EXPERIMENT NO. 8

Aim: To write test cases for white box testing.

Requirements: Windows/MAC/Linux O.S, Compatible version of JDK and Eclipse

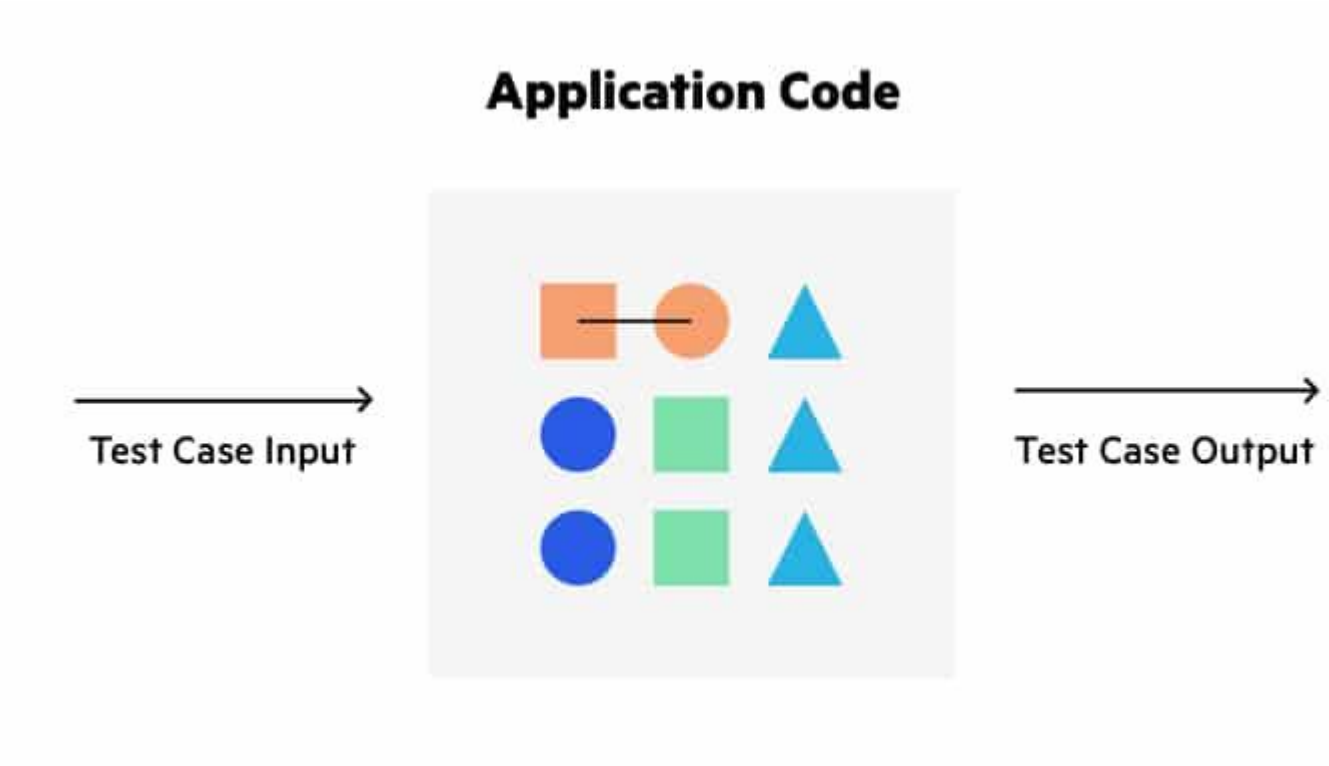
Problem Statement: To design test cases for white box testing and test it using Eclipse.

Theory:

What is White Box Testing?

White box testing is an approach that allows testers to inspect and verify the inner workings of a software system—its code, infrastructure, and integrations with external systems. White box testing is an essential part of automated build processes in a modern Continuous Integration/Continuous Delivery (CI/CD) development pipeline.

White box testing is often referenced in the context of Static Application Security Testing (SAST), an approach that checks source code or binaries automatically and provides feedback on bugs and possible vulnerabilities.



White box testing provides inputs and examines outputs, considering the inner workings (control structure) of the code.

What Does White Box Testing Focus On?

