

Lab 2- Black-box Testing (test-case design using software requirements): Automated API Unit Testing using JUnit

Developed and maintained by: Dr. Vahid Garousi

vgarousi@gmail.com
www.vgarousi.com

Part of the Open Software Testing Laboratory Courseware:

sites.google.com/view/software-testing-labs

TABLE OF CONTENTS

REVISION HISTORY OF THIS DOCUMENT:	3
1 INTRODUCTION	3
1.1 Objectives.....	3
1.2 This lab is a Group work.....	3
1.3 Viewing and browsing the lab-doc PDF file with the document map (outline)	3
1.4 Tools to be used in this lab.....	4
1.4.1 JUnit testing framework	4
1.4.2 Javadoc	5
1.4.3 Java, JDK, JRE, and Eclipse IDE versions.....	6
1.5 Software System Under Test (SUT)	6
1.6 Overview of the lab work	7
2 INSTRUCTIONS	8
2.1 Familiarization and learning.....	8
2.1.1 Creating a new Eclipse project.....	8
2.1.2 Both students of each group shall have a working set up in their own machines (laptops, PC etc.).....	9
2.1.3 Adding the necessary Java libraries to your project	9
2.1.4 Add JUnit4 (version 4) to your project	10
2.1.5 Reviewing the SUT (JFreeChart) and running its demos.....	11
2.1.6 Project set-up can be tricky, as the IDE settings are quite sensitive and error prone	12
2.1.7 Navigating Javadoc API specifications.....	13
2.1.8 Developing a simple JUnit test class for the <code>org.jfree.data.Range</code> class	14
2.1.9 Developing a simple JUnit test class for the <code>org.jfree.data.DataUtilities</code> class	18
2.1.10 Using “implementing” classes for “interfaces” in Java.....	21
2.1.11 Asserting for expected “exceptions” in JUnit	22
2.1.12 Errors versus failures in JUnit test-case executions	26
2.1.13 Designing test cases for methods / functions with no parameters and developing test code for them.....	26
2.2 Work to be done: Design and development of unit test cases.....	27
2.2.1 Test requirements (classes to be tested).....	28
2.2.2 The first step: Develop and document (write) your unit testing plan.....	28
2.2.3 Designing the test cases, BEFORE developing them in JUnit	29
2.2.4 Developing your test code based on your test-case design.....	29
3 SUMMARY OF THE LAB AND LEARNING OUTCOMES	30
4 DELIVERABLES	30
4.1 Lab Report.....	30
4.2 JUnit test-suite test-code	30
ACKNOWLEDGEMENTS	30
APPENDIX A – TYPES OF ASSERTION METHODS AVAILABLE IN JUNIT	32
APPENDIX B – JAVADOC EXAMPLE	33
APPENDIX C – CONVENTIONS FOR TEST-CODE ARCHITECTURE AND NAMING IN JUNIT	34
C1: Test-code architecture	34
C2: Test Classes	34
C3: Test methods and their naming convention	34
C4: Asserting for expected exceptions in JUnit	35

APPENDIX D – UNDERSTANDING THE STEPS OF A TEST METHOD IN XUNIT.....	37
D1: Setup	37
D2: Exercise	37
D3: Verify.....	37
D4: Tear-Down	37
APPENDIX E – A VERY SHORT EXAMPLE OF A BLACK-BOX UNIT-TEST PLAN	38

REVISION HISTORY OF THIS DOCUMENT:

Summer 2008	First version was developed by Dr. Vahid Garousi and his team at U of Calgary
September 2010-2017	Various improvements were made, using experience from using these lab in various course offerings and feedbacks from students
Fall 2021	Made various improvements using student comments, provided between 2017-2021

1 INTRODUCTION**1.1 OBJECTIVES**

There are several key objectives of this lab. The first is to introduce students to the fundamentals of automated unit testing, specifically unit testing based on requirements for each unit. The most widely used unit testing tool for Java is the JUnit framework, which is a part of the XUnit framework family.

After completing the lab, students will be able: to develop automated test code in JUnit and other XUnit testing frameworks such as NUnit, CSUnit, PHPUnit, etc.

1.2 THIS LAB IS A GROUP WORK

All the tasks of this lab should be completed in groups of two (as you have formed before). One single report has to be completed and submitted for each group. In the report, there should be a section which should discuss "How the team work/effort was divided and managed". As we discuss in Section 4, a Word template file is provided by the professor which already has a heading for this purpose.

1.3 VIEWING AND BROWSING THE LAB-DOC PDF FILE WITH THE DOCUMENT MAP (OUTLINE)

We know that reviewing and browsing large documents is always more convenient when having a document "map" (outline). We have enabled this feature in the PDF file of the lab document. You need to download the PDF and open it using the Adobe software (as shown in Figure 1) - not opening the file in Canvas via your browser. You need to enable the "Bookmark" mode in the PDF reader software, as shown in Figure 1. Then, you can easily click on each section or sub-section and the PDF reader will jump to that section, enabling you to see the "big picture" of the document and also to conveniently browsing the document.

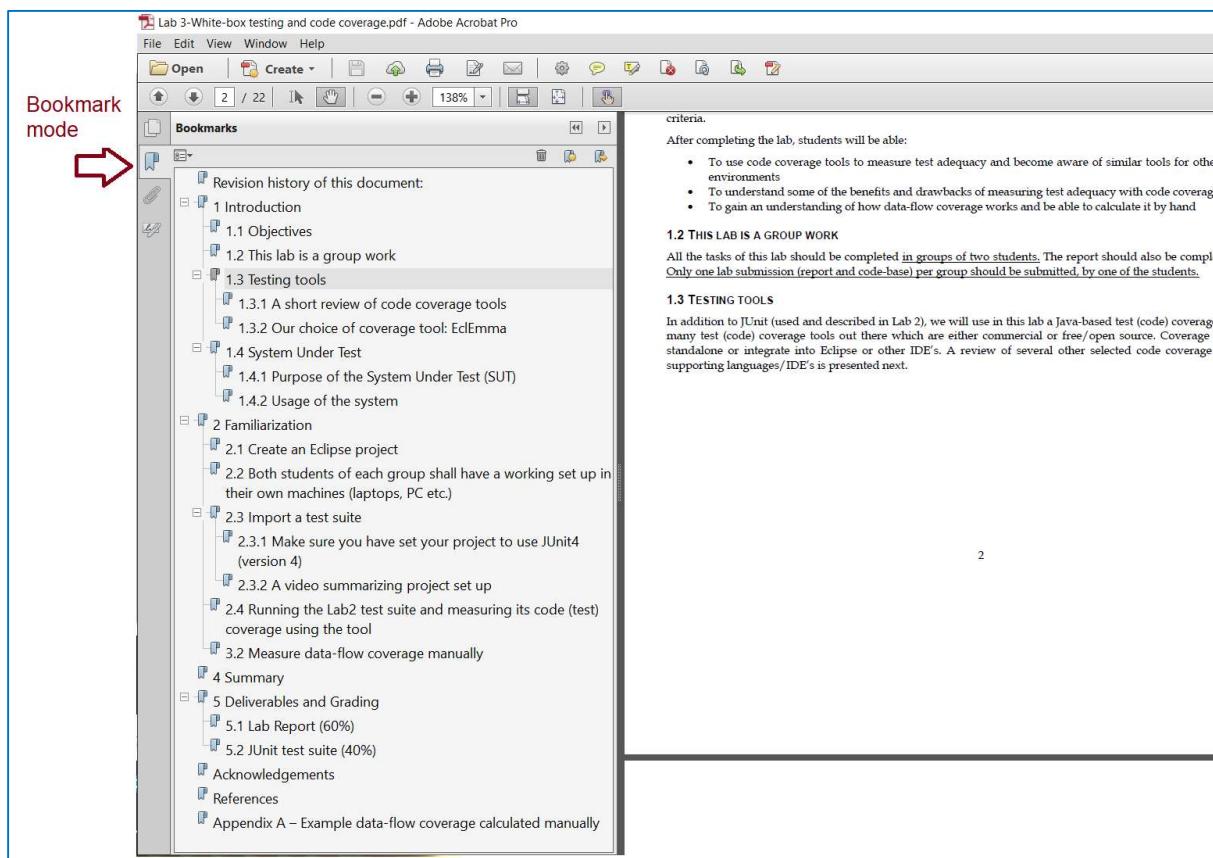


Figure 1 – Viewing the lab doc PDF file with the document map (outline)

1.4 TOOLS TO BE USED IN THIS LAB

1.4.1 JUnit testing framework

The main testing tool for this lab is JUnit (www.junit.org). JUnit is a widely-used unit testing framework for Java. JUnit has been important in the development of test-driven development (TDD), and is one of a family of unit testing frameworks which is collectively known as *xUnit* (www.wikipedia.org/wiki/XUnit), e.g., NUnit for .Net, PHPUnit for PHP, CppUnit for C++, etc.

JUnit

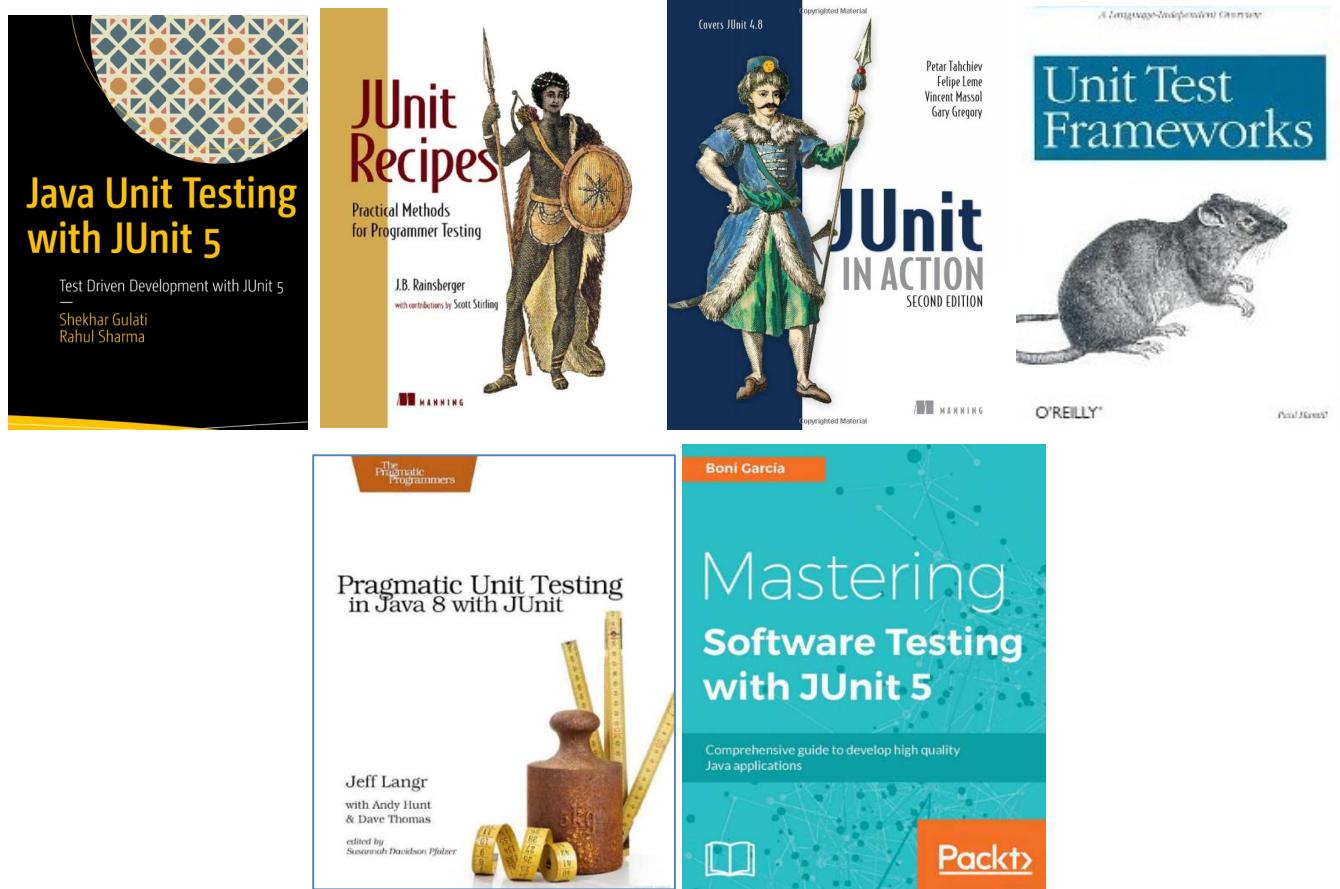
JUnit was first developed in early 2000's by a team of practitioners and soon "*JUnit took off like a rocket - and was essential to supporting the growing movement of Extreme Programming and Test Driven Development*"¹. JUnit has evolved through the years, and each new version has brought new features and improvements in it. As of 2021, its latest version is 5². But for this lab, we prefer to **use the version 4** of JUnit, since it is better for learning purposes.

JUnit has seamless integration with many Integrated Development Environments (IDE's), e.g., the Eclipse development environment [1]. The JUnit framework allows developers to quickly and easily develop unit tests and test suites, and execute them.

