|  |
| --- |
| Department of Software Engineering  Mehran University of Engineering and Technology, Jamshoro |

|  |  |  |  |
| --- | --- | --- | --- |
| Course: SW426 - Software Quality Engineering | | | |
| Instructor | Rabia Iftikhar | **Practical/Lab No.** | 05 |
| Date | 20-08-2020 | **CLOs** | CLO-3 |
| Signature |  | **Assessment Score** | 1 Mark |

|  |  |
| --- | --- |
| Topic | Unit testing using Junit |
| Objectives | * To learn Unit testing concepts * To explore Junit framework |

|  |
| --- |
| Lab Discussion: Theoretical concepts and Procedural steps |

**Unit Testing**

A class in an object-oriented programming language can be considered as a program unit. A program unit is a piece of code, such as a function or method of class, that is invoked from outside the unit and that can invoke other program units. A program unit is assumed to implement a well-defined function providing a certain level of abstraction to the implementation of high-level functions. A program unit may be viewed as a piece of code implementing a “low”-level function. There are two reasons for testing a unit in a stand-alone manner.

1. First, errors found during testing can be attributed to a specific unit so that it can be easily fixed. Moreover, unit testing removes dependencies on other program units.
2. Second, during unit testing it is desirable to verify that each distinct execution of a program unit produces the expected result.

A programmer needs to test a unit as follows:

1. Execute every line of code. This is desirable because the programmer needs to know what happens when a line of code is executed. In the absence of such basic observations, surprises at a later stage can be expensive.
2. Execute every predicate in the unit to evaluate them to true and false separately.
3. Observe that the unit performs its intended function and ensure that it contains no known errors.

Unit testing is conducted in two complementary phases:

1. **Static unit testing:** a programmer does not execute the unit; instead, the code is examined over all possible behaviors that might arise during run time. The code of each unit is validated against requirements of the unit by reviewing the code.
2. **Dynamic unit testing:** Dynamic unit testing is execution based. A program unit is executed, and its outcomes are observed.

**STATIC UNIT TESTING**

Units are individually tested by executing them, those are subject to usual review and correction as it is commonly understood. The idea behind review is to find the defects as close to their points of origin as possible so that those defects are eliminated with less effort, and the interim product contains fewer defects before the next task is undertaken. In static unit testing, code is reviewed by applying techniques commonly known as *inspection* and *walkthrough*.

1. **Inspection:** It is a step-by-step peer group review of a work product, with each step checked against predetermined criteria.
2. **Walkthrough:** It is a review where the author leads the team through a manual or simulated execution of the product using predefined scenarios.

**DYNAMIC UNIT TESTING**

Execution-based unit testing is referred to as dynamic unit testing. In this testing, a program unit is executed in isolation, as we commonly understand it. An environment for dynamic unit testing is created by emulating the context of the unit under test. The context of a unit test consists of two parts:

* a caller of the unit and
* all the units called by the unit.

The caller unit is known as a **test driver**, and all the emulations of the units called by the unit under test are called **stubs**. The test driver and the stubs are together called **scaffolding.**

* **Test Driver**

A test driver is a program that invokes the unit under test. The unit under test executes with input values received from the driver and, upon termination, returns a value to the driver. The driver compares the actual outcome, that is, the actual value returned by the unit under test, with the expected outcome from the unit and reports the ensuing test result. The test driver functions as the *main* unit in the execution process. The driver not only facilitates compilation, but also provides input data to the unit under test in the expected format.

* **Stubs**

A stub is a “dummy subprogram” that replaces a unit that is called by the unit under test. Stubs replace the units called by the unit under test. A stub performs two tasks. First, it shows an evidence that the stub was, in fact, called. Such evidence can be shown by merely printing a message. Second, the stub returns a pre-computed value to the caller so that the unit under test can continue its execution.



**Junit Framework**

The JUnit is a unit testing framework for the Java programming language designed by Kent Beck and Erich Gamma. Experience gained with JUnit has motivated the development of the TDD methodology. The idea in the JUnit framework has been ported to other languages, including C# (NUnit), Python (PyUnit), Fortran (fUnit) and C++ (CPPUnit). This family of unit testing frameworks is collectively referred to as xUnit.

**Scenario**

Suppose that we want to test the individual methods of a class called PlanetClass. Let Move()be a method in PlanetClass such that Move()accepts only one input parameter of type *integer* and returns a value of type integer. One can follow the following steps:

1. Create an object instance of PlanetClass.
2. Select a value for all the input parameters of Move()—this function has just one input parameter. Let us represent the input value to Move() by *x*.
3. Know the expected value to be returned by Move(). Let the expected returned value be *y*
4. Invoke method Move() on object Mars with input value *x*. Let *z* denote the value returned by Move().
5. Now compare *y* with *z*. If the two values are identical, then the method Move() in object Mars passes the test. Otherwise, the test is said to have failed.



**Conclusion of the Scenario**

It is useful to employ a general programming framework to code individual test cases, organize a set of test cases as a *test suite*, initialize a test environment, execute the test suite, clean up the test environment, and record the result of execution of individual test cases. In the example, creating the object Mars is a part of the initialization process. The two print() statements are examples of recording the result of test execution. Alternatively, one can write the result of test execution to a file.

