

COM1027 Programming Fundamentals

Lab 4

Using the debugger and an introduction to arrays

Purpose

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

- Use the Debugging tool in Eclipse,
- Declare and initialise Arrays
- Use Arrays and loops together to perform simple calculations (*Note: more on repetition statements in the coming weeks*)

This lab example is based on Chapter 5 of Helbert (2018), re-written for Eclipse.

- 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 marks 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>

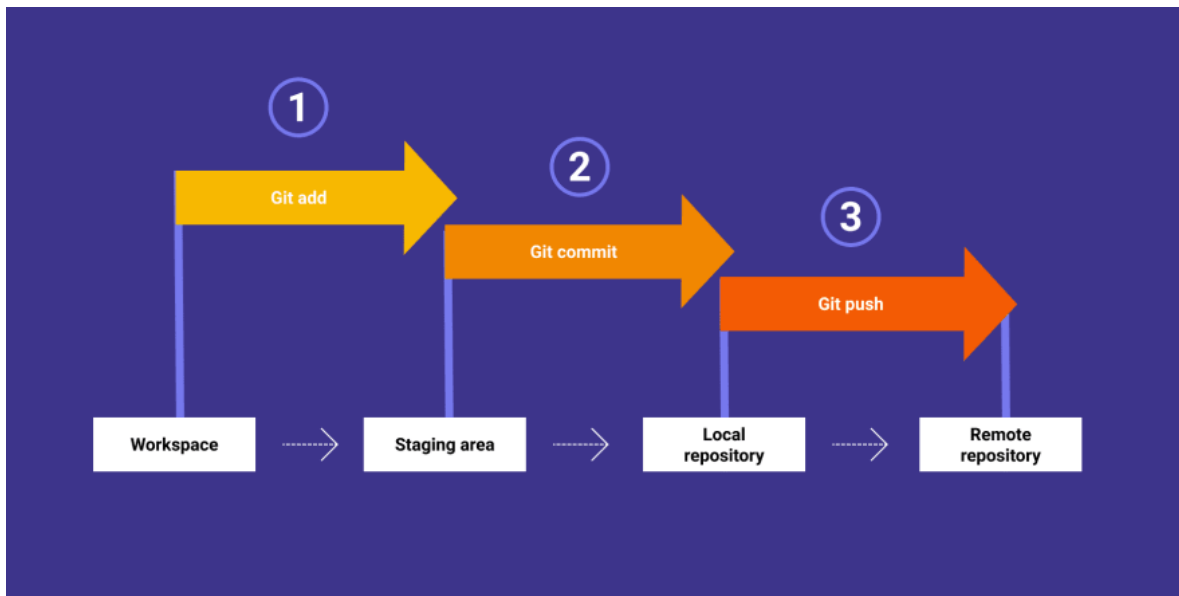
**Please ask us lots of questions during the labs, or use the discussion forums on SurreyLearn.
All the demonstrators are here to help you get a better understanding of the Java
programming concepts.**

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 (<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
- Committed 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 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-Lab04.zip`. Extract the contents of the project in the Downloads folder. Copy the uncompressed (unzipped) Lab04 folder **to your local repository**. **Note:** Do not copy the `COM1027_Lab04` folder. Navigate in the folder, and only copy Lab04 folder.

If you are using the computer labs, then your local git directory would look like this:

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

For example:

```
/user/HS223/sk0041/git/com1027sk0041/
```

where sk0041 shows a sample username.

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 Lab04 from your local git repository. Once the files are loaded, select the Eclipse project and click `Finish`.

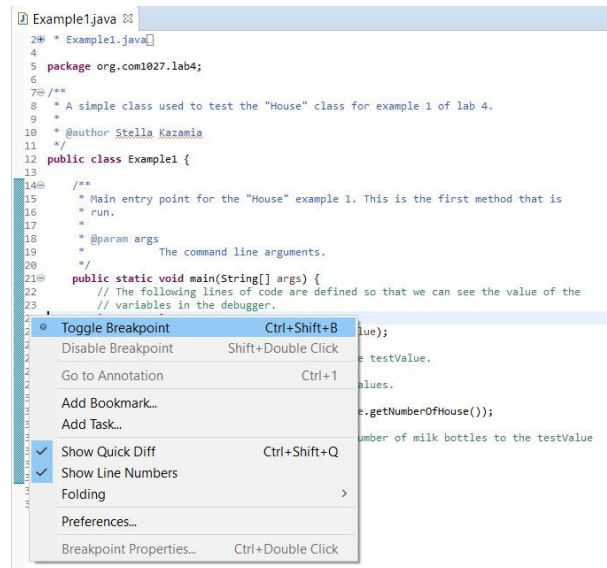
```
✓ Lab04 [com1027zf0001 master]
  ✓ src/main/java
    ✓ lab4
      > Example1.java
      > Example2.java
      > Example3.java
      > Example4.java
      > House.java
      > lab4_exercise1
      > lab4_exercise2
      > lab4_exercise3
    > src/test/java
    > JRE System Library [JavaSE-1.7]
    > Maven Dependencies
    > JUnit 4
    > src
    > target
    pom.xml
```