There are many online resources and books on JUnit. Some of the many books on JUnit are shown below:

¹ martinfowler.com/bliki/Xunit.html

² mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api



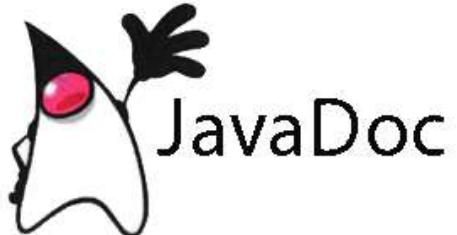
1.4.2 Javadoc

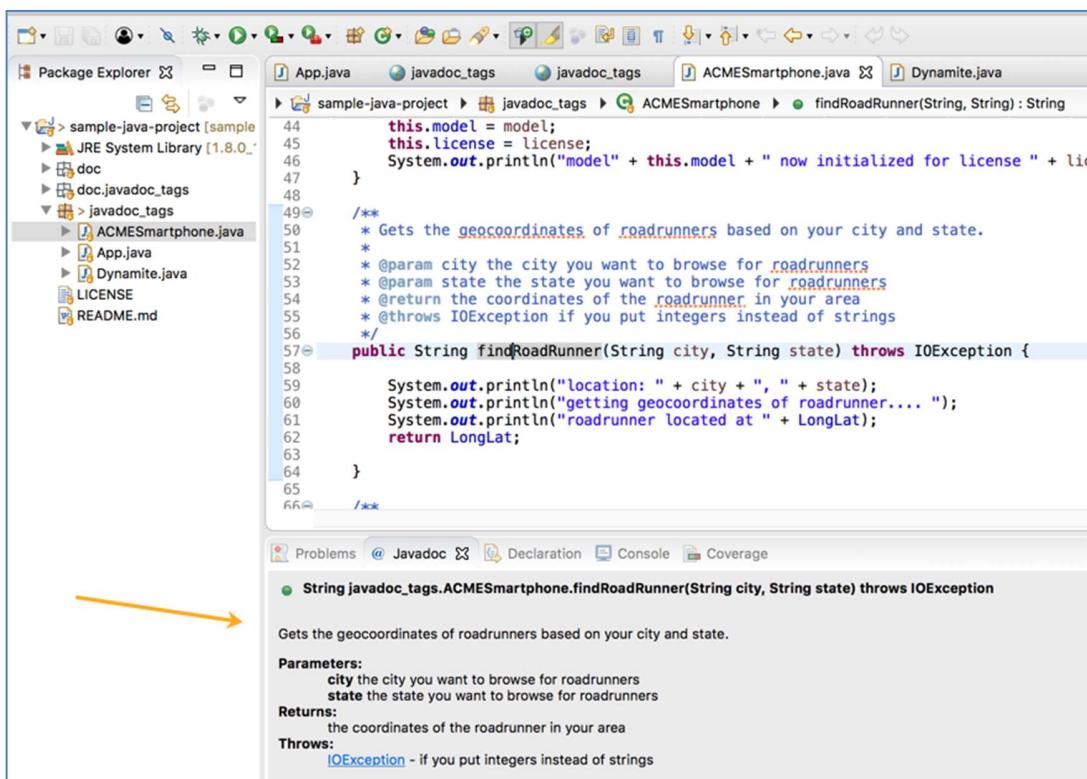
Another tool which is often used for the purpose of testing is Javadoc (www.wikipedia.org/wiki/Javadoc). Javadoc is a software documentation generator created by Sun Microsystems for Java for generating API documentation in HTML format from Java source code.

While Javadoc is not a testing tool, it will be used in the context of this lab as the format in which the requirements specification of the System Under Test (SUT) has been specified in. We will use requirements specification in the Javadoc to derive black-box test cases.

Developing the documentation and the code in the same location does not only improve communication between developers, maintainers and testers, but it also makes it simpler to keep the documentation up to date, and prevent potential redundancies and/or mistakes. For more information on Javadoc, see java.sun.com/j2se/javadoc.

A screenshot of how documentation using Javadoc is developed and derived is shown in the following.





The screenshot shows the Eclipse IDE interface. In the top right, several Java files are listed: App.java, javadoc_tags, ACMESmartphone.java, and Dynamite.java. The ACMESmartphone.java file is open in the editor, showing code related to roadrunner coordinates. Below the editor is the Javadoc view, which displays the Javadoc tags for the findRoadRunner method. A yellow arrow points from the text "Javadoc tags" in the previous slide to the Javadoc view in the IDE.

```

44     this.model = model;
45     this.license = license;
46     System.out.println("model" + this.model + " now initialized for license " + lic
47   }
48
49 /**
50  * Gets the geocoordinates of roadrunners based on your city and state.
51  *
52  * @param city the city you want to browse for roadrunners
53  * @param state the state you want to browse for roadrunners
54  * @return the coordinates of the roadrunner in your area
55  * @throws IOException if you put integers instead of strings
56 */
57 public String findRoadRunner(String city, String state) throws IOException {
58
59   System.out.println("location: " + city + ", " + state);
60   System.out.println("getting geocoordinates of roadrunner.... ");
61   System.out.println("roadrunner located at " + LongLat);
62   return LongLat;
63
64 }
65
66 /**

```

String javadoc_tags.ACMESmartphone.findRoadRunner(String city, String state) throws IOException

Gets the geocoordinates of roadrunners based on your city and state.

Parameters:

- city the city you want to browse for roadrunners
- state the state you want to browse for roadrunners

Returns:

- the coordinates of the roadrunner in your area

Throws:

- IOException - if you put integers instead of strings

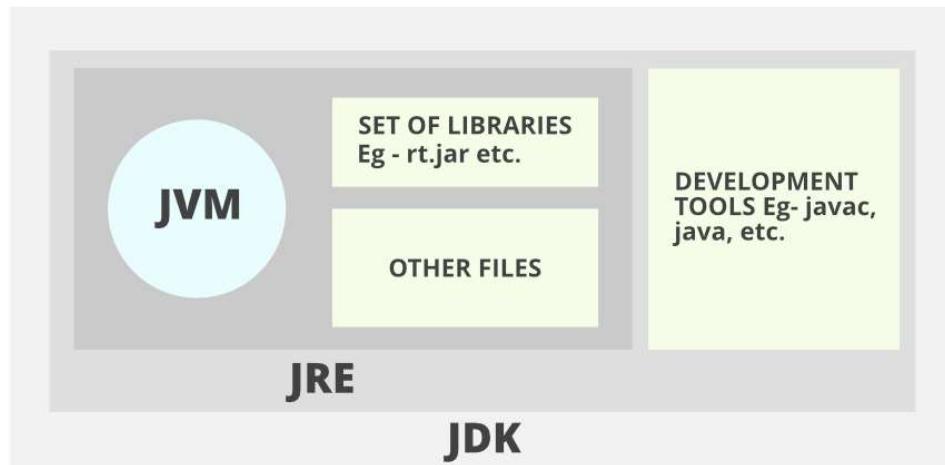
1.4.3 Java, JDK, JRE, and Eclipse IDE versions

Note that like all labs in this course, students shall use the **latest versions** of Java JDK (Java Development Kit), JRE (Java Runtime Environment), and the Eclipse IDE.

To know what JDK and JRE are, and how they differ from each other, see the figure below; and also read more here: google.com/search?q=jre+vs+jdk

Java downloads can be found in: www.oracle.com/java/technologies/downloads

You need to make sure that all your JDK / JRE files are the latest versions.



1.5 SOFTWARE SYSTEM UNDER TEST (SUT)

The System Under Test (SUT) which we will test in this lab is JFreeChart (www.jfree.org/jfreechart). JFreeChart is an open source Java framework for programmatically creating and displaying charts in Java. This framework supports many

different (graphical) chart types, including pie charts, bar charts, line charts, histograms, and several other chart types. A snapshot of four different types of charts drawn using JFreeChart is shown in Figure 2.

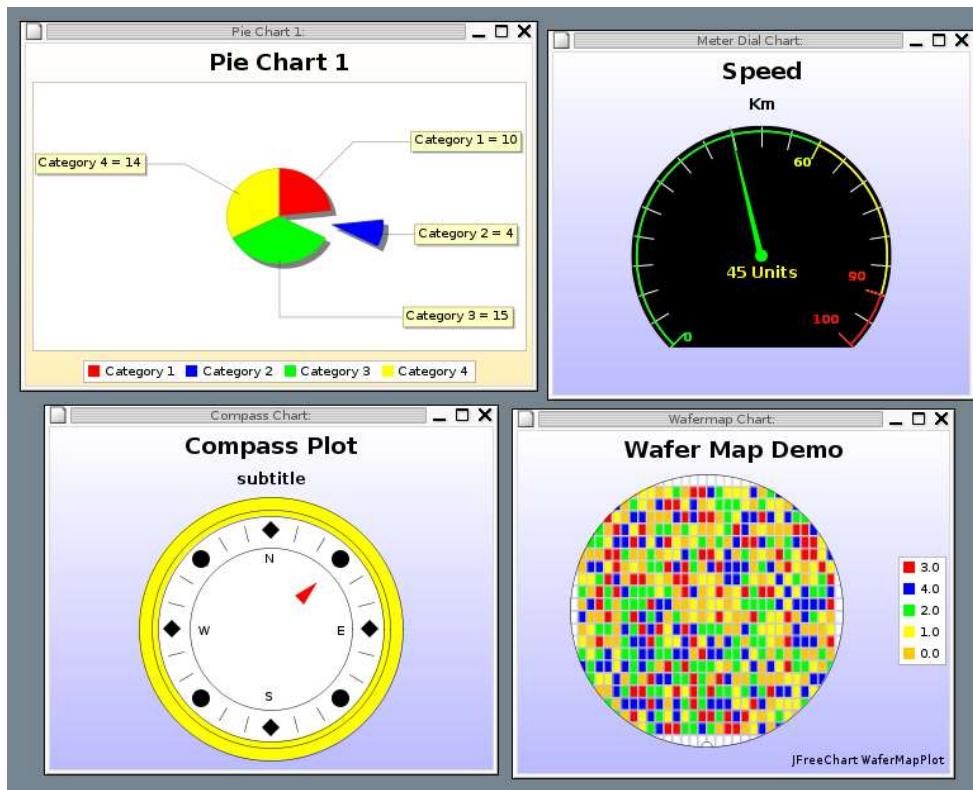


Figure 2 - A snapshot of four different types of charts drawn using JFreeChart.

To get started with the JFreeChart system, download the lab artifacts and extract the entire archive to a known location. Then, find “jfreechart.jar” file inside the artifacts. More information on how to get started with these files will be provided in the familiarization stage (Section 2.1). Note that the versions of JFreeChart distributed for this lab do not correspond with latest releases of JFreeChart. The versions have been slightly modified for the purposes of this lab.

The JFreeChart framework is intended to be integrated into other systems as a quick and simple way to add charting functionality to Java applications. With this in mind, the API for JFreeChart is required to be relatively simple to understand, as it is intended to be used by many developers as an open source off-the-shelf framework.

While the JFreeChart system is not technically a stand-alone application, the developers of JFreeChart have created several demo classes which can be executed to show some of the capabilities of the system. These demo classes have *Demo* appended to the class name. For the purpose of this lab, full knowledge of the usage of the JFreeChart API is not particularly necessary.

1.6 OVERVIEW OF THE LAB WORK

The lab-work is divided into three main sections. Similar to lab 1, the first section is familiarization (Section 2.1). During the familiarization stage, students will be shown how to set up a JUnit test project in Eclipse using JUnit, develop a simple unit test, and how to navigate through software requirements of the SUT (JFreeChart) in the Javadoc documentation.

After familiarization, the second section of the lab is designing the test cases using the software requirements specified in Javadoc, and development of unit tests. In this stage, students will develop unit test suites for several classes of the SUT. These test suites will be comprised of unit tests which have been generated based on the requirements.

Finally, upon completion of the test suites, during the last stage, the tests will be executed on several versions of the SUT and test results will be collected and recorded.

Deliverables of the lab will include: lab report (50% of the lab mark), and JUnit test suite (50% of the lab mark). Details about deliverables and grading are discussed in Section 4.

2 INSTRUCTIONS

This section details the instructions for executing the lab. All sections of this lab should be completed as a group as students have already formed.

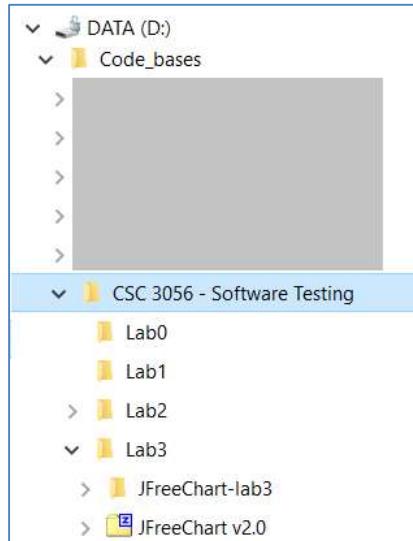
2.1 FAMILIARIZATION AND LEARNING

Both students of each group should perform this section of the lab together on a single computer. Ensure that both of you understand the concepts in this section before moving on to the rest of the lab.

1. Download the "Lab 2- artifacts" ZIP (RAR) file from the course website.
2. Extract the contents of the ZIP RAR file into a known location. The contents should include: the *jfreechart.jar*

2.1.1 Creating a new Eclipse project

3. Open Eclipse Java IDE
4. Open the *New Project* dialog by selecting the *File -> New -> Project...*
5. We **strongly** recommend that you structure your folder structure like the following: having a separate folder for each lab in this course:



6. When you are creating the New Project, in the first screen, make sure to check that both JDK (Java Development Kit), JRE (Java Runtime Environment), are properly installed on your machine and also accessible from Eclipse. To check this, you need to see the latest versions of JDK and JRE are shown in the first window of the *New Java Project* screen, as shown below. If the JDK and JRE versions are not listed in the screen below, you need to fix your JDK and JRE setup, via instructions found in online how-to's, such as: google.com/search?q=JDK+and+JRE+setup+in+eclipse

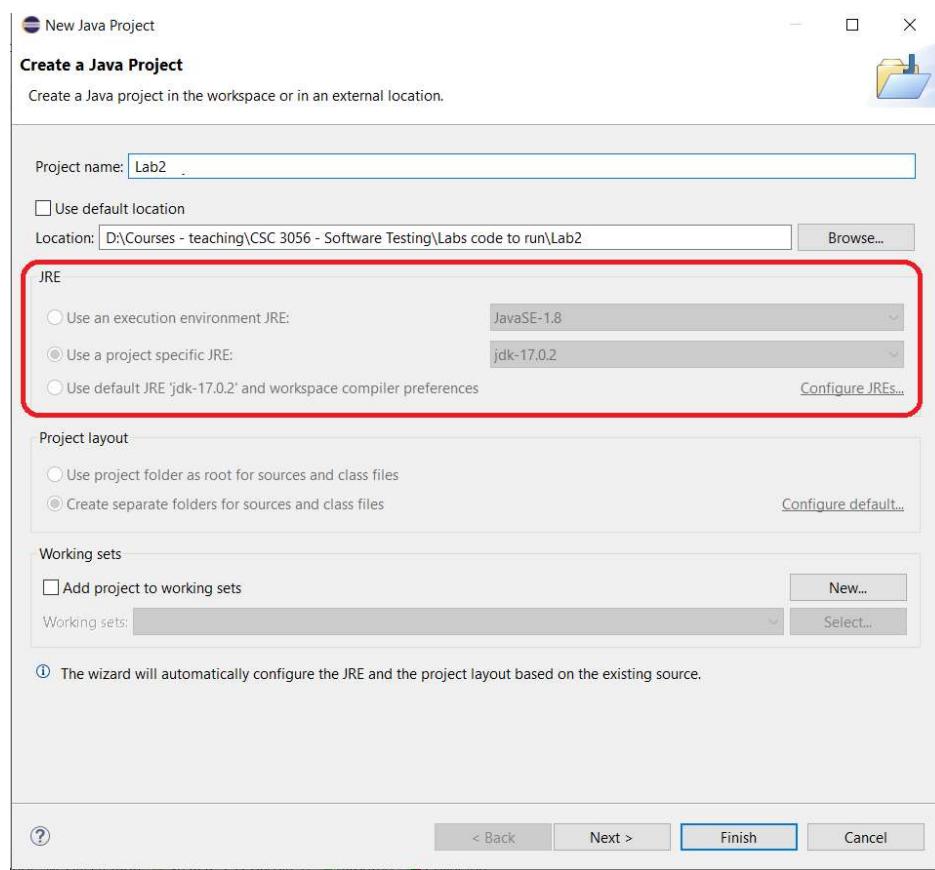


Figure 3 – Checking the latest versions of JDK and JRE in the New Project screen of Eclipse.