**JUnit Framework**

The framework provides a basic class, called TestCase, to write test cases. Programmers need to *extend* the TestCase class to write a set of individual test cases. A class may have multiple functions each for defining a test case. Programmers need to make assertions about the state of objects while extending the TestCase class to write test cases

For example, in each test case it is required to compare the actual outcome of a computation with the expected outcome. Though an if() statement can be used to compare the equality of two values or two objects, it is seen to be more elegant to write an assert statement to achieve the same. The class TestCase extends a utility class called Assert in the Junit framework.

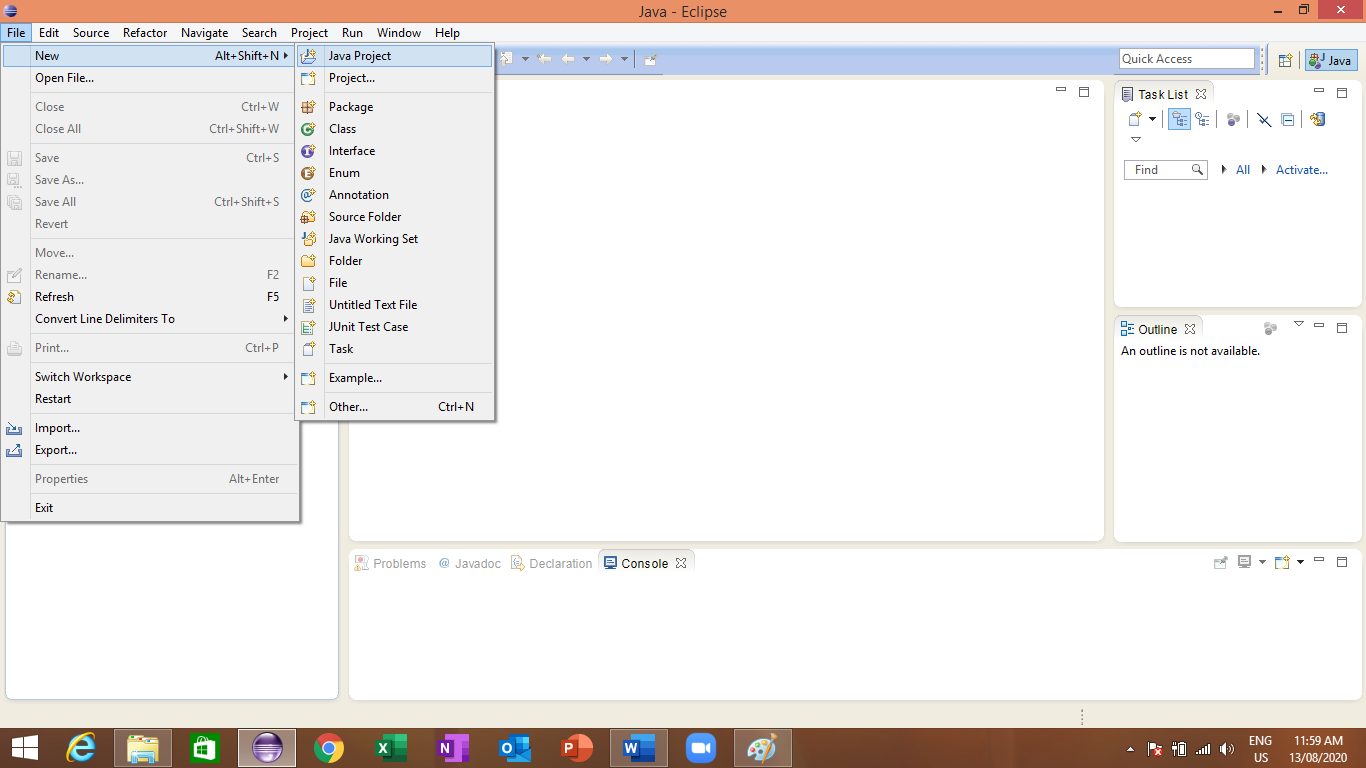
The Assert class provides methods, as explained in the following, to make assertions about the state of objects created and manipulated while testing.

