - Programming Fundamentals, by displaying their mark for each assessment as well as the module average. In this exercise, two classes are required; Assessment and Module. The Assessment and Module classes are related as shown in the following UML class diagram:



- Define an Assessment class. Each Assessment object comprises of the assessment's name and the marks achieved in that assessment. The maximum that can be achieved in an assessment is 100%. Thus, the instance variable maximumcan be set by default to 100.

  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 Module class which represents COM1027 - Programming Fundamentals. A Module object contains of two assessments; a lab assessment (worth 20%) and a coursework (worth 80%). Create a parameterised constructor that accepts both instance variables (assessment1 and assessment2 as parameters).

  As per the class diagram, the Module class also has two methods; the calculateAverage() and the toString() methods. The toString() method will be used to return the string representation of the object in the required format, and the calculateAverage() method which returns the module average.

- Define a new class called ModuleTest with a main method, to test your code. In the main method, create two Assessmentobjects, and one Moduleobject as follows:

  - 1 x `Assessment` object with the following details:

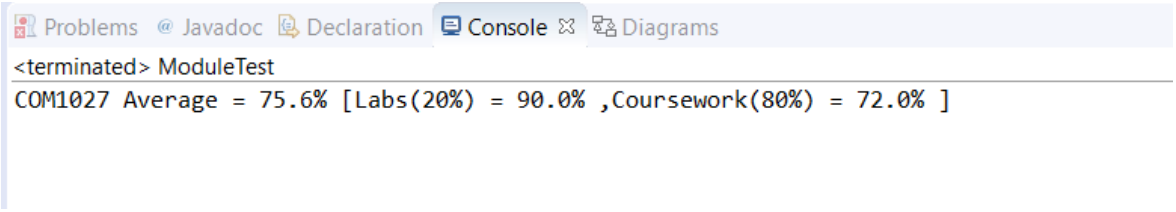
    Name: Labs , **Mark:** 90.0

  - 1 x `Assessment` object with the following details:

    Name: Coursework , **Mark:** 72.0

  - 1 x `Module` object that comprises of the above `Assessment` objects

  Confirm that you get the same output as the example displayed below:



To test the overall functionality, right-click on the project name and run the program using the `Run As > JUnit Test`. This will enable you to check the functionality of code, and compare the expected output with the actual output.

Go to next page

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 occured in the `testNoon`and on the right hand side it states the expected and actual values.

By selecting `Run As > JUnit Test` all the 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: 22, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2.687 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.