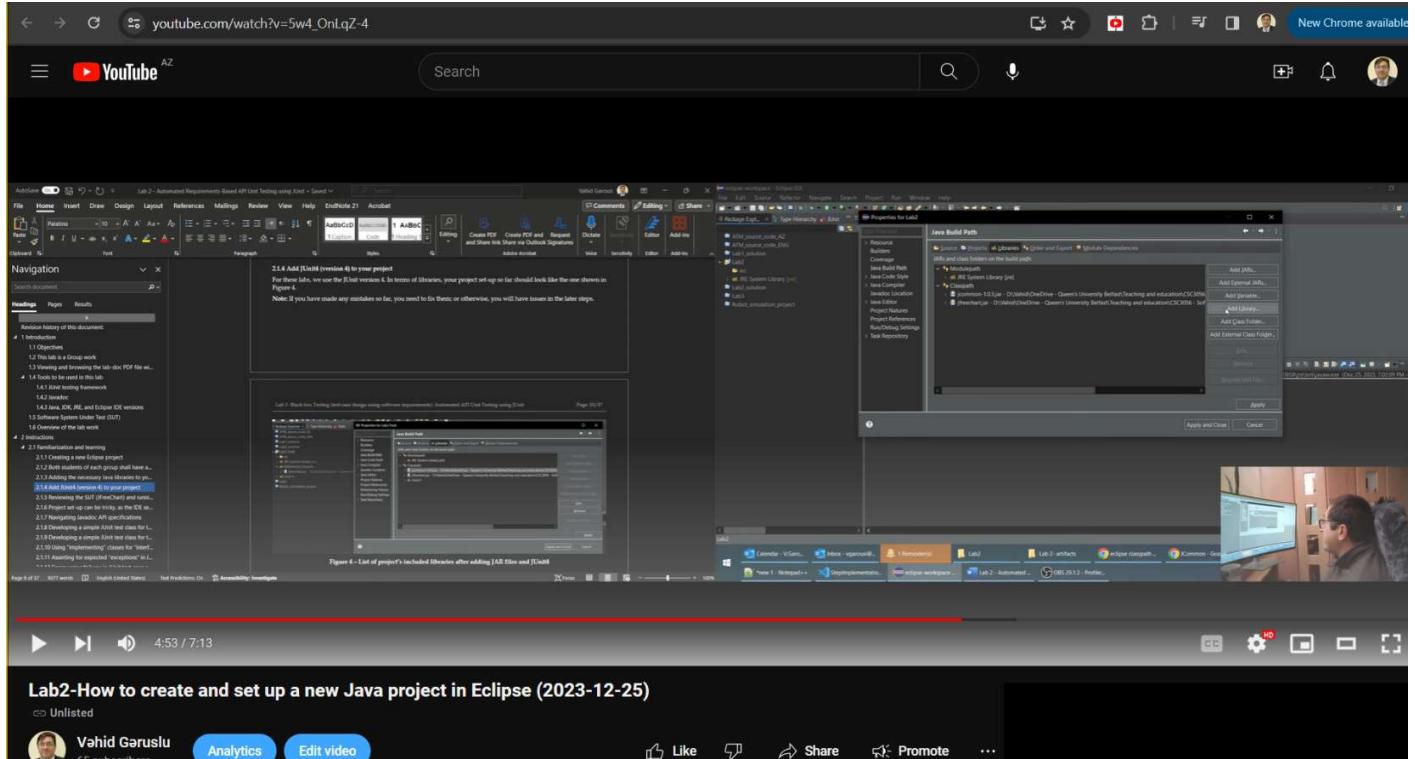
2.1.2 Both students of each group shall have a working set up in their own machines (laptops, PC etc.)

Although this is a group work, both students of each group shall have a working set up in their own machines (laptops, PC etc.), since all students shall learn the course concepts individually and also as a group.

2.1.3 Adding the necessary Java libraries to your project

7. The *Java Settings* dialog should now be displayed. This dialog has four tabs along the top: *Source*, *Projects*, *Libraries*, and *Order and Export*.
8. Move to the *Libraries* tab, and click the *Add External JARs (or Libraries)...* button. Select the *jfreechart.jar* file from the known location and click *Open*. JAR stands for Java ARchive (more info: [en.wikipedia.org/wiki/JAR_\(file_format\)](https://en.wikipedia.org/wiki/JAR_(file_format))).
9. You also need to add the JAR file *jcommon-1.0.5.jar* (provided in the artifacts file) to the libraries list. Information about JCommon library: www.jfree.org/jcommon/
10. Click *Finish*.

A short online video by Dr. Garousi explains a summary of project set-up: youtube.com/watch?v=5w4_OnLqZ-4



2.1.4 Add JUnit4 (version 4) to your project

For these labs, we use the JUnit version 4. In terms of libraries, your project set-up so far should look like the one shown in Figure 4.

Note: If you have made any mistakes so far, you need to fix them; or otherwise, you will have issues in the later steps.

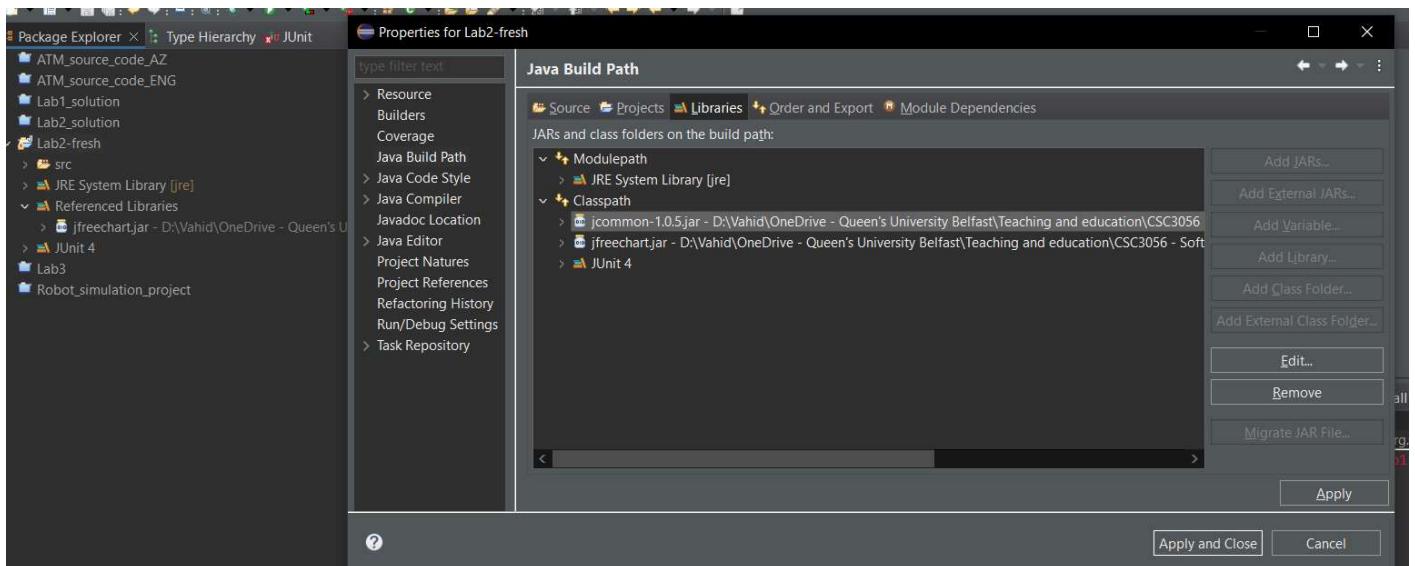


Figure 4 – List of project's included libraries after adding JAR files and JUnit4

2.1.5 Reviewing the SUT (JFreeChart) and running its demos

11. Let's take a few minutes to review the SUT (JFreeChart framework). Of course, in this lab, we do not have its source code and only have its executable file (JAR: Java ARchive file format).
12. We can see the contents of the JFreeChart library by going to *Reference Libraries* node in your newly-created project, see the large screenshot on the right side →
13. As you can see in the tree structure of JFreeChart JAR file, the framework is grouped into two main packages, (1) `org.jfree.chart` and (2) `org.jfree.data`. Each of these two packages is also divided into several other smaller packages. For the purpose of testing in this lab, we will be focusing on the `org.jfree.data` package.
14. The JFreeChart JAR file comes with several demo classes, each showing a type of chart which can be generated by JFreeChart. In the package explorer, expand the *Referenced Libraries* item in the newly-created JFreeChart project, exposing the .jar files just added. Right click on the `jfreechart.jar`, and select *Run As -> Java Application* (Figure 5).

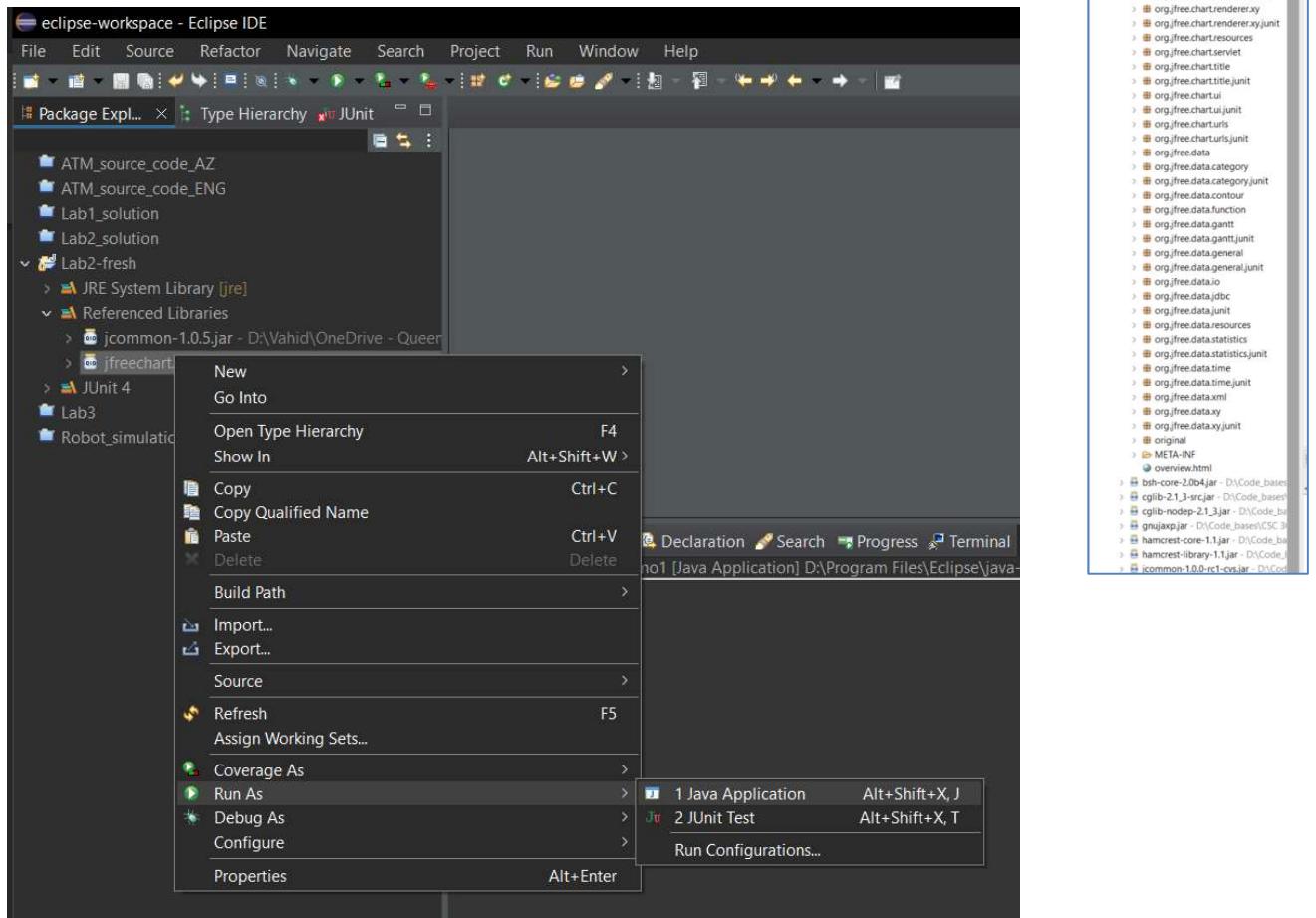


Figure 5 - Running demo classes of JFreeChart

15. In the *Select Java Application* dialog, select any of the demo applications (e.g., `CompositeDemo`), and click *OK* as shown in Figure 6. You should see some demo charts as shown in Figure 6.