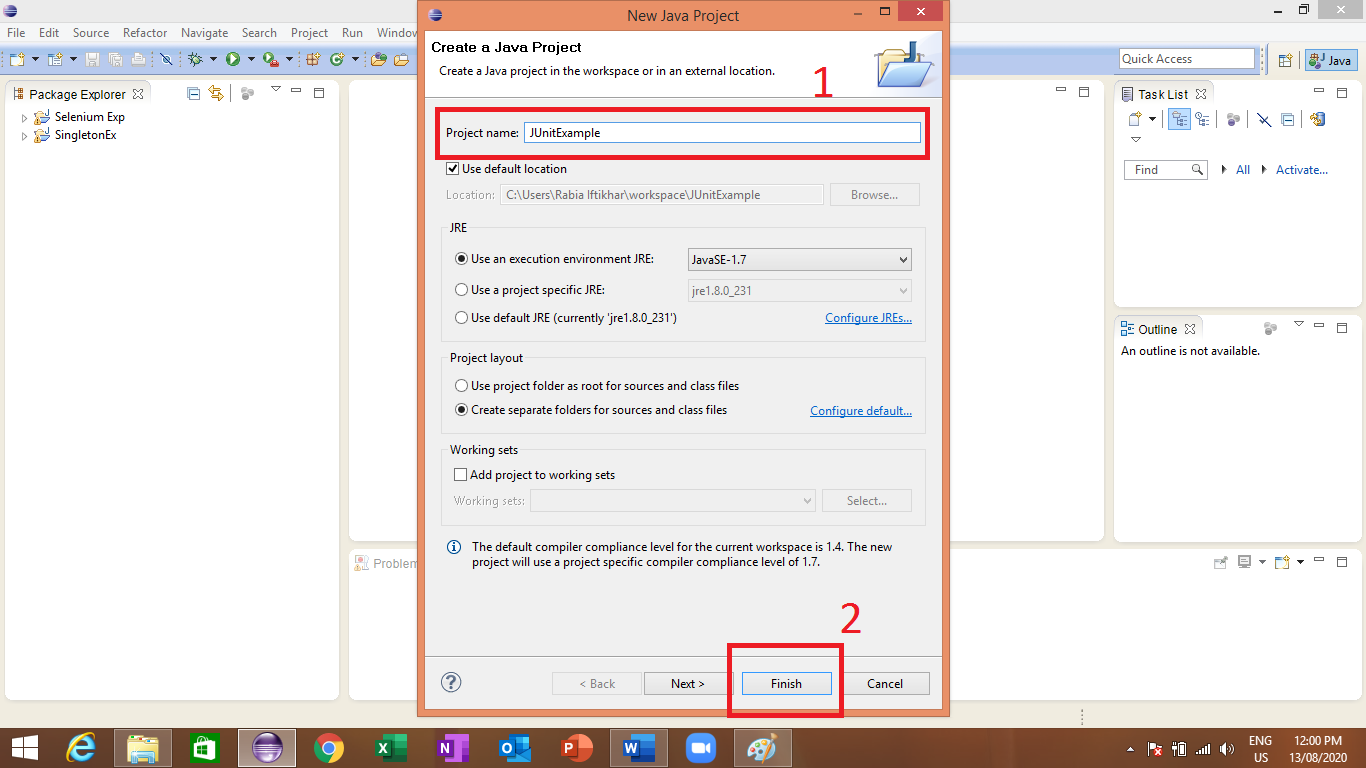
|  |  |
| --- | --- |
| *assertTrue(Boolean condition)*: | This assertion passes if the condition is *true*; otherwise, it fails. |
| *assertEquals(Object expected, Object actual)*: | This assertion passes if the expected and the actual objects are equal according to the equals() method; otherwise, the assertion fails. |
| *assertEquals(int expected, int actual)*: | This assertion passes if expected and actual are equal according to the = = operator; otherwise, the assertion fails. For each primitive type int, float, double, char, byte, long, short, and boolean, the assertion has an overloaded version. |
| *assertEquals(double expected, double actual, double tolerance):* | This assertion passes if the absolute value of the difference between expected and actual is less than or equal to the tolerance value; otherwise, the assertion fails. The assertion has an overloaded version for float inputs. |
| *assertSame(Object expected, Object actual)* | This assertion passes if the expected and actual values refer to the same object in memory; otherwise, the assertion fails. |
| *assertNull(Object testobject)* | This assertion passes if test object is null; otherwise the assertion fails. |

You usually have multiple asserts in each test method, as you prove various aspects and relationships of the method(s) under test. But when an assert fails, that test method will be aborted—the remaining assertions in that method will not be executed this time. Fix any test as soon as it fails, and keep all tests passing all the time.

