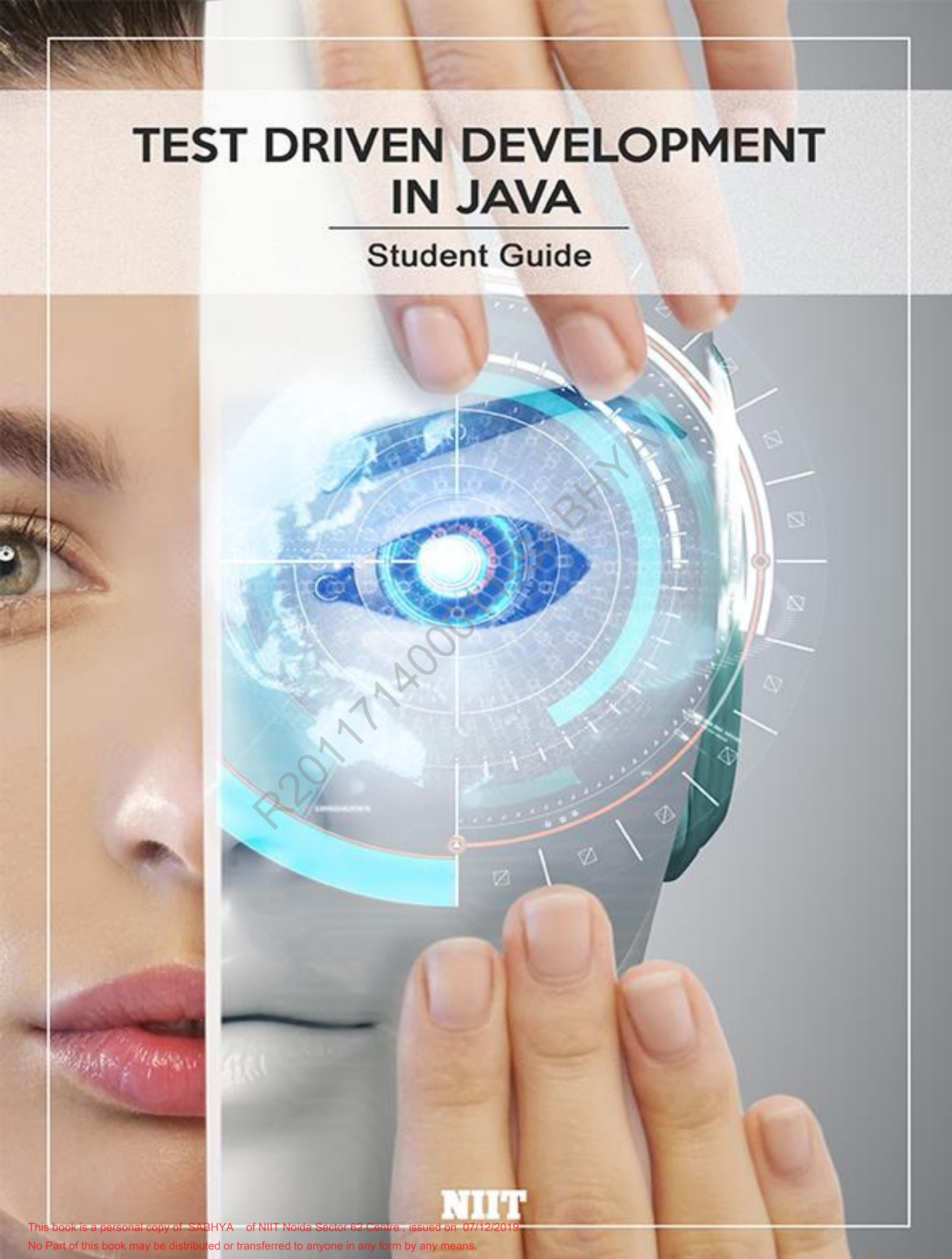
TEST DRIVEN DEVELOPMENT IN JAVA

Student Guide



NIIT

Test Driven Development in Java

Student Guide

Test Driven Development in Java/SG/18_M08_V01
Copyright ©NIIT. All rights reserved.

Course Design - Student Guide

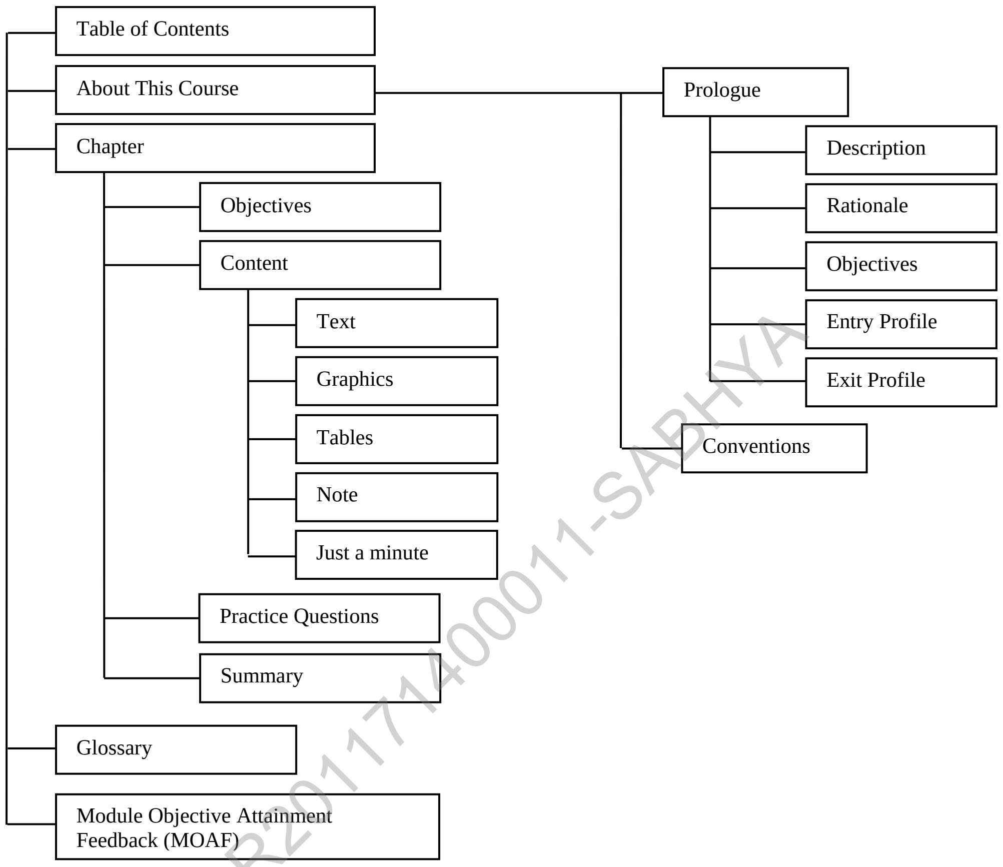


Table of Contents

Chapter 3 – Structuring Test Cases

Glossary

About This Course

Prologue

Description

Test Driven Development in Java is the most popular open-source regression-testing framework used by the developers to implement unit testing in Java, accelerate programming speed, and enhance the quality of code. This course will introduce you to the concepts of testing and allow you to test different types of Java applications. You will learn to test software so that quality software is created. It ensures that a defect-free application is developed. In addition, this course will introduce the different types and techniques of testing.

Rationale

Today, businesses run on software and are severely impacted when software applications malfunction. Software applications, particularly those on the cloud, often serve millions of end-users. High-availability and bug-free software applications are crucial to ensure customer satisfaction. Software applications that perform poorly can spoil the brand and business of an organization. Thus, software testing is vital to ensure the quality of a software application before it is deployed at a data center or released to customers.

The principle objective of software testing is to ensure that a quality product is delivered to the customer. To achieve this, a developer must test each module/method/unit of code written to create the software application. Unit testing is a good way to check that the code is functional and delivers as expected.

Objectives

After completing the Test Driven Development in Java course, a learner will be able to:

- Identify software testing
- Identify unit testing
- Write test cases in JUnit
- Add behaviors to test cases
- Test multiple tests in Junit

Entry Profile

The students who want to study this course should have knowledge of Java Programming Language (Core as well as Web-based), JSP, and Servlets. In addition, the students should have knowledge about database and MYSQL.

Exit Profile

At the end of this course, the students should be able to unit test Test Driven Development in Java framework.

Conventions

Convention	***Indicates…***
📎	*Note*
⏰	*Just a minute*
	Placeholder for an activity. The steps to complete the activity are included in the book, Test Driven Development in Java – Lab Guide.

Getting Started with Software Testing

CHAPTER 1

While developing an application, developers tend to make certain mistakes that are not easily identified by them. Post development, if an application is directly deployed at the actual server without being tested, it can lead to severe issues, such as system breakdown, customer dissatisfaction, or legal problems.

However, if the application is thoroughly tested before being deployed at the production environment, then these issues can be easily avoided. Software testing plays a vital role in development of a bug-free application and helps in delivering a defect-free product.

This chapter introduces the concept of software testing along with the software-testing life cycle. In addition, it discusses different types of testing and identifies JUnit as a unit testing tool.

Objectives

In this chapter, you will learn to:

- Identify software testing
- Identify unit testing

Introducing Software Testing

Consider a scenario of a bank application. This application has different modules, such as user account management, money transfer management, and loan management. Your organization needs to develop this application. The development team develops the application and directly deploys the same to the production environment. However, the customers of the bank encounter an error in the money transfer module of the application wherein the money got deducted from the payer's account, but was not credited to the payee's account. This incident spoils the goodwill of the bank and creates dissatisfaction among its customers. Therefore, to avoid such situations, a software application should be thoroughly tested before it is deployed to the live environment.