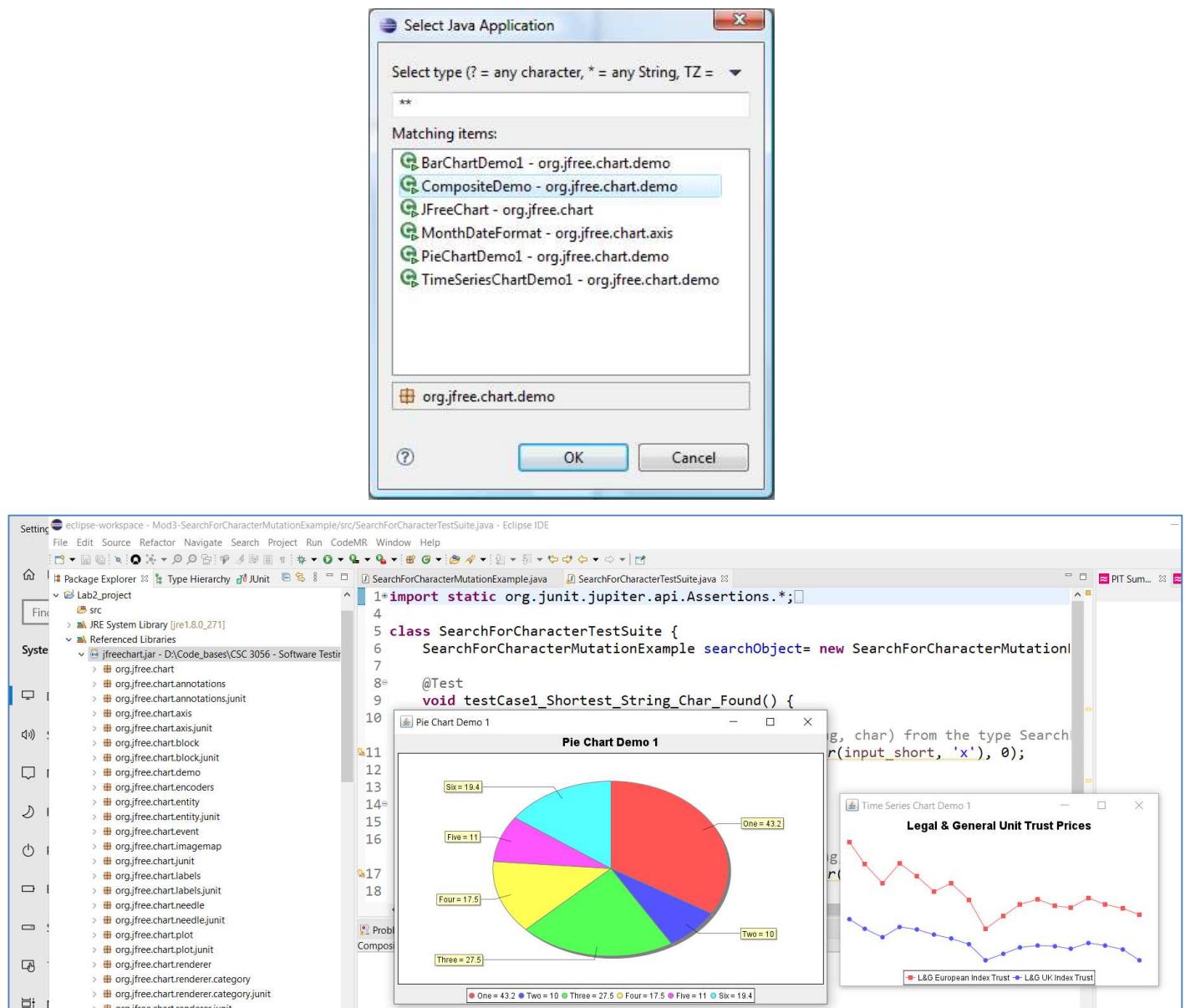


Figure 6 - Running demo classes of JFreeChart

Note: Feel free to ask the instructor or the Teaching Assistant (TA) [if we have TA(s) in a given term], if you have any issues so far.

2.1.6 Project set-up can be tricky, as the IDE settings are quite sensitive and error prone

We hope that by now, you have been able to run the demos of the JFreeChart. In some cases, we have seen that some students get various types of errors in their project set-up in Eclipse, due to minor mistake(s) in following the steps, or due to their OS / PC not having the project JDK/JRE, etc. For example, one of those many possible types of errors in their project set-up in Eclipse is the `java.lang.NoClassDefFoundError` exception (see below).

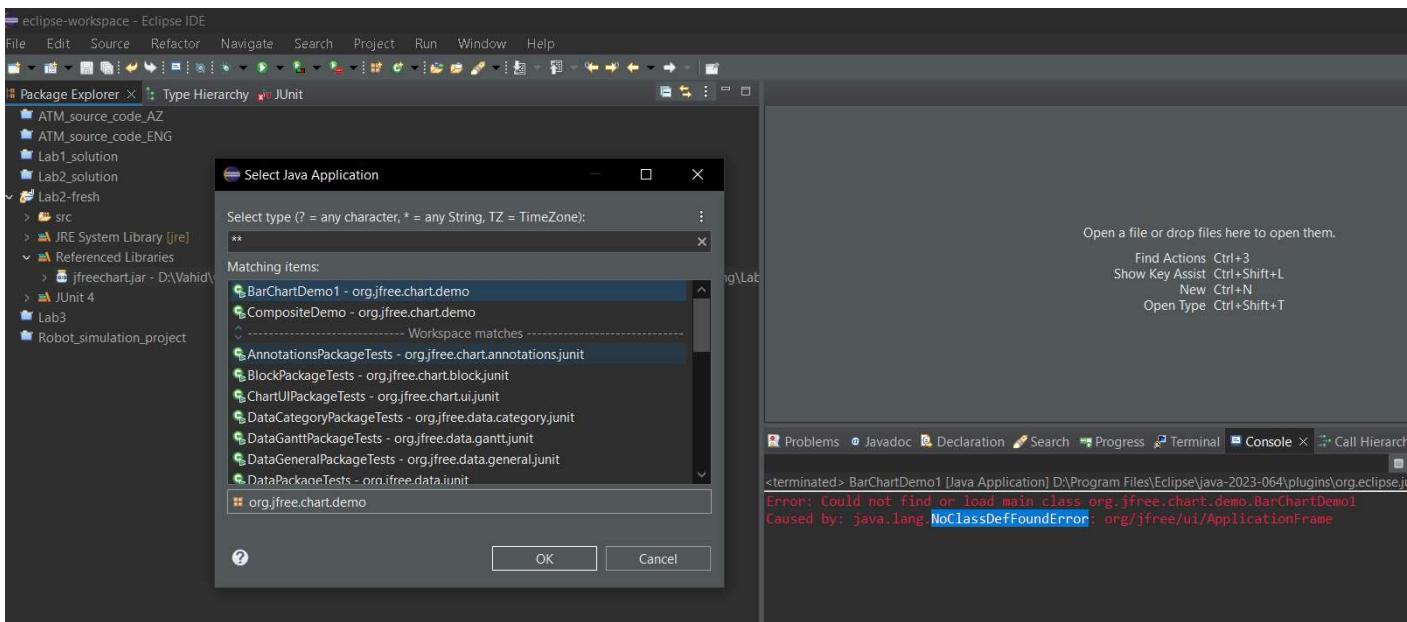


Figure 7 – It is possible to get various types of errors in project set-up

As software engineers in training, the students should realize that the space of IDEs, compiler (Java in our cases), OS are constantly evolving. Thus, settings and project parameters that were working a week ago, may not work the same in a given day. One possibility of getting issue is the proper project set-up w.r.t. libraries and JRE files. They should look like the following.

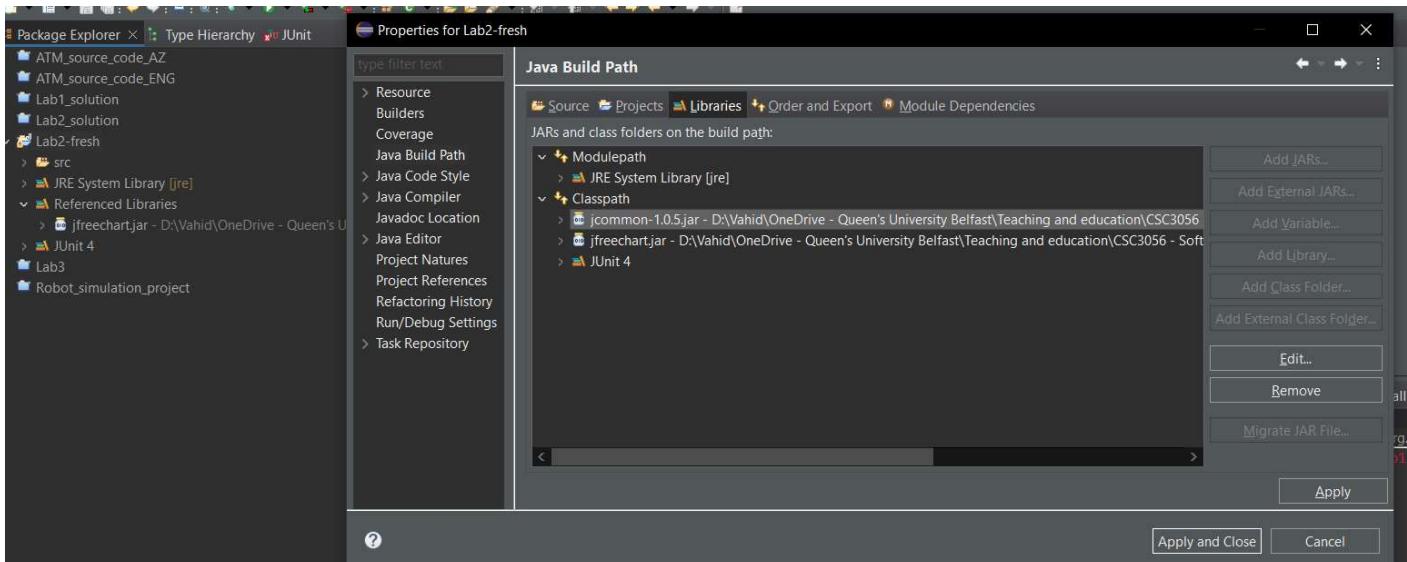


Figure 8 – Proper project set-up w.r.t. libraries and JRE files

Note that you should follow all the steps (provided in past pages of this document) properly to do the setup.

2.1.7 Navigating Javadoc API specifications

In Section 2.1.9 of this lab document, the document will ask you to design and develop unit tests for a number of classes based on specifications (requirements) contained in the Javadocs for those classes. As you have learned in the lectures, this is called Black-box testing.

16. Unzip the *JFreeChart-ModifiedJavadoc.zip* file and open the file *index.html*. Note the location of different elements in the Javadoc as shown in Appendix B. Note that Javadocs can be browsed with all classes shown, or with classes filtered by package. Each of these two approaches has its usefulness. Viewing all classes is useful if you know what

the class you are looking for is called, as they are ordered alphabetically. Viewing classes in a single package only is useful for when you're not sure exactly what class you're looking for, but know what area of the code it might be found in.

17. In the list of packages (top-left corner), scroll down to find the `org.jfree.data` package and click on it. This should show a list of classes in the class list (bottom left corner) now.
18. In the class list, click on the `Range` class. The main content pane now shows the API specifications of the `Range`. Scroll down and notice the layout of the specification. At the very top is a description of the class itself (including inheritance information), followed by nested classes, attributes, methods (starting with any constructors), inherited methods, and finally the detailed specification for each method; as shown below.

Figure 9 – Reviewing classes and their properties in Javadoc API specifications

19. Take note of the information available in the *Method Summary* and *Method Detail* sections of the main content pane, as this is what you will be testing methods against (as test oracle) in the following section. For especially effective tests, however, the specifications need to be specific, precise, clear and complete.

2.1.8 Developing a simple JUnit test class for the `org.jfree.data.Range` class

As an example class under test (CUT), let us choose the `org.jfree.data.Range`. You can find its API specification, in the Javadoc as shown in Figure 9. For developing tests for any class, you need to review the API specification of the class: its methods and their inputs and outputs, as shown in Figure 9 for this class under test (CUT).

To create a test suite containing a single unit test in JUnit, follow these steps.

20. In the package explorer, expand the *Referenced Libraries* list item to show all the archives that the project uses.
21. Expand the `jfreechart.jar` archive to expose all the packages that are contained in that archive.
22. Expand the `org.jfree.data` package within the archive to show all the .class files contained in that package.

23. Finally, expand the `Range.class` item to expose the class contained in that file, along with all the class' methods and fields contained in that class; as shown in Figure 10.
24. Right click on the `Range` class (has a green 'C' icon, denoting it is a class), and select `New -> JUnit Test Case`. Type `RangeTest` in the `Name` field and type '`.test`' in the `Package` field at the end of the default package to create a new package for the test cases. Ensuring that test classes are in a separate package makes it easier to keep the two apart. Also, ensure that the superclass is as specified in Figure 10. Also, make sure that you have checked the boxes for the method stubs specified in Figure 10, e.g., `setUp()`

