Onlinevarsity
your e-way to learning

Testing Android Applications Learner's Guide

Are you □□□□□□□□□□ with Onlinevarsity.com?	R G E T R E E S I D
Did you □□□□□□□□ this ebook?	D O O W L N A D
Do you have a □□□□□ copy of this ebook?	L G A E L
Are you a victim of □□□□□□?	P C I A R Y

Answers: REGISTERED, DOWNLOAD, LEGAL, PIRACY

Download a **LEGAL** copy of this ebook which you can say is **mine** because **PIRACY** is a **crime**

Testing Android Applications Learner's Guide

APTECH LIMITED
Contact E-mail: ov-support@onlinevarsity.com

First Edition - 2013

Dear Learner,

We congratulate you on your decision to pursue an Aptech course.

Aptech Ltd. designs its courses using a sound instructional design model – from conceptualization to execution, incorporating the following key aspects:

- Scanning the user system and needs assessment

Needs assessment is carried out to find the educational and training needs of the learner

Technology trends are regularly scanned and tracked by core teams at Aptech Ltd. TAG* analyzes these on a monthly basis to understand the emerging technology training needs for the Industry.

An annual Industry Recruitment Profile Survey# is conducted during August - October to understand the technologies that Industries would be adapting in the next 2 to 3 years. An analysis of these trends & recruitment needs is then carried out to understand the skill requirements for different roles & career opportunities.

The skill requirements are then mapped with the learner profile (user system) to derive the Learning objectives for the different roles.

- Needs analysis and design of curriculum

The Learning objectives are then analyzed and translated into learning tasks. Each learning task or activity is analyzed in terms of knowledge, skills and attitudes that are required to perform that task. Teachers and domain experts do this jointly. These are then grouped in clusters to form the subjects to be covered by the curriculum.

In addition, the society, the teachers, and the industry expect certain knowledge and skills that are related to abilities such as *learning-to-learn, thinking, adaptability, problem solving, positive attitude etc*. These competencies would cover both cognitive and affective domains.

A precedence diagram for the subjects is drawn where the prerequisites for each subject are graphically illustrated. The number of levels in this diagram is determined by the duration of the course in terms of number of semesters etc. Using the precedence diagram and the time duration for each subject, the curriculum is organized.

- Design & development of instructional materials

The content outlines are developed by including additional topics that are required for the completion of the domain and for the logical development of the competencies identified. Evaluation strategy and scheme is developed for the subject. The topics are arranged/organized in a meaningful sequence.

The detailed instructional material – Training aids, Learner material, reference material, project guidelines, etc.- are then developed. Rigorous quality checks are conducted at every stage.

- Strategies for delivery of instruction

Careful consideration is given for the integral development of abilities like thinking, problem solving, learning-to-learn etc. by selecting appropriate instructional strategies (training methodology), instructional activities and instructional materials.

The area of IT is fast changing and nebulous. Hence considerable flexibility is provided in the instructional process by specially including creative activities with group interaction between the students and the trainer. The positive aspects of web based learning –acquiring information, organizing information and acting on the basis of insufficient information are some of the aspects, which are incorporated, in the instructional process.

- Assessment of learning

The learning is assessed through different modes – tests, assignments & projects. The assessment system is designed to evaluate the level of knowledge & skills as defined by the learning objectives.

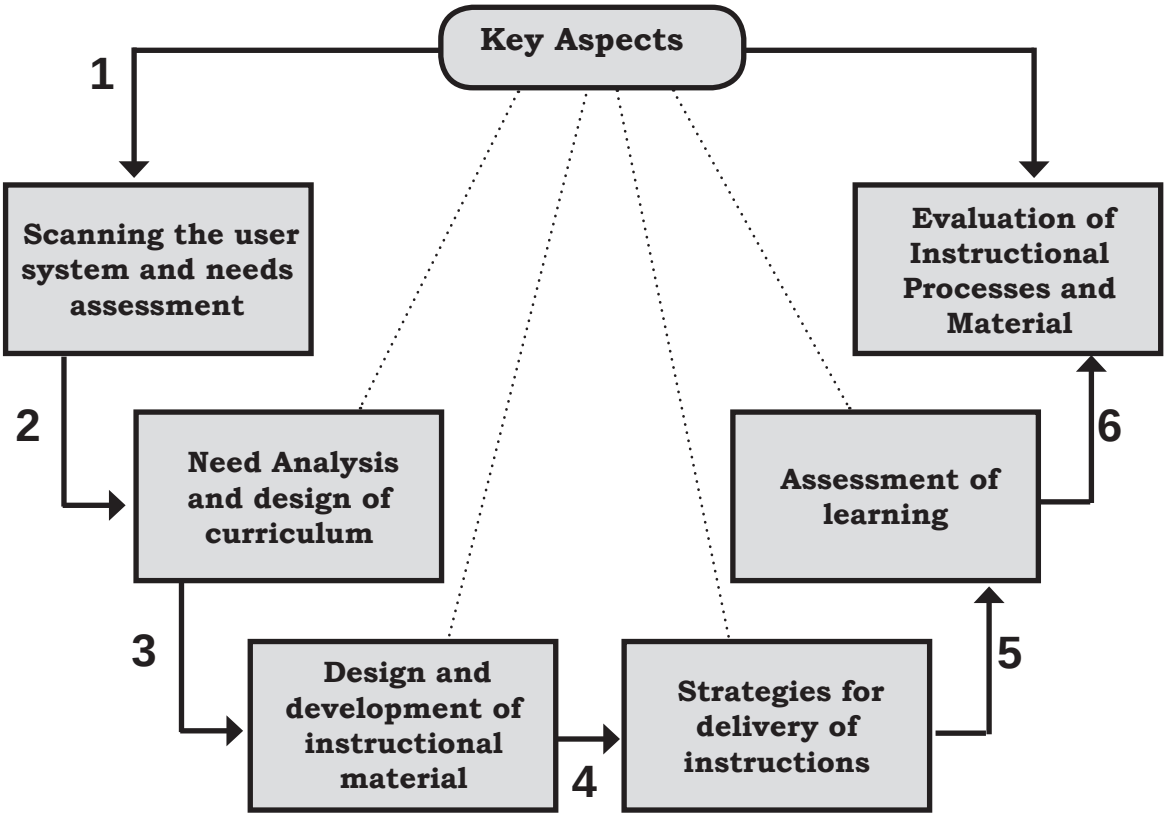
- Evaluation of instructional process and instructional materials

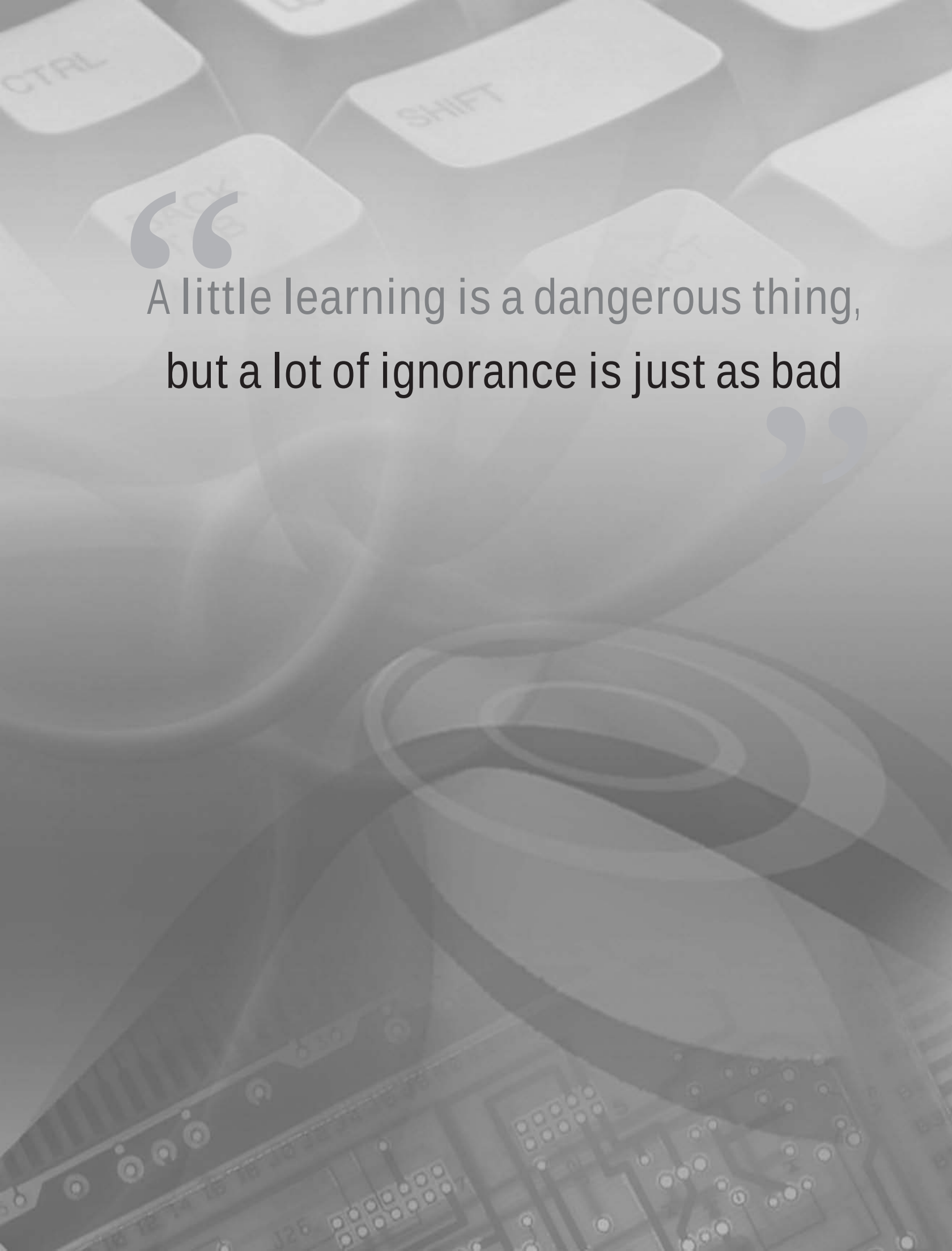
The instructional process is backed by an elaborate monitoring system to evaluate - on-time delivery, understanding of a subject module, ability of the instructor to impart learning. As an integral part of this process, we request you to kindly send us your feedback in the reply pre-paid form appended at the end of each module.

*TAG – Technology & Academics Group comprises of members from Aptech Ltd., professors from reputed Academic Institutions, Senior Managers from Industry, Technical gurus from Software Majors & representatives from regulatory organizations/forums.

Technology heads of Aptech Ltd. meet on a monthly basis to share and evaluate the technology trends. The group interfaces with the representatives of the TAG thrice a year to review and validate the technology and academic directions and endeavors of Aptech Ltd.

Aptech New Products Design Model





“A little learning is a dangerous thing,
but a lot of ignorance is just as bad”

Preface

The book, Testing Android Applications, aims to teach the basics of testing Android applications developed for Android enabled devices. The book introduces the developer to the most commonly available techniques, frameworks, and tools to improve the development of Android applications.

Android programming has undergone several reformations since its inception. This book intends to familiarize the reader with the techniques available for testing Android applications. The book begins with an introduction to the basic concepts of testing Android applications using Eclipse IDE. The book proceeds with the explanation of the various Test case classes that can be utilized for testing Android applications and also provides a clear, step-by-step instruction to show how to write tests for your application. This will enable the developer to create professional Android applications. The book concludes with an explanation of testing the different components in an Android application.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions.

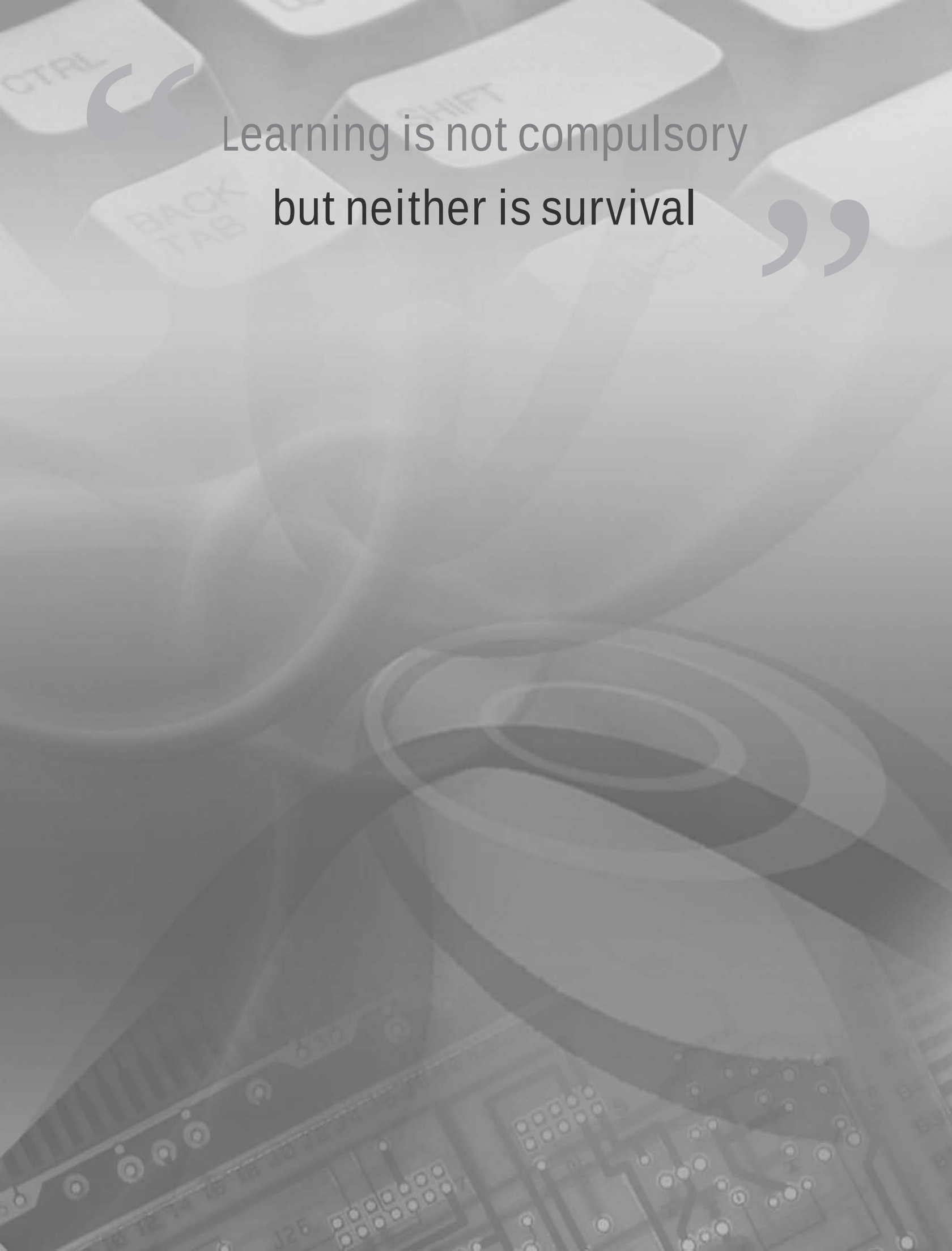
Design Team

“Nothing is a waste of time if you use the experience wisely”

Table of Contents

Module

1.	Introduction to Software Testing	1
2.	Android Testing Framework	17
3.	Components Used for Creating Tests	59
4.	Android Testing Environment	103
5.	Testing Android Projects	125
6.	Testing for Different Situations	169


“Learning is not compulsory
but neither is survival”

Session - 1

Introduction to Software Testing

Welcome to the session, **Introduction to Software Testing**.

This session explains the concept of software testing. It describes the purpose and need for testing. The session also introduces the advantages of testing. It lists the various types of testing along with their advantages and disadvantages.

In this session, you will learn to:

- Explain the concept of testing
- Explain the need and purpose for testing
- Describe the advantages of testing
- List the different types of testing

1.1 Introduction

Human beings commit errors in any process. Some of the errors can be ignored while others can break the entire system or software. Such errors should be handled in time before the software is deployed. No matter how much time is invested in design and development of software program, mistakes are inevitable and bugs will appear. Software application development and mistakes are closely related with each other.

The session explains software testing which is an activity aimed for evaluating the quality of the program and improvising it by identifying the defects and problems.

Finally, the session explains the different types of software testing.

1.2 Overview of Testing

Testing is one of the most important activities in verification and validation process. It is a process to identify the correctness, completeness, and quality of the developed software. Its goal is to create a defect free product for customers by verifying and validating whether the software is working according to the expected specifications. A simple definition of testing is that it is a process of executing a program with an intention of finding error/s. There are certain true facts about testing. They are as follows:

- ➔ The number of possible inputs is very large
- ➔ The number of possible outputs is very large
- ➔ The number of execution paths is very large

With such a large number of inputs, outputs, and execution paths, it can be stated that a good test is one that has a high probability of finding errors in the shortest time and with minimum effort. A successful test is one that can uncover the errors quickly and effortlessly.

1.2.1 What is Testing?

Software testing in simple words is a quality check process. It is a process of checking the correctness and completeness of a product before being released to the end users. The objectives of software testing are as follows:

- ➔ Meets the expected design and development specifications
- ➔ Functions as expected
- ➔ Uses same characteristics for implementation
- ➔ Conforms to the needs of end users

Software testing can be done at different points in the development process. However, the different software models determine when the testing method can be employed. While in traditional models, testing occurs after the requirements are defined and the coding process is complete, in newer models a test-driven method is executed. Here, the developer is responsible for a major portion of the testing before it passes to a team of testers. Agile software development is one such model.

There are some limitations to software testing. They are as follows:

- Testing cannot deliver an error-free product as its purpose is to detect errors.
- It cannot identify all errors within software. Its function is to compare the state and behavior of the product against certain components. The components can be standards, specifications, applicable laws, products that can be compared, previous versions of the same product, inferences about intended and expected product goal, user requirements, and so on.
- It can only identify defects under controlled situations. It cannot ascertain that a product can function properly under all conditions.