Software testing plays a vital role in the development of defect-free software. Software should be tested at the developer's end to identify all the errors and issues in the code, so that there are no leakages of errors to the production environment. To test software comprehensively, a software testing life cycle is devised. It serves as a general guideline for testing an application and can vary according to the testing strategy being used. For example, in the case of the bank application, you can develop the tests either before or after the application is developed. These are two different testing approaches wherein you first develop the code and then test it or you first develop the test cases by understanding and analyzing the scenario and then develop the code according to the test cases.

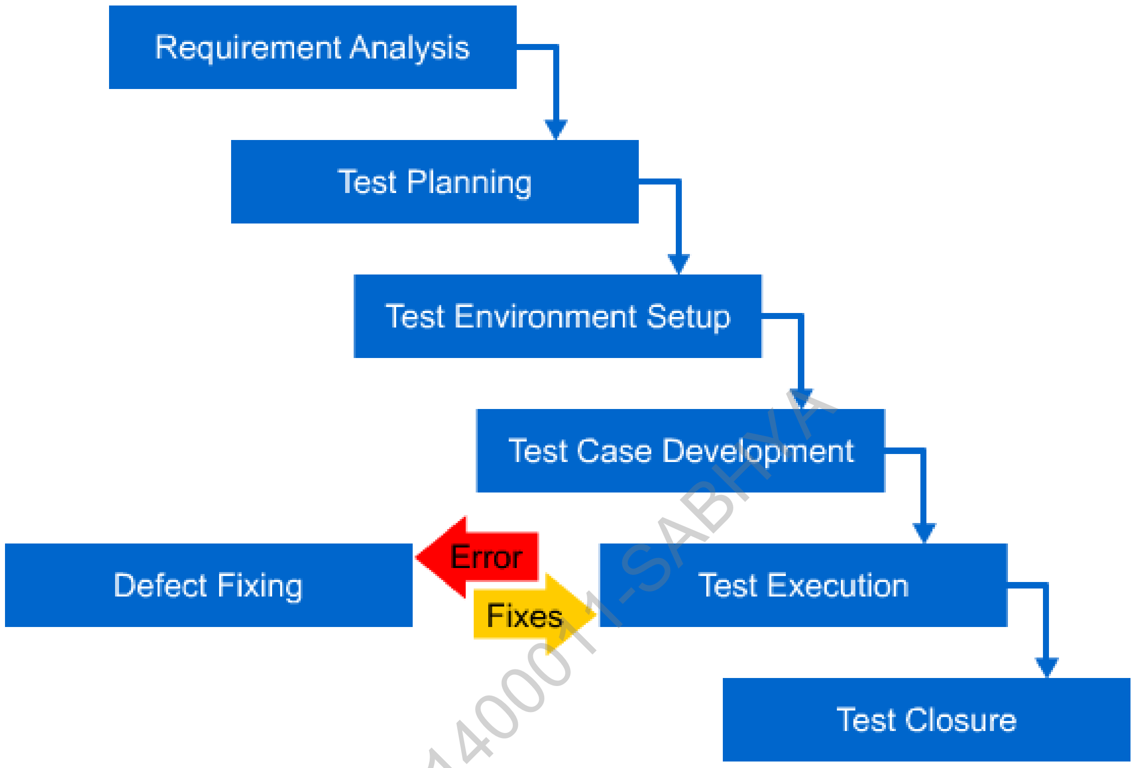
Testing strategy refers to the type of testing approach being used. For example, you can test individual modules, two or more integrated modules, or the complete application.

Let us now understand the software testing life cycle.

Software Testing Life Cycle

Software Testing Life Cycle (STLC) is a systematic and progressive approach used for testing software to enhance its quality. The software should be tested with the intent of finding every possible bug introduced in the code during the software development. It helps in ensuring the delivery of quality software to users with the minimum risk of failure and errors.

STLC comprises the six phases, as shown in the following figure.



The Six Phases of STLC

STLC consists of the following six phases:

1. **Requirement analysis**: This is the initial step of the testing process, which focuses on understanding the requirement that has been laid down in the requirement specification document. After understanding the requirements, you need to identify the elements of an application to be tested and can be tested. If you identify any requirement that is not testable, you can conduct sessions with the required people, such as stakeholders, to resolve the issue. This is the phase to reconsider any non-testable areas or requirements. Apart from identifying the testing elements, you may decide the types of testing that you will perform while testing the application.
2. **Test planning**: Test planning is the most essential step of the testing process that emphasizes on achieving an optimum balance between the specified requirements and the available resources. It also defines the objectives of the testing. Further, you need to determine the required effort and allocate the determined responsibilities to the individuals. You need to prepare the schedule for the complete testing phase of the application. In addition, you need to identify the risks involved, if any, and convey the same to the stakeholders.
3. **Test environment setup**: Setting up the test environment plays a vital role in the testing process. In this phase, you need to decide the hardware and software requirements, such as server, network, or software, needed to test the application. It includes setting up the necessary tools and infrastructure required for convenient and smooth execution of the tests.

4. **Test case development**: During this phase of STLC, the emphasis is on developing the test. In this phase, you write step-by-step instructions for the test conditions that are identified in the initial stages of STLC. It is a good practice to create detailed test conditions based on which test cases can be identified easily. In addition, breaking test conditions into various sub-conditions ensures that the entire code is tested by the test cases and no area of code is left untested. In this phase, you also need to maintain a requirement-traceable metrics, which maps all the requirements specified with the test cases designed.
5. **Test execution**: Once developed, the test cases are executed under the given test environment to verify the test results. The outcome of the test can be a successful or poor endeavor. When the outcome of a test results in failure, you need to fix the issue and retest it. Therefore, test execution phase has a small cycle that executes till the time all results are successful. This cycle involves test execution and defect fixes.
6. **Test closure**: Once the test cases are successfully executed, you need to close the testing life cycle. It includes matching the expected results of the test cases with the desired objectives. In addition, you need to generate test reports specifying the issues identified, issues rectified, test results summary, and test metrics, such as timelines, quality, coverage, and cost. These test results or facts and figures should be reported to the concerned people, such as stakeholders.

Different Types of Testing

Software testing has become a diversified field constituting different types of testing strategies and levels. The testing strategy is the approach followed by you to test software. This strategy can be of the following types:

- **Black box testing**: This testing strategy tests the software for its overall functionality and does not test each statement and branches given in the code. In this type of testing, you are not given access to the source code because you check the software's functionality in terms of input and the desired output only. You define a set of output for a respective set of input and verify if the actual output is same as the defined output. This type of testing is known as black box testing approach because the code is considered to be in a black box and, therefore, not visible. You can just verify the input and output passing though this black box.
- **White box testing**: In this testing strategy, testing is done in detail. Here, you need to understand the code and then test every statement and condition given in the code. While using this strategy, you design test cases such that the test executes all the branches and statement of the code and, therefore, you need to know the code in detail. This type of testing is also known as glass box testing because, in this case, the code is visible to you in a manner that it is kept in a transparent glass box.

Once the testing approach is decided, the testing can be done at various levels, which are:

- **Unit testing**: This type of testing tests the individual units of application, which can be a method of a class, a complete class, a set of inter-related class, or a module. Unit testing is one of the best development practices to test the smaller unit of code. It reduces the chances of encountering errors at a later stage of development and makes the code efficient. The sole objective of the unit testing is to individually perform tests on each unit of an application to ensure their functional efficiency individually.

  For example, consider the scenario of the banking application. In order to develop a modularized application, the application was broken down into various modules, such as user account management, money transfer management, and loan management. These modules are independent and provide various facilities to the users, such as maintaining their data, helping them in transferring money from one account to another, and enabling them to get the loan from the bank. Therefore, these individual modules must be tested for their desired functionality.

- **Integration testing**: When the individual units are tested, they need to be integrated and then tested. This type of testing verifies that different units, when combined together, produce the desired result. Integration testing tests different modules of the given application while integration. This type of testing examines whether each module is working perfectly along with the other modules of the same application on integration.

  For example, consider the scenario of the banking application. In this application, you defined three modules being developed individually. When any two modules of this application were ready, you could combine them together as one unit, which, in turn, could be combined to the third module. Now, integration testing aims at identifying the errors, if any, during the integration of the two modules. When two modules are inter-dependent, you may encounter certain errors on combining them into a single bigger module. Therefore, it helps in tracing the errors easily and rectifying them at an early stage of application development.

- **System testing**: It tests the complete system for the desired functionality. In the case of system testing, you need to combine all the units of an application to build the entire application. Then, this application is tested as a complete system to verify whether the application meets the requirements specified at the beginning of the development phase. System testing tests the reliability of the overall system. The system can be viewed as one unit that was requested by the customer. This level of testing generally uses black box testing approach because in this case, you test the complete functionality of the software by providing certain input and verifying the outcome of that input.

  For example, consider the scenario of the banking application. In this application, the system includes three modules to be checked in an end-to-end manner for detecting errors. It will check how all the modules are able to interact with each other and behave as a system. It covers and tests every detailed input in the application from the user's perspective to determine the probable behavior of the system.

- **Regression testing**: It refers to testing an application after it has been modified. This type of testing aims to find out bugs in the code after it has been changed. It also looks for the effect of the change in other parts of the code. Regression testing is to retest the given modules with same or different tests to ensure that changes in a module have not impacted other modules.

  For example, consider the scenario of the banking application. With the three given modules of the application, suppose a defect hits the user account management module in which the user is unable to login to an account with the given user credentials. This defect is reported to the concerned organization. Once this defect is rectified, the other modules, such as money transfer and loan management, should be tested again to check whether the change in user account management module has affected these modules.

- **Acceptance testing**: As the name implies, acceptance testing refers to the testing done at the customer's end for accepting the software. This type of testing verifies whether the developed application meets the user requirements. Acceptance testing is performed at the client-site in real-time environment. Once the system testing is performed successfully, then the proposed system is considered ready to be deployed at the client-side with all the tools and infrastructure present. The system should use the test data to verify whether it meets all the requirements specified by the customer. When the software is accepted by the user, it denotes the successful delivery of the software.