25. Click *Finish*.

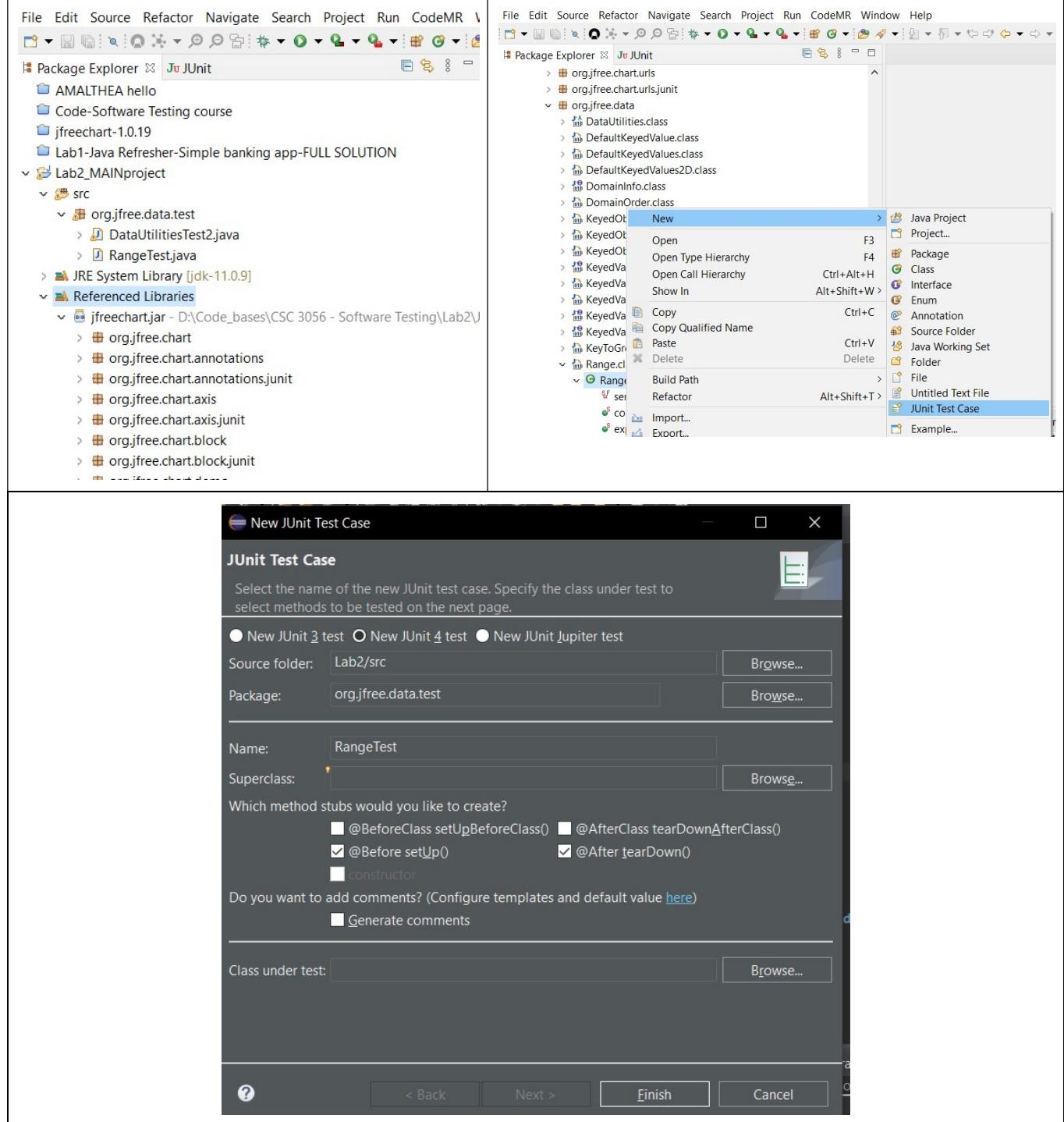
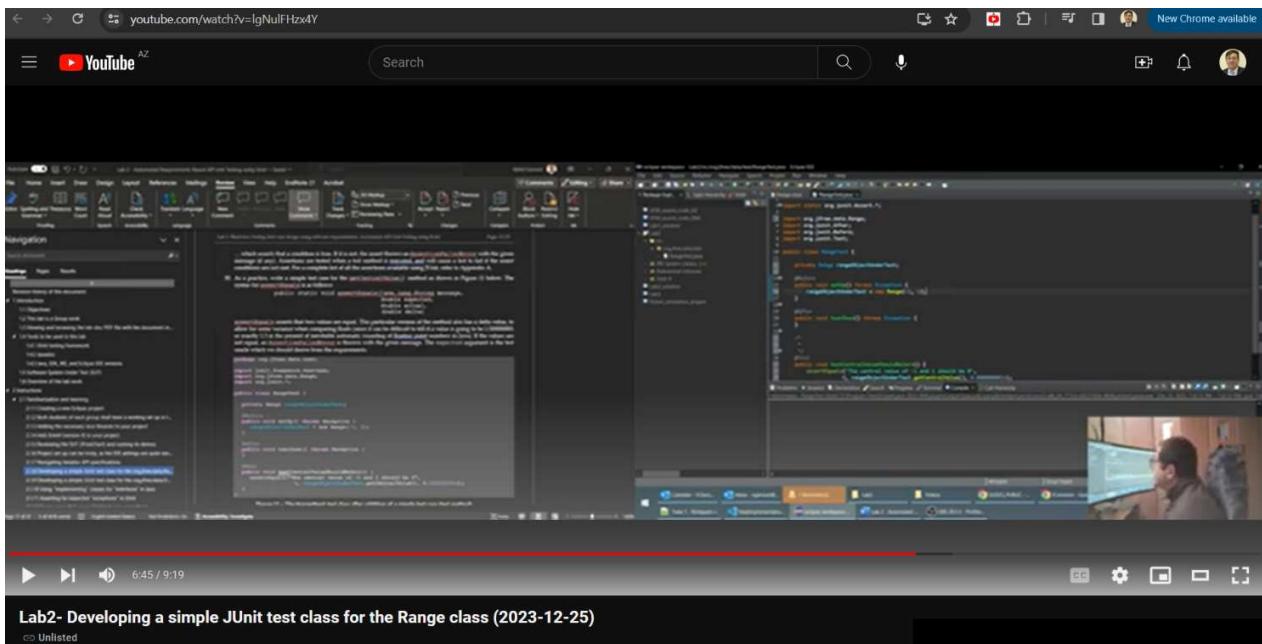


Figure 10 - Creating a new JUnit test case for the org.jfree.data.Range class

A short online video by Dr. Garousi explains a summary of developing a simple JUnit test class:
www.youtube.com/watch?v=lgNulFHzx4Y



26. The newly-created test class (RangeTest). A set of test cases which test similar functions of a class may all want to initialize an instance of that class and prepare for the test cases in the same way. The way that JUnit facilitates this is the `setUp()` and `tearDown()` methods. Write the `setUp()` and `tearDown()` methods for the RangeTest class as shown in Figure 11, including the `testRange` attribute which is required for the test cases to have access to the object initialized in the `setUp()` method.

```
package org.jfree.data.test;

import junit.framework.TestCase;
import org.jfree.data.Range;
import org.junit.*;

public class RangeTest {

    private Range rangeObjectUnderTest;

    @Before
    public void setUp() throws Exception {
        rangeObjectUnderTest = new Range(-1, 1);
    }

    @After
    public void tearDown() throws Exception {
    }
}
```

Figure 11 - Simple `setUp()` and `tearDown()`

27. It is possible that the IDE will tell you that the Range class cannot be resolved. It is best to use the IDE's QuickFix feature to fix minor issues like this. Choose to import the class from `org.jfree.data.Range` in this case, as shown in Figure 12.



Figure 12-Resolving package/module imports using the IDE's QuickFix feature for the Range class

28. Test cases in JUnit are individual methods which are usually prefixed with the word “test”, for example: `testCentralValue()`. The `@Test` annotations are also the identifiers. These test cases can perform any number of steps, and should follow four phase testing (setup, exercise, verify, and teardown). The test oracle for each test case in JUnit can be implemented in several ways, using assertions being the recommended one. An example assertion is:

```
assertTrue(("example string").length() == 14);
```

... which will always pass (assuming that the `length()` method works as expected). The syntax of the above assertion as defined in JUnit is:

```
public static void assertEquals([java.lang.String message,]
                           boolean condition)
```

... which asserts that a condition is true. If it is not, the assert throws an `AssertionFailedError` with the given message (if any). Assertions are tested when a test method is executed, and will cause a test to fail if the assert conditions are not met. For a complete list of all the assertions available using JUnit, refer to Appendix A.

29. As a practice, write a simple test case for the `getCentralValue()` method as shown in Figure 13 below. The syntax for `assertEquals` is as follows:

```
public static void assertEquals(java.lang.String message,
                               double expected,
                               double actual,
                               double delta)
```

`assertEquals` asserts that two values are equal. This particular version of the method also has a delta value, to allow for some variance when comparing floats (since it can be difficult to tell if a value is going to be 1.5000000001 or exactly 1.5 in the present of inevitable automatic rounding of floating point numbers in Java). If the values are not equal, an `AssertionFailedError` is thrown with the given message. The `expected` argument is the test oracle which we should derive from the requirements.

```
package org.jfree.data.test;

import junit.framework.TestCase;
import org.jfree.data.Range;
import org.junit.*;

public class RangeTest {

    private Range rangeObjectUnderTest;

    @Before
    public void setUp() throws Exception {
        rangeObjectUnderTest = new Range(-1, 1);
    }
}
```

```

    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void testCentralValueShouldBeZero() {
        assertEquals("The central value of -1 and 1 should be 0",
                    0, rangeObjectUnderTest.getCentralValue(), 0.000000001d);
    }
}

```

Figure 13 – The RangeTest test class after addition of a simple test case (test method)

30. Now that you have a completed test case, run the test class. To do this, right click on the RangeTest class in the Package Explorer and select *Run As -> JUnit Test*.
31. This will change the perspective to the JUnit perspective, and run all the tests within the RangeTest class. The test just written should pass, indicated by the JUnit view as shown in Figure 4 below. In JUnit a *Failure* and *Error* are similar, but differ slightly. If you get an error it means that your test method did not execute as expected (i.e., an uncaught exception), a failure, however, means that the execution was as expected, but an assertion failed.

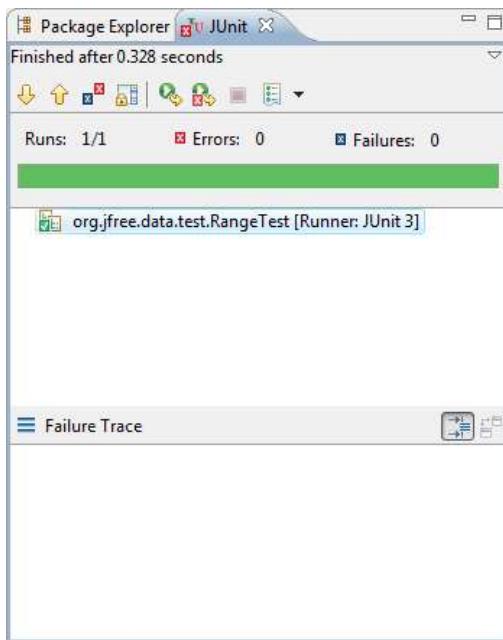


Figure 14 - JUnit view in Eclipse showing a passed test method

2.1.9 Developing a simple JUnit test class for the `org.jfree.data.DataUtilities` class

Let us show you how to develop a JUnit test class for another class under test (CUT) in the JFreeChart code-base. The class under test is `org.jfree.data.DataUtilities` and you can find its API specification, in the Javadoc as shown in Figure 15. For developing tests for any class, you need to review the API specification of the class: its methods and their inputs and outputs, as shown below.



Figure 15 - API specification of the `org.jfree.data.DataUtilities` class

The steps below are almost identical as what we did for the Range class in the previous section.

32. In the package explorer, expand the *Referenced Libraries* list item to show all the archives that the project uses.
33. Expand the *jfreechart.jar* archive to expose all the packages that are contained in that archive.
34. Expand the `org.jfree.data` package within the archive to show all the .class files contained in that package.
35. Finally, expand the `DataUtilities.class` item to expose the class contained in that file, along with all the class' methods and fields contained in that class; as shown in Figure 16.
36. Right click on the Range class (has a green 'C' icon, denoting it is a class), and select *New -> JUnit Test Case*. Type `RangeTest` in the *Name* field and type '`.test`' in the *Package* field at the end of the default package to create a new package for the test cases. Ensuring that test classes are in a separate package makes it easier to keep the two apart. Also, ensure that the superclass is as specified in Figure 10. Also, make sure that you have checked the boxes for the method stubs specified in Figure 10, e.g., `setUp()`
37. Click *Finish*.

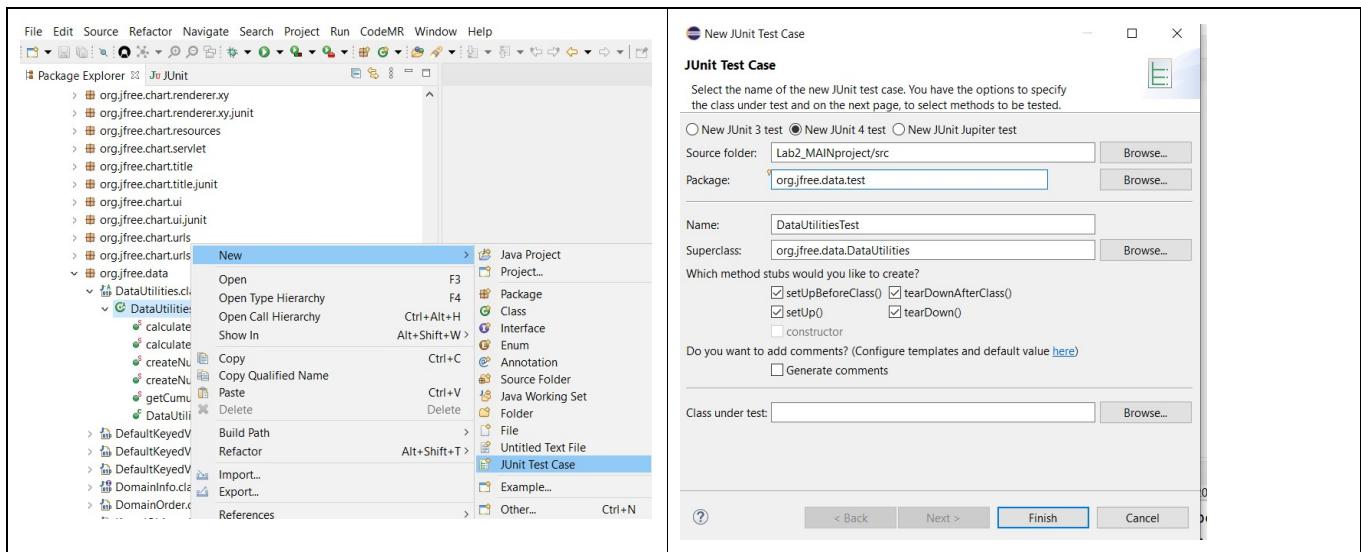


Figure 16 - Creating a new JUnit test case for the `org.jfree.data.DataUtilities` class

38. Write the test-code of DataUtilitiesTest as follows. We will explain next some new coding concepts used in this test-code snippet.

```
package org.jfree.data.test;

import org.jfree.data.DataUtilities;
import org.jfree.data.DefaultKeyedValues2D;
import org.jfree.data.Values2D;
import junit.framework.TestCase;

public class DataUtilitiesTest
{
    private Values2D values2D;

    @Before
    public void setUp()
    {
        DefaultKeyedValues2D testValues = new DefaultKeyedValues2D();
        values2D = testValues;
        testValues.addValue(1, 0, 0);
        testValues.addValue(4, 1, 0);
    }

    @After
    public void tearDown()
    {
        values2D = null;
    }

    @Test
    public void testValidDataAndColumnColumnTotal()
    {
        assertEquals("Wrong sum returned. It should be 5.0",
                    5.0, DataUtilities.calculateColumnTotal(values2D, 0), 0.0000001d);
    }
}
```

39. By the way, it will be quite likely that you will need to resolve package/module imports using the IDE's QuickFix feature, as shown in Figure 17.