White box tests can focus on discovering any of the following problems with an application's code:

- **Security gaps and vulnerabilities** — checking to see if security best practices were applied when coding the application, and if the code is vulnerable to known security threats and exploits.
- **Broken or poorly structured paths** — identifying conditional logic that is redundant, broken or inefficient.
- **Expected output** — executing all possible inputs to a function to see if it always returns the expected result.
- **Loop testing** — checking single loops, concatenated loops and nested loops for efficiency, conditional logic, and correct handling of local and global variables.
- **Data Flow Testing (DFT)** — tracking variables and their values as they pass through the code to find variables that are not correctly initialized, declared but never used, or incorrectly manipulated.

Testing Techniques and Code Coverage

One of the main goals of white box testing is to cover the source code as comprehensively as possible. Code coverage is a metric that shows how much of an application's code has unit tests checking its functionality.

Within code coverage, it is possible to verify how much of an application's logic is actually executed and tested by the unit test suite, using concepts like statement coverage, branch coverage, and path coverage. These concepts are discussed in more detail below.

Statement Coverage

Statement coverage is a white box testing technique that ensures all executable statements in the code are run and tested at least once. For example, if there are several conditions in a block of code, each of which is used for a certain range of inputs, the test should execute each and every range of inputs, to ensure all lines of code are actually executed.

Statement coverage helps uncover unused statements, unused branches, missing statement that are referenced by part of the code, and dead code left over from previous versions.

Branch Coverage

Branch coverage maps the code into branches of conditional logic, and ensures that each and every branch is covered by unit tests. For example, if there are several nested conditional statements:

```
if X then..
  if Y then..
    A
    B
  else
    if Z then..
```

C

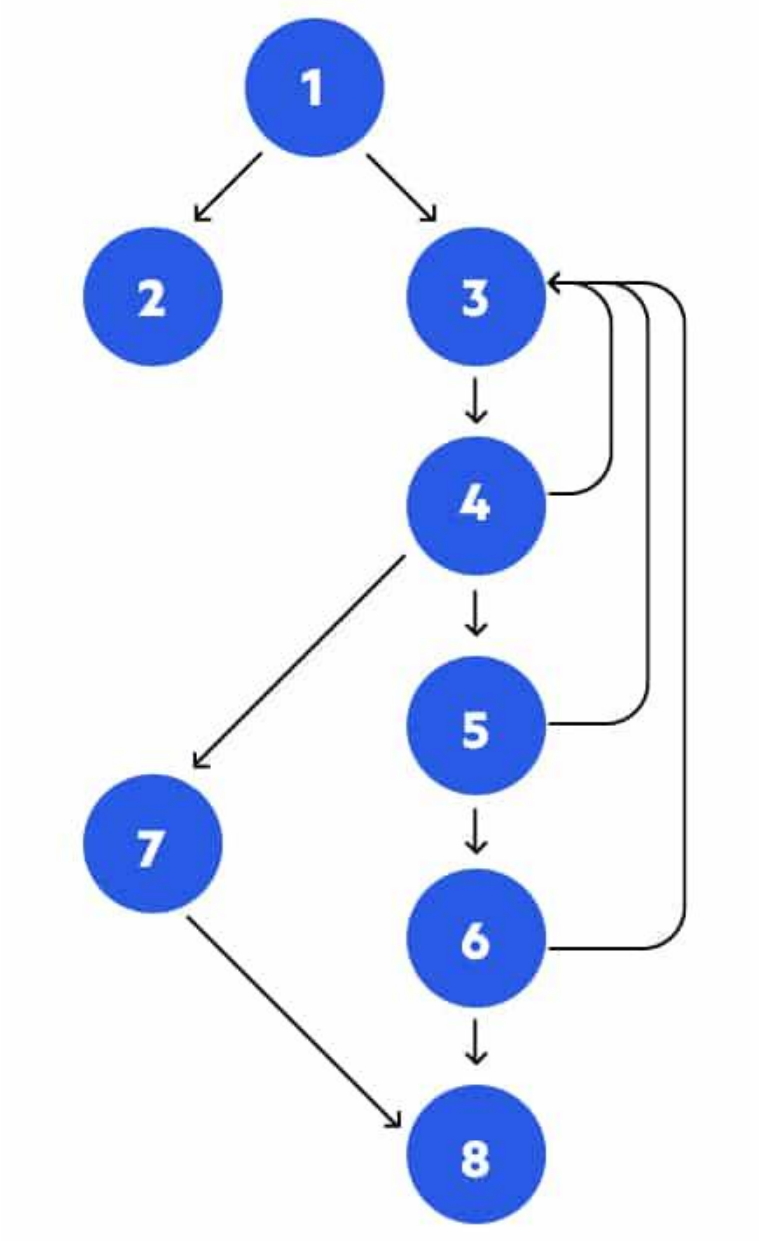
else..