  For example, consider the scenario of the banking application. At this stage, the entire flow of the application can be walked through by initiating the activity of opening a new account and then transferring money from one account to another. In addition, the user should be able to generate the request for loan, if he wants to avail the same. The walkthrough should display user-friendly messages, wherever required. All these functionalities and features should be seamlessly achieved for the client to appreciate the system.

- **Performance testing**: An application that performs all the tasks perfectly, but responds lately is considered to be inefficient, and users do not prefer using such applications. Therefore, it is essential to check the response time of the application in a usual scenario as well as in a loaded scenario. This type of testing is known as performance testing. Performance testing is to test the entire non-functional performance expected of the system. The non-functional requirement could be related with the precise time in which a Web page of a Web-based application should be opened. The non-functional parameters are not just restricted to time, but also to speed, security, reliability, and other performance-related issues of the application.

  For example, consider the scenario of the banking application. The non-functional requirement specified in the requirement document beholds that the Web page of any module, which is being requested from the user, should open in less than five seconds. Therefore, it is necessary to test this application for various non-functional aspects of the application, which can be done with the help of performance testing.

Just a minute:

Which one of the following testing strategies tests the software for its overall functionality and does not test each statement and branches given in the code?

1. *Regression testing*
2. *Black box testing*
3. *White box testing*
4. *Performance testing*

Answer:

2. Black box testing

Introducing Unit Testing

Consider the scenario of the bank application development. Jenny has been assigned the task of developing the user account management module of the application. This module has multiple classes containing multiple methods. Jenny develops the methods and integrates these together in a class. When she validates the functionality of the class, it fails to generate the desired output. There are bugs in a few methods, which propagate to other methods as well. As a result, she needs to spend time and effort in tracing and rectifying the error. If these methods were tested before they were integrated together, the application development and debugging time could have been reduced. The testing of small pieces of code, known as units, is known as unit testing. This unit can be a piece of code, such as a method, multiple methods of a class, a class, multiple classes, or even small modules of an application. The unit testing should be done to ensure that the smallest unit of the code is bug-free and reusable.

Unit testing provides the following advantages:

- It ensures that the individual units are reliable and error-free, thereby simplifying the integration testing because if individual units are working as expected, then they are expected to work fine when integrated. For example, in the scenario of the bank application, the individual units developed were not unit tested. As a result, Jenny had to spend a lot of time and effort to rectify the error.
- It eases the maintenance of the software because the test designed for the original piece of code can be executed every time the code is modified. This enables you to verify whether the change has affected the original functionality.
- It finds the errors/issues in the code at the development stage, thereby reducing the impact and the cost that might be huge when it occurs at integration testing stage or later. For example, in the scenario of the bank application, Jenny did not perform the unit testing of the methods being developed. As a result, when she integrated these units together, she encountered an error that propagated to other methods as well. Therefore, Jenny had to spend a lot of time and effort to rectify the error, which impacted the cost of the project as well.

While developing applications, you might feel unit testing to be a tedious process. Moreover, you might think that your job is to develop the code and not to test it. However, unit test ensures that the code developed by you is error-free and meets the specified requirement of the application. Therefore, you should always perform unit testing of your code and ensure the quality of the code developed. Java provides the JUnit framework to test the Java code. To test Java code, you can use NetBeans or Eclipse IDE.

Exploring IDEs for Unit Testing

You can test the Java code using the JUnit framework at the command prompt as well as in IDE. However, to use the JUnit framework at command line, you need to perform the following steps:

1. Verify if JDK is installed on your machine. If not, install JDK on your machine.
2. Set the Java environment using the **JAVA_HOME** environment variable and add the Java compiler location to the **Path** system variable.

   For example, set **JAVA_HOME** to **C:\Program Files\Java\jdk1.7.0_25** and append the **C:\Program Files\Java\jdk1.7.0_25\bin** string to **Path**.
3. Download and save the JUnit jar file in your disk.

4. Set the JUnit environment variable and the **CLASSPATH** environment variable.

   For example, if you have saved your JUnit jar file named **junit4.10.jar** to the **JUNIT** folder in the C drive, then set **JUNIT_HOME** to **C:\JUNIT** and the **CLASSPATH** variable to **%CLASSPATH%;%JUNIT_HOME%\junit-4.11.jar;%JUNIT_HOME%\hamcrest-core-1.3.jar;.;**.
5. Create the test file, which has the logic for testing, and the test runner file, which is required to execute the test on the command prompt.
6. Execute the test and verify the result.

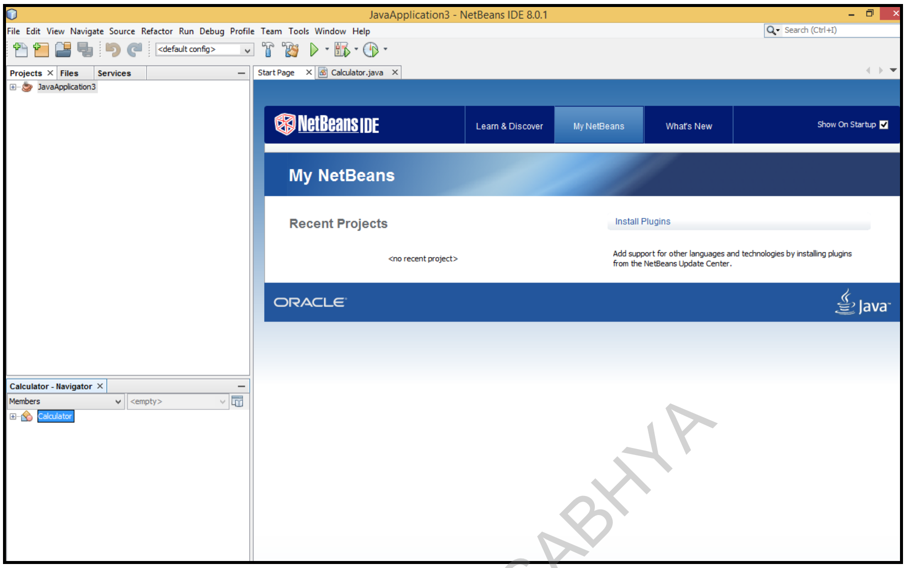
This is a complex and tedious procedure.

To simplify this procedure, there are various IDEs available, such as NetBeans and Eclipse. These IDEs automatically set up the desired environment and download the necessary jar files required to execute unit test code. You just need to write the code and its tests. These IDEs have an inherent support for the JUnit framework and, therefore, you do not need to perform any additional task for testing the code using the IDEs.

There are various IDEs available for different languages and purposes. The commonly used IDEs for the Java programing language are:

- **NetBeans**: It is an IDE primarily used for developing and testing Java applications. However, NetBeans also support other languages, such as HTML5, C, C++, and PHP. Apart from supporting various languages, NetBeans has various other features as well. Some of these features are:
  - It is a tool-based IDE. It provides source-code-related support, such as highlighting the compile-time errors in the code, completing the code intelligently, indenting the code automatically, and balancing the parentheses and braces automatically.
  - It is integrated with in-built tools, such as compiler, debugger, version controller, and profiler.
  - It supports various types of application, such as console based, mobile based, and Web based.
  - It enables you to organize your project efficiently and, therefore, lets you perform project management quickly and easily.
  - It is an open-source IDE sponsored by Oracle and is easy to set up and install.
  - It runs on cross-platform machines including Windows, Mac OS X, GNU/Linux, and Solaris, provided JVM is installed on them.
  - It comes with an in-built support for SQL, MySQL, and Oracle drivers and is easy to configure.
  - It comes with support for various plugins that are easily available within NetBeans. This feature makes NetBeans an extensible IDE.

When you open NetBeans IDE, the interface of NetBeans IDE appears, as shown in the following figure.



Add support for other languages and technologies by installing plugins from the NetBeans Update Center.

NetBeans IDE's Interface

Identifying JUnit as a Testing Tool

JUnit is an open-source testing framework provided for Java programming language. This framework provides various annotations to test the code and makes the testing process simple and quick. The JUnit framework emphasizes the idea of testing the code before it is coded. It means that first you should design the tests for a code, then write the code, and finally test the code using the designed test. This process simplifies the development and debugging of an application because you test a piece of code as soon as you write it. This helps in quickly identifying and fixing the errors, thereby making the code bug-free and reliable.

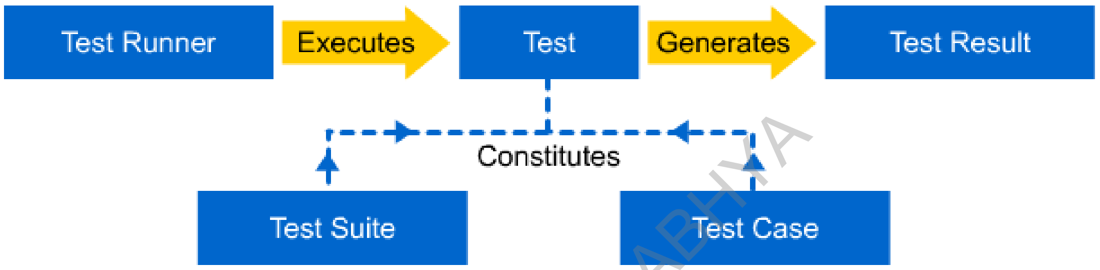
The JUnit framework has the following features:

- To test a unit of code, JUnit provides the unit test case that verifies the output of the code under test in accordance with the input for the code. It means a unit test case can be considered as a simple method that checks the output of a function of a class for a particular input for the function.
- You can automatically execute JUnit test cases individually and then obtain a summarized report in the end.
- It shows a progress bar that signifies the success or failure of test cases in red or green, respectively. Therefore, you can easily interpret the test results.
- A Java program consists of various methods that are encapsulated within a class to create a collection. Similarly, a JUnit test class consists of various test cases. When you have multiple test classes, you can organize these test classes into a collection known as test suite. A test suite is a class that comprises different test classes that are executed one after the other automatically.
- It allows execution of multiple tests concurrently and, therefore, saves time.