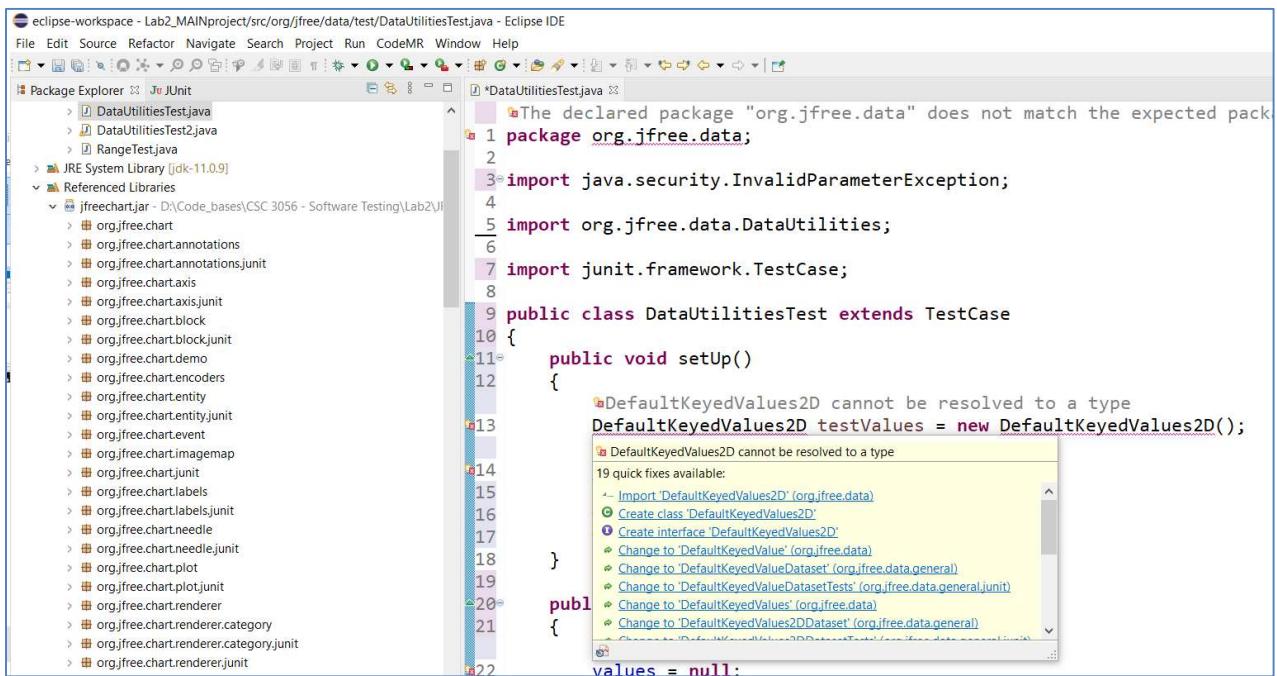


Figure 17- Resolving package/module imports using the IDE's QuickFix feature for the DataUtilities class

2.1.10 Using “implementing” classes for “interfaces” in Java

40. When developing JUnit test scripts for certain classes under test, we need to deal with the so-called “interfaces” in Java. First, read and learn the concept of Java “interfaces” in this URL: [en.wikipedia.org/wiki/Interface_\(Java\)](https://en.wikipedia.org/wiki/Interface_(Java))
41. Some methods in the class under test (CUT) DataUtilities use the interfaces Values2D and KeyedValues. In such cases, we need to find out and use the proper “implementing” classes of the interfaces, and that information is often available in Javadoc specification of the API (see Figure 18). You can use the following “implementing” classes for them: DefaultKeyedValues2D and DefaultKeyedValues, as shown in the Javadoc, shown in Figure 18.

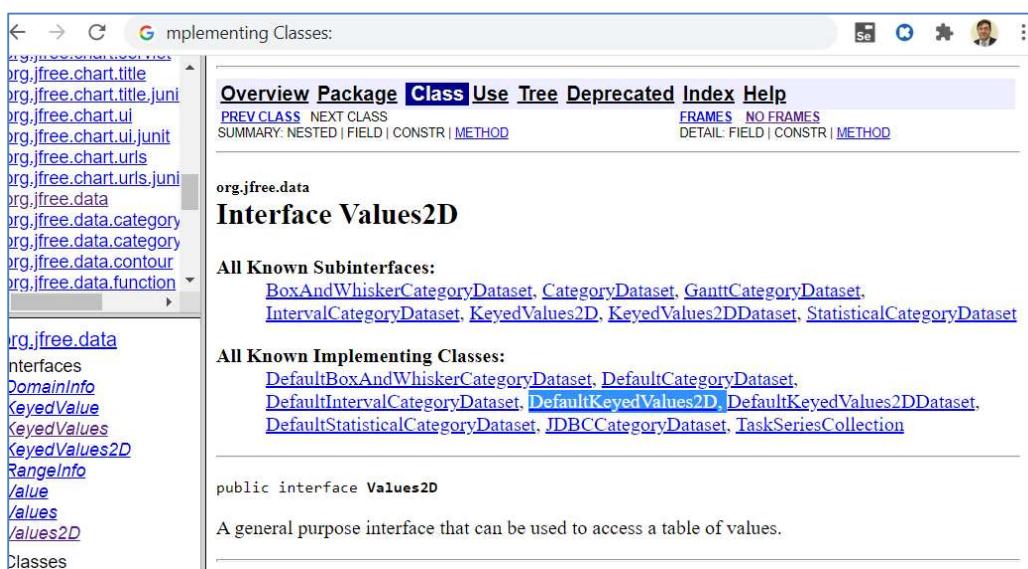


Figure 18 - “Implementing” classes of interfaces in Javadoc API specifications

42. Now look at the test-code of DataUtilitiesTest that we have provided for you above. In there, look at how the interface Values2D and class DefaultKeyedValues2D have been used. In fact, if you move the mouse over those two data types, you will see that Eclipse will tell you that they are an interface and a class, respectively.

```

8 public class DataUtilitiesTest extends TestCase
9 {
10     private Values2D values2D;
11
12     public void setUp()
13     {
14         DefaultKeyedValues2D testValues = new DefaultKeyedValues2D();
15         values2D = testValues;
16         testValues.addValue(1, 0, 0);
17         testValues.addValue(4, 1, 0);
18     }

```

Note: `value2D.addValue(1, 0, 0) // Will NOT work`

Figure 19 – Working with interfaces in Java

43. Since `Values2D` is an “interface” type in Java, its behavior “must be implemented by other classes”, as per definition of interfaces: [en.wikipedia.org/wiki/Interface_\(Java\)](https://en.wikipedia.org/wiki/Interface_(Java)). `DefaultKeyedValues2D` is one such class as per the API specification. Therefore, we declare an object of that class type and make that equal to the object of the interface type as you can see in the test-code: `values2D = testValues`. We then manipulate the object of class `DefaultKeyedValues2D` (named `testValues` in our code) afterwards, e.g., call its methods. The “scope” of the `testValues` object is only in the setup method level (“scope” means the boundary in which a variable is defined, known and can be used). However, the “scope” of the object of the interface `Values2D` (named `values2D` in our code) is in the class level, as you can see in the test-code. In other test methods, we thus need to use the `values2D` object, e.g., as we are using it in the `testValidDataAndColumnColumnTotal` test method.

44. As another example, the code example below shows how to use the `DefaultKeyedValues` class which “implements” the interface `KeyedValues`.

```

@Test // getCumulativePercentages function
public void testgetCumulativePercentages() {
    // setup

    DefaultKeyedValues keyvalues = new DefaultKeyedValues();
    keyvalues.addValue((Comparable) 0.0, 6.0);
    keyvalues.addValue((Comparable) 1.0, 11.0);
    keyvalues.addValue((Comparable) 2.0, 3.0);
    KeyedValues object_under_test = DataUtilities.getCumulativePercentages((KeyedValues) keyvalues);

    assertEquals((double) object_under_test.getValue(2), 1.0, .000000001d);
}

```

Figure 20 – How to use and work with the `DefaultKeyedValues` class which “implements” the interface `KeyedValues`

2.1.11 Asserting for expected “exceptions” in JUnit

An “exception” is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. Depending on what previous courses you have taken, you may have or have not learned about exceptions. See the links below to learn about exceptions:

- A short definition: <https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html>
- A comprehensive page: https://en.wikipedia.org/wiki/Exception_handling

Exceptions are classified using a hierarchy in every programming language. In Java, a partial snapshot from the hierarchy of exceptions is as shown in Figure 21. Note: This is a UML class-diagram¹, also called a *meta-model*². You do not have to

¹ https://en.wikipedia.org/wiki/Class_diagram

² <https://en.wikipedia.org/wiki/Metamodeling>

memorize or learn of all of this diagram in our course, but it is good to see it in case you will later hear about exceptions in your future software engineering career.

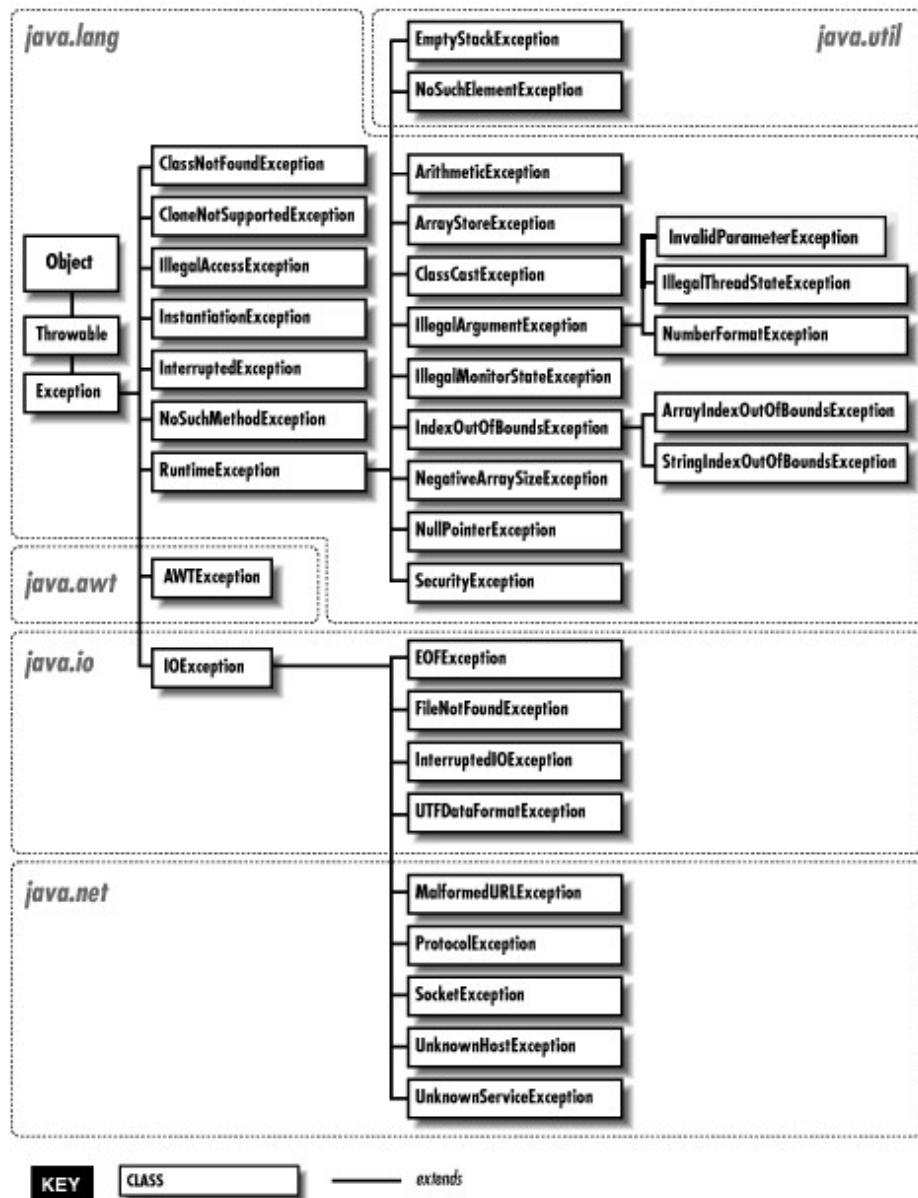


Figure 21 – Hierarchy and types of exceptions in Java

45. When we design test cases “on paper” and then write the corresponding JUnit test cases, in some instances the expected outcomes are exceptions. For example, let’s look at the API Javadoc of method *calculateColumnTotal* inside class DataUtilities, shown in Figure 22.

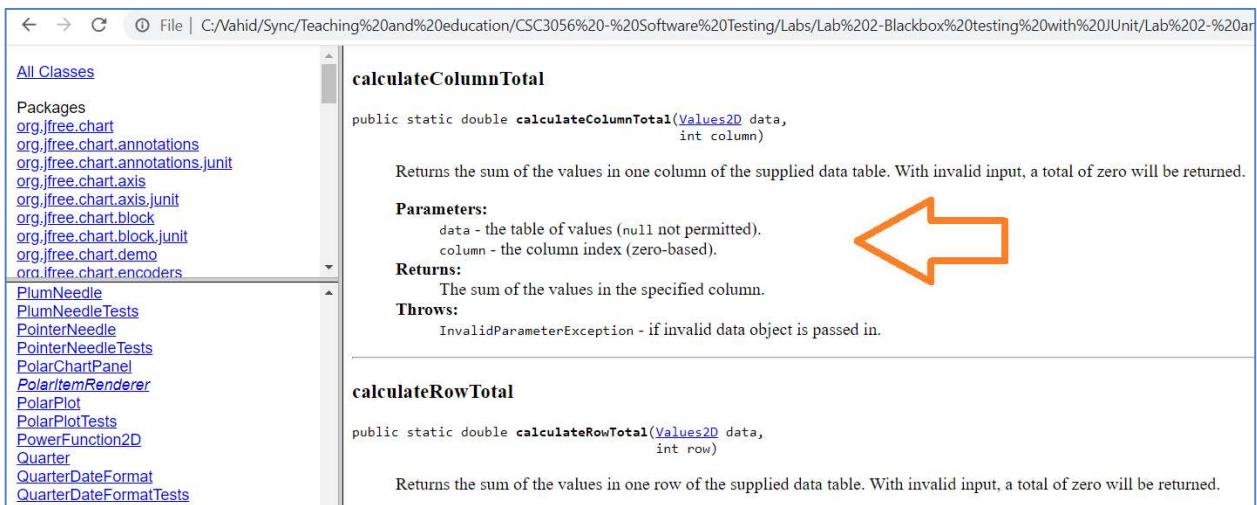


Figure 22 – API Javadoc of the method `calculateColumnTotal` inside class `DataUtilities`

46. If we pass in an invalid data object to the `calculateColumnTotal` method, it “should” return (“throw”¹) an `IllegalArgumentException`. Therefore, when we develop our JUnit test method, and let’s say, pass a NULL object to the first parameter (data) of this method, our test case shall expect to see an exception, and that is considered a “pass” for the test case. If the `calculateColumnTotal` method does not return an exception, given a null data object, the test cases will be considered a fail. With all these explanations, we now provide the test case method for the above case, below. You need to carefully review the test code below to ensure you fully understand it. If you have issues understanding each line of this test code, talk to your friends and/or the instructor.

```
public void testNullDataColumnTotal()
{
    try
    {
        DataUtilities.calculateColumnTotal(null, 0);
        fail("No exception thrown. The expected outcome was: a thrown
             exception of type: IllegalArgumentException");
    }
    catch (Exception e)
    {
        assertTrue("Incorrect exception type thrown",
                   e.getClass().equals(IllegalArgumentException.class));
    }
}
```

Figure 23 – Another test case for class `DataUtilities` catching exceptions

47. You may not have heard about try/catch blocks either. In that case, learn about it by doing a Google search.
 48. Add the above test-code into to your test-code of `DataUtilitiesTest` that we helped you develop in the previous section. Note: If the compiler would show an error message on the `IllegalArgumentException` class (saying that it is unknown), use the Eclipse’ QuickFix feature to fix the import statement needed for that (we have discussed it in the previous sections of this lab document).
 49. Run the test class `DataUtilitiesTest` and observe its outcome. You shall see the test outcome seen in Figure 24. `testValidDataAndColumnColumnTotal` shall pass. `testNullDataColumnTotal` shall fail. The latter test case mean that the SUT does not return what we had expected from it, as per its API Javadoc.