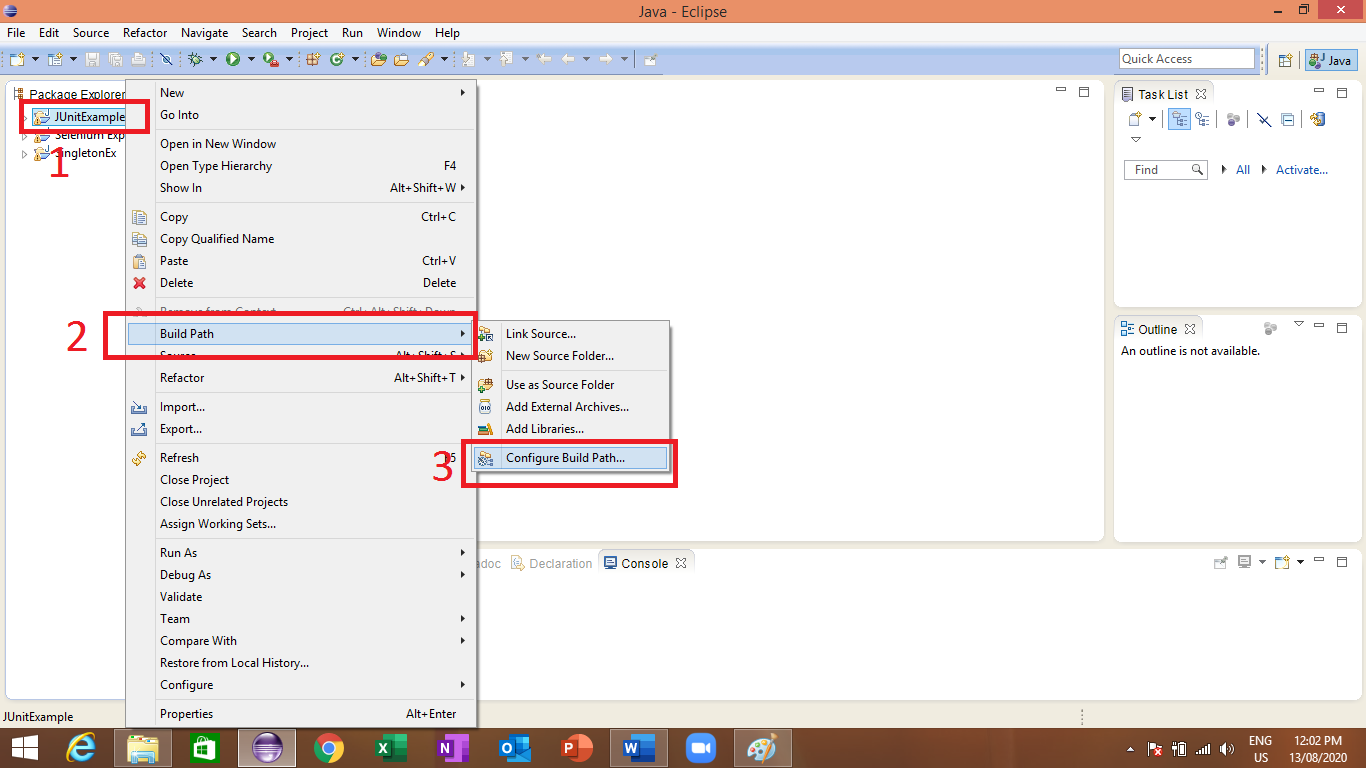
**Steps**

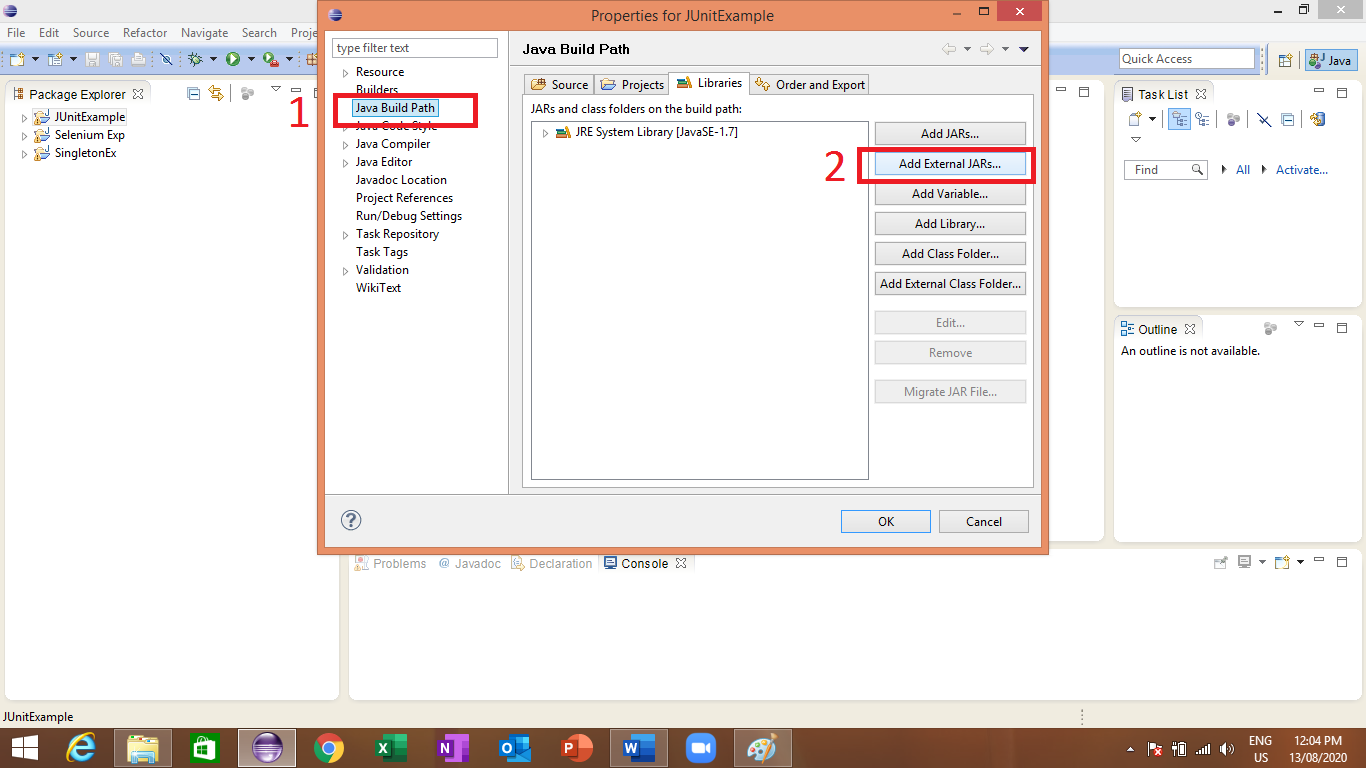
1. *Create a Java project named “JunitExample”.*