Software testing consists of examination and execution of code under various conditions as well as checking of the code against specifications. In the current software development scenario, a testing team may be separate from the development team. The skills required for a tester are varied but a major trait that they need to have is to ask questions regarding the functioning of the software and the possibility of defects due to its functioning.

1.2.2 Need for Testing

In February 2003, the US Treasury Department sent out 50,000 Social Security checks. The problem that they encountered was that they did not verify that the checks were mailed without a beneficiary's name. The error was attributed to a software maintenance defect. This is just one of the many examples that could have been avoided with software testing.

Therefore, testing helps to solve three main purposes namely verification, validation, and defect finding. Each of this process is essential to any software development process to enable them to produce a quality product.

- **Verification process:** In this process the software is tested against the given technical specifications. A 'specification' can be described as a measurable output value based on a specific input value under specification conditions. Software technical reviews represent one such method for verifying various products. A specification review will attempt to verify the specification requirement against the requirement description and the customer requirements specification. An example

of a simple specification is a SQL query to retrieve monthly data for a single account against a multi-month account-summary table. The objective is to return the result within 3 seconds of submitting the request.

➔ **Validation process:** This process is performed at the end of the development stage to ascertain that the software meets the business requirements. A business requirement may be to retrieve the income tax data from a monthly salary account-summary table. The objective is to return the result containing columns of data within 3 seconds of submitting the request.

➔ **Defect finding:** The purpose of this process is to check the actual results against the expected one. A defect determines the variance in the results. The defect can be backtracked to a cause in the specification, design, or development phases.

1.2.3 History of Testing

Software testing initially started in the fifties. Testing was primarily debugging, where the objective was to identify and eliminate bugs in the software. Slowly, it moved to destruction testing, where the purpose was to break down the code to identify the gaps present in them. In 1979, Glenford J. Myers introduced a distinction between debugging and testing. Later, the advent of prevention oriented methodologies, where the intention was to satisfy specifications and detect and prevent errors, originated.

1.2.4 Advantages of Testing

The advantages of software testing are as follows:

➔ **Time-saving:** In general, manual testing takes up more time both during software and application development. Such time can be considerably lessened with automated tools provided the scripts are standard and non-complex.

➔ **Reliability:** Automated tests remove the possibility of error, especially when the same sequence of steps need to be performed again and again.

➔ **Cost-saving:** Automated tests save money by identifying the defects at different stages of the testing process.

➔ **Development downtime:** A quick testing process avoids or reduces development time by reducing a computer's unavailability time to minimum.

➔ **Customer service:** Automated testing ensures that the customers' requirements are met by developing better applications.

- **Versions of the same software:** Testing identifies the areas for modifications and enhancements in later versions and catalogs reusable modules and components.

- **Training:** Another advantage of testing is identifying the training areas for programmers and developers.

1.3 Types of Testing

Software testing is of different types depending upon where they are introduced in the software development process. The different types of software testing are as follows:

1.3.1 Unit Testing

Unit testing of a software application is done by the developer during the coding process. The objective of unit testing is to isolate each part of the program and verify that each individual part is correct. It ensures that the codes used by the software application work independently. Unit testing may also be known as component testing. In other words, unit testing verifies that the lowest independent entity in a software application is working correctly.

The advantages of unit testing are as follows:

- **Facilitates change:** With unit testing it is possible to refractor code at a later date to ensure that the module continues to work correctly. Test cases are written for all functions and methods, so that whenever faults occurs it can be quickly identified and corrected.

- **Simplifies integration:** Testing parts of the program and then the sum of the parts makes integration testing easier. It may lessen uncertainty in the units themselves and use a bottom-up testing style approach.

- **Documentation:** Each unit test consists a documentation of its functionality and characteristics that define the appropriate/inappropriate use of the unit.

- **Design:** An unit-driven test approach can be viewed as a formal design of each unit that specifies the class, method, and observable behavior of that unit.

The drawbacks of unit testing are as follows:

- Unit testing cannot identify system-level errors or integration errors.

- It cannot show an error-free product but can only record the presence of defects.

- It is necessary to maintain records of the tests performed on each of the unit but also of all the modifications performed on the source code. Use of a version

Concepts

control system becomes essential to detect units that have failed the current version test though they have been cleared in the previous versions.

➔ It is also important to follow a sustainable process to address test case failures immediately. In the absence of such a process, the application may become out of sync with the unit test suite and record false positives, thereby reducing the effectiveness of the test.

1.3.2 Bottom-up Testing Strategy

The bottom-up testing strategy tests different subsystems in the hierarchy. In this testing strategy, the lowest layer of subsystems is tested individually. Following this, the next level of subsystems are tested that call for the previously tested subsystems. This is continued until all subsystems are included.

The advantages of using this strategy are as follows:

➔ **Stubs:** There is no need for stubs. A stub is a component that the TestUnit depends on for implementation, though partially and for fake values.

➔ **Other systems:** This strategy is useful for integration testing of others systems such as object-oriented systems, real-time systems, and systems with strict performance requirements.

The disadvantages of using this strategy are as follows:

➔ Drivers are required for this type of testing. A driver is a component that calls the TestUnits and controls the test cases.

➔ The User Interface (UI) is the most important subsystem that is tested last.

1.3.3 Top-down Testing Strategy

Contrary to the bottom-up testing strategy, the top-down testing strategy tests the top most layers or the controlling subsystem first. The remaining subsystems that are called by the tested subsystems are combined together and the resulting collection of subsystem is tested. This is continued until all subsystems are incorporated in the test. Stubs are required for this type of testing when the sub modules have not been developed. Stub is a temporary program.

The advantages of using this strategy are as follows:

➔ **Drivers:** Drivers are not required for this test.

➔ **Test cases:** These can be defined in terms of functional requirements.

The disadvantages of using this strategy are as follows:

- Stubs must define all possible conditions of testing so writing them may be difficult.
- In case the lowest level of the system has many methods, a large number of stubs maybe needed.
- It is not possible to test some interfaces separately.

1.3.4 Integration Testing

Integration testing, as the name suggests integrates the individual units (components) and tests it as a group. Its aim is to expose defects in the interaction between the components. Integration testing is sometimes, termed as I&T (Integration and Testing), String testing, or Thread testing. Integration testing is done after unit testing and is used for detecting the problems in the interface of the two units.

The advantages of using this strategy are as follows:

- **Time saving:** System testing becomes time consuming if integration testing is not performed.
- **Off-the-shelf components:** Many off-the shelf components cannot be unit tested.
- **Deduction of errors:** Many system failures are avoided by reduction of errors during the interaction of the subsystems.

The disadvantages of using this strategy are as follows:

- Failures that are not deducted by integration testing will be found after the system is deployed and will prove to be expensive.
- Conditions that are not defined in the integration tests will not be tested.
- Conditions that do not confirm to the execution of design items are not tested.

1.3.5 Functional/Acceptance Testing

Functional testing refers to testing the software application in accordance with the specified requirements. It focuses on the whole output and ignores the internal components. In other words, it ensures that the functionality as specified in the system requirement works.

Acceptance testing is a type of testing that is usually done at the client's end. The purpose is to check whether the output meets the specified requirements and works as expected. Acceptance testing are of different kinds but the typical types are as follows:

- **User acceptance testing:** This type of testing is based on the location of the test—factory or site. Factory acceptance testing is conducted by factory users before moving the product or application to its destination site. Site acceptance testing is done by users at the site.

- **Operational acceptance test (OAT):** This test is also referred to as the operational readiness testing. The test is done to check that the processes and procedures are in place for a system to be used and maintained. The checks include back-up facilities, disaster recovery procedures and end user training, maintenance, and security procedures.

- **Contract and regulation acceptance testing:** A system is considered to be accepted after it is tested against the acceptance specifications documented in a contract. In regulation acceptance testing the system is tested against government, legal, and safety standards.

- **Alpha and beta testing:** Alpha testing is testing of the software in a controlled setting at the developer's environment with the developer fixing errors immediately. Beta testing is testing done at the client's end in the absence of the developer. The software is tested in a realistic target environment and changes, if any are made based on the feedback before being released to the end user. This type of testing is also called 'field testing'.

The advantages of using this strategy are as follows:

- **Cost-saving:** Running tests manually is expensive compared to creating an automated test.

- **Customer support:** When a project passes acceptance test, future releases, and long-term contract support becomes worthwhile.

The disadvantages of using this strategy are as follows:

- There is a possibility of repeated execution changes from project to project and long-term support and maintenance issues.

- After a product is launched it needs to be maintained. Sometimes, the investment to set and maintain a test for the product life becomes expensive. It may, however be less costly to manually test the product when the need arises.

1.3.6 Performance Testing

This test assesses the speed and effectiveness of the software application as given in the performance requirements. It ensures that the results are generated within the specified time and determines the stability of the application under varying user loads.

Performance testing is done by generating some tasks or activity on the system and then testing by using some performance tools.

There are many types of performance testing. They are as follows:

- **Stress testing:** evaluates the stress limits of the system
- **Volume testing:** determines the working of the system when large data are fed
- **Configuration testing:** tests various software and hardware configurations
- **Compatibility testing:** estimates the compatibility with the existing systems
- **Timing testing:** determines the response and the performance time to complete a function
- **Security testing:** tests violation of security standards
- **Environmental testing:** tests tolerance to heat, humidity, and motion
- **Quality testing:** determines the reliability, maintainability, and availability
- **Recovery testing:** estimates the time taken to respond to error detection and data loss
- **Human factors testing:** tests done with end users
- **Load testing:** determines the application's response to various load levels within its acceptable limits. The main parameter here is the response time.
- **Soak/endurance testing:** measures the application's ability to handle continuous load for an extended period of time. The main parameter here is memory.

The advantages of using this strategy are as follows:

- **System performance:** Eliminates system failure, late deployment, and rework due to performance issues. Also, removes avoidable system tuning efforts.

Concepts

➔ **Overhead costs:** Avoids additional overhead operational costs for handling system issues due to performance failures.

The disadvantages of using this strategy are as follows:

➔ Testing is expensive when it is done at each phase of the SDLC.

➔ The whole system can be seen as useless or the defect removal cost higher if performance activities are done after complete development.

➔ It may also be difficult to rectify the performance defects after the development process.

1.3.7 System Testing

Testing of the complete and fully integrated software application with the full computer system is called system testing. It consists of a series of different tests to evaluate its compliance with specified requirements.

The major types of system testing are as follows:

➔ **Alpha testing:** Performed by the test team in a controlled setting in the developing environment.

➔ **Beta testing:** Conducted by a select group of users offering valuable feedback to the developer on the working of the system.

➔ **Acceptance testing:** Performed by the customer to check the acceptance of the system for delivery.

The advantages of using this strategy are as follows:

➔ **End-to-end testing:** Provides a complete testing where every single input and its outcome is verified.

➔ **User experience:** Tests the experience of the user in working with the application.

The disadvantages of using this strategy are as follows:

➔ This test requires detailed test cases and test suites to test each part of the application.

➔ The type of system test to take is dependent on various factors such as availability of time, resource, and budget, the tester's experience, and the company the tester works for. The method used by large companies is different than those used by small and medium sized companies.

1.4 Check Your Progress

1. The primary objective of prevention oriented testing methodologies is to:

(A)	satisfy specifications, detect, and prevent errors	(C)	break code and identify gaps
(B)	identify and eliminate bugs	(D)	identify gaps and satisfy specifications

2. The three main purposes of software testing are _____, _____, and _____.

(A)	verification, validation, and fault finding	(C)	verification, examination, and execution
(B)	verification, fault finding, and examination	(D)	validation, examination, and fault finding

3. Which of the following tests isolates and verifies the correctness of each individual part of the program?

(A)	Component testing	(C)	Functional testing
(B)	System testing	(D)	Integration testing

4. The goal of string testing is to:

(A)	integrate components and test it as a group	(C)	check against specifications
(B)	test each unit individually	(D)	focus on the whole application

5. Which one of this test is performed by the client?

(A)	integration testing	(C)	performance testing
(B)	system testing	(D)	acceptance testing

6. Which one of the following test uses a stub for testing subsystems?

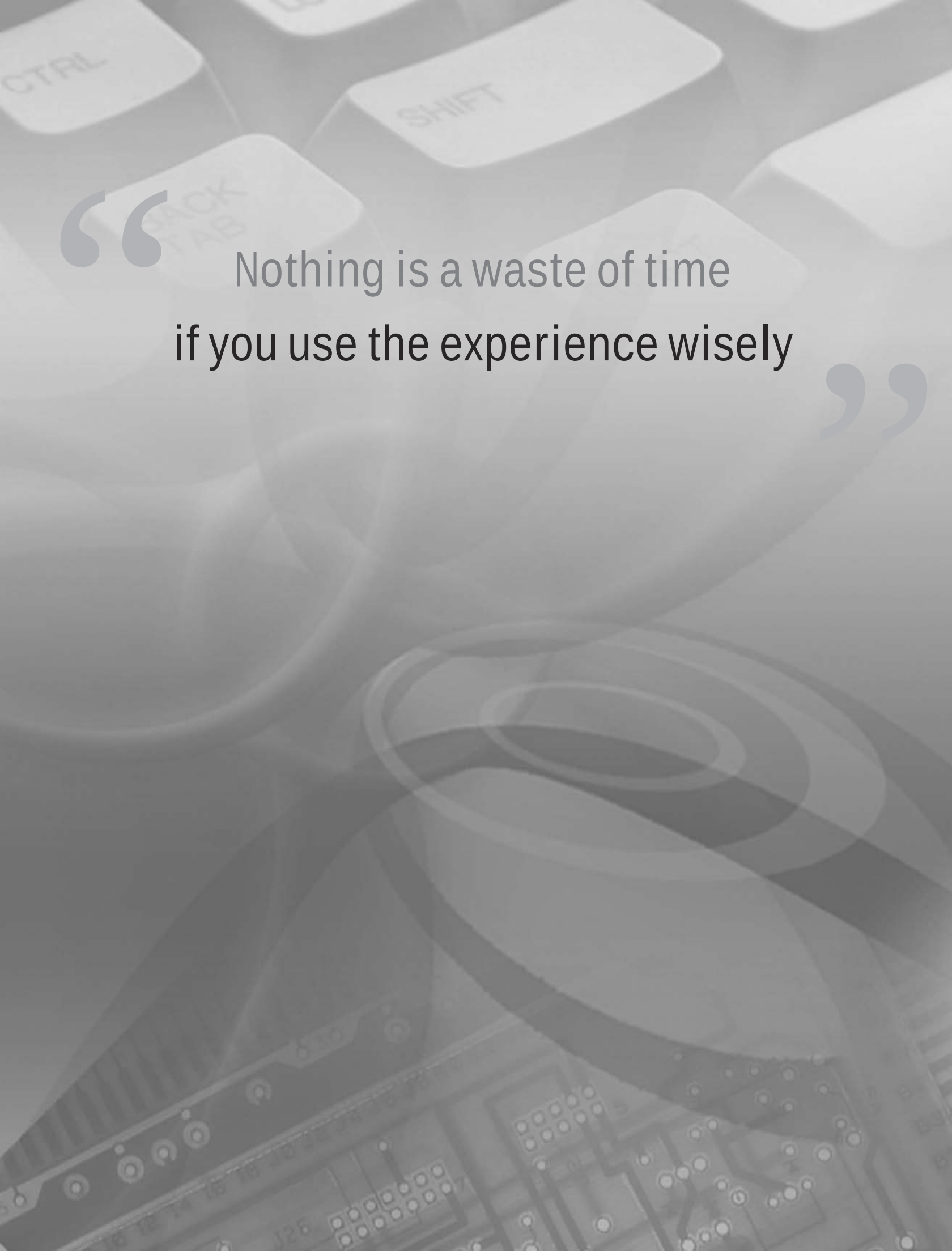
(A)	Top-down testing	(C)	Alpha testing
(B)	Bottom-up testing	(D)	Beta testing

1.4.1 Answers

1.	A
2.	A
3.	A
4.	A
5.	D
6.	A

Summary

- Software testing checks the correctness and completeness of the project before its release to end users.
- Unit testing tests each part of the program and verifies that each individual part is correct.
- The bottom-up testing strategy tests different subsystems in hierarchy.
- The top-down testing strategy tests the top most layers or the controlling subsystem first and then moves down to testing the remaining subsystems.
- Integration testing integrates the individual units (components) and tests it as a group.
- Functional testing checks the software application in accordance with the specified requirements.
- Acceptance testing checks whether the output meets the specified requirements and works as expected.
- Performance testing assesses the speed and effectiveness of the software application as given in the performance requirements.
- System testing checks the complete and fully integrated software application with the full computer system.



“Nothing is a waste of time
if you use the experience wisely”

Session - 2

Android Testing Framework

Welcome to the session, **Android Testing Framework**.

This session describes the test structure and the Android Testing API. The concept of `JUnit` and instrumentation will also be discussed.

In this session, you will learn to:

- Describe the Android test structure
- Explain the Android testing API
- Identify Android specific-testing classes

2.1 Introduction

The Android testing framework is an essential element of the development environment. It constitutes the basic architecture and tools that help in testing every aspect of an application at various levels.

Android test suites are based on JUnit. The Android testing framework has a set of control methods that call for lifecycle methods (`onCreate`, `onResume`, `onPause`, `etc.`) of components to show how the application will behave under certain conditions. The SDK also provides other test tools for stress testing of the application.

2.2 Test Structure in the Android Testing Framework

Android has a built-in testing framework that helps in testing every unit of an application. It is an advanced testing framework which extends the industry standard JUnit with specific features suitable for different types of testing. The test structure is a fundamental feature of the Android testing framework. Android build and test tools assuming that the test projects consist of an organized and standard structure of tests, test case classes, test packages, and test projects. Figure 2.1 displays the testing framework.