Note:

¹ <https://rollbar.com/guides/java/how-to-throw-exceptions-in-java/>

- The instructor has intentionally injected an undisclosed number defects in the SUT. Thus, a number of your test cases in this lab-work shall fail and of course, you should leave them as they are, and report the failures in your lab report.
- But for “failed” test cases, you need to double check that all your test-case design is done correctly, and the expected outputs (encoded via assertions) are put in place properly.
- But you need to ensure that you should not have any “Errors” in the JUnit output (like the following) in your final submitted work. Also, ensure that your test cases are designed properly (on “paper”, before coding them) and then coded properly in Java/JUnit.

The screenshot shows the Eclipse IDE interface with the following details:

- Title Bar:** eclipse-workspace - Lab2_MAINproject/src/org/jfree/data/test/DataUtilitiesTest.java - Eclipse IDE
- Toolbar:** File Edit Source Refactor Navigate Search Project Run CodeMR Window Help
- Package Explorer:** Shows the project structure and the test class `DataUtilitiesTest`.
- JUnit View:** Displays the test results: Runs: 2/2, Errors: 0, Failures: 1. It lists two tests: `testValidDataAndColumnColumnTotal` (0.000 s) and `testNullDataColumnTotal` (0.000 s).
- Code Editor:** The code for `DataUtilitiesTest.java` is shown. It includes imports for `DefaultKeyedValues2D`, `Values2D`, and `TestCase`. The class `DataUtilitiesTest` extends `TestCase`. It contains two methods: `setUp()` and `tearDown()`. The `testValidDataAndColumnColumnTotal` method uses `assertEquals` to check if the sum is 5.0. The `testNullDataColumnTotal` method uses a try-catch block to assert that an `IllegalArgumentException` is thrown when passing null to the `calculateColumnTotal` method.
- Failure Trace:** Shows the stack trace for the failure in the `testNullDataColumnTotal` method.

```

6 import org.jfree.data.DefaultKeyedValues2D;
7 import org.jfree.data.Values2D;
8 import junit.framework.TestCase;
9
10 public class DataUtilitiesTest extends TestCase
11 {
12     private Values2D values2D;
13
14     public void setUp()
15     {
16         DefaultKeyedValues2D testValues = new DefaultKeyedValues2D();
17         values2D = testValues;
18         testValues.addValue(1, 0, 0);
19         testValues.addValue(4, 1, 0);
20     }
21
22     public void tearDown()
23     {
24         values2D = null;
25     }
26
27     public void testValidDataAndColumnColumnTotal()
28     {
29         assertEquals("Wrong sum returned. It should be 5.0",
30                     5.0, DataUtilities.calculateColumnTotal(values2D, 0), 0.0000001d);
31     }
32
33     public void testNullDataColumnTotal()
34     {
35         try
36         {
37             DataUtilities.calculateColumnTotal(null, 0);
38             fail("No exception thrown");
39         }
40         catch (Exception e)
41         {
42             assertTrue("Incorrect exception type thrown",
43                         e.getClass().equals(InvalidParameterException.class));
44         }
45     }
46 }

```

Figure 24 – Outcome of JUnit test execution of two test cases for class `DataUtilities`

More details and examples for asserting for expected exceptions in JUnit are provided in Appendix C.

2.1.12 Errors versus failures in JUnit test-case executions

50. When you execute a set of JUnit test-cases, you will see three possible outcomes in the JUnit execution view of Eclipse:

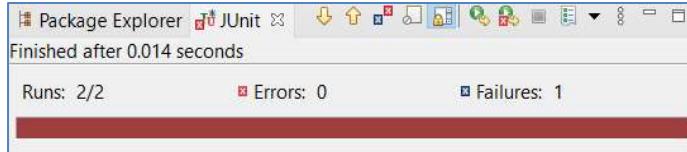


Figure 25 – Three possible outcomes for test cases in the JUnit execution view (Pass, Error and Failure)

51. Failures are counted and shown in the JUnit execution view when any of your test cases fail – i.e. when the condition of any is evaluated to False.
52. On the other hand, Errors in the JUnit execution view correspond to problems (often defects) in your test-code itself. They are often unexpected errors that occur while trying to actually run the test suites (the set of all test cases in your JUnit code-base), such as improper exception handling, etc. If your test method or the unit under test throws an exception which does not get properly handled through your test-code and the Assertion framework in JUnit, it gets reported as an Error in the JUnit execution view. For example, a *NullPointerException*, or a *ClassNotFoundException* exception will report an error, if they are not properly handled using the mechanism presented in the above example code snippet.
53. As a rule, when your final-version JUnit test-suite is executed, you should NOT see any “Errors” in the JUnit execution view. If you see any, you shall carefully inspect your test code, fix the issue rerun it and ensure number of errors=0, in the JUnit execution view. Also, if you are getting unwanted exceptions in your test code (from the SUT), you shall add the proper try/catch blocks, to convert the Error to Failure in test execution.
 - But of course, seeing Failures in JUnit execution view is fine, and they will denote existence of faults in the SUT
 - Errors mean problems / issues in test code
 - Failures mean problems / issues in the product code (SUT). But of course, we still need to inspect the test code to ensure that we are doing the test assertions, leading to Failures, properly
54. **Important note:** we have intentionally injected several defects into the SUT, thus a number of your test cases will and should Fail. Note, you should leave those failing test cases as they are.
55. **In summary, you need to ensure that your test cases are designed properly (on “paper”, before coding them) and then coded properly in Java/JUnit.**

2.1.13 Designing test cases for methods / functions with no parameters and developing test code for them

56. We have seen in the lectures and sections above many examples of how to design test cases for methods with one or more parameters and developing test code for them. But what about designing test cases for methods with no parameters? That is an interesting and important question!
57. Let's take an example from JFreeChart, the method `getLength()` inside class `Range`. Its API spec is as follows:

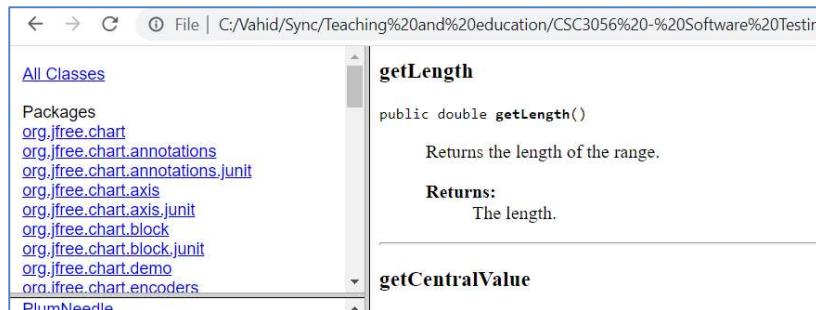


Figure 26 – Designing test cases for methods / functions with no parameters

58. To apply equivalence-class partitioning to this method, we need to be creative and smart, since it does not have any input. However, we should note that it is a method of a class, and look at its business logic: it should return the length of the range. Furthermore, it is easy to find out, from the class constructor, `Range(double lower, double upper)` in the API spec, that the range is determined by two numbers, which would be the member variables (attributes)

of the class: a lower double variable, and a upper double variable. These contexts can help us realize that we need to treat those member variables when testing the method `getLength()`. No, can you apply equivalence-class partitioning to this method? Think and if you cannot find out, read on...

59. From the definition of the equivalence-class partitioning technique, we need to partition the input space in which input data have the same effect on the method under test (e.g., the result in the same output). So we can find out that:

- If both *lower* and *upper* variable of the object have the same value, `getLength()` should return the value of zero (0)
- Else, `getLength()` should return the value of *upper-lower*
- To be on the safe side, we can also test when both are *lower* and *upper* positive values, when both are negative values and also when one is +ve and one is -ve. Thus, in a minimalist test design manner, that will give us at least five test cases as follows:

TC	Inputs		Expected output
	lower	upper	
1	2	2	0
2	4	9	5
3	-99	-99	0
4	-11	-4	7
5	-5	3	8

- The JUnit test code of the above test suite will be:

```

public void testGetLengthBothPositiveAndEqual() {
    Range r1 = new Range(2, 2);
    assertEquals("getLength: Did not return the expected output", 0.0, r1.getLength());
}

public void testGetLengthBothPositiveAndNotEqual() {
    Range r2 = new Range(4, 9);
}

public void testGetLengthBothNegativeAndEqual() {
    Range r3 = new Range(-99, -99);
    assertEquals("getLength: Did not return the expected output", 0.0, r3.getLength());
}

public void testGetLengthBothNegativeAndNotEqual() {
    Range r4 = new Range(-11, -4);
    assertEquals("getLength: Did not return the expected output", 7.0, r4.getLength());
}

public void testGetLengthOnePositiveOneNegative() {
    Range r5 = new Range(-5, 3);
    assertEquals("getLength: Did not return the expected output", 8.0, r5.getLength());
}

```

2.2 WORK TO BE DONE: DESIGN AND DEVELOPMENT OF UNIT TEST CASES

Now that you have finished the Familiarization section above, we now provide in this section the instructions for what should actually be done for your lab-work, and to be reported in your report.

This section is recommended to be performed as a group, however the work may be divided and completed individually, or you may wish to employ “peer programming”. There is a huge number of online resources about peer programming, and you can learn from them using Google search¹ or YouTube video search.

¹ www.google.com/search?q=peer+programming

2.2.1 Test requirements (classes to be tested)

60. In this section, you will be required to design and develop unit tests for the following classes, using their specifications (as mentioned in Javadocs). The two classes to be tested are the followings, which we have seen them to some extent in the Familiarization section above:

- `org.jfree.data.Range` (in the package `org.jfree.data`): Has 15 methods in total
- `org.jfree.data.DataUtilities` (in the package `org.jfree.data`): Has 5 methods in total

Take a few minutes to browse and review the API specifications of each of these classes in Javadoc, and their list of methods.

The lab workload expectation: To keep your workload manageable, we would like you to design test cases for all five (5) methods of `DataUtilities` class, and only five (5) out of 15 methods for `Range` (feel free to choose any random five of those methods).

Note: In the lectures, we have done problem-solving on black-box test-case design for a number of the methods of the SUT JFreeChart, including some methods of `Range` and `DataUtilities`. Note that those solved example methods CANNOT be included as part of your five and five methods in your lab-work. You can of course learn from them in the lectures, and then choose OTHER five and five methods from the classes under test, to develop test cases for.

In total, you need to have 10 Java class methods to develop Unit tests for. Thus, the effort level that you will spend to design and develop the test cases will be reasonable (not too high), but still you will near test automation in a realistic setting (similar to industrial settings).

2.2.2 The first step: Develop and document (write) your unit testing plan

61. As with any testing to be done, to begin with, a plan must first be developed and documented. Document this test plan, as it should be followed carefully in your testing work and also be included in your lab report. For this test plan, you need to include the following components:

- List of the methods under test
- The black-box test technique(s) that you plan to use to design test cases, and why you have decided to use those test technique(s)
- How you will organize your JUnit test suites (based on Appendix C)
- A very short (conceptual) example of a black-box unit-test plan has been provided to assist you, in Appendix E
- Note: We do not require from you a very extensive test plan for this lab, but when you will work in industry as a test engineer, a typical test plan could include the following sections. You are welcome to Google "test plan examples" and learn from the many examples provided by test engineers online. One example from the internet is shown below:

Below find important constituents of a test plan-

- [1 Introduction](#)
- [1.1 Scope](#)
 - [1.1.1 In Scope](#)
 - [1.1.2 Out of Scope](#)
- [1.2 Quality Objective](#)
- [1.3 Roles and Responsibilities](#)
- [2 Test Methodology](#)
 - [2.1 Overview](#)
 - [2.2 Test Levels](#)
 - [2.3 Bug Triage](#)
 - [2.4 Suspension Criteria and Resumption Requirements](#)
 - [2.5 Test Completeness](#)
- [3 Test Deliverables](#)
- [4 Resource & Environment Needs](#)
 - [4.1 Testing Tools](#)
 - [4.2 Test Environment](#)

Figure 27 -Typical contents of a test plan document in software industry

2.2.3 Designing the test cases, BEFORE developing them in JUnit

62. After you documented your test plan, you should discuss and document in your report how you designed the test cases (recall the “test-case design” lectures from the class). Since you are only given the requirements of the SUT, you should apply black-box test-case design techniques: equivalence classes partitioning, and boundary value analysis (BVA).
63. When using the equivalence-classing method, you need to generate the multi-dimensional (“strong”) equivalence-classes and, clearly, those multi-dimensional test suites will automatically contain the one-dimensional (“weak”) equivalence-classes test suites. When applying these techniques, make sure to explicitly follow and document in your report all the steps of the equivalence-classing method, as discussed in the class, e.g.,
 - 1-Identify the input domain (of method/function parameters)
 - 2-Equivalence classing of method input(s): shall be presented "visually", like done in the lectures
 - 3-Combining equivalence classes of the different inputs, using the multi-dimensional approach, i.e., Strong Equivalence-Class Testing (SECT)
64. Reminder: You need to include the details of how you design test cases in your report, just like done in the lectures.
65. You should ensure that the requirements are adequately tested.
66. Carry out your test plan and design your test-cases on paper (your lab report) first.
 - **Important: Only after you have designed your test cases on paper, you should develop them in JUnit.**

2.2.4 Developing your test code based on your test-case design

67. The next step is to develop your test code in the JUnit framework based on the list of test cases you have designed on paper, in the previous section. Note: Each test method should include one test-case only. For example: `testPositiveValuesForMethodX()` and `testNegativeValuesForMethodX()`, instead of a single `testMethodX()`. This will help to keep test cases consistent, and make analysis of test case impact simpler later on.
68. Before writing your test methods, you should first read the test-code architecture and also naming conventions provided in Appendix C, and follow them in your test code development.