Once imported, run the `Example1` class under the `lab4` package as a Java application (right-click on `Example1.java` and select `Run As > Java Application`). You will see the following displayed in the console:

```
my value is 10
my house is 4
```

1. Use the debugging tool

We are now going to run the same program but using the debugger. Set a break point on the line of code which assigns the value 10 to `testValue`. To do this, right-click in the margin of the code and choose Toggle Breakpoint:



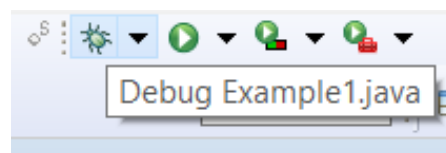
You should now be able to see a blue blob next to the line of code:

```

21 public static void main(String[] args) {
22     // The following lines of code are defined so that we can see the value of the
23     // variables in the debugger.
24     int testValue = 10;
25     System.out.println("my value is " + testValue);
26
27     // Write a line of code that increments the testValue

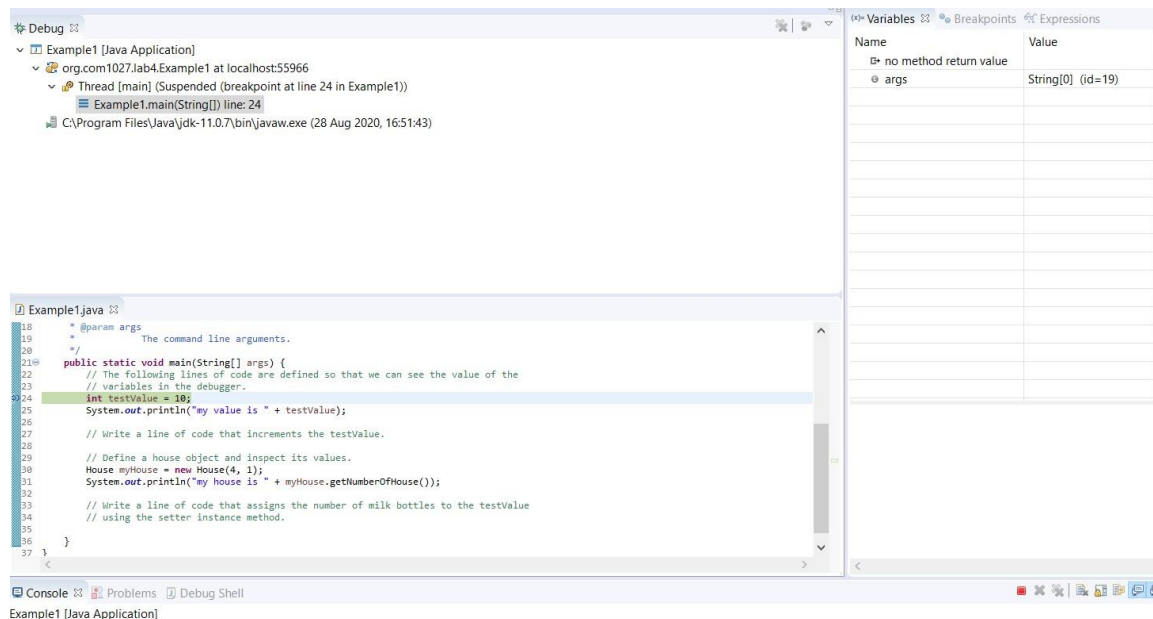
```

Run the code again in Debug mode by clicking on the icon.



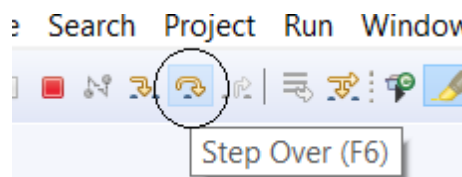
If you cannot locate the Debug icon, go to the 'Run' tab on the top of your screen, and then select 'Debug' from the drop-down menu.