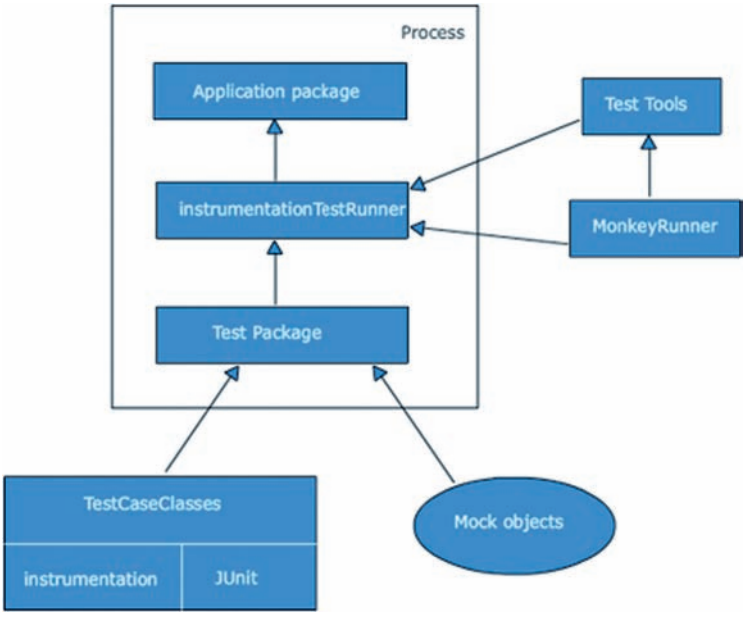


Figure 2.1: Android Testing Framework

Some of the key features of the testing framework are as follows:

- Android test suites are generally based on JUnit API. JUnit is an open source unit testing framework for Java programming language. A JUnit test is a set of Java classes used to test individual methods in a Java class. It is used to build test cases when using agile development process. For a beginner who is not familiar with Android testing, simple test case classes like `AndroidTestCase` can be used as the initial test case classes followed by more complex test classes.

- The function of Android JUnit extensions is to provide component-specific test case classes that help in creating mock objects and methods to control the lifecycle of a component.

- Also, there is no need to learn new set of tools or techniques to design or build tests. This is because test suites are similar to main application packages as they are contained in test packages.

- The SDK tools are a set of building and debugging tools. They are used with other IDEs for conducting tests in Eclipse with ADT plugin and for testing in command-line form. The function of these tools is to obtain information from the project of the application which is under test. Then, they use this information to generate the build files, manifest file, and directory structure for the test package.

- Another feature of SDK tools is to send pseudo-random events to a device with the use of `monkeyrunner` and UI/Application Exercise Monkey. While `monkeyrunner` is an API testing device used with Python programs, UI/Application Exercise Monkey is a command tool for stress-testing UIs.

  The following example demonstrates the use of JUnit in Eclipse. In this example a new project is created in eclipse which will multiply two numbers and will be tested with the help of JUnit. The steps are as follows:

  1. Click **File → New → Project** to display the **New Project** dialog box.
  2. Select **Android → Android Application Project** as shown in figure 2.2.

Concepts

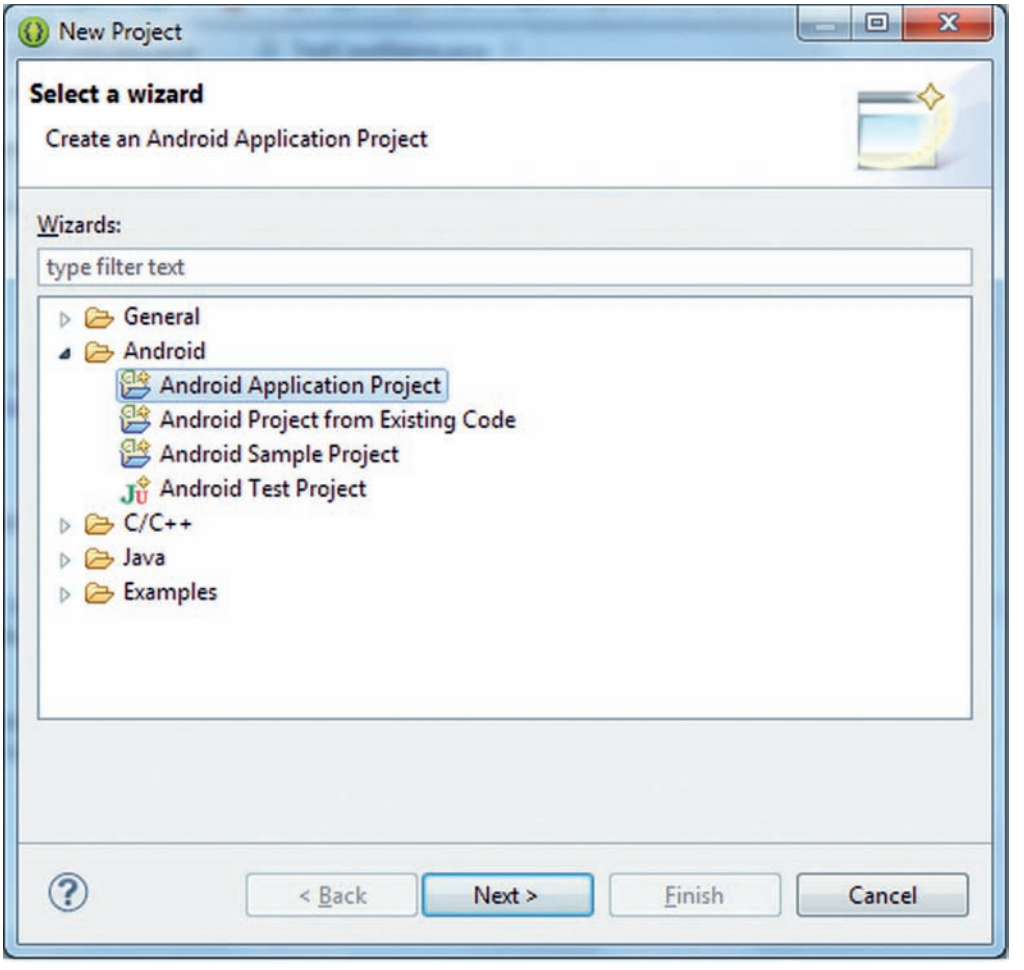


Figure 2.2: New Project Dialog Box

3. Click **Next**.

4. Type **`Android Project`** in the **Application Name** box. By default the **Project Name** box displays **Android Project** and **Package name** box displays **com.example.androidproject** as shown in figure 2.3.

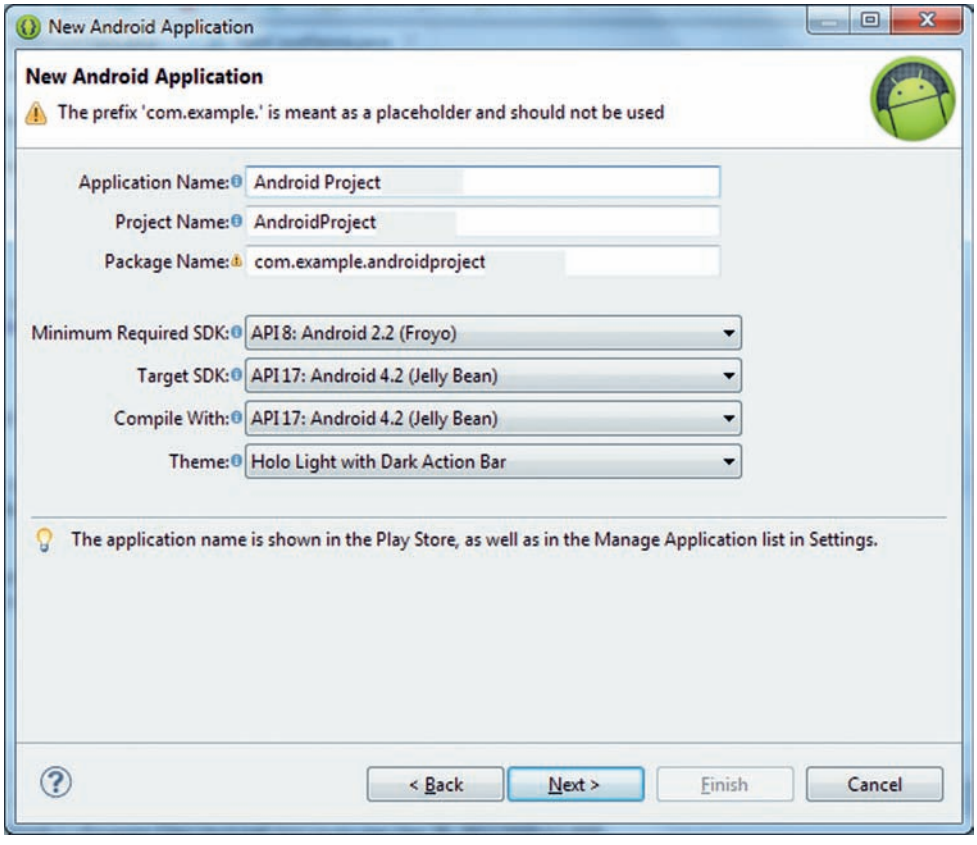


Figure 2.3: New Android Application Dialog Box

5. Click **Next** to display the **Configure Project** pane of **New Android Application** dialog box as shown in figure 2.4.

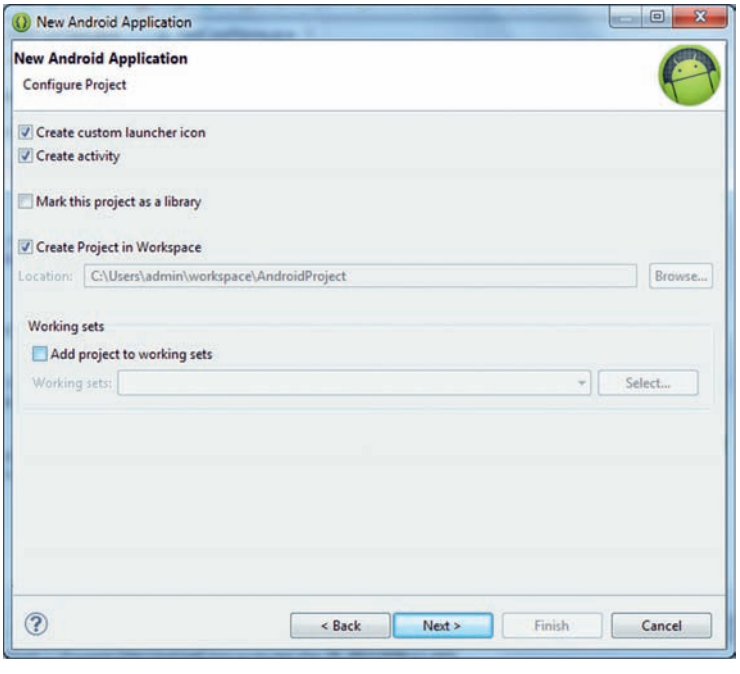


Figure 2.4: New Android Application – Configure Project

6. Click **Next**.
7. Click **Next**.
8. Click **Finish**.
9. Navigate to the **src** → **com.example.androidproject**.
10. Right-click the folder to display the context menu.
11. Select **New** → **Class** as shown in figure 2.5.

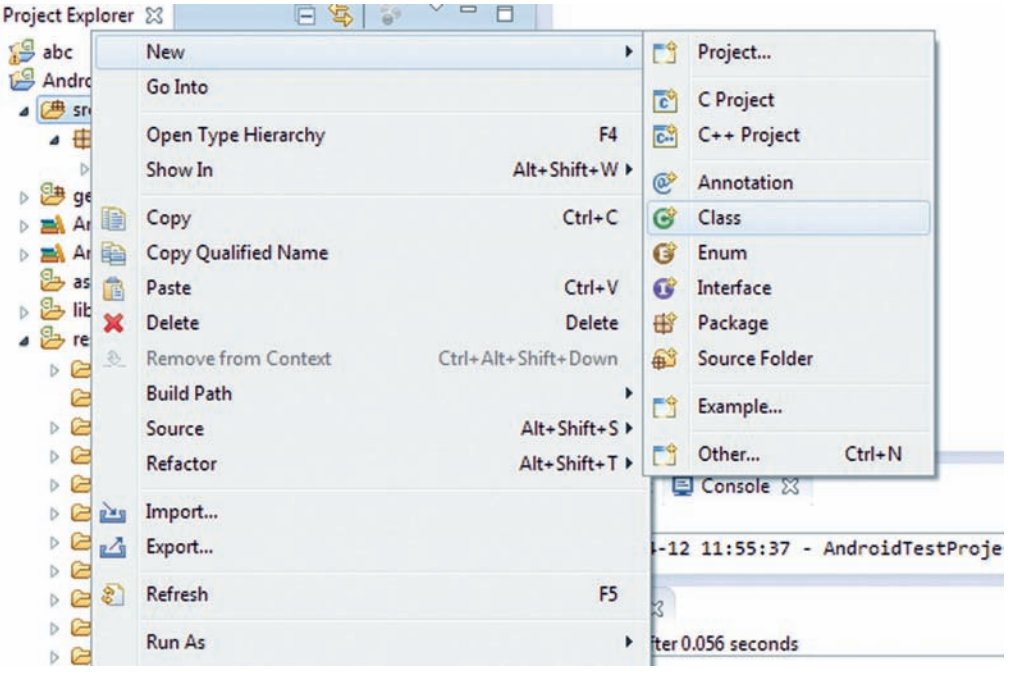


Figure 2.5: Creation of New Class

12. Type **`JunitTestClass`** in the **Name** box.
13. Add the code as shown in code snippet 1.

Code Snippet 1:

```
package com.example.androidproject;

public class JunitTestClass {

     public int multiply(int a, int b) {
          return a * b;
     }

}
```

Concepts

Next a project will be created where JUnit framework will be used to test the method of application. The steps to be performed are as follows:

1. Click **File → New → Other** to display the **New** dialog box.
2. Select **Android → Android Test Project** as shown in figure 2.6.