69. If, in your group, you have divided the test cases between team members, then upon completion of the tests, review each other's test methods, looking for any inconsistencies or defects in the tests themselves.
70. While you develop your test cases, in intervals, execute your partial test suite on *jfreechart.jar*. Note that we have intentionally injected various (random) defects in the SUT! And thus a number of your tests should fail. Therefore, to write your test methods, you need to follow the specifications, not the actual results returned by the methods (functions). Note that the injected defects in the lab3 version of the SUT are different than from the lab2 version of the SUT.
71. Note that we do not prescribe any upper limit on the number of your test cases for any of the classes under test. Depending on how you will use the black-box test-case design techniques (such as equivalence classes, and boundary value analysis), you would end up with different number of test cases for each of the classes under test. But of course, you should use the black-box test-case design techniques in a logical and proper manner, just like how we learned and practiced with them in the lectures.
 - Note: If your test suite has much more or much less than the number of tests that you would normally result from "proper" application of those black-box test-case design techniques, your report will lose some marks. Note: A "few" tests more or less will not lose marks!

3 SUMMARY OF THE LAB AND LEARNING OUTCOMES

Upon completion of this lab, students should have a reasonable understanding of unit testing based on requirements using the JUnit framework. Note that unit testing and JUnit are very comprehensive and it takes quite a lot of time to be an expert in them. So do not expect to be JUnit experts just by completing this lab. If you would like to have a career path in this industry-hot topic, you will need to study this popular framework in more detail and perform more exercises to be skillful.

The unit testing knowledge you gained in this lab can be scaled up to much larger systems, and can be very useful in industry.

4 DELIVERABLES

Only one submission per group, by either of the two students.

4.1 LAB REPORT

To be consistent, **student MUST** use the template Word file "Lab 2 Report Template.doc" provided online.

- Please keep the section names as provided in the lab template file.
- You should upload each lab's report in Word DOC / DOCX format in the course management system.
- The lab report Word file shall be uploaded in Canvas, NOT in GitHub

4.2 JUNIT TEST-SUITE TEST-CODE

- Your test-suite Java files should be uploaded to your team's GitHub repository.
- Your code in the GitHub repository should be updated (committed) during the lab work duration (two weeks) by both students, showing that the team has actively worked on the lab work.
- The folders of your team's GitHub repository should be properly organized for each lab. Reminder: you should only have one single repository for the entire team, and only have one single repository for the entire module, it should have folders for each lab in its root, named as Lab0, Lab1, ...
- You should not submit your JUnit test-suite test-code in Canvas
- Your GitHub repo should ONLY include the source-code files (Java, etc.) that the lab document has asked for. Do NOT upload unwanted files such as .CLASS files, JAR files, copy of project folders outside the code that you have developed, etc. Having extra unwanted files in GitHub could lead to mark deductions.
- Store the JUnit test suite you have developed in this lab, in a safe folder; since you will use and improve these test suites in Lab 3.

ACKNOWLEDGEMENTS

This lab is part of a software-testing laboratory courseware available under a Creative Commons license. The laboratory courseware has been used by 50+ testing educators world-wide. More details can be found in the courseware's website:

sites.google.com/view/software-testing-labs/

The courseware has been made a reality thanks to many people, including Michael Godwin, Christian Wiederseiner, Yuri Shewchuk, Riley Kotchorek and Negar Koochakzadeh for their helps in developing and testing these exercises.

APPENDIX A – TYPES OF ASSERTION METHODS AVAILABLE IN JUNIT

The following is a list of assertions available for use in JUnit test cases. Note that assertions with ‘message’ arguments are equivalent to those without the message argument, but will display that message when the test fails on that assertion. These can be useful for debugging purposes. For more information, see:

<https://junit.org/junit4/javadoc/latest/org/junit/Assert.html>

- `assertEquals(expected, actual)`
- `assertEquals(message, expected, actual)`
- `assertEquals(expected, actual, delta)` - used on doubles or floats, where delta is the allowable difference in precision
- `assertEquals(message, expected, actual, delta)` - used on doubles or floats, where delta is the allowable difference in precision
- `assertFalse(condition)`
- `assertFalse(message, condition)`
- `assertNotNull(object)`
- `assertNotNull(message, object)`
- `assertNotSame(expected, actual)`
- `assertNotSame(message, expected, actual)`
- `assertNull(object)`
- `assertNull(message, object)`
- `assertSame(expected, actual)`
- `assertSame(message, expected, actual)`
- `assertTrue(condition)`
- `assertTrue(message, condition)`
- `fail()` - Fails a test with no message.
- `fail(message)` - Fails a test with the given message.
- `failNotEquals(message, expected, actual)`
- `failNotSame(message, expected, actual)`
- `failSame(message)`

APPENDIX B – JAVADOC EXAMPLE

View All Classes → All Classes

View Classes in a Single Package → Packages → org.jfree.chart

Class List → Packages → org.jfree.chart

Packages

Link	Description
org.jfree.chart	Core classes, including JFreeChart and ChartPanel .
org.jfree.chart.annotations	A framework for adding annotations to charts.
org.jfree.chart.axis	Axis classes and interfaces.
org.jfree.chart.block	Blocks and layout classes used extensively by the LegendTitle class.
org.jfree.chart.demo	Some basic demos to get you started.
org.jfree.chart.encoders	Classes related to the encoding of charts to different image formats.
org.jfree.chart.entity	Classes representing components of (or entities in) a chart.
org.jfree.chart.event	Event classes and listener interfaces, used to provide a change notification mechanism so that charts are automatically redrawn whenever changes are made to any chart

Main Content Pane: Initially shows packages, shows class API when class is selected

APPENDIX C – CONVENTIONS FOR TEST-CODE ARCHITECTURE AND NAMING IN JUNIT

Just like we use certain naming conventions in our working code for deciding how to name classes, interfaces, and variables, we also use conventions when naming our tests and test classes.

C1: TEST-CODE ARCHITECTURE

Recall from your other courses that software and code architecture are important topics in software engineering. Test-code architecture is a topic which is important in our course context. Essentially, a recommended good practice to modularize unit test code with respect to the production code (source code under test) is visualized in Figure 28. Note that the numbers (indices) are hypothetical and of course you should use the exact method names instead of the numbers. It is important to follow this architecture in your test code development.

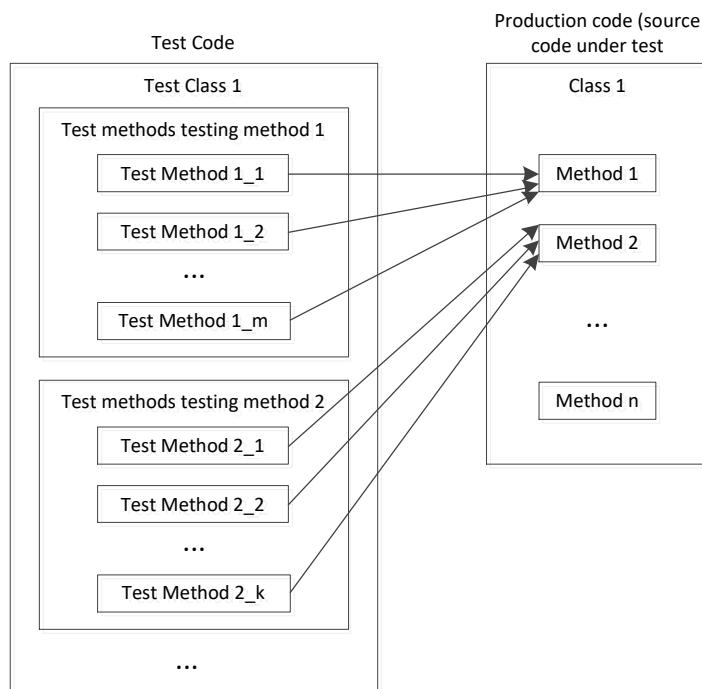


Figure 28 - A good practice to modularize unit test code architecture with respect to the the production code (source code under test)

C2: TEST CLASSES

Some rules are much the same as the Java conventions that you are used to, such as capitalization. Also, each test class should be in its own file and you should avoid defining other classes in the same file.

Test classes should be named by the class they are testing with the appended word ‘Test’ after the class name. For example, if you were to write unit test code for classes Range and DataUtilities, you would develop two Java class files each named: RangeTest.java and DataUtilitiesTest.java and you would develop the classes RangeTest and DataUtilitiesTest respectively in each of those two files.

C3: TEST METHODS AND THEIR NAMING CONVENTION

Each test method should include only one test case.

If you are familiar with JUnit 3.x, you may have become familiar with the naming convention of prefixing the name of each test with the word ‘test’. This was a necessity in JUnit 3.x in order for the compiler to understand that the method was indeed a test method. Starting in JUnit 4, this is no longer a necessity. Instead we must include the “annotation” @Test before such methods. JUnit 4 testers are divided when it comes to following the rule of prefixing name of each test with the word ‘test’. For example you may see some tests declared as:

```
@Test public void testMyMethod() ...
```

while others would name the method as follows:

```
@Test public void myMethod() ...
```

You may choose to follow either convention, but be consistent, please choose only one.

Each test method should include only one test case, and the purpose of the test should be immediately clear by reading the method name. Remember that we will not be calling these methods in our code so there is no foul against giving test methods what would otherwise be considered exceptionally long names. For example, the following are some examples of clearly named test methods:

```
void myMethodThrowsExceptionWhenInputIsNull() {}  
void myMethodReturnsNullWhenInputIsEmpty() {}  
void myMethodSucceedsWhenGivenExpectedInput() {}
```

and here are some poorly named examples:

```
void myMethod() {}  
void myMethodTest() {}  
void myMethodTest2() {}  
void myMethodSucceeds() {}  
void myMethodFails() {}  
void myMethodThrowsException() {}
```

Naming your methods in this way will help you to understand what the purpose was later on and will also help you understand what has happened when your test suite fails.

For example, for the given SUT, a properly-named test method will be like:

```
testIntersectsUpperEqualToLower();
```

C4: ASSERTING FOR EXPECTED EXCEPTIONS IN JUNIT

If the method you are testing can throw an exception, you should test that it does so under the right circumstances. There are several ways to handle this situation.

Alternative 1:

One way (alternative) to do this would be to surround the test code in a try/catch block and include a `fail()` statement as the last line of the code in the try section. An example is shown below:

```
1 package net.codejava;  
2  
3 import static org.junit.Assert.assertEquals;  
4 import static org.junit.Assert.fail;  
5  
6 import org.junit.Test;  
7  
8 public class UserTest {  
9     @Test  
10    public void testUserNameTooShort() {  
11        try {  
12            User user = new User();  
13            user.setName("Jo");  
14  
15            fail();  
16        } catch (IllegalArgumentException ex) {  
17            assertEquals("Username is too short", ex.getMessage());  
18        }  
19    }  
20}  
21}
```

As you can see, we use the `fail()` statement at the end of the catch block so if the code doesn't throw any exception, the test fails. And we catch the expected exception by the catch clause, in which we use `assertEquals()` methods to assert the exception message. You can use this structure to test any exceptions.

And there are many resources on this in the internet, see:

<https://www.google.com/search?q=junit+handle+exception>

Alternative 2:

The second alternative is to use a specific annotation of JUnit. JUnit 4 provides an annotation that will in essence do this for you. Simply include '(expected=Exception.class)' after the @Test annotation as follows:

```
@Test (expected=NullPointerException.class)  
void myMethodThrowsExceptionWhenInputIsNull() {}
```

In the above example, if a NullPointerException is thrown at any time during the test, the test will pass, otherwise it will fail.

APPENDIX D – UNDERSTANDING THE STEPS OF A TEST METHOD IN XUNIT

There are normally four steps to every test; the boundaries between each of these steps should be made immediately clear when reading test code.

D1: SETUP

Note: to learn more about how to implement setup and tear methods, see Appendix C.

Setup for a test usually happens in one or more of the following places:

- a setup method that is run once before any tests are executed (`@BeforeClass` annotation)
- setup method that is run every time before each test is executed (`@Before` annotation)
- setup code inside of your test method

Use the setup to construct the scenario for the test which usually includes constructing data objects but may also include running methods or inputting data. Using the suite-wide initialization (`@BeforeClass`) is not very common, it is best to initialize any data you need for each test in your other setup methods unless you can be absolutely sure that that data being used cannot be manipulated by the tests and therefore cause your subsequent tests to be unreliable. It is however best to abstract as much as possible from each test into your setup methods in order to make your tests more consistent and easier to read.

Although it is possible to write test cases with no setup at all, in most tests you will write, the bulk of your code for each will be found here.

D2: EXERCISE

Most, if not all, of the unit tests you will code for this lab will include an ‘exercise’ of only one line as you will be testing one function at a time. The exercise is simply the execution of the code of interest for the particular test.

D3: VERIFY

Much of the time, similar to the previous step, all of your verification for one test is executed in one statement. This is usually done through assert statements `assertEquals()`, `assertTrue()`, or `assertNull()` for example. Much of the time, it is also possible to include your exercise and verification in the same statement. However this is usually avoided in favour of clarity.

D4: TEAR-DOWN

Including tear-down code in Java tests is much less common than in other languages. Normally tear-down code is simply used for cleaning up after a test has executed so you will need to implement this step to close a file if one is opened during your setup for example.

More reading:

We encourage you to read and learn more about the above test phases in Google:

google.com/search?q=xunit+setup+pattern+teardown

APPENDIX E – A VERY SHORT EXAMPLE OF A BLACK-BOX UNIT-TEST PLAN

Methods to be tested:

- `org.jfree.data.Range` (in the package `org.jfree.data`): Choose and itemize five of the 15 methods of this class, as per the test requirements (see Section 2.2.1)
- `org.jfree.data.DataUtilities` (in the package `org.jfree.data`): List all its 5 methods, as per the test requirements (see Section 2.2.1)

Black-box testing techniques that we plan to use and the rationale of choosing it:

We plan to use a combination of Equivalence Class testing (using the strong equivalence method) and Boundary Value analysis (BVA). We believe these two techniques are the right choice for black-box testing of the above methods under test since ...[provide your reasoning]...

How we will use the above test technique(s):

Describe how you will apply the technique on a given method, using the process taught in the lectures.

...[expand]...

How we will organize our JUnit test suites:

We will have two test class files, one for Range and one for DataUtilities. There will be one test method devoted to each test case, therefore each class will have one JUnit test method per test case. A number of variables and assertions will be ...[expand]...

We will also add the annotation '@Test' so that ...[expand]...