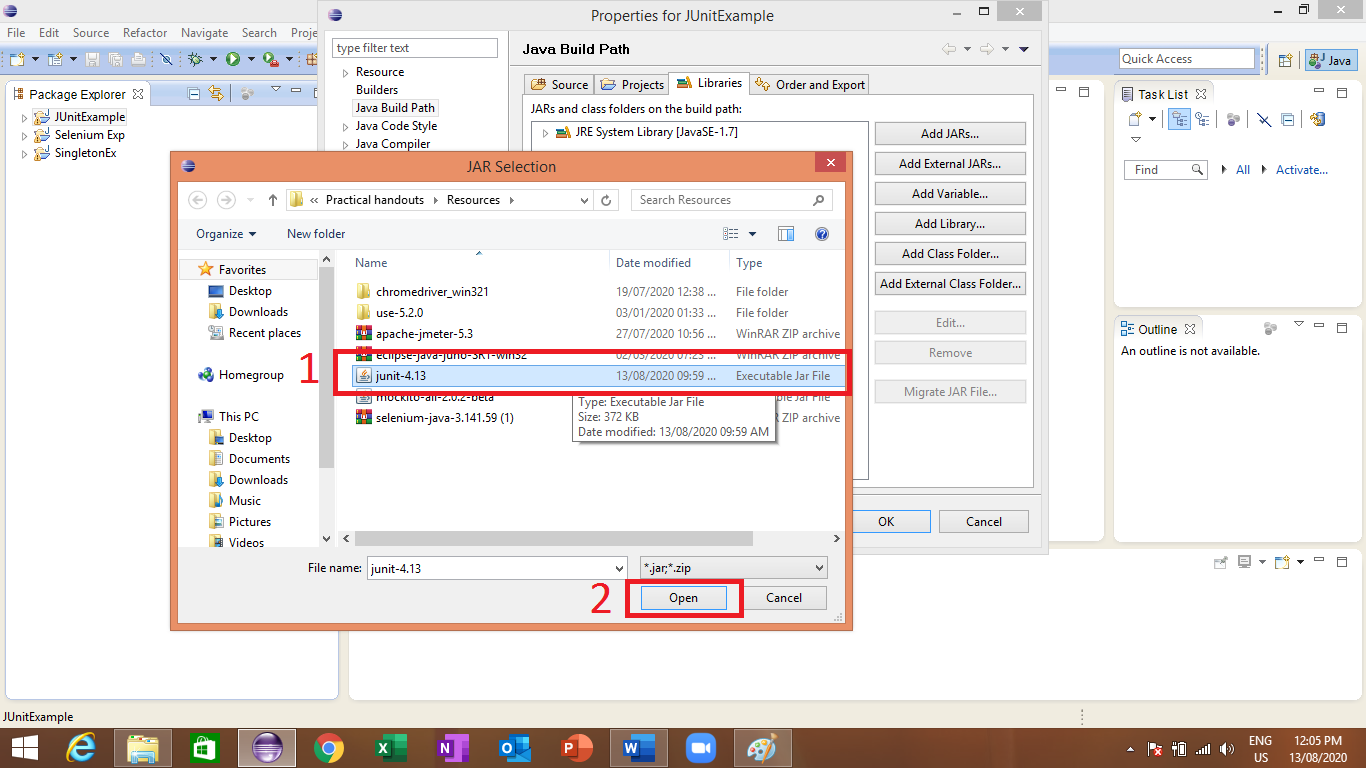
**

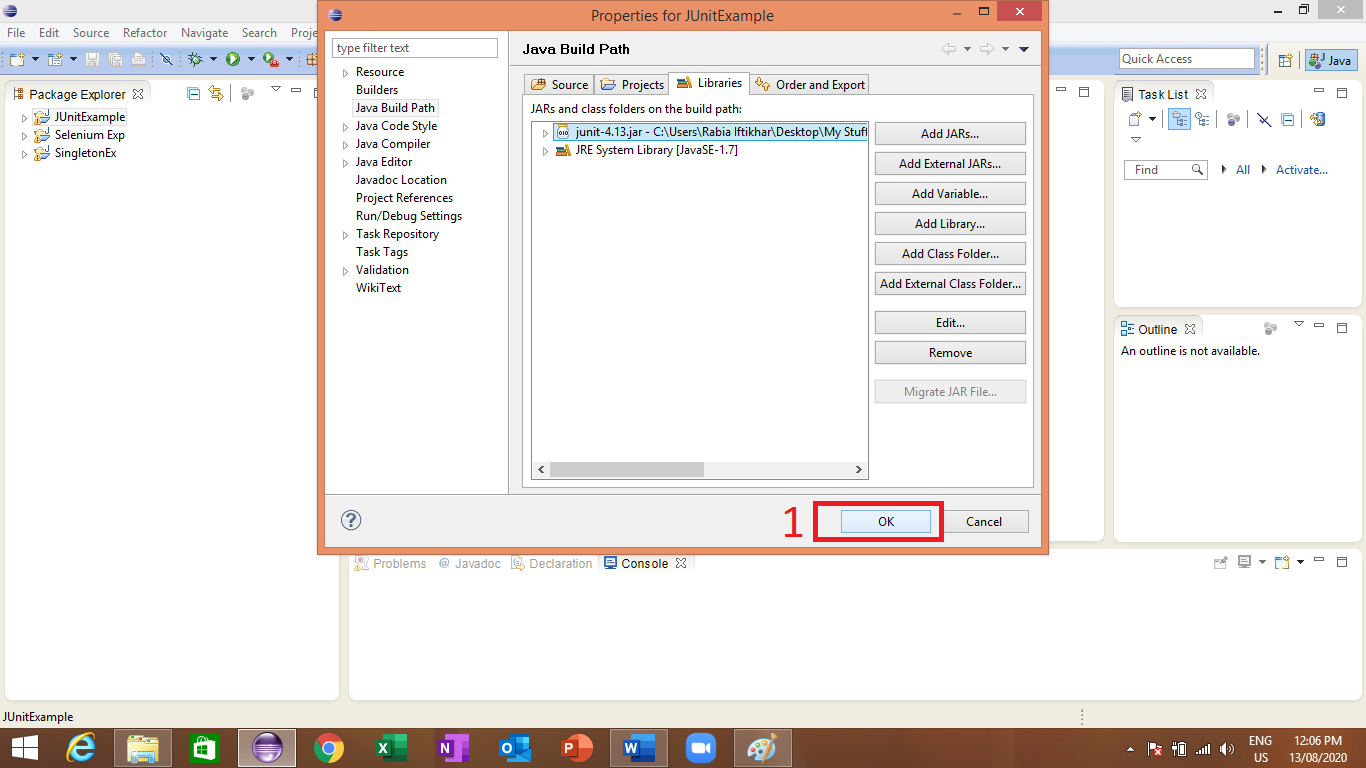
**

1. *Configure build path of your project to add Junit. Jar file to it (note: jar file is attached with this file under the resources section).*