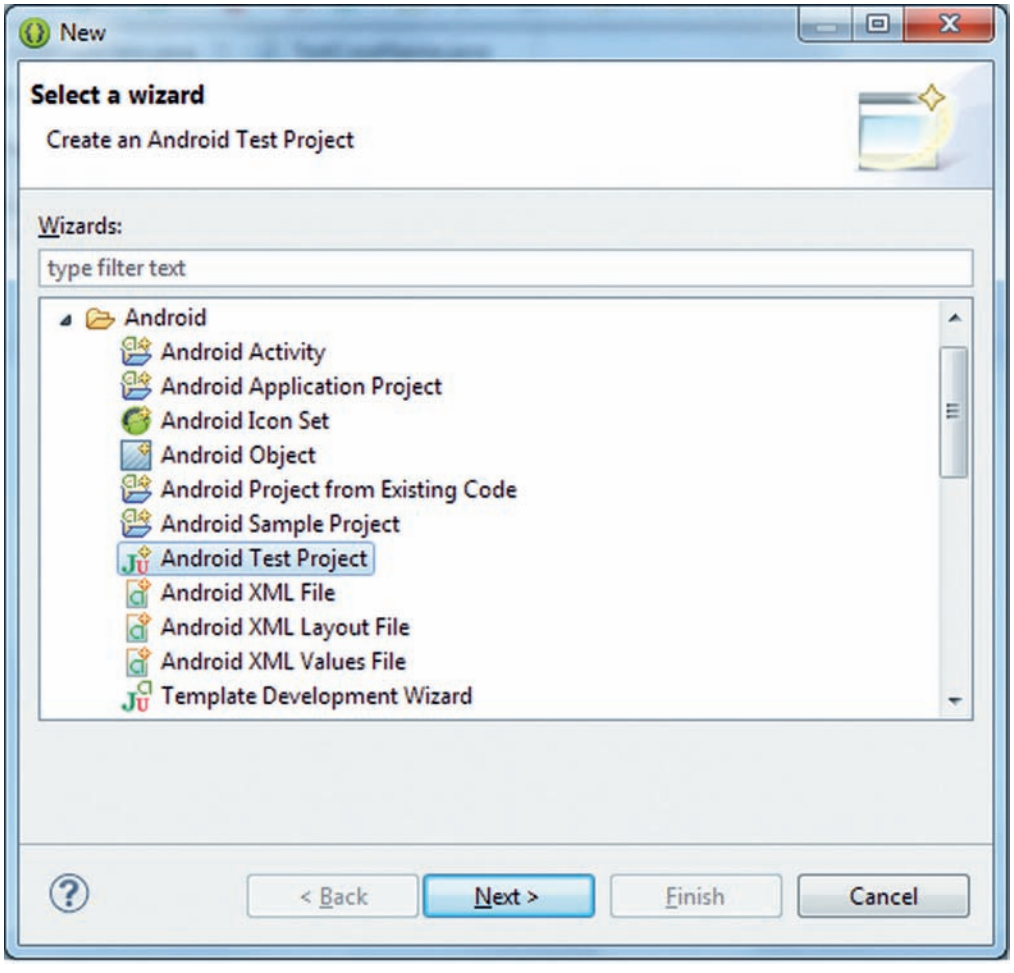


Figure 2.6: New Dialog Box

3. Click **Next** to display the **New Android Test Project** dialog box.

Concepts

4. Type **`AndroidTestJunit`** in the **Project Name** box as shown in figure 2.7.

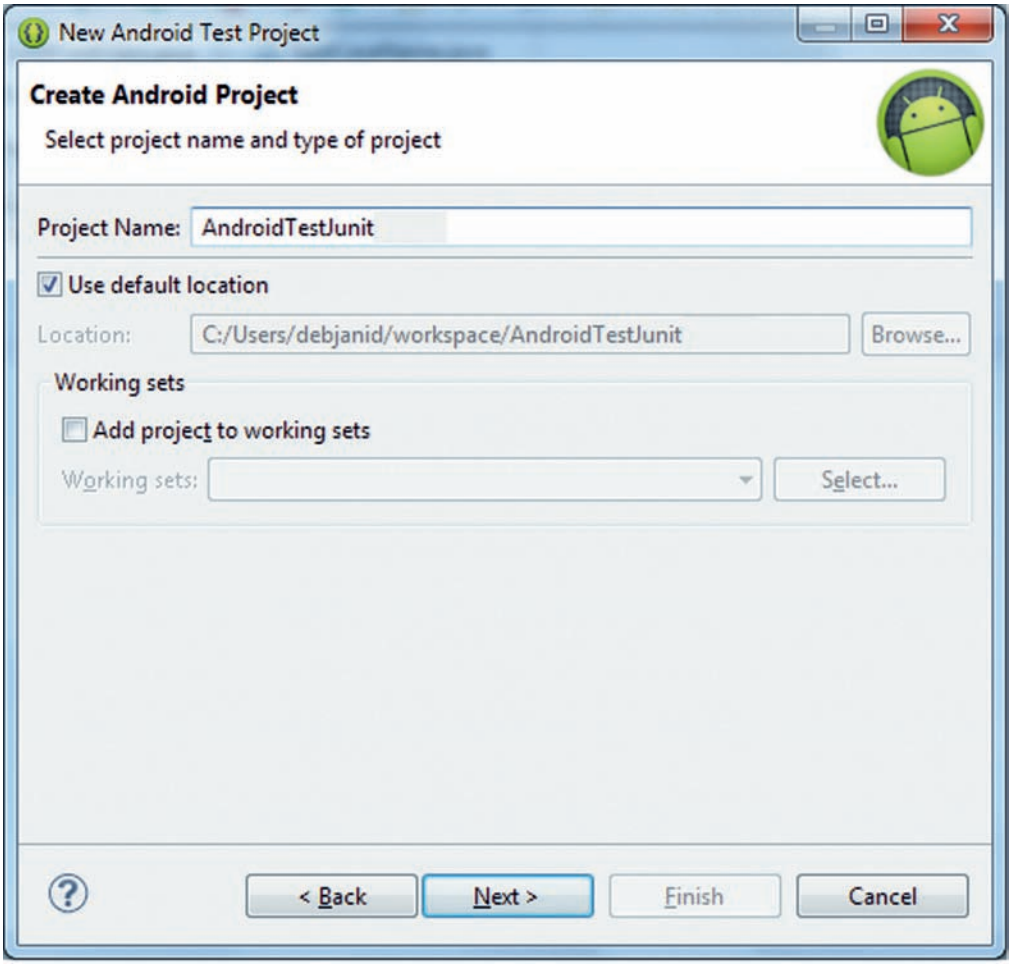


Figure 2.7: New Android Test Project

5. Click **Next** to display the **Select Test Target** pane of the **New Android Test Project** dialog box.

6. Choose the existing Android Project named **AndroidProject** as shown in figure 2.8.

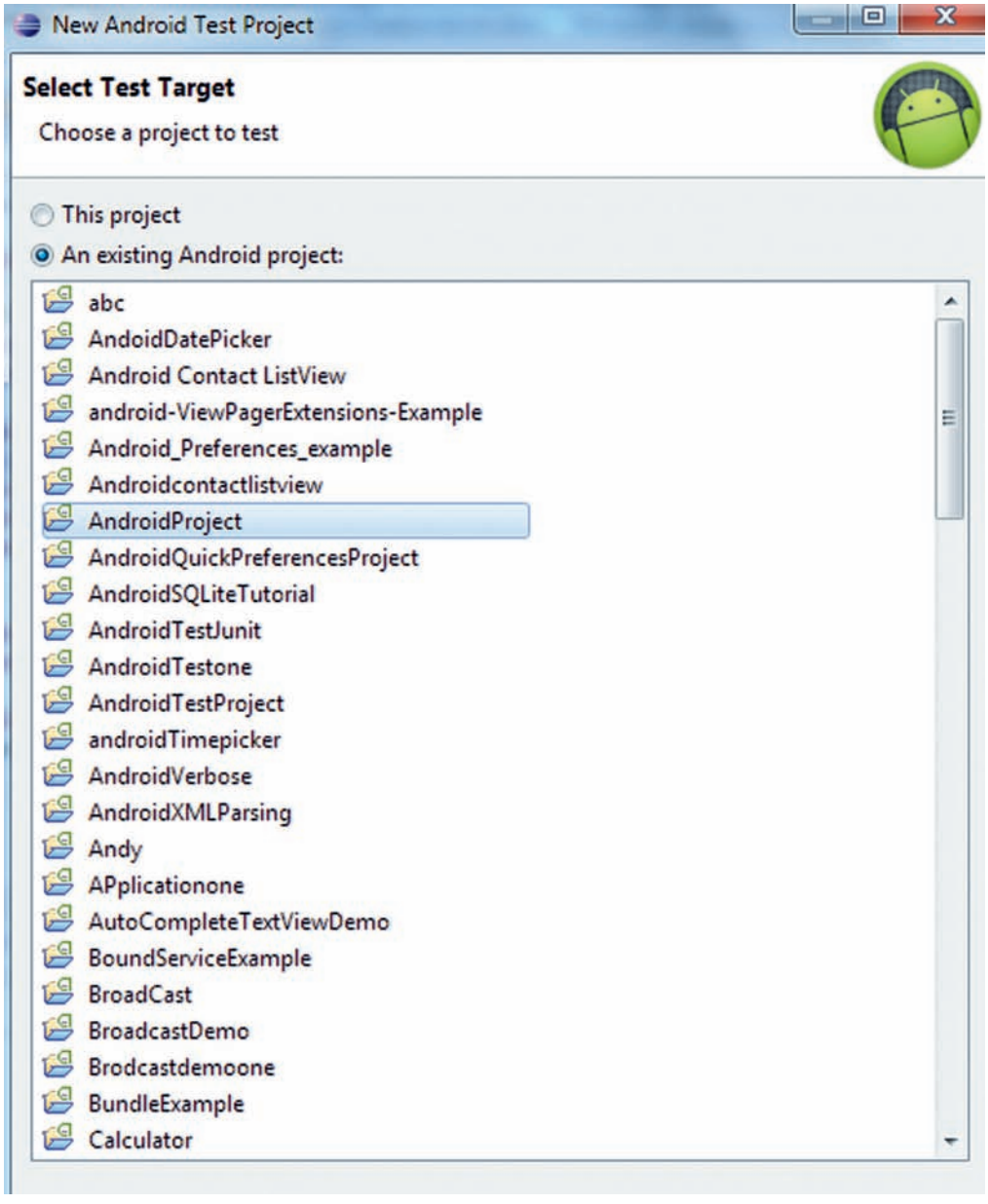


Figure 2.8: Select Test Target Pane

7. Click **Next**.
8. Click **Finish**.
9. Right-click the project to display the context menu.

Concepts

10. Select **New → Other → Java → JUnit → JUnit Test Case** to display the **Select a wizard** pane as shown in figure 2.9.

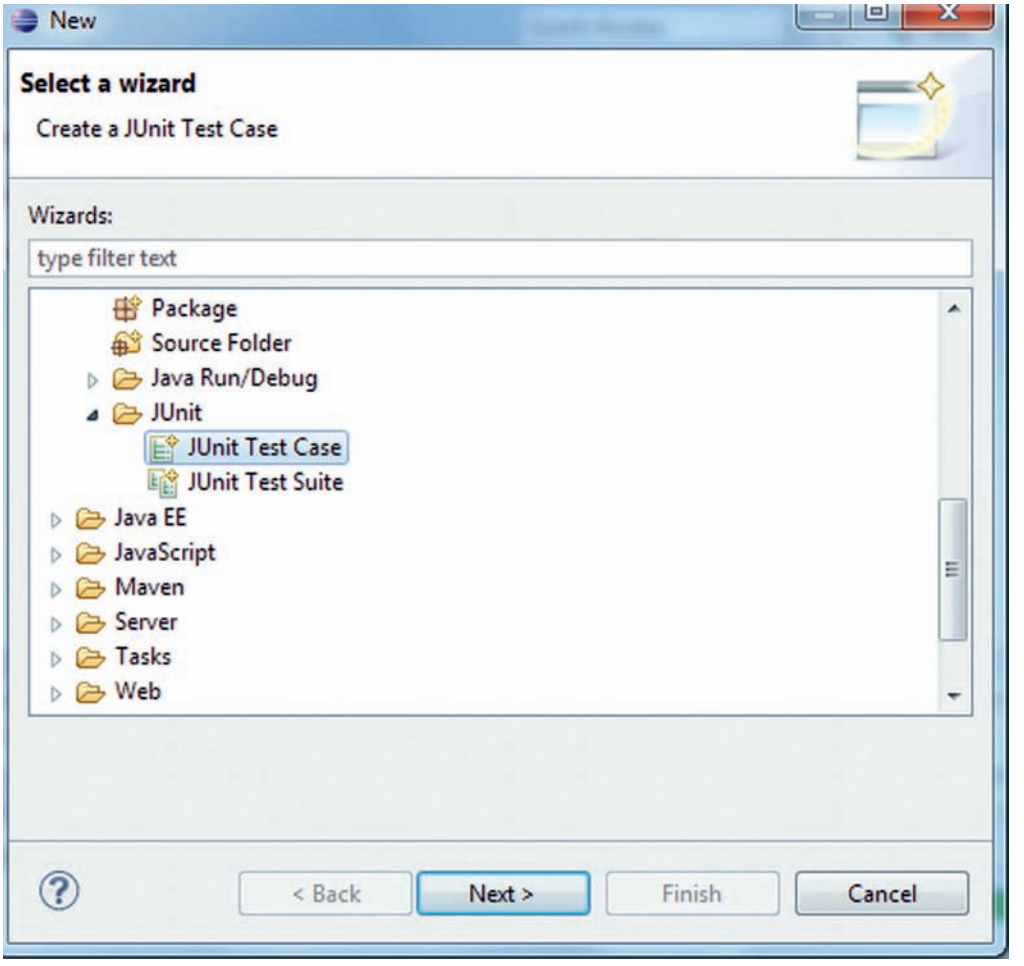


Figure 2.9: Select a wizard Pane

11. Click **Next** to display the **New JUnit Test Case** dialog box as shown in figure 2.10.

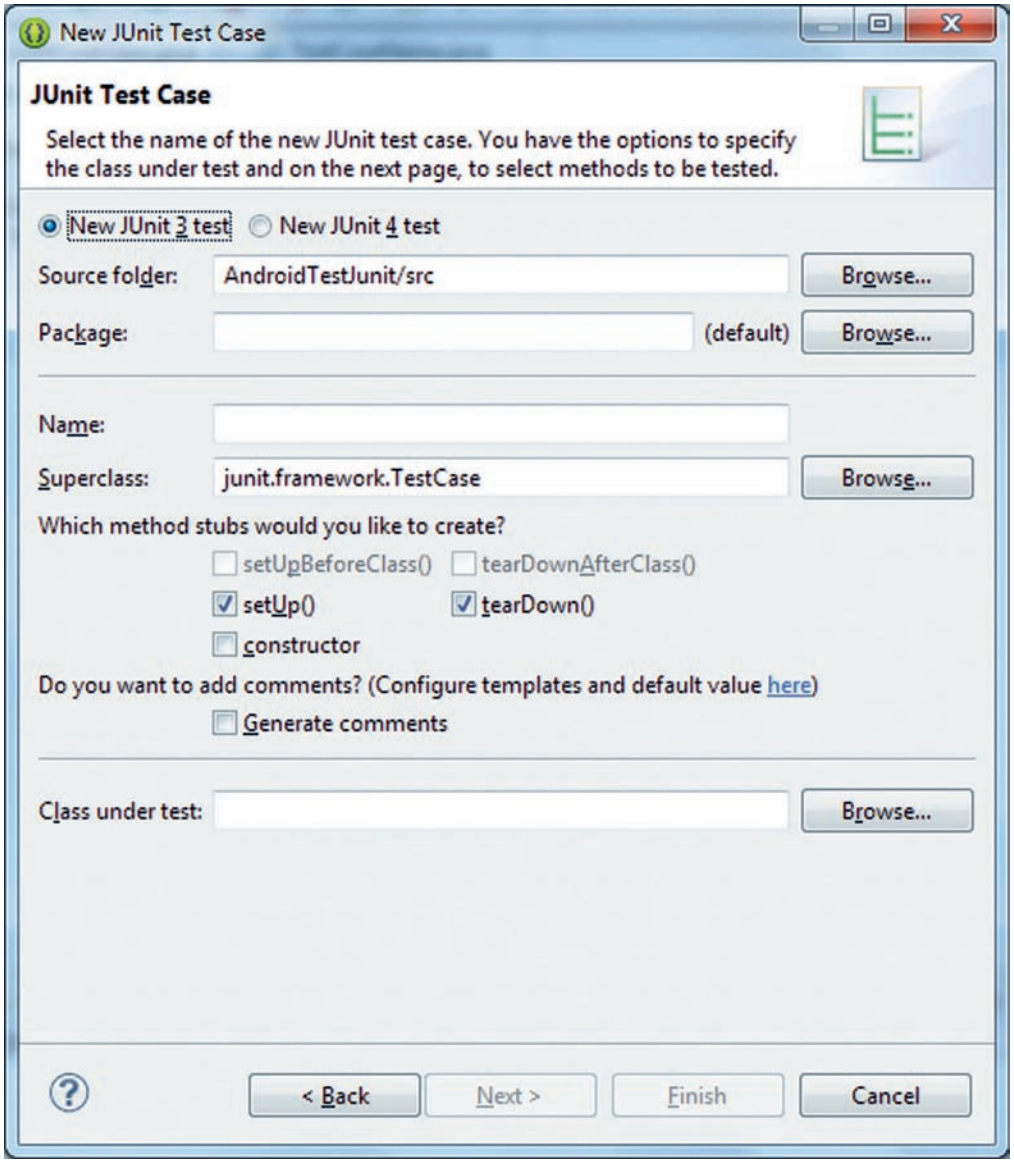


Figure 2.10: New JUnit Test Case Dialog box

12. Type **`TestCaseName`** in the **Name** box.
13. Click **Finish**.

Concepts

The automatically generated code for the test case will be as shown in code snippet 2.

Code Snippet 2:

```
import junit.framework.TestCase;

public class TestCaseName extends TestCase {
     protected void setUp() throws Exception {
          super.setUp();
     }

     protected void tearDown() throws Exception {
          super.tearDown();
     }

}
```

The `setUp()` method executes once before all the test methods are invoked whereas the `tearDown()` method executes after all the test methods are invoked.

14. Modify the code to add a testing method and perform some initialization in the `setUp()` method as shown in code snippet 3.

Code Snippet 3:

```
import com.example.androidproject.JunitTestClass;

import junit.framework.TestCase;

public class TestCaseName extends TestCase {

     JunitTestClass junitTestClass;
     int mArg1;
     int mArg2;

     protected void setUp() throws Exception {
          junitTestClass = new JunitTestClass();
          mArg1 = 6;
          mArg2 = 3;

          super.setUp();
     }

     protected void tearDown() throws Exception {
```

```
        super.tearDown();
    }

    public void testAdd(){
        assertEquals(18, junitTestClass.multiply(mArg1,
        mArg2));
    }

}
```

Note - To add any method to test cases, developer needs to prefix the method name with **test**.

15. Click **Run → Android Junit Test**.

Figure 2.11 displays the output after execution of the application.

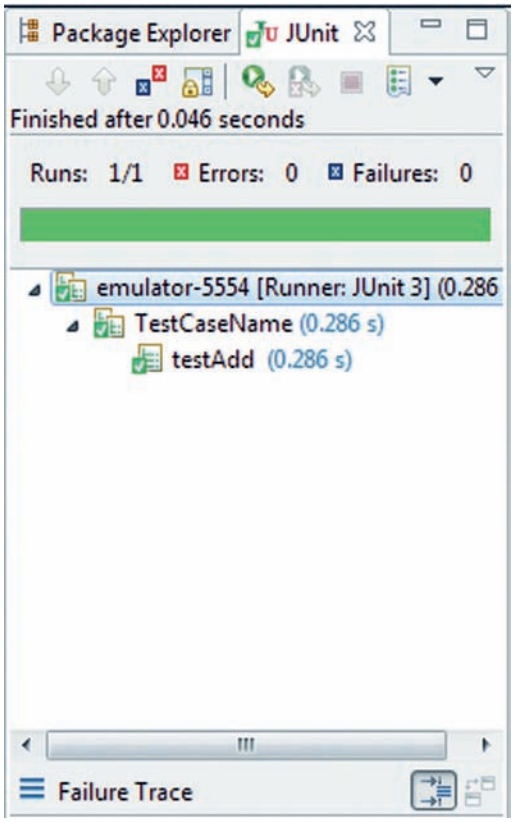


Figure 2.11: Test Output

2.2.1 Android Tools

Tests are put together into projects. A test project is a directory or an Eclipse project that is used to create the source code and manifest files for a test package. Android SDK tools are used to create test projects. The tools that are used for Eclipse with ADT and

command-line create directories and update test projects. The directories are used for source code and resources and the manifest files for the test package. And build files are created by command-line tools.

The other functions and uses of Android tools include:

➔ Assigns `InstrumentationTestRunner` automatically as the test case runner to run JUnit tests for the test package under test as seen in code snippet 4.

Code Snippet 4:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
     package="com.example.androidproject.test"
     android:versionCode="1"
     android:versionName="1.0" >
     <uses-sdk android:minSdkVersion="8" />
     <instrumentation
          android:name="android.test.InstrumentationTestRunner"
          android:targetPackage="com.example.androidproject" />
     <application
          android:icon="@drawable/ic_launcher"
          android:label="@string/app_name" >
          <uses-library android:name="android.test.runner" />
     </application>
</manifest>
```

➔ Conceives and provides a relevant name for the test package. Consider, **`com.example.myfirstapp`** as the package name for the application under test, and then generates and provides the test package name as **`com.example.myfirstapp.test`**. Such a naming system helps in identifying the relationship between the application package and the test package and also prevents conflicts within the system.

➔ Creates proper build files, manifests files, and directory structure. This is an automated function for the test project. The Android tools thus, helps in building the test package and establishing a connection between the test package and the application under test. It does not require the developer to modify the build files.

Concepts

➔ Creates a test project anywhere in the file system. Though this may be the quick way, it is advised to add the test project at the root. Adding the test project the root makes it easier to trace the tests associated with an application, because the root directory **tests/** is at the same level as the **src/** directory of the main application's project. Consider the root directory of an application project as **MyFirstProject** then the directory structure will be as shown in figure 2.12.