Let us now understand the architecture of the JUnit framework.

The JUnit Framework Architecture

The working of the JUnit framework is a simple process, which is easy to understand. JUnit is primarily designed to perform unit testing. To perform testing, you need to create test cases for the methods that you intend to test. These tests need to be executed for verifying the functionality of the application being tested. Therefore, when you perform unit testing using JUnit, the JUnit framework performs certain tasks. The process used by the JUnit framework to execute a test and display the result constitutes the architecture of JUnit. The JUnit framework has the following architecture.



The JUnit Framework Architecture

The architecture of the JUnit framework comprises the following elements:

- **Test case**: This is the smallest unit of any JUnit test that verifies the functionality of a method being tested.
- **Test suite**: When you have multiple test classes and you want to automatically execute these classes one after the other, you can create a test suite.
- **Test**: It is a collection of one or more test cases or test suites that are executed to perform the testing of an application.
- **Test runner**: When you execute a normal Java program file, the output of the program is displayed to you. However, when you execute a test Java file, test results are displayed to the user in form of red or green bar. This is because the execution process of a test is different from a simple Java file. When you execute a test, the test runner is executed in the background, which displays the result of the test in terms of pass or fail.
- **Test result**: It refers to the outcome of a test, which is displayed to the user. This test result is collected from the test class.

Just a minute:

Which one of the following options describes a class that comprises different test classes that are executed one after the other automatically?

1. *Test result*
2. *Test runner*
3. *Test suite*
4. *Test case*

Answer:

3. *Test suite*

Activity 1.1: Creating Test Scenarios

Activity 1.2: Executing Test Cases in JUnit

Practice Questions

1. Which one of the following phases of STLC includes setting up the necessary tools and infrastructure required for convenient and smooth execution of the tests?
   a. Test environment setup
   b. Requirement analysis
   c. Test planning
   d. Test execution

2. Which one of the following phases of STLC involves fixing of defects till the time all the results are successful?
   a. Requirement analysis
   b. Test closure
   c. Test execution
   d. Test case development

3. Which one of the following types of testing ensures that the individual units are reliable and error-free?
   a. Integration testing
   b. System testing
   c. Acceptance testing
   d. Unit testing

4. Which one of the following elements of JUnit architecture is the smallest unit of any JUnit test that verifies the functionality of a method being tested?
   a. Test case
   b. Test suite
   c. Test
   d. Test runner

5. Which one of the following elements of JUnit architecture is executed in the background to execute a test that displays the result of the test in terms of pass or fail?
   a. Test result
   b. Test runner
   c. Test
   d. Test suite

Summary

In this chapter, you learned that:

- Software testing plays a vital role in the development of defect-free software.
- STLC is a systematic and progressive approach used for testing software to enhance its quality.
- STLC consists of the following six phases:
  a. Requirement analysis
  b. Test planning
  c. Test environment setup
  d. Test case development
  e. Test execution
  f. Test closure
- The black box testing strategy tests the software for its overall functionality and does not test each statement and branches given in the code.
- In white box testing strategy, you need to understand the code and then test every statement and condition given in the code.
- The testing of small pieces of code, known as units, is known as unit testing.
- The unit testing should be done to ensure that the smallest unit of the code is bug-free and reusable.
- JUnit is an open-source testing framework provided for Java programming language.
- The architecture of the JUnit framework comprises the following elements:
  - Test case
  - Test suite
  - Test
  - Test runner
  - Test result

NIIT

Implementing Test Cases

CHAPTER 2

Testing an application plays a crucial role in the development process. Testing helps in identifying the correctness and quality of the developed application. Once developed, an application performs a specific task. Usually, an application takes in a set of values on which it operates, and then returns an output. To test such an application, you need to write a code that matches the generated output with the expected output and tests the application against the desired conditions. Such a code is referred to as a test case.

This chapter teaches how to perform testing on an application. In addition, it discusses about the various techniques to enhance your testing.

Objectives

In this chapter, you will learn to:

- Write test cases in JUnit
- Add behaviors to test cases

Writing Test Cases in JUnit

When you test a Java application, you generate a set of test scenarios that can be applied over the application to test its functionality. To implement these test scenarios, you need to create test cases. A test case carries a set of instructions based on which you can test the functional aspect of the code being tested. A test case basically consists of three parts: an input, an event, and an expected response. Consider a scenario where you need to test the `add()` method that adds two numbers passed as parameters. To test this function, you need to pass two values to it as parameters, and then compare the actual result with the expected result, as shown in the following code snippet:

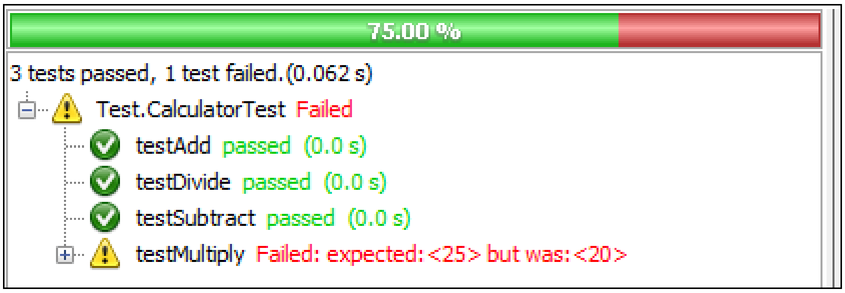
```
public void test()
{
    int actual_result, expected_result;
    actual_result = add(2,3);
    expected_result = 5;
    if(actual_result == expected_result)
    {
            System.out.println("Test passed");
    }
    else
    {
            System.out.println("Test failed");
    }
}
```

In the preceding code snippet, the `add()` method takes two values, 2 and 3, and adds them. The result of the `add()` method is stored in the `actual_result` variable. In addition, the `expected_result` variable stores the value that is expected to be returned by the method. When the `test()` method executes, the `add()` method is called and the result gets stored in the `actual_result` variable. This result is then compared with the expected value specified in the `expected_result` variable. If the actual result matches the expected result, the test passes. Otherwise, the test fails. The `test()` method can be referred to as a test case that tests the `add()` method.

In JUnit, you write test cases by creating test classes in a separate package to separate the test code from the application code. You can have multiple test cases inside a test class or you can have multiple test classes containing class-specific test cases. Consider a scenario of a calculator application that you have developed in Java. It consists of various Java methods, such as `add`, `subtract`, `multiply`, and `divide`, to implement the required functionality. Now, to test this application, you can have multiple test classes that contain test cases specific to the methods, such as `testAdd.java`, `testSubtract.java`, `testMultiply.java`, and `testDivide.java`, or you can have a single test class, such as `CalculatorTest.java`, containing multiple test cases for the methods of the calculator application.

Once the test cases are ready, they can be used to test the application. A test case either passes or fails. If a test case passes, it is indicated by a green signal. If a test case fails, it is indicated by a red signal. If a test class contains multiple test cases, some of them may pass, while others may fail.

In such a case, the result of each test case, whether passed or failed, is displayed to the user, as shown in the following figure.


75.00 %

3 tests passed, 1 test failed.(0.062 s)

```
Test.CalculatorTest Failed
    testAdd passed (0.0 s)
    testDivide passed (0.0 s)
    testSubtract passed (0.0 s)
    testMultiply Failed: expected:<25> but was:<20>
```

The Result of Test Cases

In the preceding figure, there are four test cases—`testAdd`, `testSubtract`, `testMultiply`, and `testDivide`—inside the test class, `CalculatorTest.java`. Of these four test cases, three of them, `testAdd`, `testSubtract`, and `testDivide`, are passed, while the `testMultiply` test case is failed. The overall test, `Test.CalculatorTest`, is failed as all the test cases within the test are not passed.

To write a test case, you need to configure the methods inside the test class. Configuring the methods require you to mark the methods with defined JUnit annotations. In addition, you need to learn about the assertion methods that help you verify the actual result with the expected result.

Identifying JUnit Annotations

Consider a scenario where you need to create a test case. Now, you want certain functions to be called before and after the execution of the test cases. Even if you write all those functions, how will you let the complier know the function that needs to be called and when? JUnit provides you with a set of annotations that can help you with such tasks. JUnit annotations help you identify and configure various components of a test class. These annotations make your test code readable and easier to understand.

JUnit provides you with the following commonly used annotations that are used to configure test cases:

- `@Test`
- `@Before`
- `@After`
- `@BeforeClass`
- `@AfterClass`

@Test Annotation

The `@Test` annotation informs JUnit that the method, it has been attached to, needs to be run as a test case. To run the method, JUnit first builds a new instance of the class, and then invokes the annotated method. If the method throws any exception, JUnit will report the exception as a failure. If no exceptions are thrown, the test is considered to be passed.