Confirm that you wish to open the debugger perspective and you will see something like this:



Explanation: The top left Debug pane shows the stack of execution. This means that you can see which Java class you are currently running. The pane on the right-hand side shows the variables and breakpoints. In the middle we have the source code. Please note that other panes might be visible on your screen, including the ‘Workspace’, ‘Project Explorer’ etc.

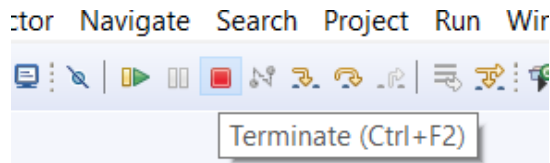
We are at the point where we can now create the `testValue` variable and assign it the value 10. You will see that the line is highlighted in green. Step over the code using the icon:



By doing this, the variable gets the value 10, and this can be seen in the debugging window. Also the green highlight has moved down a line.

(x)= Variables Breakpoints Expressions	
Name	Value
no method return value	
args	String[0] (id=19)
testValue	10

Now terminate the running of the code using the red square icon:



Click on the Java perspective in the top right to return to the package and code view.



Add the line of code underneath the comment that asks you to increment the `testValue`. (The toggle breakpoint should still be there on the previous line of code.)

Without running or debugging the program, add the line of code underneath the assignment of `myHouse` that asks you to assign the number of milk bottles to the `testValue`. Run the code again using the debugger and step through until you can see that the value has changed to 11.

<div> (x)= Variables Breakpoints Expressions </div>	
Name	Value
no method return value	
args	String[0] (id=19)
testValue	11

The line of code that creates a `House` object, should also be highlighted as green. You can now choose to step into using the ‘Step into’ icon:



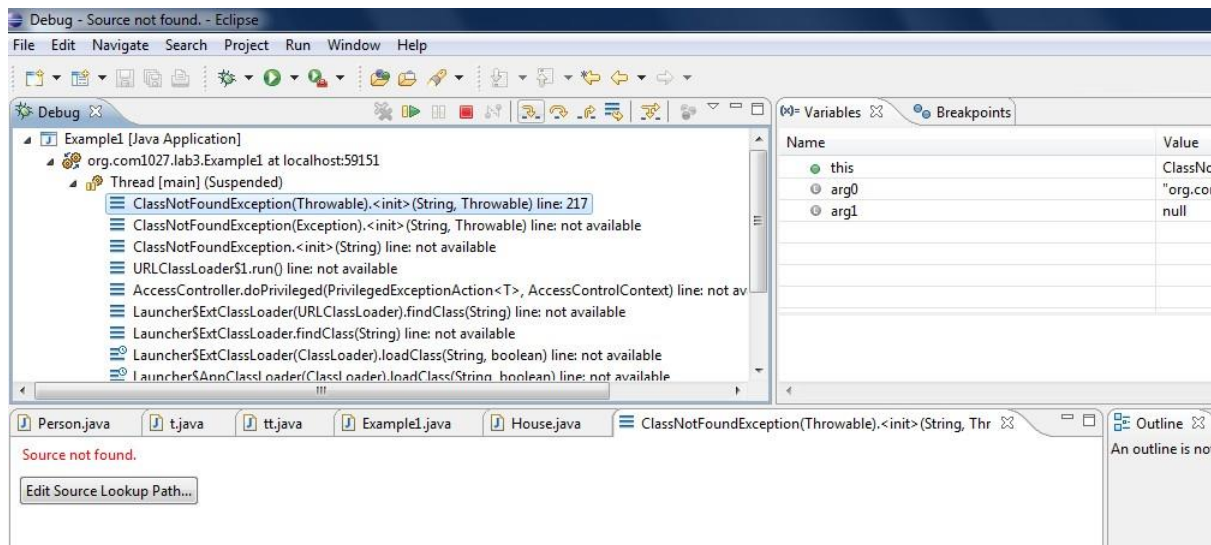
This should allow you to step into the constructor, which will enable you to see the details of the `House` constructor. *Note: If the debugging tool redirects you to the `ClassLoader` make sure that you click ‘Step Over’ and then ‘Step Into’.* This will allow you to exit the `ClassLoader` class and inspect the `House` constructor.

```

23  /**
24   * Constructor for class allowing the house number and number of milk bottles to be defined.
25   *
26   * @param numberOfHouse
27   *       the house number.
28   * @param numberBottles
29   *       the number of milk bottles required.
30   */
31  public House(int numberOfHouse, int numberBottles) {
32      super();
33      this.numberOfHouse = numberOfHouse;
34      this.numberBottles = numberBottles;
35  }