D

A, C, and D are **conditional branches**, because they occur only if a condition is satisfied. B is an **unconditional branch**, because it is always executed after A. In a branch coverage approach, the tester identifies all conditional and unconditional branches and writes code to execute as many branches as possible.

Path Coverage

Path coverage is concerned with linearly independent paths through the code. Testers draw a control flow diagram of the code, such as the example below.



Control flow diagram used to design tests in a path coverage approach

In this example, there are several possible paths through the code:

- 1, 2
- 1, 3, 4, 5, 6, 8
- 1, 3, 4, 7, 6, 8
- etc.

In a path coverage approach, the tester writer's unit tests to execute as many as possible of the paths through the program's control flow. The objective is to identify paths that are broken, redundant, or inefficient.

ECLIPSE:

Eclipse is an integrated development environment that is used in computer programming.

It is the mostly widely use Java IDE and contains a base workspace and an extensible plug-in system for customising the environment.

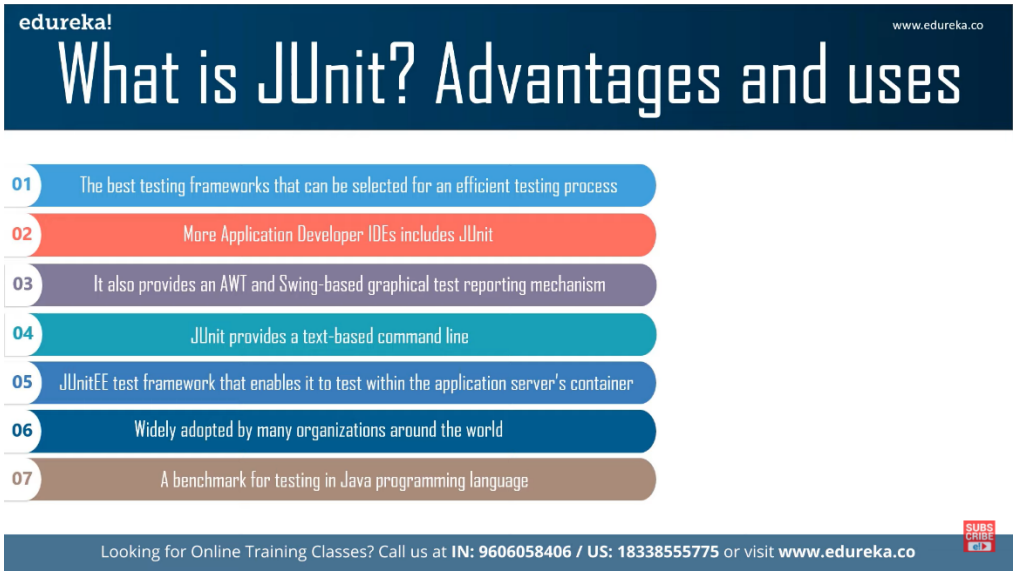
The platform has been designed to build integrated web and application development tooling.

It is designed to not offer a huge amount of end user functionality but the value of the platform comes with its ability to encourage the rapid development of integrated features based on a plug-in model.