The syntax for the `@Test` annotation is:

```
@Test
public void testXXX()
{
........................
........................
}
```

Consider the scenario of the `Calculator.java`, which includes the functionalities of the `add()`, `subtract()`, `multiply()`, and `divide()` methods, as shown in the following code:

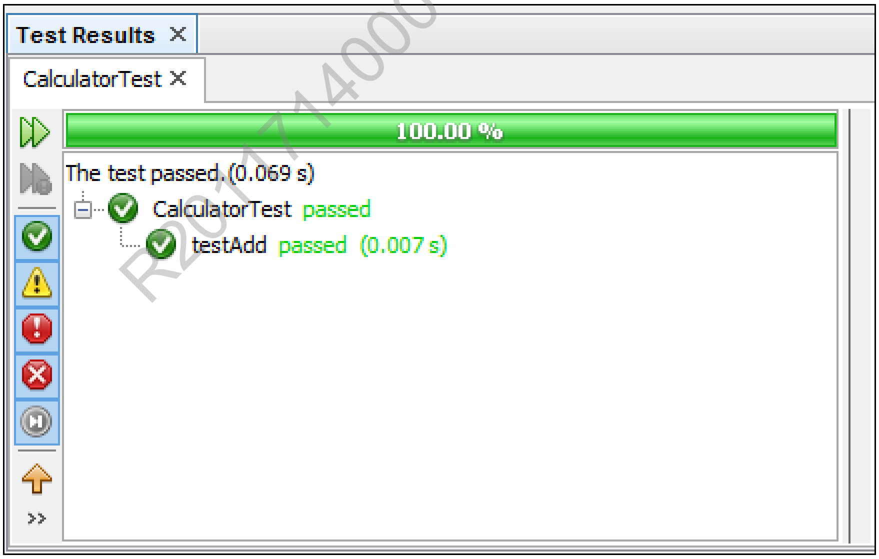
```
public class Calculator {

   public int add(int firstNumber,int secondNumber)
   {
       return firstNumber+secondNumber;
   }
public int subtract(int firstNumber,int secondNumber)
    {
        return firstNumber-secondNumber;
    }
    public int multiply(int firstNumber,int secondNumber)
    {
        return firstNumber*secondNumber;
    }
    public int divide(int firstNumber,int secondNumber)
    {
        return firstNumber/secondNumber;
    }
}
```

The following code displays the JUnit test method, `testAdd()`, which tests its corresponding `add()` method in the `Calculator.java` class:

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculatorTest {

    public CalculatorTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
```

```
        public static void tearDownClass() {
        }

        @Before
        public void setUp() {
        }

        @After
        public void tearDown() {
        }
        /**
         * Testing the add method of the Calculator class
         */
        @Test
        public void testAdd() {
        Calculator c1=new Calculator();
    int expectedresult=8;
    assertEquals(expectedresult,c1.add(3,5));
      }
    }
```

In the preceding code snippet, the `testAdd()` method, which is a test method, has been created. The `testAdd()` method is a public method marked with the `@Test` annotation. It creates an instance, `c1`, of the `Calculator` class and initializes the variable, `expectedresult`, as `8`. It also uses the assert method, `assertEquals()`, which verifies whether the expected result matches the actual result. The following figure displays the test results of the preceding code snippet.



The test passed. (0.069 s)
CalculatorTest passed
testAdd passed (0.007 s)

The Test Results Window

In the preceding figure, the `testAdd()` method in the test class, has been assessed to check the functionality of its corresponding `add()` method in the `Calculator.java`. The test methods can pass or fail based on the accuracy of its functionality. The `testAdd()` has passed the test which is represented in a green color.

@Before Annotation

The `@Before` annotation causes the method, it has been attached to, run before the test method. The `@Before` annotation is ideally used when several tests require similar objects to be created before they are executed. The method annotated as `@Before` must be a public method and should not return any value. The syntax for the `@Before` annotation is:

```
@Before
public void setUp(){
//runs before every test method and initializes the object for every test
method
…
}
```

The following code snippet displays the JUnit `setUp()` method that runs before every test method to initialize the objects of the test:

```
@Before
public void setUp()
{
value1=3;
value2=5;
}
```

In the preceding code snippet, the `@Before` annotation is written before the `setUp()` method to execute this method before every test method in the test class. Here, it initializes the variable, `value1`, to `3` and the variable, `value2`, to `5`.

@After Annotation

The `@After` annotation causes the method, it has been attached to, run after the test method. All the methods annotated with `@After` will run even if a test method throws an exception. The method annotated as `@After` must be a public method and should not return any value. The syntax for the `@After` annotation is:

```
@After
public void tearDown(){
//runs after every test to perform tasks, such as releasing the memory
…
}
```

The following code snippet displays the JUnit `tearDown()` method that runs after every test method to reset the variable's values to 0:

```
@After
public void tearDown()
{
value1=0;
value2=0;
}
```

In the preceding code snippet, the `@After` annotation is written before the `tearDown()` method to execute this method after every test method in the test class. It resets the variables, `value1` and `value2`, to `0`.

@BeforeClass Annotation

The `@BeforeClass` annotation causes the method, it has been attached to, run once before any of the test methods in the class. The method annotated as `@BeforeClass` must have the following properties:

- It should be a public method.
- It should be declared static.
- It should not return any value.
- It should not contain any arguments.

The syntax for the `@BeforeClass` annotation is:

```
@BeforeClass
public static void setUpClass(){
//runs once before all the test cases
…
}
```

The following code snippet displays the `setUpClass()` method, which is executed once before all the test cases:

```
@BeforeClass
public static void setUpClass()
{
System.out.println("@BeforeClass - oneTimeSetUp");
}
```

In the preceding code snippet, the `@BeforeClass` annotation is written before the `setUpClass()` method. It executes the `setUpClass()` method once, before the execution of all the test cases. You can use this annotation to perform certain initializations, such as connecting to databases or opening a file, which need to be performed only once.

The annotations, `@BeforeClass` and `@Before`, are almost similar. However, the difference between the two annotations is that the method annotated with the `@BeforeClass` annotation will be called once per test class, whereas the method annotated with the `@Before` annotation will be called once per test case.

@AfterClass Annotation

The `@AfterClass` annotation causes the method, it has been attached to, run after all the methods in the class have been executed. All the methods annotated with the `@AfterClass` annotation will run even if the `@BeforeClass` annotated method throws an exception. The method annotated as `@AfterClass` must have the following properties:

- It should be a public method.
- It should be declared static.
- It should not return any value.
- It should not contain any arguments.

The syntax for the `@AfterClass` annotation is:

```
@AfterClass
public static void tearDownClass(){
//runs once after all the test cases
…
}
```

The following code snippet displays the `tearDownClass()` method, which is executed once after the end of all the test cases:

```
@AfterClass
public static void tearDownClass()
{
System.out.println("@AfterClass - oneTimeTearDown");
}
```

In the preceding code snippet, the `@AfterClass` annotation is written before the `tearDownClass()` method. It executes the `tearDownClass()` method once after all the test cases are executed. It can be used to perform one-time cleanup in the code, such as closing all the resources' connections or closing all the opened files.

Identifying JUnit Assert Statements

Once you have annotated a particular function as a test case using the `@Test` annotation, you need to write a code to test the functionality of a function. This code would require you to verify the output of the function being tested with the expected output. Consider a scenario of a simple Java application to calculate the square root of a number. While writing test cases for this application, you provide a set of input values on which testing needs to be performed. For the provided input values, you will be carrying a set of expected results against which the actual results will be tested. JUnit provides static methods in the `Assert` class to verify the expected and actual results. An assertion method compares the expected value with the actual value returned by the test. If the comparison fails, an exception, `AssertionException`, is thrown.

The following table displays the commonly used JUnit assert statements.

Assert Statement	***Description***
`assertTrue([message], boolean condition)`	*Checks that the specified boolean condition is true. The message parameter is optional.*
`assertFalse([message], boolean condition)`	*Checks that the specified boolean condition is false.*
`assertEquals([message], expected, actual)`	*Checks whether the expected and actual values are same.*

Assert Statement	***Description***
`assertEquals([message], expected, actual, tolerance)`	*Checks whether the expected and actual values are same for float or doubles values. The tolerance is the amount of difference that can be allowed in the expected and actual values for them to be considered equal. Therefore, the test will pass, if the condition, (difference between expected and actual) < tolerance, is true.*
`assertNull([message], object)`	*Checks if the specified object is null.*
`assertNotNull([message], object)`	*Checks if the specified object is not null.*
`assertSame([message], expected, actual)`	*Checks if the expected and actual variables refer to the same object.*
`assertNotSame([message], expected, actual)`	*Checks if the expected and actual variables refer to different objects.*

The Commonly Used JUnit Assert Statements

Let us see some examples to make you more familiar with the assert statements.

Consider the scenario of the application calculating the square root of a number, as shown in the following code snippet:

```
public class Arithmetic {
//method to find square root of a number
    public double findSquareroot(double num)
    {
      return Math.sqrt(num);
    }
}
```

In the preceding code snippet, the `findSquareroot()` method returns the square root of a number passed as an argument.

To test the preceding code snippet, you need to create a test class, which verifies the expected square root of a number to be equal to the actual value returned from the `findSquareroot()` method. The following code snippet displays the code for the test class:

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class ArithmeticTest {
    public ArithmeticTest() {
    }
    /**
     * Test of findSquareroot method, of class Arithmetic.
     */
    @Test
```

```
        public void testFindSquareroot() {
            Arithmetic instance = new Arithmetic();
            double expResult = 2.5;
            double result = instance.findSquareroot(6.25);
            assertEquals("findSquareroot",expResult, result, 0.0);
        }
    }
```

In the preceding code snippet, the `testFindSquareroot()` method is the test case created to test the functionality of the `findSquareroot()` method. It creates an object, `instance`, of the `Arithmetic` class. Next, the `findSquareroot()` method of the `Arithmetic` class is invoked by passing `6.25` as its argument. Finally, the `assertEquals()` method is used to check whether the actual result equals the expected result.

Consider another scenario of an application, which checks if a number is even or odd, as shown in the following code snippet:

```
    public class EvenOrOdd {
        public boolean isEvenNumber(int number){

            boolean result = false;
    //to check if the number is completely divisible by 2
    //and the remainder is 0
            if(number%2 == 0){
                result = true;
            }
            return result;
        }
    }
```

In the preceding code snippet, the `isEvenNumber()` method checks if the number passed to the method is even or odd and returns the Boolean value, `true` or `false`, accordingly.