```

In older versions this was not the case, and instead you could only see the following warning:



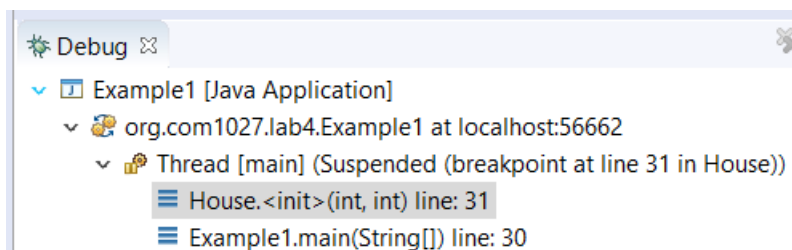
To be more precise with your debugging, choose to terminate the debugging (again using the red square) and return to the Java perspective.

Go into the `House` class and add a breakpoint on the first line of code in the constructor, `super()`. Now go back to the `Example1` class and run the debugger again. When you reach the line of code that creates a `House` object choose to step over this time.

```
public static void main(String[] args) {
    // The following lines of code are defined so that we can see th
    // variables in the debugger.
    int testValue = 10;
    System.out.println("my value is " + testValue);

    // Write a line of code that increments the testValue.
    testValue++;
    // Define a house object and inspect its values.
    House myHouse = new House(4, 1);
    System.out.println("my house is " + myHouse.getNumberOfHouse());
}
```

By having a breakpoint in the constructor, we can now see that the debugging tool looks into the `House` class directly. This is shown in the 'Debug' pane, where the `House` class is at the top of the stack which means that it is currently running:



Expand the variables of the current object (this). There are set to their default values. Keep the expanded view visible, so that we see how the values of the parameters will update them in the following lines of code:

(x)= Variables ☒ Breakpoints	
Name	Value
▼ this	House (id=40)
▪ numberBottles	0
▪ numberOfHouse	0
④ numberOfHouse	4
④ numberBottles	1

Step over the lines `super();` and `this.numberOfHouse = numberOfHouse;`. The instance variable `numberOfHouse` should be assigned a new value.

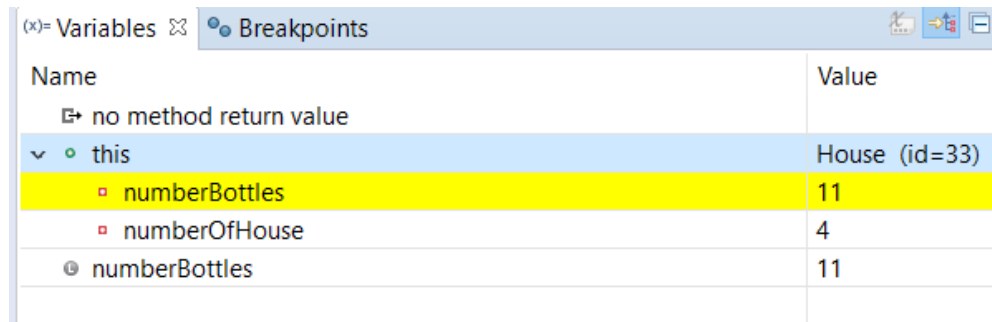
(x)= Variables ☒ Breakpoints	
Name	Value
✚ no method return value	
▼ this	House (id=40)
▪ numberBottles	0
▪ numberOfHouse	4
④ numberOfHouse	4
④ numberBottles	1

Step over again to assign the `numberBottles` instance variable too.

Step over again to return to the creation line in the `Example1` class code. Once you step over again you will see that the `myHouse` reference variable has been assigned appropriately:

(x)= Variables ☒ Breakpoints	
Name	Value
✚ no method return value	
④ args	String[0] (id=16)
④ testValue	11
▼ ④ myHouse	House (id=40)
▪ numberBottles	1
▪ numberOfHouse	4

On the line of code you added to assign the `testValue`, choose 'Step into' this time. This enables us to go into the instance method `setNumberBottles` to see the instance variable `numberBottles` updated to the following:



The screenshot shows the 'Variables' window in an IDE. The 'Breakpoints' tab is active. The 'Variables' section shows a tree structure. The 'this' object is expanded, showing its instance variables: `numberBottles` (value 11), `numberOfHouse` (value 4), and `numberBottles` (value 11). The `numberBottles` variable is highlighted in yellow.