```
MyFirstProject/
        AndroidManifest.xml
    res/
        ...(resources for main application)
    src/
        ...(source code for main application)...
    tests/
        AndroidManifest.xml
    res/
        ...(resources for tests)
    src/
        ...(source code for tests)
```

Figure 2.12: Directory Structure of the Test Project

2.3 Android Testing API

The Android testing API is modeled on the JUnit API. It has an instrumentation framework and Android-specific testing classes.

Unit test is a software test that helps to isolate the component under test and tests it repeatedly. It is written in a programming language for the programmers by the programmers. JUnit is the default standard for unit tests on Android. It automates unit testing.

A JUnit can be called as a method that allows testing of each part of an application. Test methods are organized into classes called test cases or test suites. Each test method is a stand-alone test conducted on an individual module present in the application under test. And, each class is a collection of related test methods. The class may also be responsible for providing helper methods too.

Similar to JUnit, Android too uses SDK build tools to create one or more test source files into class files in an Android test package. While in JUnit, a test runner is used to execute test classes, an Android uses test tools to execute an Android-specific test runner. The test tools first load the package and the application under test and then operate the Android-specific test runner.

AndroidTestCase extends the JUnit TestCase. JUnit TestCase class is called when unit testing is done on a class that does not use Android APIs. Android-dependent objects can be tested using AndroidTestCase class. Besides this, AndroidTestCase class also provides Android-specific setup, teardown, and helper methods. Figure 2.13 displays the hierarchy of AndroidTestCase class.

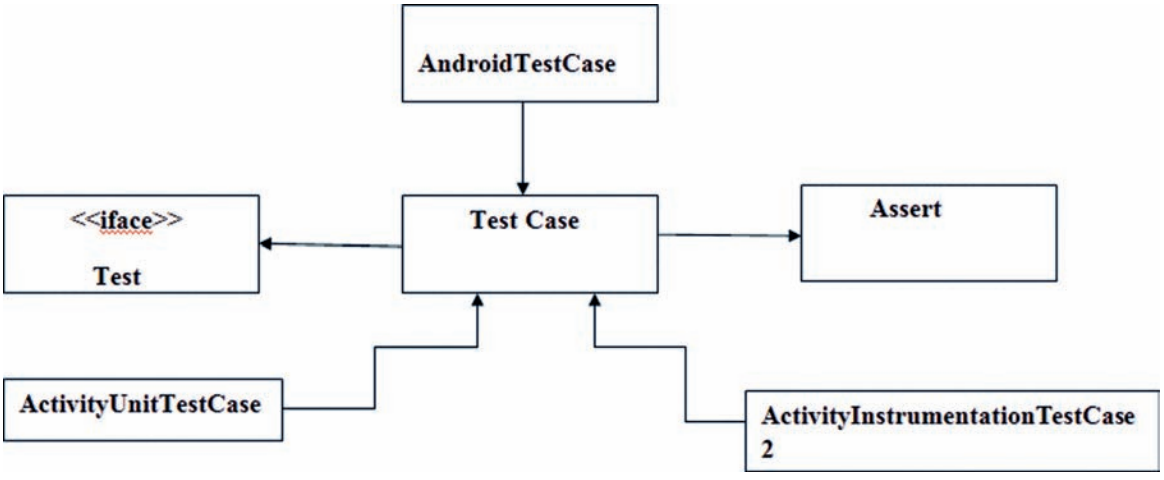


Figure 2.13: AndroidTestCase class

2.3.1 AndroidTestCase Base Class

As the name suggests, this class is the base class for testing Android test cases. It is present in the **android.test** package. It extends both the TestCase and Assert class. This class is used when access to an Activity Context in the file system is required. Activity Context can be resources, databases, or files in the file system. This class stores Context object as a field referred to as mContext. Both mContext and getContext() methods can be used inside tests if needed. Also, Context.startActivity() method can be used to start more than one Activity. The other test cases that extend this base class are as follows:

- ApplicationTestCase<T extends Application> - A class for testing an entire application.
- ProviderTestCase2<T extends ContentProvider> - A class for isolated testing of a single Content Provider.
- ServiceTestCase<T extends Service> - A class for isolated testing of a single Service.

Apart from these, this class provides methods for testing permissions and a method that prevents memory leaks. It removes certain class references to protect from memory leaks. Figure 2.14 shows the UML class diagram of AndroidTestCase and its related classes.

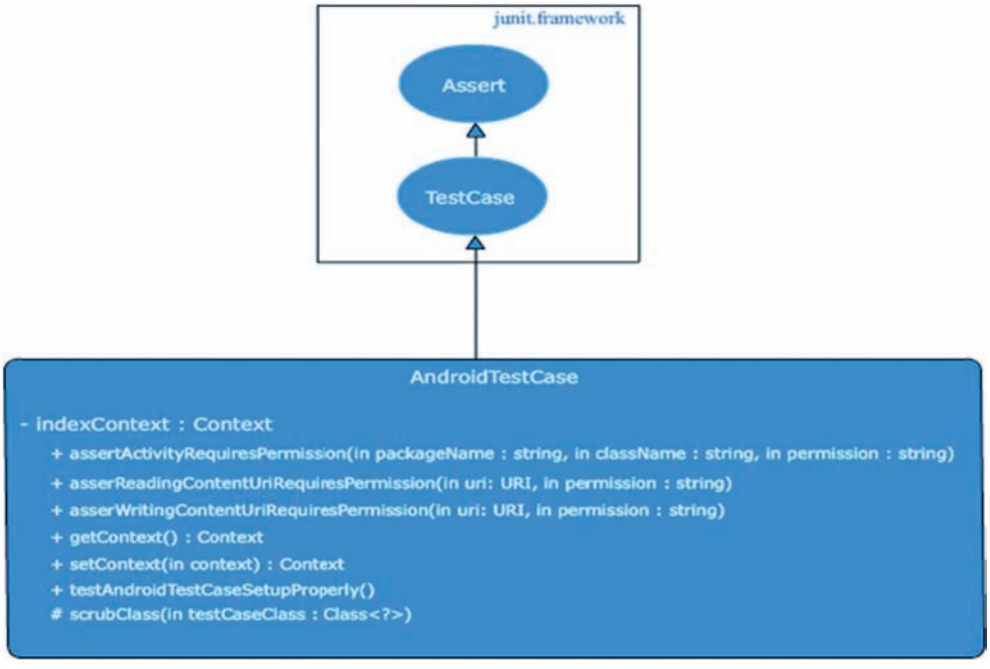


Figure 2.14: UML Class Diagram of AndroidTestCase

➔ ApplicationTestCase class

This class is used for testing the application classes, more clearly to test the setup and teardown of all application objects. The function of these objects is to take care of the global state of information that is applicable to all the components in the application package. The test case also verifies whether the `<application>` element in the manifest file is set up correctly. The `<application>` tag in the **AndroidManifest.xml** file will cause the class to be instantiated when the application/package is created. It needs to be remembered that this test case does not allow control testing of the components in the application package.

➔ ProviderTestCase2

It is a class that is used for isolated testing of a single **ContentProvider**. This test case has its own internal map which it uses to resolve providers that are given an authority. This enables injecting test providers that are to be used and at the same time nulling out providers that are unused. The base class for this class is `AndroidTestCase` class. The template parameter **T** represents the `ContentProvider` under test and uses an `IsolatedContext` and a `MockContentResolver` for implementation of this test.

A few mock objects that are set up by this class are as follows:

- `IsolatedContext`
  - Allows tests to run file and perform database work.
  - Stubs out Context methods that affect functioning of the rest of the system.

Concepts

- `MockContentResolver`
  - Uses `IsolatedContext` to function like a regular content resolver.
  - Stubs out the method `notifyChange(Uri, ContentObserver, boolean)` to prevent the test from affecting the functioning of the rest of the system.
- An instance of the provider case under test, which runs in an `IsolatedContext`.

The base class' `setUp()` method automatically sets up the framework for the `ProviderTestCase` class. In case there is a need to override this method, the `super()` method is invoked and should be the first statement that needs to be invoked in the overridden method.

For successful running of these tests, concrete subclasses must support their own constructor with no arguments.

Figure 2.15 shows the UML diagram of `ProviderTestCase2` and the closest related classes.

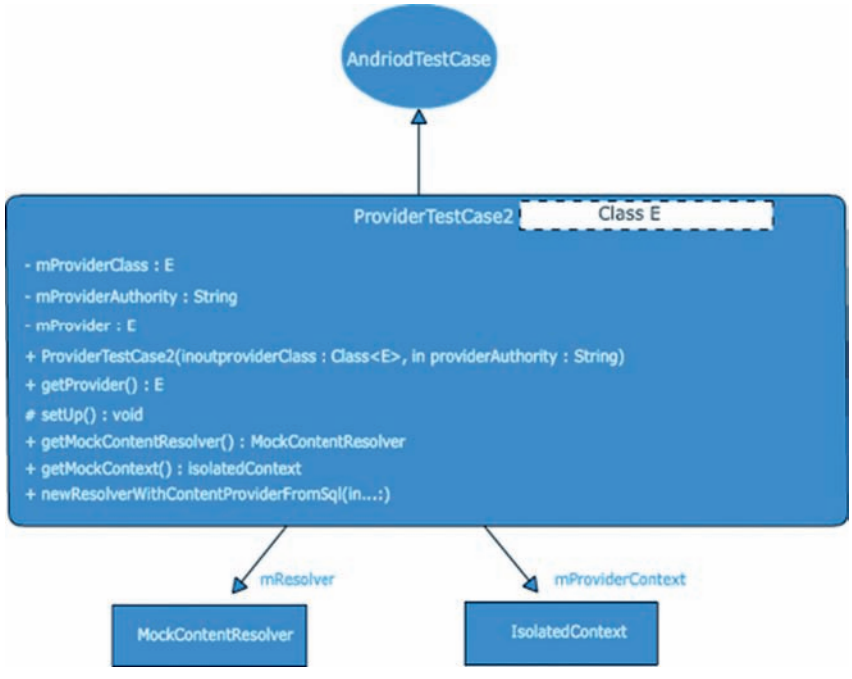


Figure 2.15: ProviderTestCase2

The only public non deprecated constructor of this class is:

```
ProviderTestCase2(Class<E>providerClass, String
providerAuthority)
```

It is invoked using an instance of the `ContentProvider` class which is also used as the first parameter. The authority for the provider, defined as a constant named `AUTHORITY` in the `ContentProvider` class is the second parameter.

Code Snippet 5 displays an example of testing the Content Provider.

Code Snippet 5:

```
...
public void testCheckQuery(){
        Uri uri=Uri.withAppendedPath(
    MyProvider.CONTENT_URI, "dummy");
    final Cursor ap=mProvider.query(uri, null, null, null, null);
    finalint anticipated=4;
    finalint original=ap.getCount();
    assertEquals(anticipated, original);
    }
...
```

In the code snippet, the `Cursor` object is anticipated to return a result containing 4 rows. This test can be asserted for more than four rows.

In the `setup()` method, the `getProvider()` method is invoked to obtain a reference of the provider named, `mProvider`. An interesting point to observe is that since these tests are using `MockContentResolver` and `IsolatedContext`, the content of the real database is not affected. This allows running tests as shown in code snippet 6.

Code Snippet 6:

```
...
public void testCheckDelete(){
        Uri uri=Uri.withAppendedPath(
    MyProvider.CONTENT_URI, "dummy");
    final int original=mProvider.delete(
    uri, "_id=?", new String[]{"1"});
    final int anticipated=1;
    assertEquals(anticipated, original);
}
…
```

Though this test deletes some content of the database, it restores the initial content so that other tests are not affected.

➔ The ServiceTestCase<T>

This test case tests services. In other words, it tests the `Service` class. Service lifecycle methods such as `setupService()`, `startService()`, `bindService()`, and `shutDownService()` are also included in this class. The only public non deprecated constructor of this class is:

```
ServiceTestCase(Class<T>serviceClass)
```

This test provides the basic support to test `Service` classes in a controlled environment. It provides the framework for the lifecycle of the service. It also serves links for injecting various dependencies to test the `Service` class in a controlled environment. Figure 2.16 shows that this class extends the `AndroidTestCase` class.

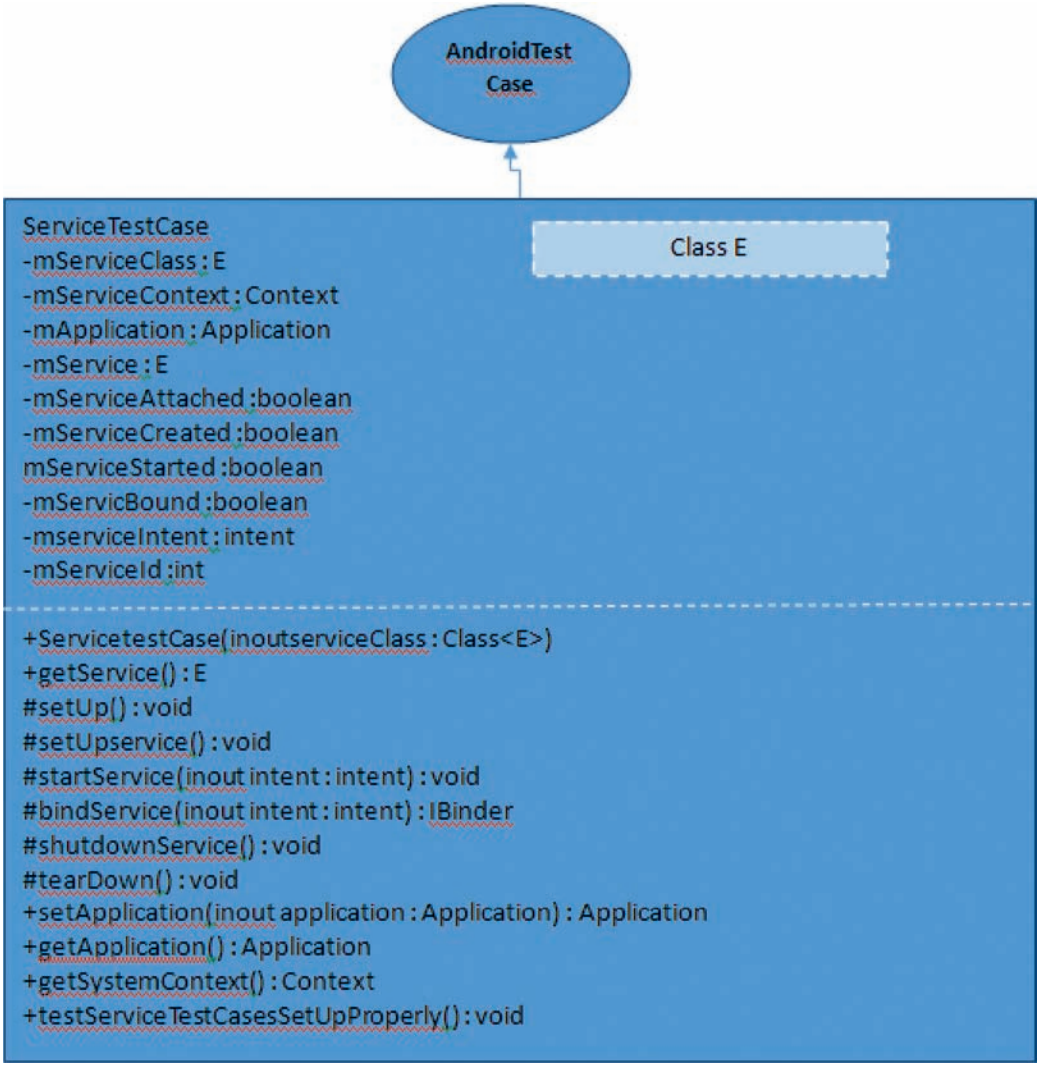


Figure 2.16: UML Class Diagram of ServiceTestCase<E>

2.3.2 Android Instrumentation

The Android system has a set of control methods or 'hooks' called Android instrumentation. The hooks function as a control:

➔ for an Android component to run independently of its normal cycle.

➔ on how an Android loads an application/applications.

Generally, the system determines the lifecycle of an Android component. For example, the lifecycle of an `Activity` object begins when it is activated by an Intent. The object of the `Activity` class invokes the `onCreate()` method followed by the `onResume()` method. When another application is started, the `onPause()` method is invoked. The `onDestroy()` method is invoked when the `Activity` code invokes the `finish()` method. These callback methods cannot be invoked directly with the use of the Android framework API but can be done using instrumentation.

Normally, the system runs all components of an application in the same process or sometimes for some components, such as content providers, it runs as a separate process. However, it is not possible for the system to run an application in the same process as another application that is already running. Such a disability can be overrun with the Android instrumentation. Android instrumentation runs step-by-step through the lifecycle of a component by invoking callback methods in the test code.

The following example shows the use of Android Instrumentation in Eclipse. The example will create a new project in Eclipse where two numbers will be added and will be tested with the help of Android Instrumentation.

1. Start Eclipse IDE.
2. Create a project named **`Add Number`**.
3. Navigate to **res → layout** folder.
4. Open **activity_main.xml** file.
5. Modify the code as given in code snippet 7.

Code Snippet 7:

```
<RelativeLayout xmlns:android="http://schemas.android.
com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        tools:context=".MainActivity" >
```

```
<EditText
    android:id="@+id/editText1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/textView1"
    android:ems="10"
    android:hint="Enter first number"
    android:singleLine="true" >
</EditText>

<EditText
    android:id="@+id/editText2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentLeft="true"
    android:layout_below="@+id/textView1"
    android:layout_marginTop="99dp"
    android:ems="10"
    android:hint="Enter second number"
    android:singleLine="true" >

<requestFocus />
</EditText>

<TextView
    android:id="@+id/textView5"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignRight="@+id/textView1"
    android:layout_below="@+id/editText2"
    android:layout_marginTop="35dp"
    android:text="Output" />
```

```
    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_alignRight="@+id/editText1"
        android:gravity="center"
        android:text="Add Two Number" />