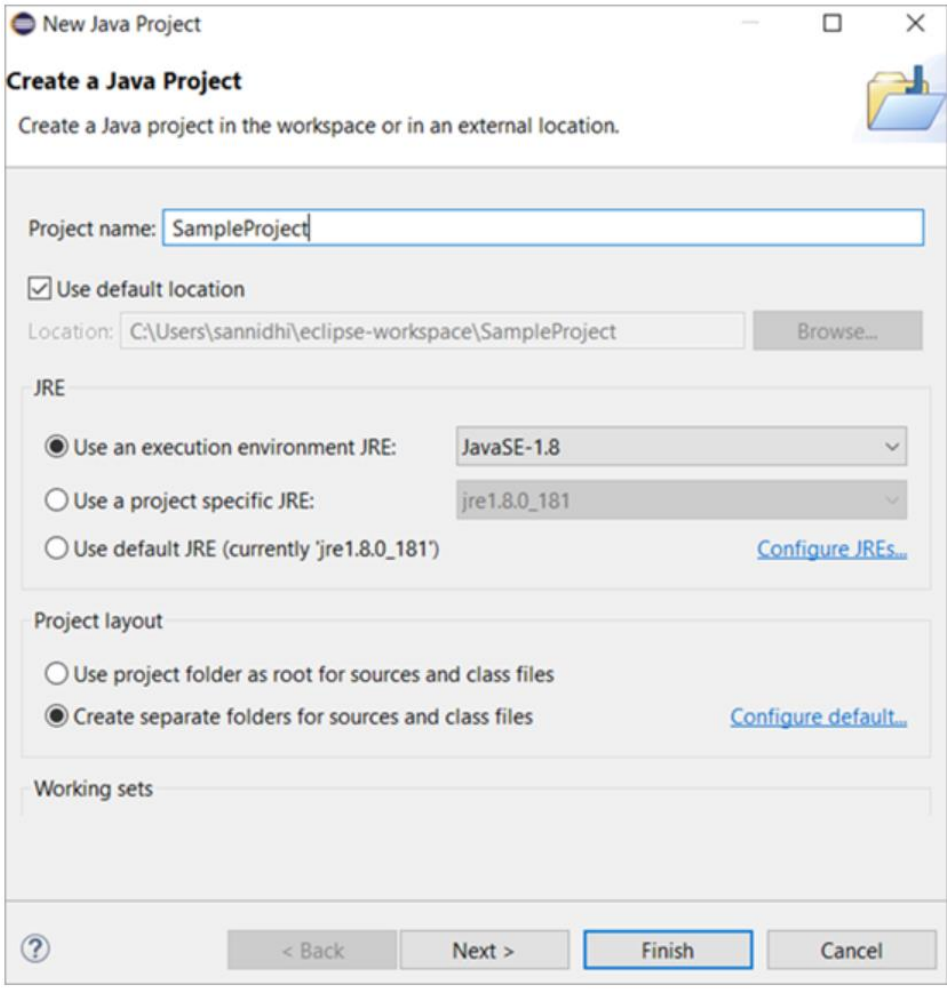
Eclipse provides a common user interface model for working with tools and is designed to run on multiple operating systems.

JUNIT:

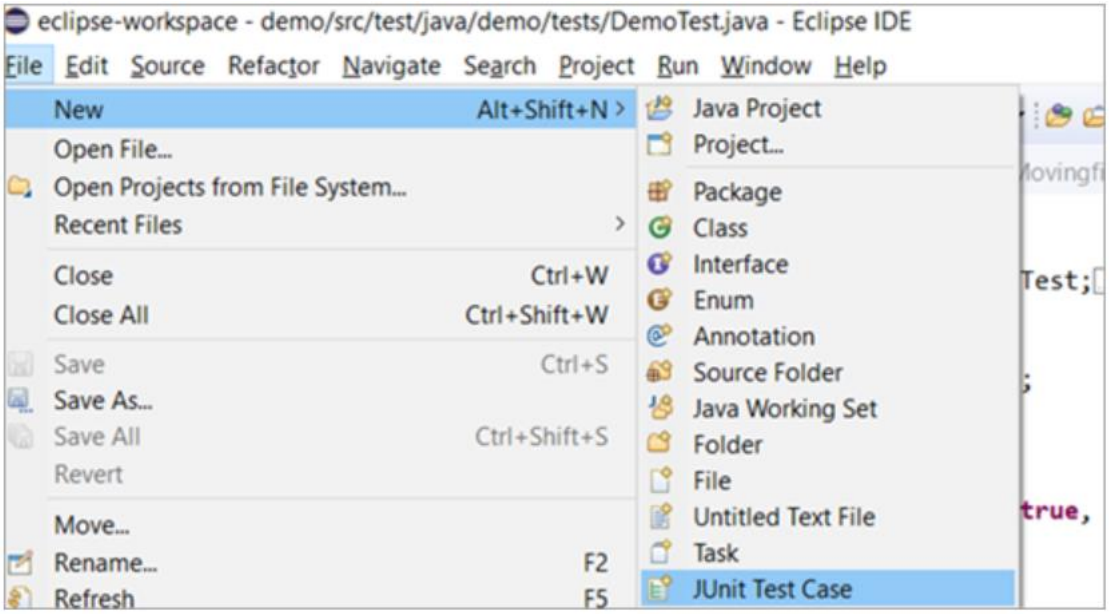
JUnit is a unit testing framework for Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks collectively known as xUnit, that originated with JUnit.