Name	Value
no method return value	
this	House (id=33)
numberBottles	11
numberOfHouse	4
numberBottles	11

Explanation: This time since the object had been created we can use step into. We only have to be careful with trying to debug the code of the constructor. We could have put a breakpoint in the setter and chosen to step over, as we did with the constructor code. Then the debugging would have given us the same result as we have just seen.

Step over to return to the `Example1` code. If you continue trying to step over after the last left curly brace `}` you will see that the debugger continues executing, until it reaches the end of the stack.

This means that the execution has reached a point outside of your code and therefore we can no longer see the local variables.

Click on the red square to terminate the debugging session and return to the Java perspective (as shown earlier).

2. Creating arrays using primitive data types

An array is a collection of variables of the same type, referred to by a common name. In Java, arrays can have one or more dimensions, although the one-dimensional arrays are the most commonly used. In previous labs, we looked at simple variables of primitive or non-primitive data types. The principal advantage of an array is that it organises multiple data in such a way that it can be easily manipulated.

Concept Reminder:

To create an array, we need to first declare an array reference variable. For example `int[] houses;`.

We then need to allocate memory for the array, assigning a reference to that memory to the array variable using the `new` operator. For example, `houses = new int[size];`.

The number of elements that the array will hold is determined by `size`, which is always set as an integer value. In Java programming we can retrieve an array's size by using the built-in variable `length`. For example `houses.length;`

Go to `Example2`. Enter the following code inside the main method and then run the program.

```
// Code needing a new variable for each house.
int house1 = 3;
int house2 = 0;
int house3 = 4;

System.out.println("How many bottles in house 1 is " + house1);
System.out.println("How many bottles in house 2 is " + house2);
System.out.println("How many bottles in house 3 is " + house3);
```

Explanation: Clearly if you were trying to do this for lots of houses it just would not make sense, so let us define an array to hold the integer values.

Use the above concept reminder, to create an array called `houses` that holds three integer values. Make sure that you write the lines of code within the main method.

Although arrays in Java can be used just like arrays in other programming languages, they have one special attribute: they are implemented as objects, which means that we cannot simply display its values unless we call each element or use a loop.

In the lectures you have been introduced to a FOR loop, but in the following weeks we will explore loops in more details (including FOR EACH, WHILE, etc.). Loops, also known as repetition statements, allow us to loop through the same lines of code multiple times. Continue to add code at the bottom of the main method in `Example2`. Type in the following code after what you have entered so far and run the code:

```
// Create an array of fixed size.

houses[0] = 3;
houses[1] = 0;
houses[2] = 4;

System.out.println("How many bottles in house 1 is " + houses[0]);
System.out.println("How many bottles in house 2 is " + houses[1]);
System.out.println("How many bottles in house 3 is " + houses[2]);
```

Explanation: These two pieces of code have the same effect. They both store the number of bottles required for each house. There are two differences: 1) we use an array in the second piece of code; and 2) array indexes start at 0 and hence house 1 is referred to as `houses[0]`.

Now add the following line of code after all the other code so far in the main method of `Example2` and run it again as a Java Application:

```
// Will cause a runtime exception because [3] is not valid.  
houses[3] = 5;
```

Explanation: You will get a run time error because you are trying to assign to the 4th item in the array but earlier we declared the array to be of size 3: `new int[3]`. Indexing starts at 0, so the elements of the array are 0, 1 and 2.

Remove this line of code so that the program runs properly again.

The lines printing out the number of bottles per house are tedious because all we are doing is printing out each element of the array. We can use a loop to do this better.

After all the code you have written so far add:

```
for (int i = 0; i < houses.length; i++) {  
    System.out.println("How many bottles in house " + (i + 1) + " is " + houses[i]);  
}
```

Run the code as a Java Application. You should see that the printing is similar but more compact code. To make it clearer to the user we have used `i + 1` in the printing to offset the array index so that index 0 is associated with house 1.

Explanation: Note we are iterating through the whole array starting at 0 and stopping at the last index which is one less than the length of the array. The length of array is available from its `length` field, so `houses.length` is an integer value. We can also use this loop to count up the total number of milk bottles for the houses.

Declare a local variable just before the code entering the last for loop you have included. It will be of type `int` to hold a running counter. Then in the body of the loop, write one line of code which adds the number of the milk bottled for house `i` to the running total. After the for loop write a print statement which displays the total number of bottles, which should display the value 7.