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/editText2"
        android:layout_centerHorizontal="true"
        android:text="Button" />


    </RelativeLayout>
```

6. Navigate to **src → com.example.addnumber** folder.
7. Open **MainActivity.java** file.
8. Modify the code as shown in code snippet 8.

Code Snippet 8:

```
package com.example.addnumber;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
```

Concepts

```
public class MainActivity extends Activity {

    EditText first, second;
    Button res;
    TextView tx;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        first = (EditText) findViewById(R.id.editText1);
        second = (EditText) findViewById(R.id.editText2);
        tx = (TextView) findViewById(R.id.textView5);
        res = (Button) findViewById(R.id.button1);

        res.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                // TODO Auto-generated method stub
                float result = Float.parseFloat(first.
getText().toString())
                        + Float.parseFloat(second.
getText().toString());
                tx.setText(Float.toString(result));
            }
        });

    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar
        if it is present.
```

```
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
}
```

Once you execute the application, the output will be as shown in figure 2.17.

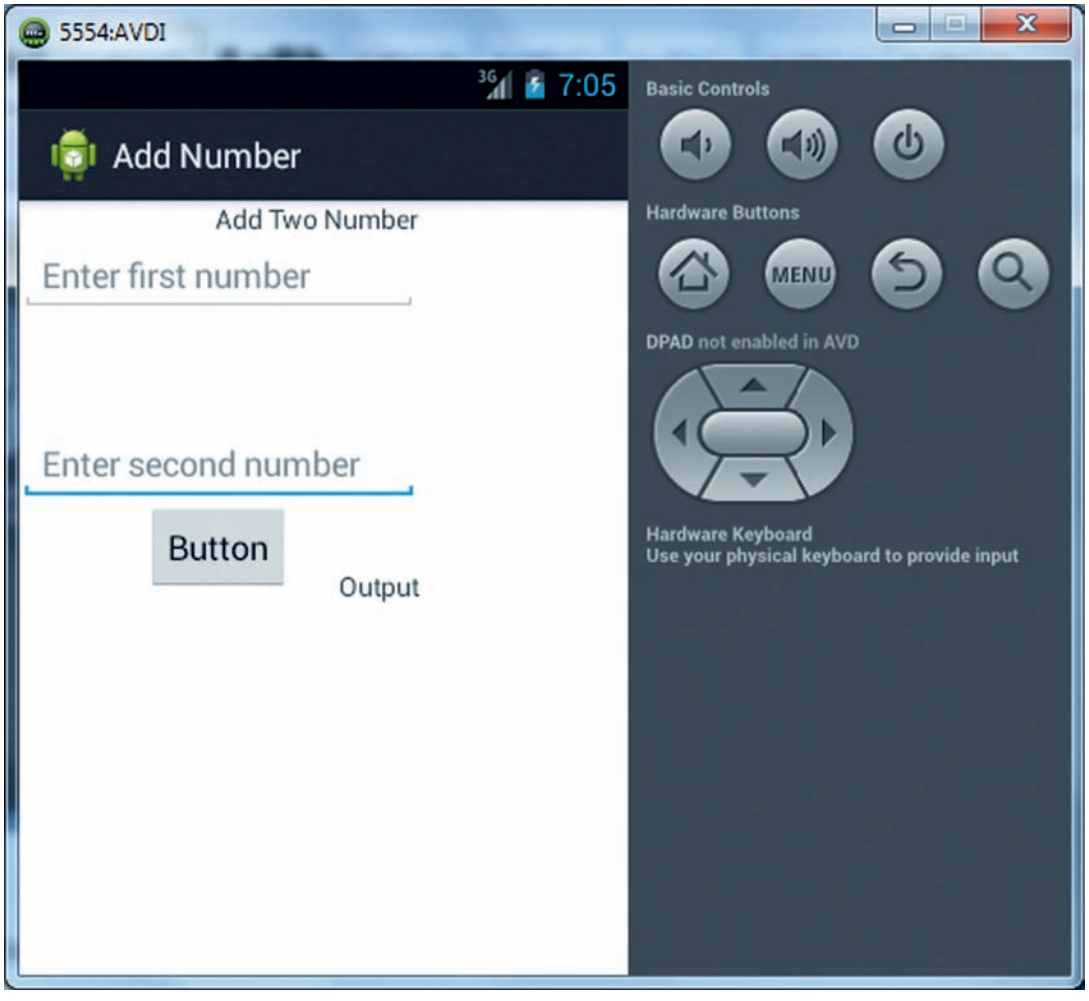


Figure 2.17: Add Number Application

Next a project, **AddNumberTest** will be created which will use Instrumentation framework to test the method of application.

1. Click **File → New → Other** to display the **New** dialog box.
2. Select **Android → Android Test Project** as shown in figure 2.18.

Concepts

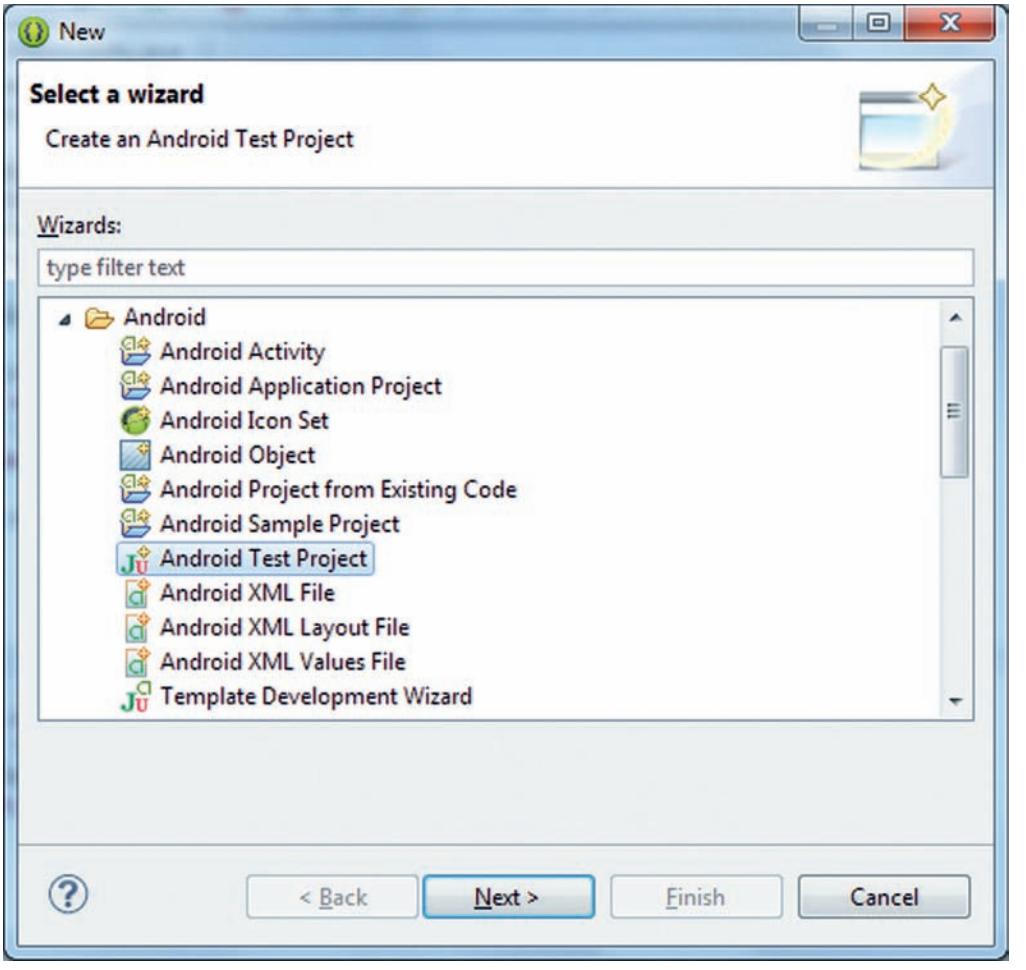


Figure 2.18: New Dialog Box

3. Click **Next**.

4. Type **AddNumberTest** in **Project Name** box as shown in figure 2.19.

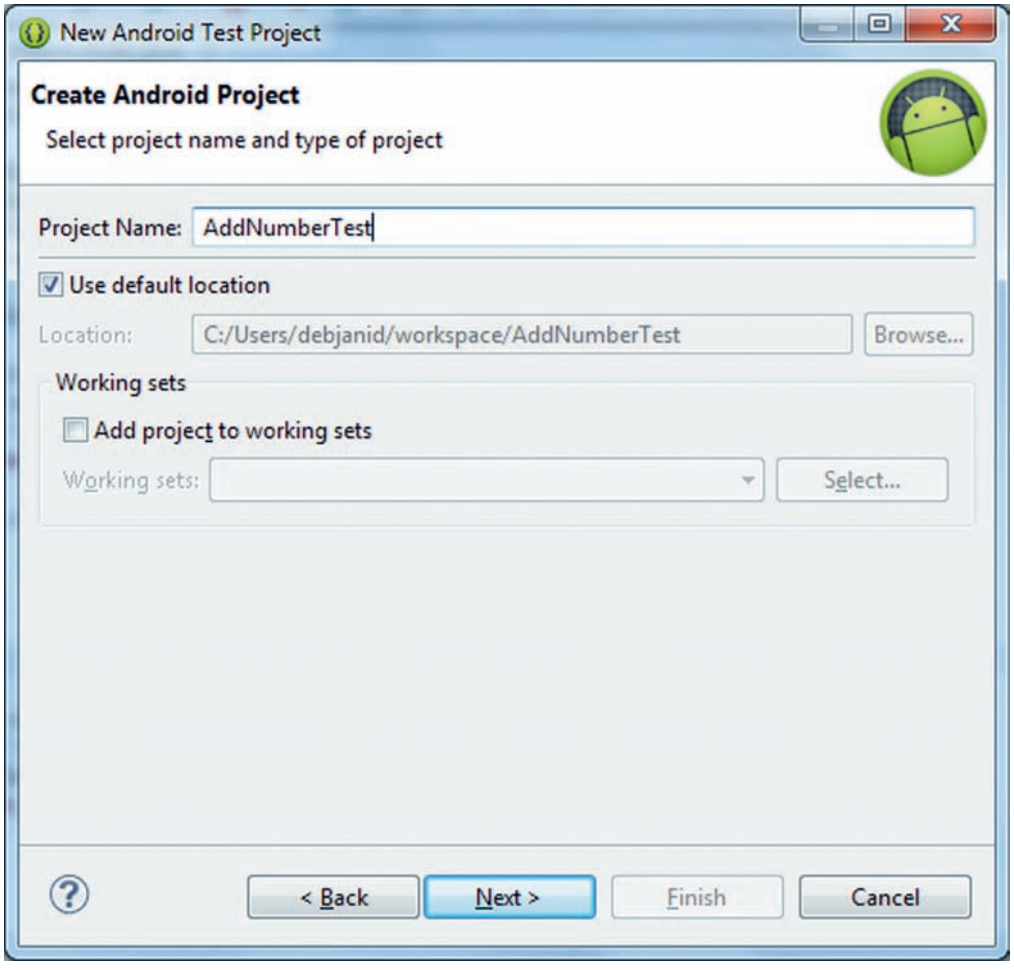


Figure 2.19: New Android Test Project

5. Click **Next**.

Concepts

6. Select **AddNumber** as the project for which the test project is being created as shown in figure 2.20.

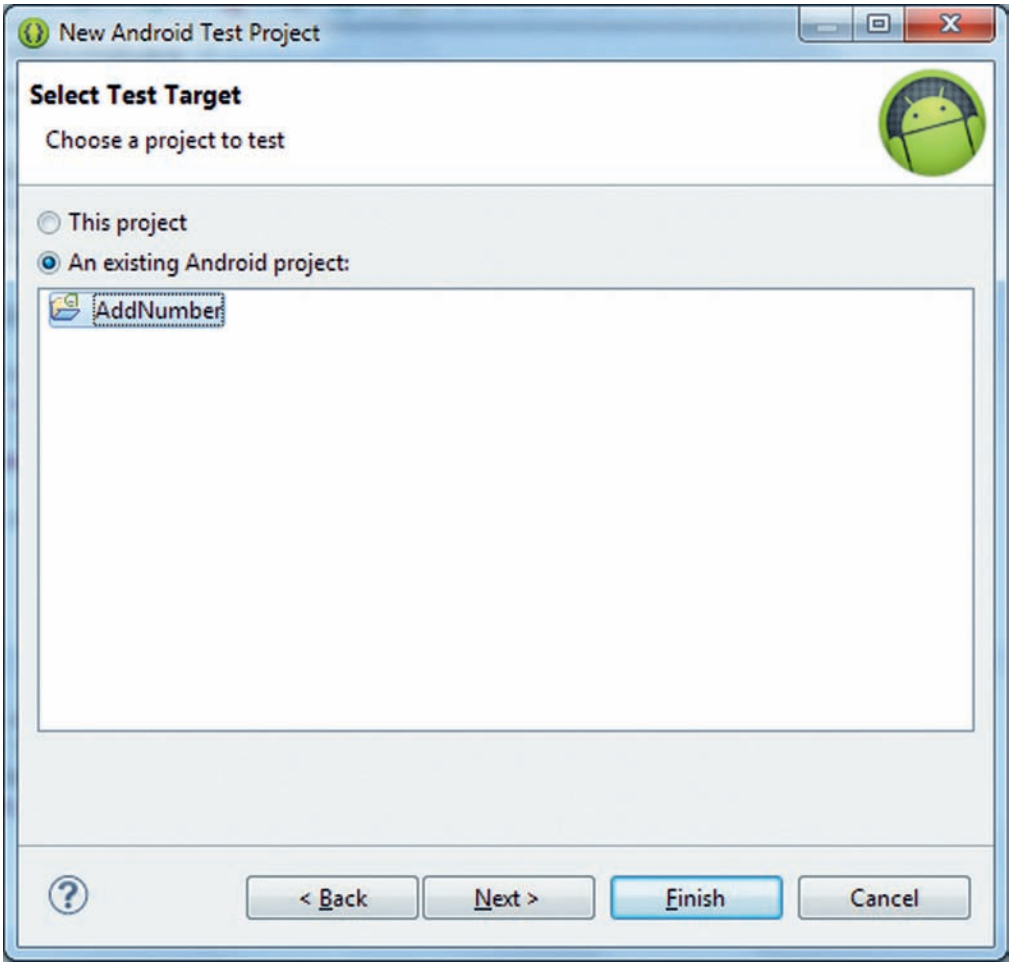


Figure 2.20: Select Test Target Pane

7. Click **Next**.
8. Click **Finish**.
9. Navigate to **src** → **com.example.addnumber.test** folder.
10. Right-click the folder to display the context menu.
11. Click **New** → **Class**.
12. Type **AddnumberTest** in the **Name** box.
13. Modify the code as given in code snippet 9.

Code Snippet 9:

```
package com.example.addnumber.test;

import com.example.addnumber.MainActivity;

import android.test.ActivityInstrumentationTestCase2;
import android.test.TouchUtils;
import android.test.suitebuilder.annotation.SmallTest;
import android.view.KeyEvent;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Spinner;
import android.widget.TextView;

public class AddnumberTest extends ActivityInstrumentation
TestCase2<MainActivity> {

     MainActivity mActivity;
     /** Operator of the Calculator activity */
     Spinner mOperator;

     /** Operand1 */
     EditText mNumber1 ;

     /** Operand2 */
     EditText mNumber2;

     /** Result */
     TextView mResult ;

     /** Button to perform calculation */
     Button mBtnResult;
     @SuppressWarnings("deprecation")
     public AddnumberTest() {
```

```
        super("com.example.addnumber",MainActivity.
        class);
    }
    @Override
    protected void setUp() throws Exception {
        super.setUp();
        /** Starting Calculator Activity */
        mActivity = getActivity();

        /** Getting reference to Operand1 */
        mNumber1 = (EditText) mActivity.findViewById(
                com.example.addnumber.R.id.editText1);

        /** Getting reference to Operand2 */
        mNumber2 = (EditText) mActivity.findViewById(
                com.example.addnumber.R.id.editText2);

        /** Getting reference to Result */
        mResult = (TextView) mActivity.findViewById(
                com.example.addnumber.R.id.textView5);

        /** Getting reference to the calculation performing
        button */
        mBtnResult = (Button) mActivity.findViewById(
                com.example.addnumber.R.id.button1);
    }

    @Override
    protected void tearDown() throws Exception {
        super.tearDown();
    }

    /** Testing Add Operation */
    @SmallTest
```

```
    public void testAddition(){

        String strNum1 = "100 PERIOD 5";
        String strNum2 = "50";
        float expected = 150.5f;

        /** Inputting Operand1 */
        TouchUtils.tapView(this, mNumber1);
        sendKeys(strNum1);

        /** Inputting Operand2 */
        TouchUtils.tapView(this, mNumber2);
        sendKeys(strNum2);

        /** Performs calculation */
        try {
            runTestOnUiThread(new Runnable() {
                @Override
                public void run() {
                    mBtnResult.performClick();
                }
            });
        } catch (Throwable e1) {
            e1.printStackTrace();
        }

        /** Getting the result shown */
        float actual = Float.parseFloat(mResult.getText().
        toString());