To test the preceding code snippet, you need to create a test class, which verifies if the expected result is equal to the actual result when an even as well as an odd number is passed. The following code snippet can be used to test the `EvenOrOdd` class:

```
    import org.junit.After;
    import org.junit.AfterClass;
    import org.junit.Before;
    import org.junit.BeforeClass;
    import org.junit.Test;
    import static org.junit.Assert.*;
    public class EvenOrOddTest {

        public EvenOrOddTest() {
        }
         /**
         * Test of isEvenNumber method, of class EvenOrOdd.
         */
        @Test
        public void testIsEvenNumber() {
            EvenOrOdd e = new EvenOrOdd();
            assertTrue(e.isEvenNumber(4));
            assertFalse(e.isEvenNumber(3));
        }
    }
```

In the preceding code snippet, the `testIsEvenNumber()` method is the test method, which creates an instance, `e`, of the `EvenOrOdd` class. With this instance, the `isEvenNumber(4)` method is invoked by passing `4` as an argument. The `assertTrue()` method is a static method of the `Assert` class that checks whether the condition is `true`. The `assertFalse()` method is also a static method of the `Assert` class that checks whether the condition is `false`. Therefore, in this test case, when you pass an even number, the `isEvenNumber()` method returns `true` and vice-versa.

Consider a scenario of a student application that checks whether the details of the students have been entered.

The following code snippet can be used to create this application:

```
public class assertNullOrNotNull {
   String name=null;
   int marks=0;
   //check whether the name is null or not null

   public String verifyName(String fullName)
   {
       name=fullName;
       return name;
   }
   //check whether the marks is null or not null

   public int verifyMarks(int totalMarks)
   {
       marks=totalMarks;
       return marks;
   }
}
```

In the preceding code snippet, the variable, `name`, of the type, `String, has been initialized with null, which means it is currently not referring to any object in the memory`, and another variable, `marks`, of the type, `int`, has been initialized to `0`. Both the `verifyName()` and `verifyMarks()` methods are used to verify if the `name` variable refers to any object and if the `marks` variable stores any value, respectively.

The following code snippet can be used to test the `assertNullOrNotNull` class:

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class assertNullOrNotNullTest {
    public assertNullOrNotNullTest() {
    }
     /**
     * Testing the verifyName method
     */
    @Test
    public void testVerifyName() {
        assertNullOrNotNull instance = new assertNullOrNotNull();
        String result = instance.verifyName(null);
        assertNull(result);
    }

    /**
     * Testing the verifyMarks method
     */
    @Test
    public void testVerifyMarks() {
        assertNullOrNotNull instance = new assertNullOrNotNull();
        int result = instance.verifyMarks(87);
        assertNotNull(result);
    }
}
```

In the preceding code snippet, the `result` variable in the `testVerifyName()` method stores the value, `null`, as the `verifyName()` method is invoked by passing `null` as an argument. Then, the `assertNull()` method is invoked by passing `result` as an argument. As the `result` variable contains `null`, the test passes. Similarly, the `result` variable in the `testVerifyMarks()` method stores the value, `87`, which is not null. Therefore, the `assertNotNull(result)` statement asserts that the `result` variable is not null and the test passes.

Consider a scenario of an employee application that can be used to enter the relevant data of an employee and store it in a respective object. This object, passed as a parameter in a method, has a functionality to save the data in the database. The application verifies if the data returned from the database is same as the given details of an employee. The following code snippet can be used to create this application:

```
    public class Employee {
        int empid;
        String ename;
        double salary;
        public Employee enterData(Employee e)
        {
            this.empid=e.empid;
            this.ename=e.ename;
            this.salary=e.salary;
            return e;
        }
        public void display()
        {
            System.out.println(this.empid);
             System.out.println(this.ename);
              System.out.println(this.salary);
        }
    }
```

In the preceding code snippet, the employee details have been passed in the employee object, `e`, which holds the employee details, such as employee id, employee name, and employee salary. This information, which is stored in the database, is retrieved. This is to verify whether the information retrieved from the database is same as information being provided as a sample value.

To test the preceding code snippet, you can create a test class, as shown in the following code:

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class EmployeeTest {
    Employee e = new Employee();

    public EmployeeTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    @Before
    public void setUp() {
        e.empid=21341;
        e.ename="Goerge Simon";
        e.salary=1000000;
    }

    @After
    public void tearDown() {
    }
    @Test
    public void testEnterData() {
        System.out.println("enterData");
        Employee instance=new Employee();
        Employee result = instance.enterData(e);
        assertSame(e, result);
        assertNotSame(instance,result);

    }
}
```

In the preceding code, the `testEnterData()` method is created. It is a test method that comprises two assert methods. The `assertSame()` method is a static method of the Assert class that will pass if the object, `result`, points to the same location where the object, `e`, is stored. The `assertNotSame()` method is also a static method of the Assert class that will pass if the object, `instance`, points to some another location, which is different from the location of the object, `result`.

Just a minute:

Which one of the following annotations informs JUnit that the method, it has been attached to, needs to be run as a test case?

1. `@Test`
2. `@Before`
3. `@After`
4. `@BeforeClass`

Answer:

1. `@Test`

Activity 2.1: Executing JUnit Test on Command Prompt

Activity 2.2: Writing Test Cases in JUnit

Adding Behaviors to Test Cases

Consider a scenario where you need to create test cases for the calculator application that performs basic calculations, such as addition, subtraction, multiplication, and division. While testing the methods of this application, you might encounter situations where some of the methods throw an exception, such as divide by zero exception. Therefore, you want your test cases to handle those exceptions.

In addition, some of the methods might not be fully functional and you want the test cases associated with those methods to be ignored while testing. Suppose you want to add an additional function to the application that calculates the percentage. The method to implement this functionality is not yet ready, and you want the test cases associated with this method to be ignored.

Moreover, in order to check how optimized the code is, you want to specify a time under which the test case should successfully complete. Suppose you want the application to calculate and return the result within a second. However, there might be situations where the user provides large input values and the method takes longer time than usual to execute. In such cases, you want your test cases to check whether the method is executed within the specified time.

To add such behaviors into your test cases, JUnit provides you with various annotations and attributes.

Handling Exceptions

Consider the scenario where you have been asked to create test cases for the calculator application. However, the divide method of this application might throw the divide by zero exception. Therefore, you need to write a test case that handles the divide by zero exception. If the exception is caught, the test passes. Otherwise, it fails. To implement such functionality in your test case, JUnit provides you with the `expected` attribute inside the `@Test` annotation. This attribute specifies that when the test method is executed, the exception of the specified type is expected to be thrown by the method being tested.

The syntax for handling exceptions is:

```
@Test(expected=Exception.class)
public void testmethod ()
{
........................
........................
}
```

In the preceding syntax, `Exception` is the name of the exception class to be handled by the test method.

The following table shows some of the important exception classes that handle the exceptions.

Exception Class	***Description***
`IllegalArgumentException`	*It is used for handling illegal or inappropriate arguments.*
`AssertionError`	*It is used for handling assertion errors, such as user passing wrong inputs that produce some unexpected results.*
`RuntimeException`	*It is used to handle exceptions that occur during the execution of a program.*
`StackOverflowError`	*It is used for handling the stack overflow error that occurs when a recursive function is invoked n number of times such that the stack, storing the values of the function, overflows. It is equivalent to method calling itself without an exit condition.*
`ArithmeticException`	*It is used to handle arithmetic errors, such as divide by zero exception.*

The Exception Classes

Consider a scenario of a method that accepts the length and breadth of a rectangle and calculates its area. The length and breadth of a rectangle can never hold negative values. Therefore during testing, you need to ensure that user never passes a negative input value. Such exceptions, where user passes invalid input values and derives unexpected results, can be handled using the `AssertionError` class.

Consider the following code snippet that shows the `divide()` method of the calculator application:

```
public class Calculator {
    public int divide(int firstNumber,int secondNumber)
    {
        int result=firstNumber/secondNumber;
        return result;
    }
}
```

In the preceding code snippet, the `divide()` method has been created to divide the variable, `firstNumber`, with the variable, `secondNumber`, and save the result in the variable, `result`.

To test the preceding code snippet for the `Calculator.java` application, you need to create a test case that handles the exception. The following code snippet has a test case, with the `expected` attribute, to handle the exception:

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculatorTest {

    public CalculatorTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    @Before
    public void setUp() {
    }

    @After
    public void tearDown() {
    }

    /**
     * Test of divide method, of class Calculator.
     */
    @Test(expected=ArithmeticException.class)
    public void testDivide() {
       Calculator instance = new Calculator();
       instance.divide(4, 0);
    }
}
```

In the preceding code snippet, the `testDivide()` method is a test method that handles the arithmetic exception.

Practice Questions

1. Which one of the following annotations causes the method, it has been attached to, run before the test method?
   a. `@Before`
   b. `@BeforeClass`
   c. `@After`
   d. `@AfterClass`

2. Which JUnit assert statement checks whether the expected and actual values are same?
   a. `assertTrue()`
   b. `assertFalse()`
   c. `assertEquals()`
   d. `assertEqual()`

3. Which one of the following annotations is used with the `expected` attribute?
   a. `@AfterClass`
   b. `@Test`
   c. `@BeforeClass`
   d. `@Before`

4. Which one of the following exception classes is used to handle exceptions that occur during the execution of the program?
   a. `IllegalArgumentException`
   b. `AssertionError`
   c. `RuntimeException`
   d. `StackOverflowError`

Summary

In this chapter, you learned that:

- A test case carries a set of instructions based on which you can test the functional aspect of the code being tested.
- The `@Test` annotation informs JUnit that the method, it has been attached to, needs to be run as a test case.
- The `@Before` annotation causes the method, it has been attached to, run before the test method.
- The `@After` annotation causes the method, it has been attached to, run after the test method.
- The `@BeforeClass` annotation causes the method, it has been attached to, run once before any of the test methods in the class.
- The `@AfterClass` annotation causes the method, it has been attached to, run after all the methods in the class have been executed.
- JUnit provides static methods in the `Assert` class to verify the expected and actual results.
- An assertion method compares the expected value with the actual value returned by the test.
- The `expected` attribute specifies that when the test method is executed, the exception of the specified type is expected to be thrown by the method being tested.

Structuring Test Cases

CHAPTER 3

When a large Java application is developed, a number of developers work on multiple Java classes of the application. A particular class, along with its methods, needs to be tested with all possible input values to verify its stability and accuracy. However, writing test cases for each input value is not efficient as it consumes a lot of time. Instead, it would be convenient to run the same test case, defined for a particular method, with a different set of values. For this, you can create parameterized tests.

In addition, your test cases should be easily understood by any developer and generate readable error messages.

This chapter discusses how to effectively write and test multiple test classes in an application.

Objectives

In this chapter, you will learn to:

- Test multiple tests in Junit

Testing Multiple Tests in JUnit

While writing test cases for a method, you may realize that you have created a number of unit test cases with similar functionality but different input and expected values. For example, consider a scenario of a basic calculator application. You need to create unit test cases for different methods, such as `add()` and `subtract()`, of this application. While writing test cases for the `add()` method, you realize that you have been writing similar test cases with different input values, such as `add(5,8)`, `add(5,7)`, and `add(2,8)`. This strategy not only makes the code lengthy but also reduces the readability for others who try to analyze the code. Therefore, you want to create a single test case that allows you to pass a different set of values as parameters. To implement this functionality, JUnit provides you with parameterized tests that allow you to run a test case with multiple sets of input and expected values.

In addition, you have added some additional sets of functionality, such as reversing a number and finding the factorial of a number, to the calculator. The code for the additional functionality has been written in a separate Java class file. After testing this code, you want to combine all the test cases from different test classes and run them together. This will help you to test the functionality of the application as a whole. JUnit allows you to create test suites that let you group multiple unit test cases from different classes and run them together.

Creating Parameterized Tests

Consider the scenario of the basic calculator application. You need to write test cases for the `add()`, `subtract()`, `multiply()`, and `divide()` methods. The following code snippet displays the test cases for the `add()` method:

```
@Test
    public void testAdd() {
        System.out.println("add");
        int a = 3;
        int b = 7;
        Calculator instance = new Calculator();
        int expResult = 10;
        int result = instance.add(a, b);
        assertEquals(expResult, result);

    }
    @Test
    public void testAdd2() {
        System.out.println("add");
        int a = 5;
        int b = 6;
        Calculator instance = new Calculator();
        int expResult = 11;
        int result = instance.add(a, b);
        assertEquals(expResult, result);
    }
    @Test
    public void testAdd3() {
        System.out.println("add");
        int a = 7;
        int b = 2;
        Calculator instance = new Calculator();
        int expResult = 9;
        int result = instance.add(a, b);
        assertEquals(expResult, result);
    }
```

In the preceding code snippet, the same set of statements is being executed in every test case with a different set of input and expected values. As a result, the test code appears to be very long. To abbreviate the test code, JUnit allows you to create parameterized tests that let users pass and run multiple sets of input and expected values as parameters inside a single test case. To create a parameterized test, you need to:

1. Create a public static method annotated with the `@Parameters` annotation that returns a collection of objects (as array) as a test data set.
2. Annotate the test class with the `@RunWith(Parameterized.class)` annotation.
3. Create a public constructor that takes in data equivalent to one row of test data.
4. Create an instance variable for each column of test data.
5. Create the test cases using instance variables.

To use the `@RunWith(Parameterized.class)` annotation in your test class, you need to use the following import statements:

```
    import org.junit.runner.RunWith;
    import org.junit.runners.Parameterized;
```

To use the `@Parameters` annotation in your test class, you need to use the following import statement:

```
    import org.junit.runners.Parameterized.Parameters;
```

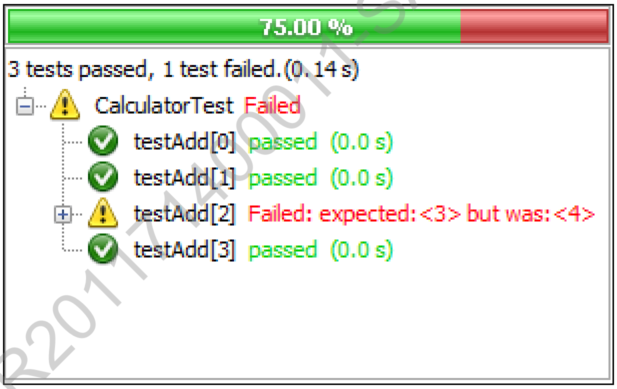
The constructor of the test class is invoked for each set of parameterized test data. Therefore, the number of values in each set of parameterized test data should be equal to the number of parameters passed to the constructor of the test class. For example, consider the following code snippet of the calculator application:

```
    public class Calculator {
        public int add(int firstNumber,int secondNumber)
        {
            return firstNumber+secondNumber;
        }
        public int subtract(int firstNumber,int secondNumber)
        {
            return firstNumber-secondNumber;
        }
        public int multiply(int firstNumber,int secondNumber)
        {
            return firstNumber*secondNumber;
        }
        public int divide(int firstNumber,int secondNumber)
        {
            return firstNumber/secondNumber;
        }
    public void percentage(){
    }
    }
```

The following code snippet creates the parameterized test for the `add()` method of the calculator application:

```
    import java.util.Arrays;
    import java.util.Collection;
    import org.junit.After;
    import org.junit.AfterClass;
    import org.junit.Assert;
    import org.junit.Before;
    import org.junit.BeforeClass;
    import org.junit.Test;
    import static org.junit.Assert.*;
    import org.junit.Ignore;
    import org.junit.runner.RunWith;
    import org.junit.runners.Parameterized;
    import org.junit.runners.Parameterized.Parameters;

    @RunWith(Parameterized.class)
    public class CalculatorTest {
        private int a;
        private int b;
        private int expected;

        public CalculatorTest(int a,int b,int expected) {
            this.a=a;
            this.b=b;
            this.expected=expected;

        }

        @BeforeClass
        public static void setUpClass() {
        }

        @AfterClass
        public static void tearDownClass() {
        }

        @Before
        public void setUp() {
        }

        @After
        public void tearDown() {
        }
        @Test
        public void testAdd() {
            Calculator instance = new Calculator();
                int actual=instance.add(a, b);
                Assert.assertEquals(expected, actual);
        }
     @Parameters
        public static Collection<Integer[]> getParameters(){
        Integer [][]inputArr;
        inputArr=new Integer[][]{{1,2,3},{-1,2,1},{2,2,3},{4,0,4}};
        return Arrays.asList(inputArr);
        }
    }
```

In the preceding code snippet, the `@RunWith(Parameterized.class)` annotation declares a class, `CalculatorTest.java`, as a test class having parameterized test cases. This class looks for a static method annotated with `@Parameters`. In such a case, the `getParameters()` method is the parameterized method, which takes a collection of integer values. For each of the integer values in the collection, the constructor of the test class is invoked. The first and second parameters of the constructor represent the integer values to be added, and the third parameter is the expected sum value of these two integer values. Therefore, the parameterized values are passed in that sequence only. The execution of the preceding test generates the output, as shown in the following figure.


```
75.00 %
3 tests passed, 1 test failed.(0.14 s)
CalculatorTest Failed
    testAdd[0] passed (0.0 s)
    testAdd[1] passed (0.0 s)
    testAdd[2] Failed: expected:<3> but was:<4>
    testAdd[3] passed (0.0 s)
```

The Test Result Using the @Parameters Annotation

In the preceding figure, of the given four tests, three tests have passed.

Implementing Test Suites

Consider the scenario of the basic calculator application. Now, you need to add some additional functionality to the calculator, such as reversing a number and finding the factorial of a number. You write the code for the preceding functionality in separate Java class files.

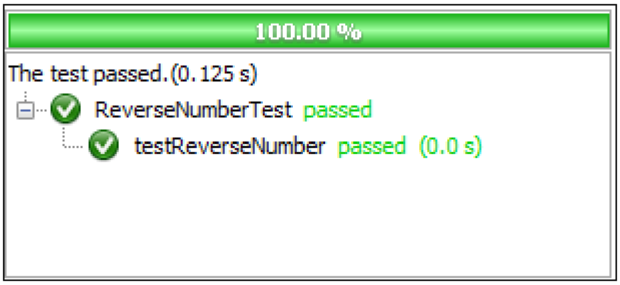
For example, you can write the code for reversing a number in a Java file named `ReverseNumber`, as shown in the following code snippet:

```
public class ReverseNumber {
    public int reverseNumber(int number){
        int reverse = 0;
        while(number != 0){
            reverse = (reverse*10)+(number%10);
            number = number/10;
        }
        return reverse;
    }
}
```

To test the `ReverseNumber.java` file, you need to create a test file, as shown in the following code snippet:

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class ReverseNumberTest {

    public ReverseNumberTest() {
    }
    @BeforeClass
    public static void setUpClass() {
    }
    @AfterClass
    public static void tearDownClass() {
    }
    @Before
    public void setUp() {
    }
    @After
    public void tearDown() {
    }

    @Test
    public void testReverseNumber() {
        ReverseNumber instance = new ReverseNumber();
        int expResult = 86871;
        int result = instance.reverseNumber(17868);
        assertEquals(expResult, result);
    }
}
```

In the preceding code snippet, the `result` variable stores the value, `86871`, which is same as the value stored in the `expResult` variable. Therefore, the test passes, as shown in the following figure.



```
100.00 %
The test passed.(0.125 s)
  ReverseNumberTest passed
    testReverseNumber passed (0.0 s)
```

The Test Result for the ReverseNumber Class

Further, consider the factorial functionality that you want to add to the calculator application. The following code snippet can be used to implement the factorial functionality:

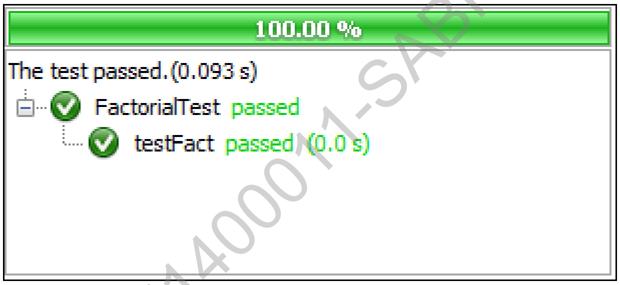
```
public class Factorial {
    public int fact(int n)
   {
        int output;
        if(n==1){
          return 1;
        }
        //Recursion: Function calling itself!!
        output = fact(n-1)* n;
        return output;
   }
}
```

In the preceding code snippet, a Java file named `Factorial.java` is created for calculating the factorial of a number. To test the preceding class, you need to create a test class, as shown in the following code snippet:

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class FactorialTest {

    public FactorialTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }
    @AfterClass
    public static void tearDownClass() {
    }
```

```
        @Before
        public void setUp() {
        }
        @After
        public void tearDown() {
        }

        @Test
        public void testFact() {
            int num = 6;
            Factorial instance = new Factorial();
            int expResult = 720;
            int result = instance.fact(num);
            assertEquals(expResult, result);
        }
    }
```

In the preceding code snippet, the factorial of the number, `6`, is calculated and stored in the variable, `result`. The `result` variable stores the value, `720`, which is same as the value stored in the `expResult` variable. Therefore, the test passes, as shown in the following figure.