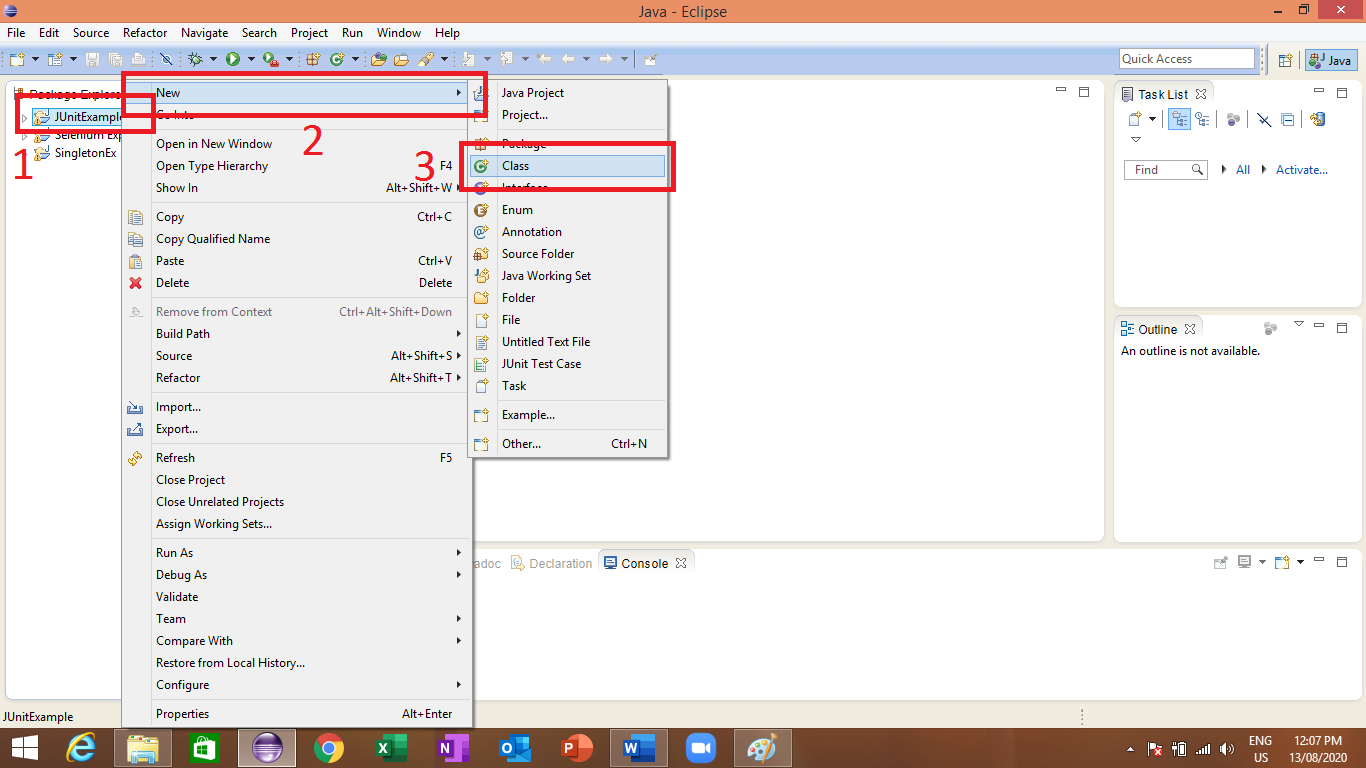
**

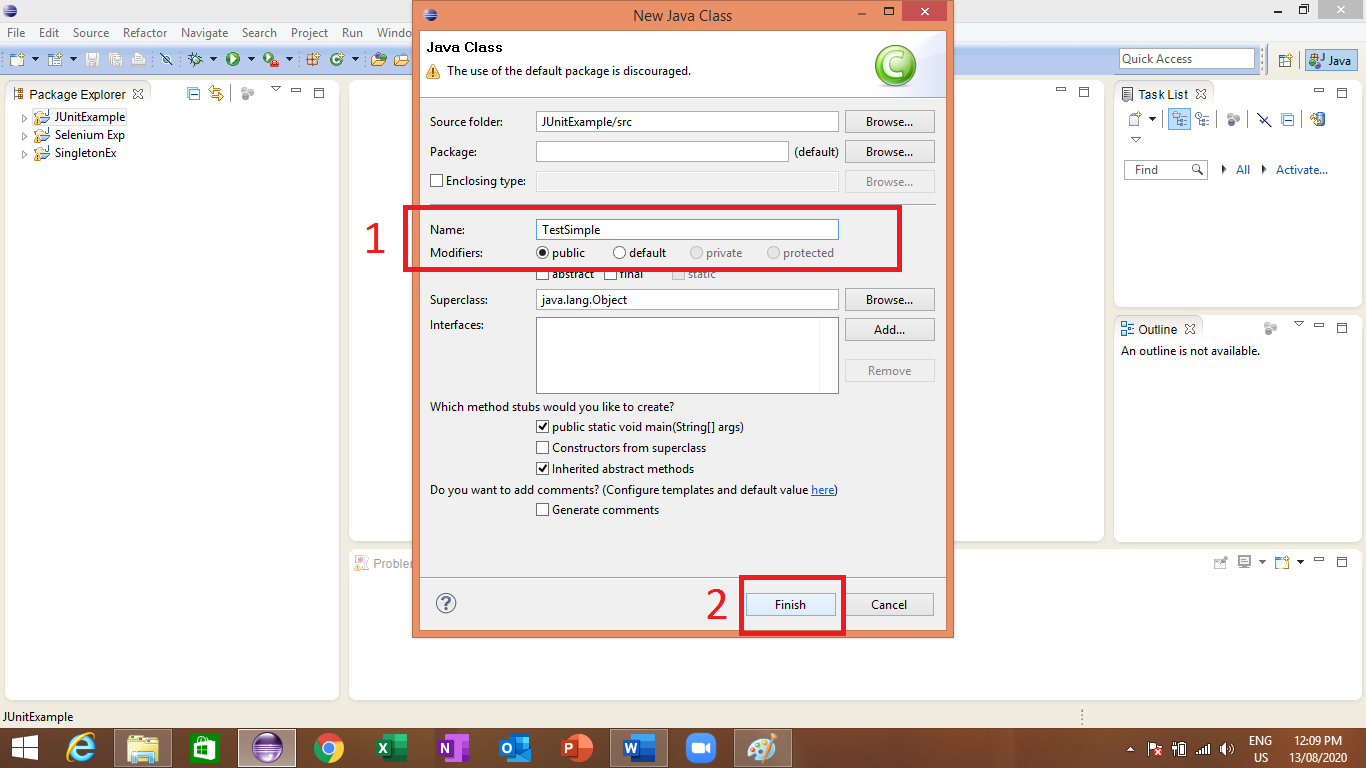
**

**

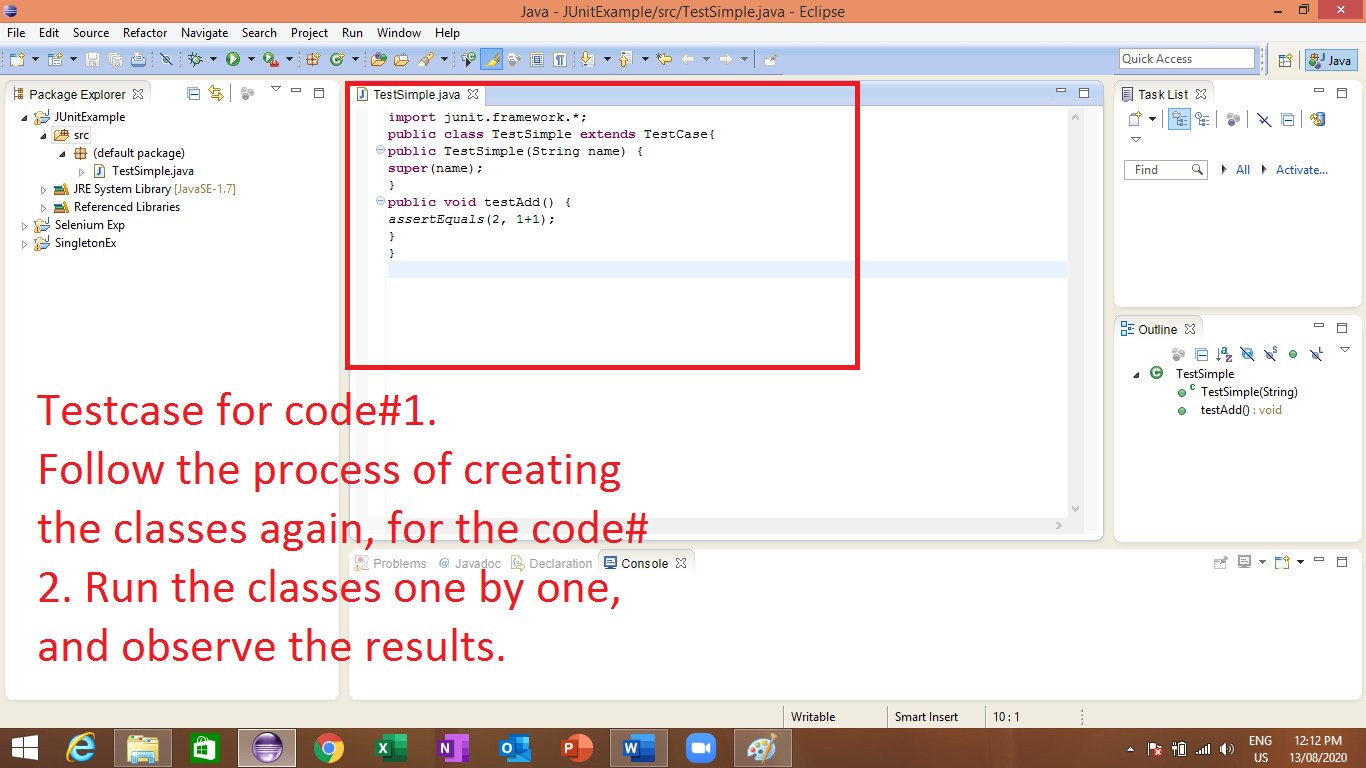
**

1. *Create a class in your project named “TestSimple”.*

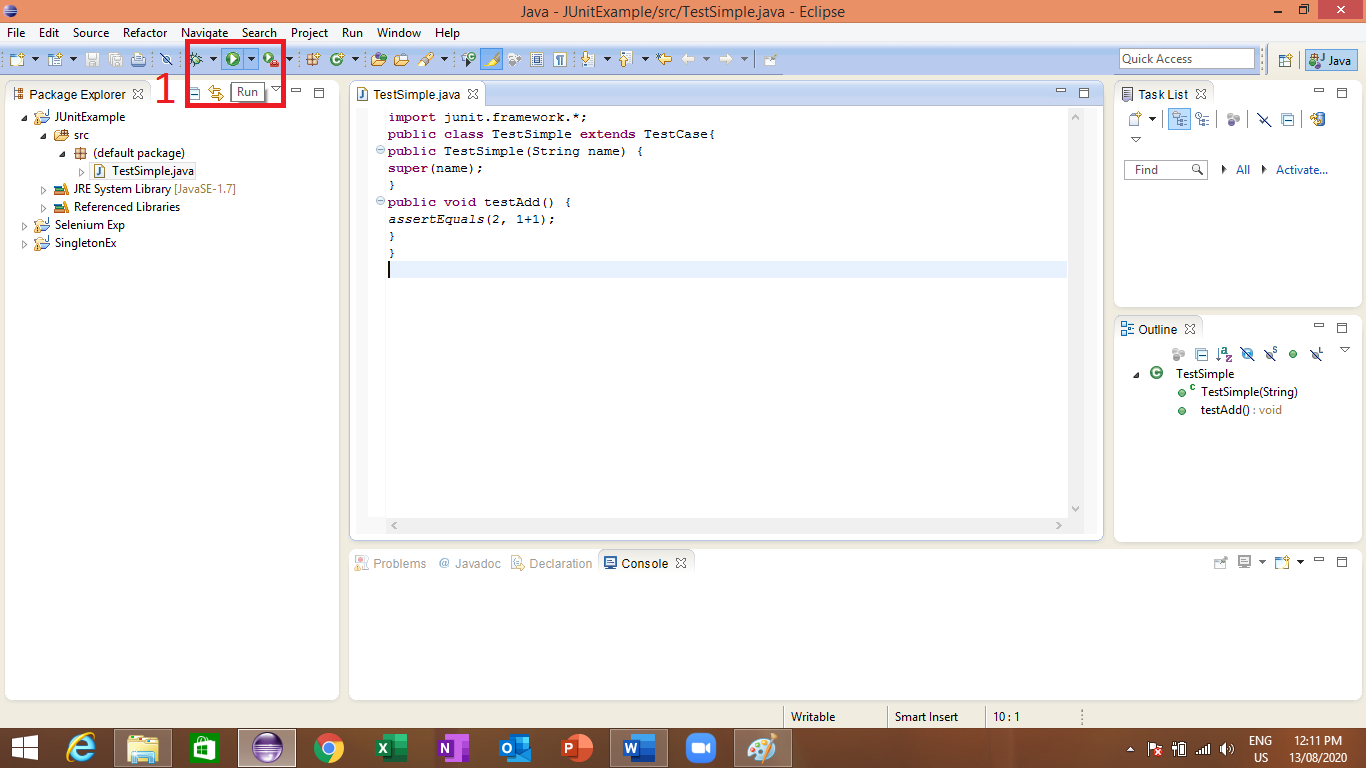
**

**

1. *Write the code in the examples given below (i.e. separate testcases for all the two example codes).*

**

1. *Run the test classes one by one to observe the results.*

**

**Example Code**

**Code# 1**

import junit.framework.\*;

public class TestSimple extends TestCase{

public TestSimple(String name) {

super(name);

}

public void testAdd() {

assertEquals(2, 1+1);

}

}

**Code# 2**

import junit.framework.\*;

public class TestSimple extends TestCase{

public TestSimple(String name) {

super(name);

}

public void testAdds() {

assertEquals(2, 1+1);

assertEquals(4, 2+2);

assertEquals(-8, -12+4);}

}

The above codes will run and return true or false based on the returned assert statements results.

|  |
| --- |
| Lab Tasks |

1. Rewrite the codes written in this handout and observe the results. Also, explain the results.
2. Create a program that checks the response when valid email address and password is entered. Test the program using JUnit framework.
3. Create a calculator program that checks all its operations using JUnit framework.

|  |  |
| --- | --- |
| Lab Tasks Assessment/Rubrics along with Score/Marks | |
| *Rubric Description* | ***Rubric Marks*** |
| 1. Test coverage | 0.5 |
| 1. Testing correctness | 0.25 |
| 1. Test completeness | 0.25 |