An even more compact way of printing the contents of an array is to use a '`foreach`' statement on the array. Type in the following code and run the code again:

```
for (int house : houses) {  
    System.out.println("How many bottles in house is " + house);  
}
```

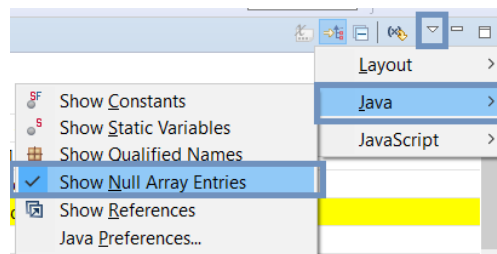
3. Arrays are fixed

Look at the `Example3.java` file and run it as a Java Application. Set a break point on line 12 of the code which assigns `names[1] = "Stella";`

Now run the code again in debug mode, and when the execution stops on line 12 expand the `names` array in the Variables view:

Variables Breakpoints	
Name	Value
no method return value	
args	String[0] (id=16)
names	String[20] (id=17)
[0]	"Helen" (id=20)
[1]	null
[2]	null
[3]	null
[4]	null
[5]	null
[6]	null
[7]	null
[8]	null
[9]	null
[10]	null

You will see that an array of 20 elements has been constructed and that the first element has been set to “Helen” but the others are set to null. If the null values don’t come up automatically then you will need to set the layout view. You can do this by doing the following:



Step over the line of code and you will see that the second element of the array is assigned the value “Stella”.

Variables Breakpoints	
Name	Value
no method return value	
args	String[0] (id=16)
names	String[20] (id=17)
[0]	"Helen" (id=20)
[1]	"Stella" (id=26)
[2]	null
[3]	null
[4]	null
[5]	null
[6]	null
[7]	null
[8]	null
[9]	null

names	String[20] (id=17)
[0]	"Helen" (id=20)
[1]	"Stella" (id=26)
hash	0
value	(id=27)
[0]	S
[1]	t
[2]	e
[3]	l
[4]	l
[5]	a

Terminate the code.

Explanation: This example shows simply that when we declare an array, which always has a fixed size, that memory is declared to hold the array elements. In the debugger we saw that the 20 elements of the array already existed before we put values into each element. *Note: In this example, each array element is a reference to a String object. In the next weeks, we will be discussing that String objects are constructed as an array of char values. This is also shown in the picture above.*

Consider what would happen if we wanted to add 21 names to the array? What would we have to do?

Try to do this by assigning a new value at position 20.

```
names[20] = "Bobby";
```

Save your work and run it again.

This should throw an `ArrayIndexOutOfBoundsException` exception on the console.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 20 out of bounds for length 20
    at org.com1027.lab4.Example2.main(Example2.java:23)
```

More information here:

<https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/IndexOutOfBoundsException.html>

To add another name in the array (at index 20), we will need to change the array declaration from:

```
String[] names = new String[20];
```

to:

```
String[] names = new String[21];
```

This can only happen at compile time unless we use a variable to define the length of the array. However the array size is still fixed and we cannot change the size of the array once it has been created.

What if we did not know how many array elements we needed until we started put values into the array?

4. Introducing ArrayLists

Re-write lines 7-18 from `Example3` in the corresponding methods in the `Example4` using `ArrayLists`.

- Create a new `ArrayList` that holds `String` objects, with the name `namesList`. This needs to be declared as a field.
- Define a default constructor
- Initialise the `ArrayList` with two values, “Helen”, “Stella”. This needs to be added in the `addNames()` method.
- Loop through the `ArrayList` using a loop of your choice (i.e. `for` loop, `for each`, etc.) This needs to be added in the `displayNames()` method, which is responsible of returning all the elements of the `ArrayList`, without any spaces. *Note: As this is a function, the `System.out.print` statement would need to be replaced with an appropriate `return` statement.*

Concept Reminder:

To create an array list, we need to first declare an array list reference, by using the `List` interface or the `ArrayList` class. For example `List<DataType> credits;`

We then need to allocate memory for the array list, assigning a reference to that memory to the array list variable using the `new` operator. For example `credits = new ArrayList<DataType>`, where the data type cannot be a primitive data type.

The number of elements that the array list will hold is not determined by a `size` at the point of declaration, like an array. In Java programming, array lists are dynamic and flexible and can increase or decrease their size during the runtime of the program. To retrieve an array list's size, you can use the built-in method `size()`. For example `credits.size()`.

Other built-in methods include the `add()`, `remove()`, `isEmpty()` methods. You can find a list of all the built-in methods by adding a fullstop at the end of the array list. This will return a drop-down menu with the various methods that can be applied to that array list.

Alternatively, you can visit the `ArrayList` class in the Java API specification which includes a short description of each built-in method: <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/ArrayList.html>

Save all your work, and run the Java project as a JUnit Test. To do this, right-click on the project name, and select ‘Run As > JUnit Test’.

Exercise 1 (3 Marks)

This exercise requires you to use the `House` class from this week's lab sheet.

First, ensure that the code for this exercise is added in the `lab4_exercise1` package. Copy the class `House` from the package `lab4` and make the relevant changes to a new class called `Neighbourhood`:

- Declare and initialise an array field called `houses`. The array should contain 3 elements of type `House` (instead of type `int` as we saw in the examples above). A partly complete code is shown below:

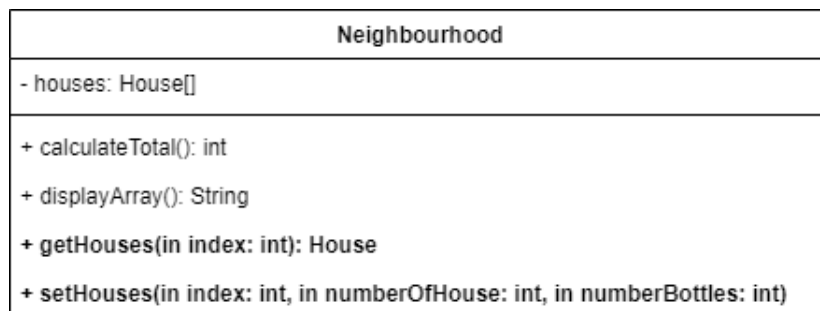
```
..... = new House[3];
```

- Create a default constructor. Within the constructor, initialise each element of the array. An example is shown below:

```
..... House(3, 4);
```

which represents a `House` object; house number is 3, and expects the delivery of 4 milk bottles. Use the expected output to guide you on how to initialise the three elements in the array.

Define setters and getters for the array field as shown in the UML diagram below. **Note: Do not auto-generate these as they will not match the UML below:**



- Define two new methods called `calculateTotal` and `displayArray`.
 - The `calculateTotal` method should be defined as a function with an `int` return type.
 - The `displayArray` method should also be defined as a function with a `String` return type.
- Both methods should include a repetition statement (loop) of your choice (see previous examples), in order to provide the following output:

```
Number of bottles in House number 3 is 4
Number of bottles in House number 5 is 1
Number of bottles in House number 7 is 2
Total number of bottles to be delivered: 7
```

Note: If you are not confident yet in using loops, consider using individual lines of code, i.e. `house[0]`, `house[1]` etc. to calculate the total, and display its values.

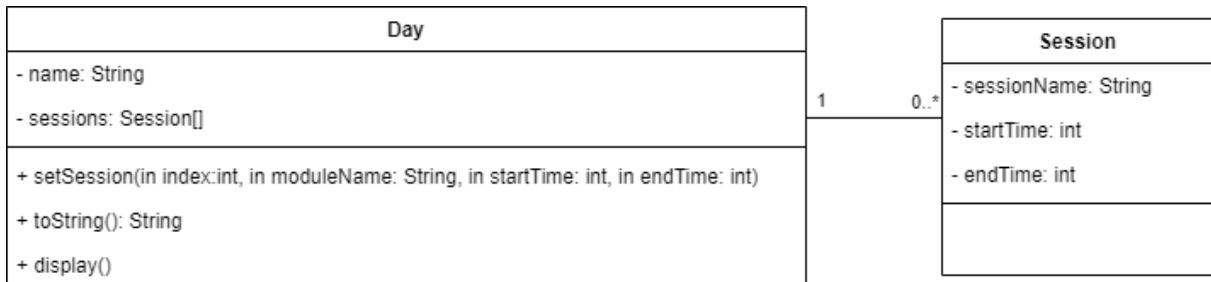
- To display the output on the console and test your code, create a `main` method. Within the `main` method define the appropriate objects and call the `displayArray` user-defined method with the use of the `System.out.println` statement.
- Test your code by right-clicking the project name and then `Run 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!

Exercise 2 (3 Marks)

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

The code for this exercise, will need to be added in the `lab4_exercise2` package.

The aim of this exercise is to create a brief timetable for a given day, which consists of the session's name, and its start and finish time. The `Day` and `Session` classes are related as shown in the following UML class diagram:



- Define a `Session` class. Each `Session` object comprises of a name and also its start and finish time. The time is represented as an integer, for example 9 meaning 9:00am and 20 meaning 8:00pm. You need to decide on whether getters and setters are required for each (or some) attribute.
- Define a `Day` class. A `Day` object contains an array of all the sessions scheduled. It also holds the name of the day, i.e. "Monday". Both can be defined as instance variables.
- Within the `Day` class, define a parameterised constructor with one parameter. Initialise the array in the constructor as follows:

```
this.sessions = new Session[4];
```

As per the class diagram, the `Day` class also has three methods; a setter, a `toString()` and a user-defined method called `display()`.

- Define the methods in the `Day` class. The `toString()` method will be used to return the string representation of the object in the required format (only the values of the `sessions` variable). For example, "COM1027: 9 - 10" which represent the name of the session, as well as its start and end time. The `display()` method needs to return the output on the console with the use of the `System.out.println()` statement (including the name of the day).
- Define a new class called `DayTest`. This is a temporary class which will help us during the development of our code. This time, make sure that you create a main method for the class, and call the relevant methods to check your work. Confirm that you get the same output as the example displayed below:

```

<terminated> Day [Java Application] C
Monday
COM1027: 9 - 10
COM1025: 15 - 17

Tuesday
COM1026: 13 - 15
  
```

The screenshot shows the output of a Java application. It displays the days of the week and the sessions scheduled for each day. For Monday, sessions COM1027 (9-10) and COM1025 (15-17) are listed. For Tuesday, session COM1026 (13-15) is listed.

- Before moving to Exercise 3, test your code by right-clicking the project name and then Run As > JUnit Test. **It is important that Exercise 2 works before moving to Exercise 3.**

Stretch & Challenge Exercise (6 Marks)

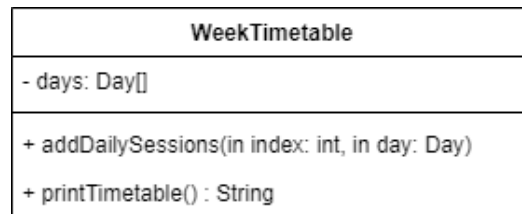
This exercise requires you to use the `Day` and `Session` classes from the previous exercise.

First, make sure that you use the correct package for your code:

```
src/main/java/lab4_exercise3.
```

The aim of this exercise is to represent a simple timetable for a given week. **Note:** No changes are required in the pre-defined classes, but a new class called `WeekTimetable` needs to be introduced.

Copy the `Day` and `Session` classes to the new package, and create a new class called `WeekTimetable`. The UML class diagram below, shows the attributes and methods of the class.



- Each `WeekTimetable` object comprises of an array that represents the five working days of the week. In previous activities, we made use of getter and/or setter methods. Are they required in this exercise?
- Define a default constructor for the class that allows an object to be created without any parameters. Initialise the array in the constructor!
- Add the `addDailySessions()` method. This should be written in a concise manner, by accepting a `Day` object as input. For example, `addDailySessions(0, day1)` where `day1` is an object of `Day`. The value 0 (zero) represents the index of the `days[]` array, therefore by calling `addDailySessions(0, day1)`, the first element of the `days[]` array is initialised with the `day1` object.
- Add the `printTimetable()` method. This should return a `String` with the required information in the specified format:

```

Monday
COM1027: 9 - 10
COM1025: 15 - 17

5 Tuesday
COM1026: 13 - 15

Wednesday
Sport: 13 - 18
10 Thursday
Spanish Language Lessons: 9 - 11
COM1027: 11 - 13

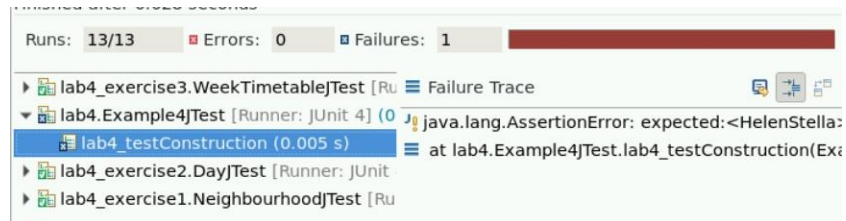
15 Friday
```

- You can test your code, by creating a `TimetableTest` class with multiple objects. Try to recreate the above output in your program.

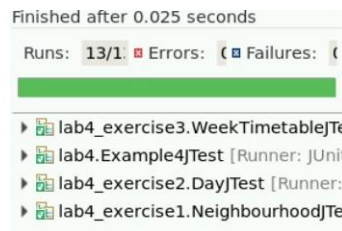
Final Steps

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 (Lab04) 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 `testConstruction` 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: 13, Failures: 0, Errors: 0, Skipped: 0
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 4.208 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.