```
100.00 %
The test passed.(0.093 s)
FactorialTest passed
    testFact passed (0.0 s)
```

The Test Result for the Factorial Class

Once the individual testing has been performed for the `Factorial` and `ReverseNumber` classes, a final testing that executes all the test classes together needs to be performed. This will help you verify the overall functionality and determine the stability of the application. JUnit provides you with test suites that let you combine test cases from different test classes and run them together.

To create a test suite, you need to annotate the test class with `@RunWith(Suite.class)`. In addition, you need to specify the reference of the test classes that you want to be a part of the test suite by using the `@Suite.SuiteClasses` annotation. The syntax for using this annotation is:

```
    @Suite.SuiteClasses({TestClass1.class, TestClass2.class, ........})
```

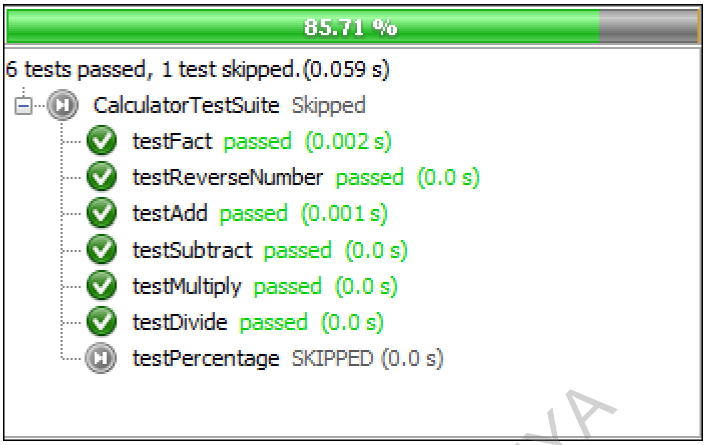
In the preceding syntax, `TestClass1` and `TestClass2` represent the name of the test classes that need to be a part of the test suite.

For example, you can create a test suite for the calculator application, as shown in the following code snippet:

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({FactorialTest.class, ReverseNumberTest.class,
CalculatorTest.class})
public class CalculatorTestSuite {

    @BeforeClass
    public static void setUpClass() throws Exception {
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
    }

    @Before
    public void setUp() throws Exception {
    }

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

In the preceding code snippet, the `CalculatorTestSuite.java` class is annotated with `@RunWith(Suite.class)` for creating the JUnit test suite. This class is also annotated with `@Suite.SuiteClasses({…,…,…})` to specify three classes, `FactorialTest.class`, `ReverseNumberTest.class`, and `CalculatorTest.class`, which are part of the test suite.

On executing the preceding test suite, all the test cases within the specified test classes are executed one after the other, as shown in the following figure.


```
85.71 %
6 tests passed, 1 test skipped.(0.059 s)
CalculatorTestSuite Skipped
    testFact passed (0.002 s)
    testReverseNumber passed (0.0 s)
    testAdd passed (0.001 s)
    testSubtract passed (0.0 s)
    testMultiply passed (0.0 s)
    testDivide passed (0.0 s)
    testPercentage SKIPPED (0.0 s)
```

The Test Result for the CalculatorTestSuite Class

In the preceding figure, all the six test cases have passed successfully, except the testPercentage() method. This method was skipped to be executed later. Hence, the success of the test cases is represented with 85.71% green signal.

Just a minute:

Which one of the following options allows users pass and run multiple sets of input and expected values as parameters inside a single test case?

1. *Parameterized tests*
2. *@Parameters annotation*
3. *@RunWith(Suite.class) annotation*
4. *@RunWith(Parameterized.class) annotation*

Answer:

1. *Parameterized tests*

Activity 3.1: Creating Parameterized Tests

Practice Questions

1. Which one of the following annotations you need to annotate the test class with to create a test suite?
   a. `@RunWith()`
   b. `@RunWith(Parameterized.class)`
   c. `@RunWith(Suite.class)`
   d. `@Suite.SuiteClasses()`

2. Which one of the following annotations you need to specify the reference of the test classes that you want to be a part of the test suite?
   a. `@RunWith(Suite.class)`
   b. `@RunWith(Parameterized.class)`
   c. `@Suite.SuiteClasses()`
   d. `@Suite.SuiteClass()`

Summary

In this chapter, you learned that:

- JUnit provides you with parameterized tests that allow you to run a test case with multiple sets of input and expected values.
- JUnit allows you to create test suites that let you group multiple unit test cases from different classes and run them together.
- To create a parameterized test, you need to:
  a. Create a public static method annotated with the `@Parameters` annotation that returns a collection of objects (as array) as a test data set.
  b. Annotate the test class with the `@RunWith(Parameterized.class)` annotation.
  c. Create a public constructor that takes in data equivalent to one row of test data.
  d. Create an instance variable for each column of test data.
  e. Create the test cases using instance variables.
- To create a test suite, you need to annotate the test class with `@RunWith(Suite.class)`. In addition, you need to specify the reference of the test classes that you want to be a part of the test suite by using the `@Suite.SuiteClasses` annotation.

Glossary

A

Acceptance testing

It refers to the testing done at the customer's end for accepting the software.

Annotations

These annotations make your test code readable and easier to understand.

B

Black box testing

This testing strategy tests the software for its overall functionality and does not test each statement and branches given in the code.

Boundary condition

A boundary condition verifies if the parameters passed are in the correct order, or, within the specified range, and in compliance with the expected value's format.

C

Code coverage analysis

It is a measure of the quality of tests and enables you to track the portions of the code that are not evaluated by the test cases.

Components

The functions, statements, conditional constructs, and loop constructs, which are part of the application, are known as its components.

Condition coverage

It analyzes each condition in a conditional construct and returns the outcome of each condition, which can be true or false.

Continuous integration testing

An approach where modules of an application are integrated with each other at intervals to verify the overall functionality and stability of an application is known as continuous integration testing.

D

Data-driven testing

Data-driven testing is a technique where input and expected values can be passed as separate data files.

E

EclEmma

It is an open-source plugin available within Eclipse for analyzing code coverage of Java applications designed using Eclipse.

F

Function coverage

It analyzes whether test cases invoke each function in the source code.

H

Hamcrest

Hamcrest framework is used for writing customized assertion matchers that allow match rules to be defined declaratively.

I

Integrated development environment

It automatically sets up the desired environment and download the necessary jar files required to execute unit test code.

Integration testing

It tests different modules of the given application while integration.

J

JaCoCo

The JaCoCo tool enables you to generate reports that help you analyze the percentage of the code covered during testing.

JMeter

It is a tool used to test the functional behavior and measure the performance of Web applications under different workloads.

JUnit

It is the open-source testing framework provided for Java programming language.

L

Loop Coverage

It helps you analyze whether a loop was executed and the number of times it was executed.

M

Mocking

The process of creating mock objects to verify the functionality of the methods is known as mocking.

P

Performance testing

It is to test the entire non-functional performance expected of the system.

R

Regression testing

It refers to testing an application after it has been modified.

S

Statement coverage

It checks the execution of various statements in a program and ensures that every statement in the code is tested.

Stubs

Stubs are a mechanism used for faking the behavior of real code that may not be available in the application yet.

System testing

System testing tests the reliability of the overall system.

T

Test

It is a collection of one or more test cases or test suites that are executed to perform the testing of an application.

Test-driven development

The approach, where test cases are written before the code is developed, is known as Test-driven development.

Test case

This is the smallest unit of any JUnit test that verifies the functionality of a method being tested.

Test plan

It defines a series of tasks that will be performed to determine the performance of a Web application.

Test result

It refers to the outcome of a test, which is collected from the test class.

Test runner

When you execute a test, the test runner is executed in the background, which displays the result of the test in terms of pass or fail.

Test suite

A test suite is a class that comprises different test classes that are executed one after the other automatically.

Transaction

Transaction is a sequence of operations performed to accomplish a task.

U

Unit testing

The testing of small pieces of code, known as units, is known as unit testing.

W

White box testing

In white box testing strategy, you need to understand the code and then test every statement and condition given in the code.

Objectives Attainment Feedback

Test Driven Development in Java

Name: ____________ Batch: ____________ Date: ____________

The objectives of this course are listed below. Please tick whether the objectives were achieved by you or not. Calculate the percentage at the end, fill in your name and batch details, and return the form to your coordinator.

S. No.	***Objectives***	***Yes***	***No***
1.	Identify software testing	☐	☐
2.	Identify unit testing	☐	☐
3.	Write test cases in JUnit	☐	☐
4.	Add behaviors to test cases	☐	☐
5.	Test multiple tests in JUnit	☐	☐

Percentage: (# of Yes/5) * 100 ____________