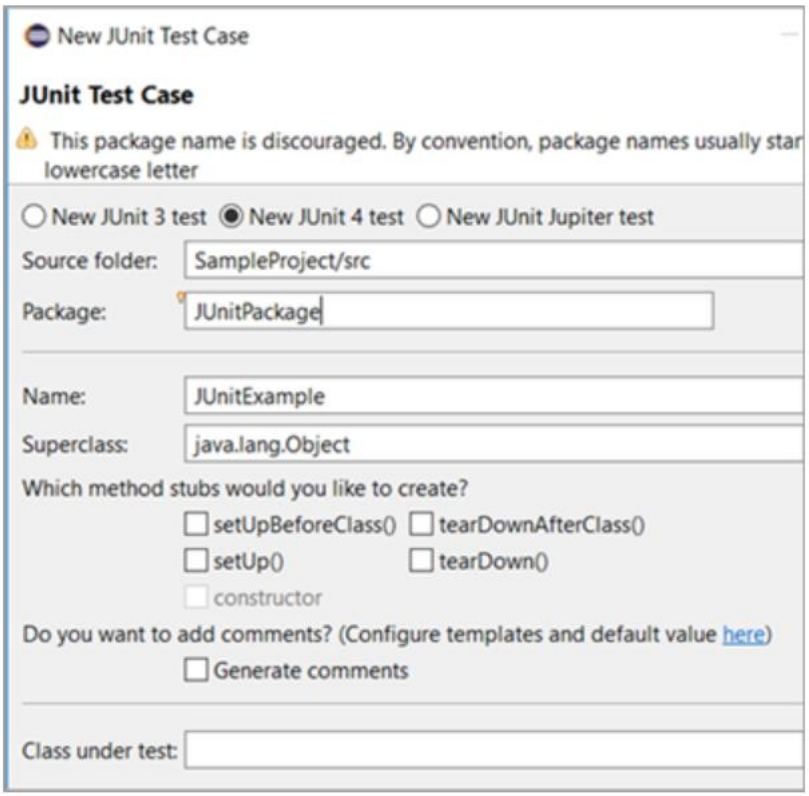
Output:



After installing Junit tool, we have to add new Junit test case into project folder.



Then we have to select source folder path in source folder package name, stub method if any and then click on finish.



Now you will get default template.

```
1  package demo.tests;
2
3  import static org.junit.Assert.*;
8
9  public class JUnitProgram {
10
11     @Before
12     public void setUp() throws Exception {
13     }
14
15     @After
16     public void tearDown() throws Exception {
17     }
18
19     @Test
20     public void test() {
21         fail("Not yet implemented");
22     }
23
24 }
```

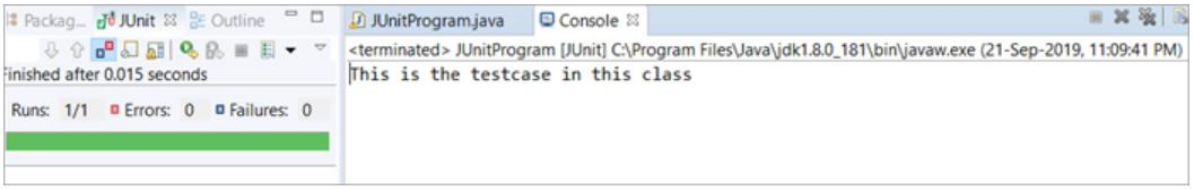
Test Case:

The test case should verify the string statements, if the string typed by user matches with the given string statement then we can say that the test case is successfully completed and if the user enter another statement then we can say that test case is unsuccessful.

```
package demo.tests;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
public class JUnitProgram {
    @Test
    public void test_JUnit() {
        System.out.println("This is the testcase in this class");
        String str1="This is the testcase in this class";
        assertEquals("This is the testcase in this class", str1);
    }
}
```

Here, it verifies it str1 variable and string passed in the condition are both equal.

The comparison of expected condition has been performed by **assertEquals()** method which is JUNIT specific method.



The JUNIT result tab displays mainly number of test cases run, number of errors and number of failures encountered.

Conclusion: We have successfully design test cases for white box testing and implemented them using JUnit with Eclipse in Java.