        /** Asserts, expected and actual are same */
        assertEquals(expected, actual, 0);
    }
}
```

14. Right-click **AddNumberTest.java** to display the context menu.
15. Select **Run As → Android JUnit Test**.

The output will be as shown in figure 2.21.

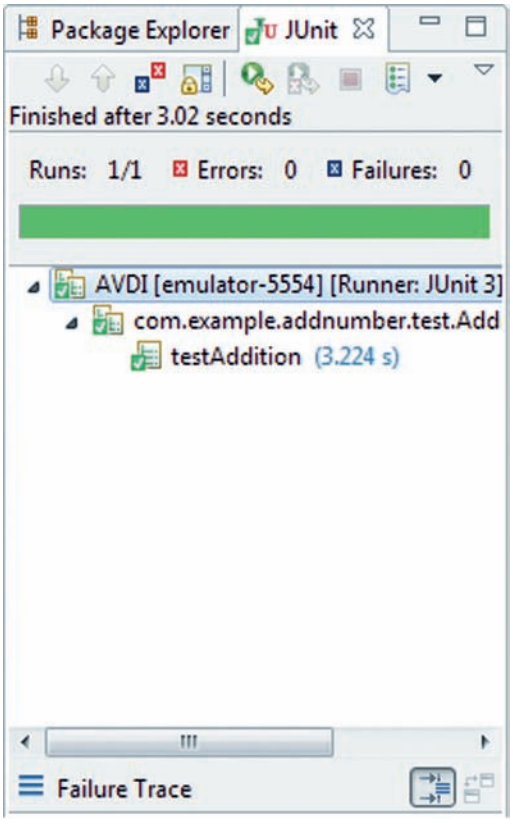


Figure 2.21: Test – Output

The key method used in the code snippet is `getActivity()`. This method is part of the instrumentation API. The Activity class which is being tested does not begin unless this method is invoked. It is possible to set the test fixture in advance and later call this method to start the activity.

Also, Instrumentation uses the same process to loading a test package and the application under check. As a result of this, tests can invoke methods and modify and examine fields in the units of the application being tested.

The various test cases that have access to Instrumentation have the `InstrumentationTestCase` class as the base class, either directly or indirectly. The most important subclasses for which `InstrumentationTestCase` class may be a direct or indirect base class are as follows:

- `ActivityTestCase`— base class for activity test classes.
- `SingleLaunchActivityTestCase`— tests a single activity.
- `SyncBaseInstrumentation`— tests synchronization of a content provider.

Concepts

- `ActivityUnitTestCase`— isolated test of a single activity.
- `ActivityInstrumentationTestCase2`— tests a single activity within a normal system environment.

➔ Using the ActivityTestCase class

This class stores common code for other test cases. This class can be used in cases where the existing alternatives do not fit requirements.

Other classes available for meeting the requirements are as follows:

- `ActivityInstrumentationTestCase2<E extends Activity>`
- `ActivityUnitTestCase<E extends Activity>`

Figure 2.22 is an UML class diagram of `ActivityTestCase` and its closely related classes.

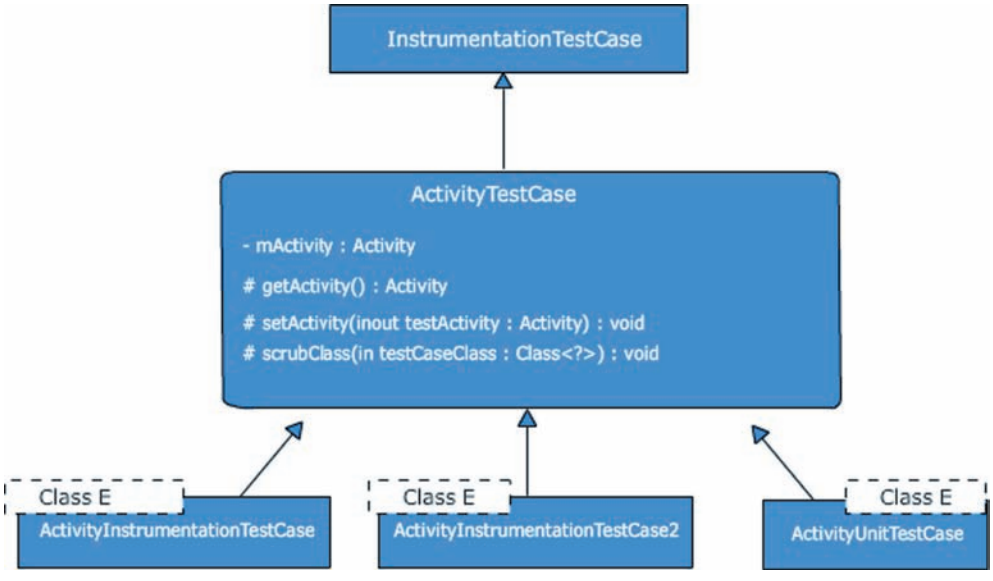


Figure 2.22: UML Class Diagram of `ActivityTestCase` and its Close Related Classes

It can be seen in figure 2.22 that the `ActivityTestCase` class extends from the `android.test. InstrumentationTestCase` class and is the base class for other classes such as `ActivityInstrumentationTestCase`, `ActivityInstrumentationTestCase2`, and `ActivityUnitTestCase`.

- **scrubClass method**

  One of the protected methods in this class is the `scrubClass()` method. The syntax of this method is:

Concepts

Syntax:

```
protected void scrubClass(Class<?>testCaseClass)
```

where,

`testCaseClass`: represents the class of the derived `TestCase` implementation.

This is invoked from the `tearDown()` method while implementing various test cases to clean up class variables. These may be manifested as non-static inner classes to avoid the need to have references to them.

In case there is a problem in accessing these variables, `IllegalAccessException` is raised.

➔ ActivityInstrumentationTestCase2 class

This is the most commonly used class for writing Android test cases because it allows functional testing of a single `Activity` class. The class invokes the `InstrumentationTestCase.launchActivity()` method and uses the system infrastructure to create the `Activity` class to be tested. Such action is possible because this class has access to Instrumentation.

Figure 2.23 shows the UML class diagram of `ActivityInstrumentationTestCase2` and its closely related classes.

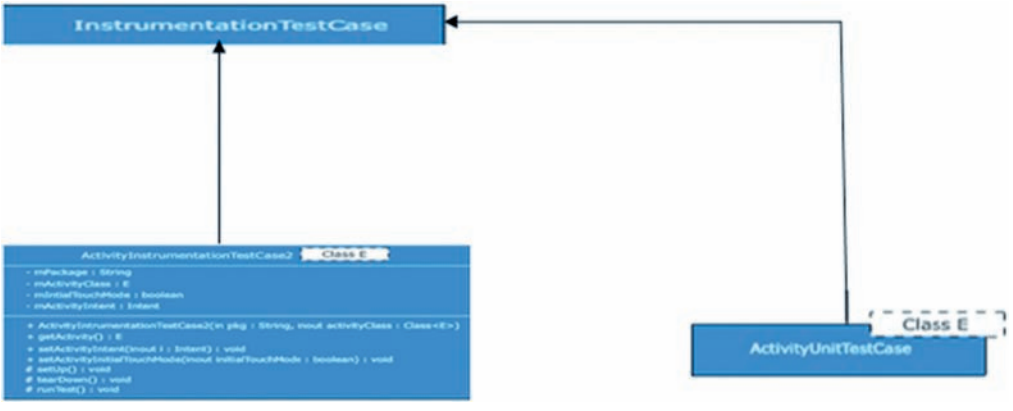


Figure 2.23: UML Class Diagram of ActivityInstrumentationTestCase2 and its Close Related Classes

Figure 2.23 shows that the `ActivityInstrumentationTestCase2` and `ActivityUnitTestCase` is derived from the `ActivityTestCase` class. Class template parameter **E**, as seen in figure 2.23, represents the Activity class. This enables the Activity to be manipulated and monitored after creation.

Suppose there is a requirement to provide a custom Intent to start an Activity, then `setActivityIntent(Intent intent)` method should be introduced before invoking `getActivity()` method. This test is very useful for testing user interaction through the interface as events can be introduced to simulate user behavior.

The syntax of the public non deprecated constructor for this class is as follows:

```
ActivityInstrumentationTestCase2(Class<E> activityClass)
```

The constructor can be invoked with the same `Activity` class that is used as the class template parameter.

- **setUp method**

  This method is used for initialization of test case fields and other fixture components that need initialization. Code Snippet 10 displays the use of the `setUp()` method.

  Code Snippet 10:

```
...
protected void setUp() throws Exception {
     junitTestClass = new JunitTestClass();
     mArg1 = 6;
     mArg2 = 3;

     super.setUp();
}
...
```

- **tearDown method**

  This method is used for cleaning up of whatever fields that was initialized in the `setup()` method. Code snippet 11 demonstrates the use of `teardown()` method.

  Code Snippet 11:

```
…
protected void tearDown() throws Exception {
super.tearDown();
}
...
```

- **testPreconditions method**

  This method checks the preliminary conditions so that the test could be executed correctly. Though it is not mandatory that this test will be executed before all other tests, it is advisable to group all preconditions tests under this custom name. Code snippet 12 demonstrates the use of the `testPreconditions()` method.

  Code Snippet 12:

```
...
public void testPreconditions(){
     assertNotNull(indexActivity);
     assertNotNull(indexInstrumentation);
     assertNotNull(indexLink);
     assertNotNull(indexMessage);
     assertNotNull(indexCapitalize);
}
...
```

  In the code snippet, only the **Not Null** values are checked. By testing the **Not Null** values, it can be confirmed that the Views were acquired using the specific IDs with the correct types. When **Not Null** values are unchecked they are assigned using the setup method.

2.4 Check Your Progress

1. Which of the following test method statements test a part of the application under test?

(A)	JUnit test	(C)	AndroidTestCase
(B)	Instrumentation	(D)	ApplicationTestCase

2. A _____ is a directory or an Eclipse project that creates the source code and manifest files for a test package.

(A)	test project	(C)	SDK tools
(B)	command-line tools	(D)	JUnit tests

3. One of the functions of Android instrumentation is to:

(A)	run an Android component independently of its normal cycle	(C)	automatically assign InstrumentationTestRunner
(B)	build files and manifests files and directory structure	(D)	test each part of an application

4. Which class is used to test a single activity within a normal system environment?

(A)	ActivityUnitTestCase	(C)	ActivityTestCase
(B)	SingleLaunchActivityTestCase	(D)	ActivityInstrumentationTestCase2

5. A class that is used for isolated testing of a single ContentProvider is _____.

(A)	ProviderTestCase2	(C)	ServiceTestCase
(B)	ActivityUnitTestCase	(D)	ApplicationTestCase

2.4.1 Answers

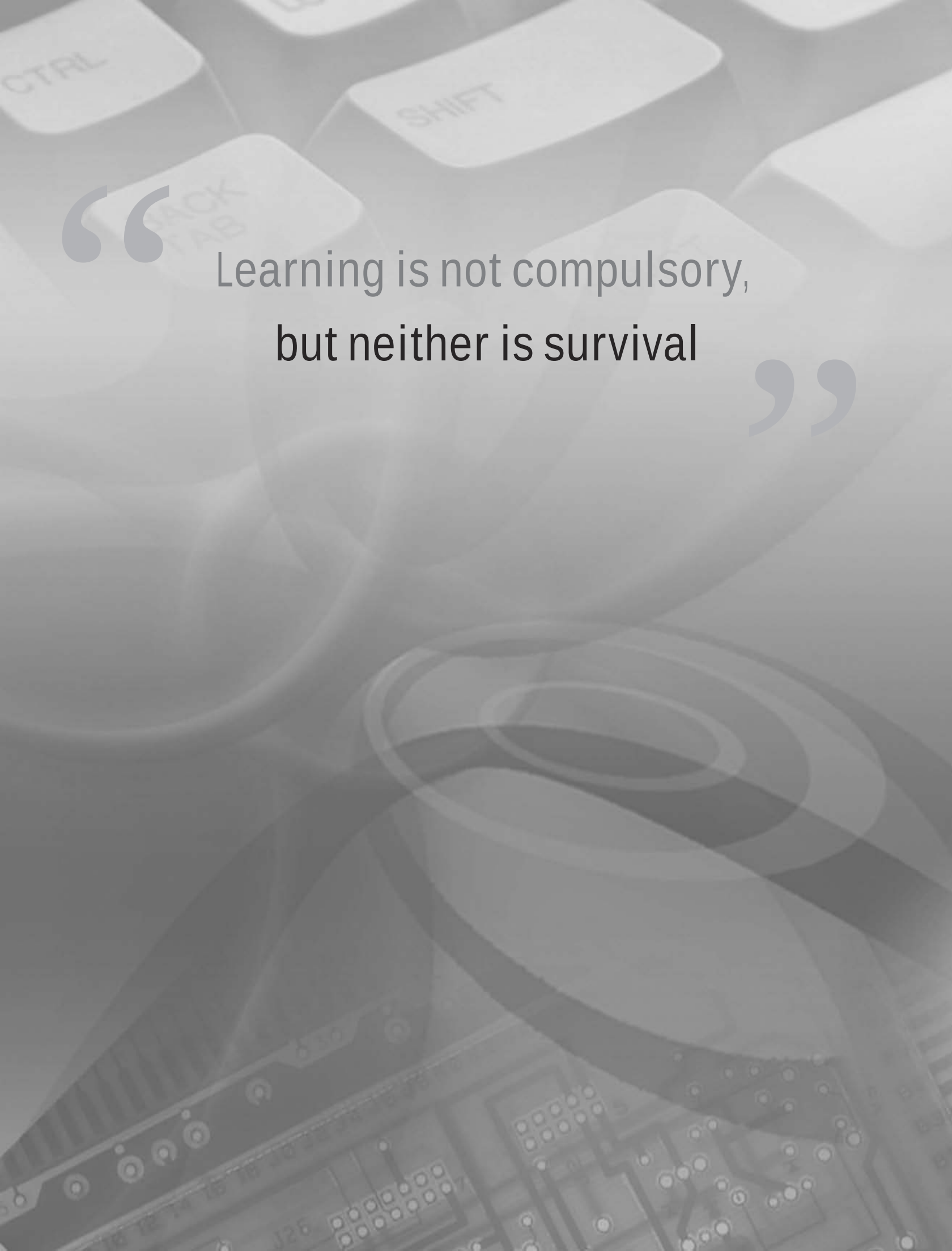
1.	A
2.	A
3.	A
4.	D
5.	A

Summary

- Android has a built-in testing framework that helps in testing every unit of an application.
- Android build and test tools assume that the test projects consists of an organized and standard structure of tests, test case classes, test packages, and test projects.
- The Android testing API is modeled on the JUnit API. It has an instrumentation framework and Android-specific testing classes.
- A JUnit can be called as a method that allows testing of each part of an application.
- The Android system has a set of control methods that enables an Android component to run independently of its normal cycle and determines how an Android loads an application/applications.
- ActivityInstrumentationTestCase2 is the most commonly used class for writing tests. It tests a single activity within a normal system environment.

Try it Yourself

Create an Android project that will perform the multiplication of two numbers. Create a test project for that application that will test the multiplication function. The test project will contain the relevant code for the `setUp()` and `tearDown()` methods.



“Learning is not compulsory,

but neither is survival”

Session - 3

Components Used for Creating Tests

Welcome to the session, **Components Used for Creating Tests**.

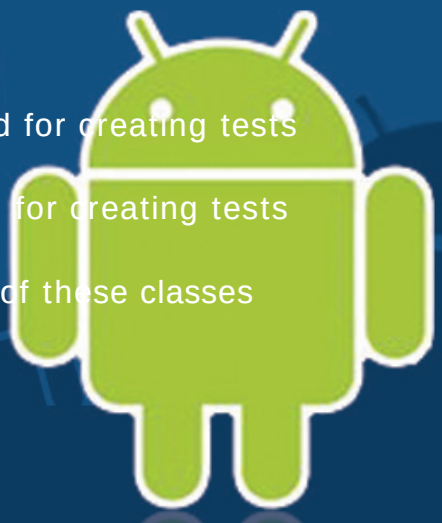
This session discusses about the various components that are required to create a test. These can be broadly divided into following classes namely,

- Assert class,
- TouchUtils class,
- Mock Object class, and
- TestCase base class.

This session will describe these classes along with their subclasses and examples.

In this session, you will learn to:

- Identify the different classes used for creating tests
- Explain the different classes used for creating tests
- Discuss the methods under each of these classes



3.1 Introduction

`Assert` class, `TouchUtils` class, `Mock object` class, and `TestCase` base class are some of the building blocks required to create tests.

Assert classes is the base class for all test case classes and are used for writing tests.

TouchUtils class extends from the `InstrumentationTestCase` and consists of a number of reusable methods for generating touch events.

MockObjects classes stub out the corresponding system object class and override methods to provide mock dependencies. Mock objects are duplicate objects that are called instead of the real domain objects to allow testing of units in isolation.

TestCase base class is the base class for all test cases in the `JUnit` framework and provides the basic methods in the JUnit framework.

3.2 Android Assertion Class

Assertion is a method that evaluates a condition and depending on the result of evaluation throws an exception and aborts the program or normally executes the program. Assertion methods can be used to display the result of tests as Android test case classes extend JUnit. It is more convenient to use and provides improved test performance.

The characteristics of the Android `Assert` class are as follows:

- ➔ Is a part of the JUnit API base class for all of the test case classes.
- ➔ Has several assertion methods that are used for writing tests. Assertion methods evaluate the actual results to an expected result value.
- ➔ Present the possible types of comparisons.
- ➔ Test the UI.
- ➔ Test a variety of conditions.
- ➔ Support different parameter types.
- ➔ Displays messages only when assertion fails.

Depending on the conditions that they check, they can be grouped into different sets as listed in table 3.1.

Methods	Description
`assertEquals`	Affirms that both objects passed as parameters are equal by invoking the `equals()` method of the object.
`assertFalse`	Affirms that the condition is false.
`assertNotNull`	Affirms that an object is not null.
`assertNotSame`	Affirms that both the objects do not refer to the same object.
`assertNull`	Affirms that an object is null.
`assertSame`	Affirms that both the objects refer to the same object.
`assertTrue`	Affirms that the condition is true.
`Fail`	Fails a test with a given/no message.

Table 3.1: Assert Methods and their Description

As the name indicates, each condition tested is identified by the method name. When any one of these conditions fails, the execution of the test is aborted and an `AssertionFailedError` is thrown.

The `fail()` method is used under the following two conditions:

- When there is a need to create a test that is not implemented immediately but is flagged for later use. In this condition the `fail()` method always fails and indicates the condition with a custom message as seen in code snippet 1.

  Code Snippet 1:

```
…
public void testNotdone(){
     fail("Not done yet");
}
…
```

- When there is need to test if a method has thrown an exception. In such a case the method invocation code is surrounded by a `try-catch` block, and the `fail()` method is forcefully invoked if the exception is not thrown. Code snippet 2 demonstrates the use of this condition.

Code Snippet 2:

```
…
public void testCheckShouldThrowException(){
try{
     HelloProjectActivity
     .hellomethodThatShouldThrowException();
     fail("Exception not thrown");
} catch(Exception ex){
     //do nothing
}
}
…
```

3.2.1 Using Custom Messages

All methods of the `Assert` class have an overloaded version which includes a custom `String` message. When an assertion fails, the custom messages are printed by the test runner instead of a default message. Custom messages help in identifying the failure and are often prescribed as the best practice. Code snippet 3 demonstrates the use of custom message.

Code Snippet 3:

```
…
public void testMax(){
     final int x=1;
     final int y=2;
     final int anticipated=y;
     final int original=Math.max(x, y);
     assertEquals("Exception"+anticipated+"but was"+original,
     anticipated, original);
     }
…
```

3.2.2 Importing Static Methods

Some assertion methods need specific import statements to improve the readability of the tests, though the basic assertion methods extend from the `Assert` base class. A specific practice is followed to import assert methods from the corresponding classes. The developer can either use the static assert method prefixed with the package and class name or use the static import statements. Code snippet 4 demonstrates the use of the static assert method prefixed with the package and the class name.

Code Snippet 4:

```
…
public void testAlignment(){
     final int edge = 0;
       ...
     android.test.ViewAsserts.assertRightAligned(
       indexMessage, indexCapitalize, edge);
   }
…
```

Code snippet 5 demonstrates the use of static import statement for importing the static method and then using the static method within another method in the class.

Code Snippet 5:

```
import static android.test.ViewAsserts.assertRightAligned;


  public void testAlignment(){
         final int edge = 0;
     assertRightAligned(indexMessage, mindexCapitalize, edge);
  }
…
```

Concepts

To work with static import methods, add it to the **Favorites** list of Eclipse IDE. To add the static imports method, perform the following steps:

1. Click **Window** → **Preferences** as shown in figure 3.1. This will display the **Preferences** dialog box.

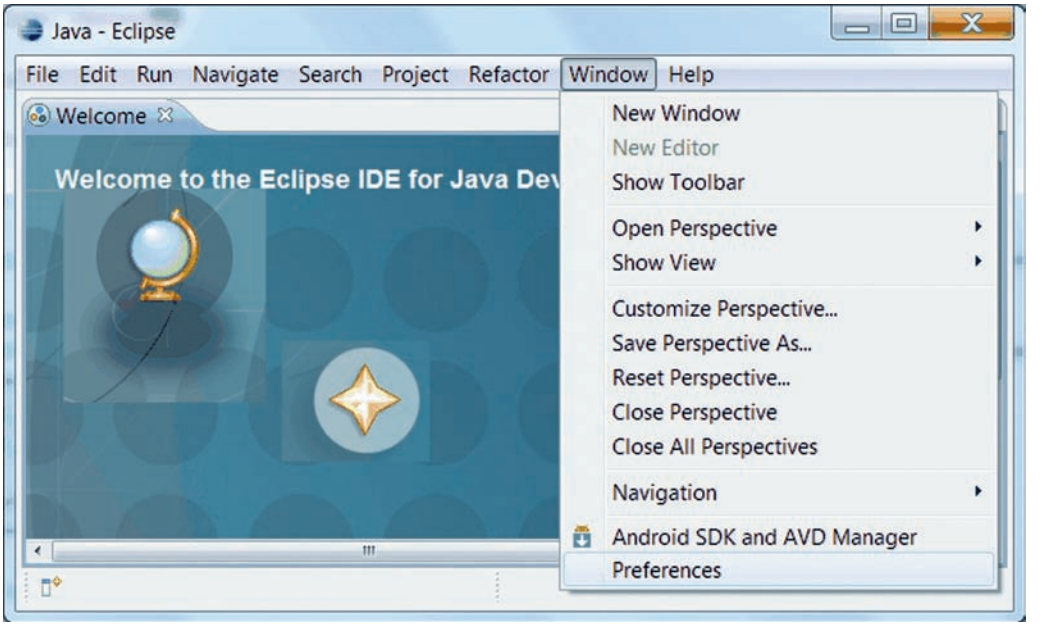


Figure 3.1: Selecting Window → Preferences

2. In the left pane of the **Preferences** dialog box, select **Java → Editor → Content Assist → Favorites** to view the **Favorites** pane as shown in figure 3.2.

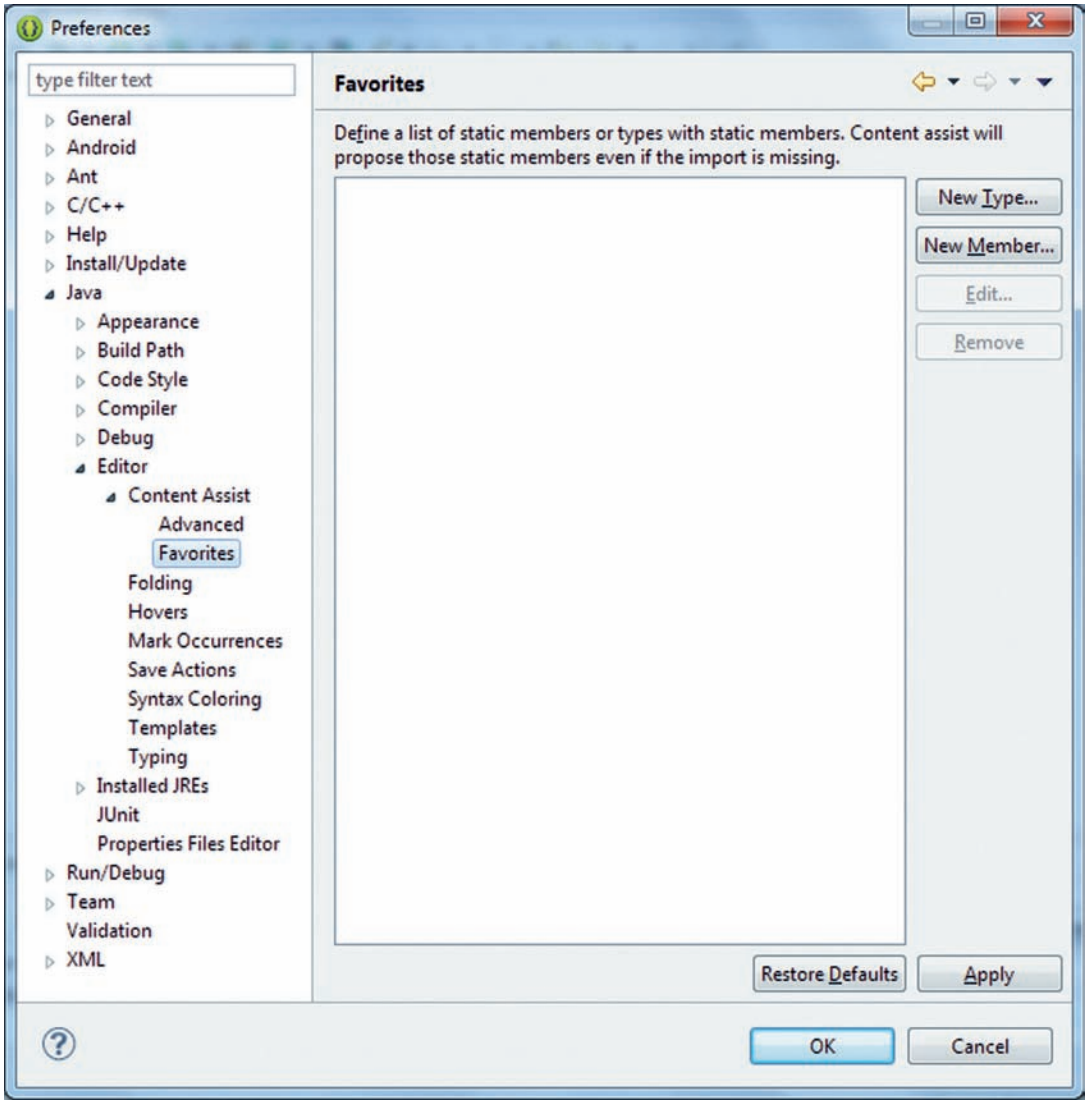


Figure 3.2: Viewing the Favorites Pane in the Preferences Dialog Box

Concepts

3. Click **New Type** to display the **New Type Favorite** dialog box as shown in figure 3.3.

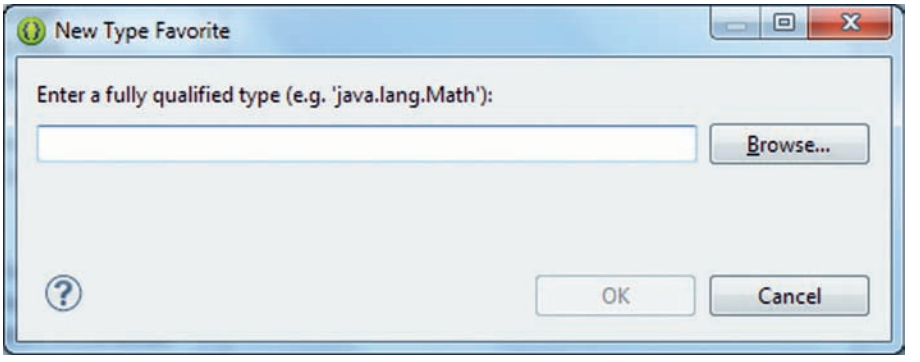


Figure 3.3: New Type Favorite Dialog Box

4. Type **`android.test.MoreAsserts`** to add the assert class.

5. Follow the steps again to add **`android.test.ViewAsserts`** class as shown in figure 3.4.

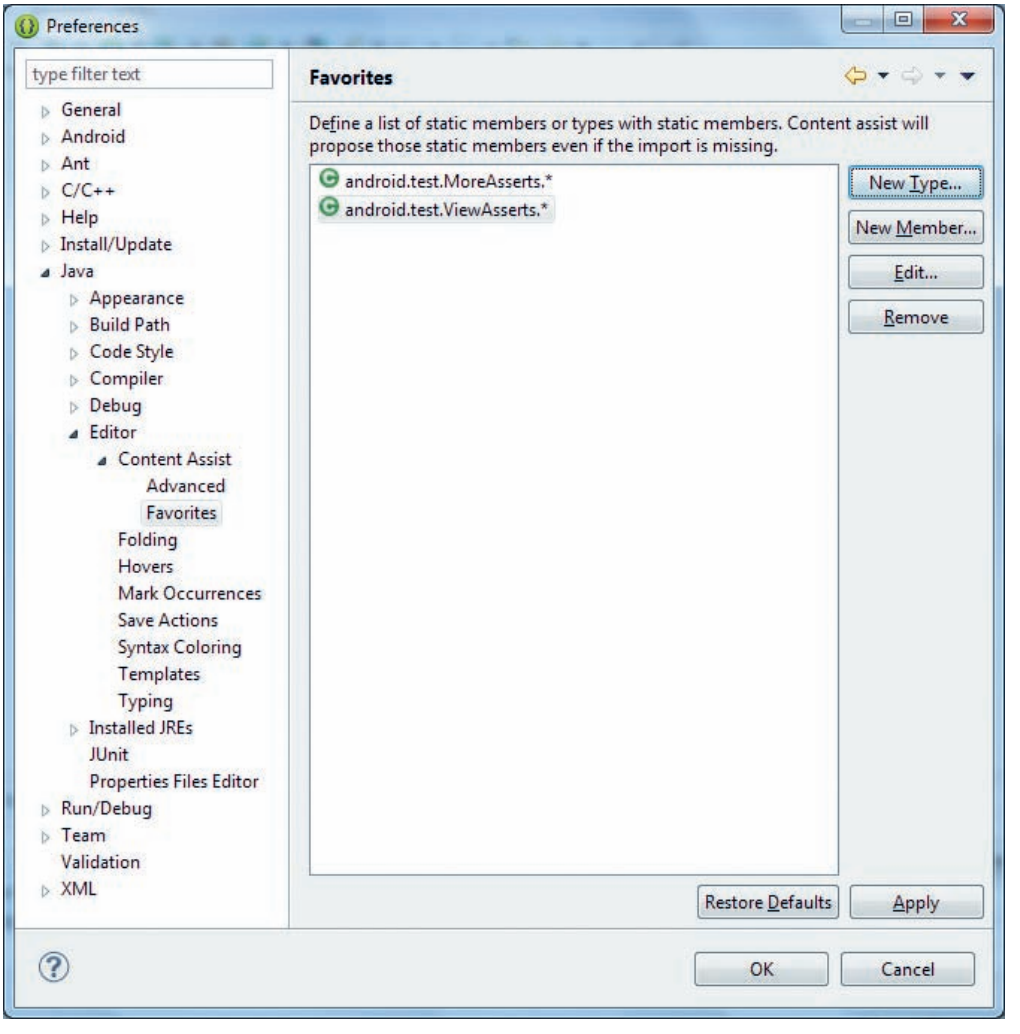


Figure 3.4: Adding Classes to the Favorites List

3.2.3 View Asserts Class Methods

Besides the JUnit `Assert` class that is used to test simple conditions or objects, Android provides the `ViewAsserts` and `MoreAsserts` classes for testing complex conditions. These classes are present in the `android.test` package. `ViewAsserts` class provides advanced methods that are used for testing user interfaces. The `android.test.ViewAsserts` has different assertion methods that test not only the views on screen but also their absolute and relative positions on the screen. Table 3.2 lists some of the methods of the class along with their description.

Methods	Description
`static void assertBaselineAligned (View v1, View v2)`	The method checks that the two views are baseline aligned with their baselines on the same y location.
`assertBottomAligned (View v1, View v2)`	The method checks that the two views have their bottom aligned with their bottom edges on the same y location.
`assertGroupContains (ViewGorup Parent, View child)`	The method affirms that the specified group contains a specific child once and only once.
`AssertGroupNotContains (ViewGorup Parent, View child)`	The method checks that the specified group does not contain a specific child.
`assertHasScreenCoordinates (View orgin,View view, int x, int y)`	The method tests whether there is a particular x and y position for each view on the visible screen.
`assertHorizontalCenterAligned (View reference, View Test)`	The method checks that the test view is horizontally center aligned in relation to the reference view.
`assertOffScreenAbove (View orgin, View view)`	The method reviews that the specified view is visible above the visible screen.
`assertOffScreenBelow (View orgin, View view)`	The method reviews that the specified view is visible below the visible screen.
`assertOnScreen (View orgin, View view)`	The method affirms that a view is visible on the screen.
`assertRightAligned (View first, View Second, int margin)`	The method asserts that two views are right-aligned with their right edges on the same x location. An optional margin can also be given.
`assertTopAligned (View First, view Second, int margin)`	The method tests that two views are top-aligned with their top edges on the same y location. An optional margin can also be specified.
`assertVerticalCenterAligned (View reference, View Test)`	The method checks whether the test view is vertically center aligned in relation to the reference view.

Table 3.2: ViewAsserts Methods for Different Conditions

For further understanding of the role of ViewAsserts class, consider the code snippet 6 where ViewAsserts class is used to test the user interface layout.

Code Snippet 6:

```
…
@Test
public final void testFontSizes() {
        final float expected = 24.0f;
        assertEquals(expected, textView1.getTextSize(), 0);
        assertEquals(expected, textView2.getTextSize(), 0);
      }

@Test
public final void testMargins() {
          LinearLayout.LayoutParams lp;
          final int expected = 6;
          lp = (LinearLayout.LayoutParams) Layout1.getLayoutParams();
      assertEquals(expected, lp.leftMargin);
      assertEquals(expected, lp.rightMargin);
      lp = (LinearLayout.LayoutParams) Layout2.getLayoutParams();
      assertEquals(expected, lp.leftMargin);
      assertEquals(expected, lp.rightMargin);
      }
…
```

3.2.4 MoreAsserts Class Methods

MoreAsserts class is responsible for regular expression matching as they contain methods such as assertContainsRegex (String, String). Similar to ViewAsserts, this class contains additional assertion methods. Table 3.3 lists some of the methods of this class along with their description.

Methods	Description
assertAssignableFrom	The method check that class expected is assignable from the object actual.
assertContainsRegex	The method tests the match between an expected Regex and any substring of the specified String. When there is no match the specified message is failed.
assertContainsInAnyOrder	The method reviews whether the specified Iterable contains the expected elements, but in any order.
assertContainsInOrder	The methods affirms that the specified Iterable contains the expected elements, in the same order.
assertEmpty	The method asserts that an Iterable is empty.
assertEquals	The method confirms that the JUnit asserts does not cover some Collections.
assertMatchesRegex	The method checks whether there is an exact match between the specified Regex and the String. When there is no match the provided message is failed.
assertNotContainsRegex	The method affirms that there is no match between the specified Regex and any substring of the specified String, fails with provided message if it does.
assertNotEmpty	The method confirms that some Collections not covered in JUnit asserts are not empty.
assertNotMatchesRegex	The method tests that there is no exact match between the specified Regex and the specified String, fails with the provided message if it does.
checkEqualsAndHashCodeMethods	The method tests equals( ) and hashCode( ) results at once. Tests that equals() applied to both objects matches the specified result.

Table 3.3: Methods of MoreAsserts Class

3.3 Using TouchUtils Class

The TouchUtils class provides one of the simplest ways to stimulate different kinds of touch events in the UI. This class is used with the InstrumentationTestCase or ActivityInstrumentationTestCase2 for interaction with the application through a touch screen. It consists of a number of reusable methods for generating touch events.

TouchUtils class supports the following reusable methods.

- Clicking a View and releasing.
- Tapping on a View and releasing.
- Long click a View.
- Dragging the screen.
- Dragging Views.

Table 3.4 lists the methods of the TouchUtils class.

Method	Description
clickView(InstrumentationTestCase test, View v)	The method is invoked for simulating touching at the center of a view and releasing it.
Drag((InstrumentationTestCase test, float fromX, float toX, float fromY, float toY, int stepCount)	The method is invoked for simulating touching at a specific location to dragging it to a new location.
dragQuarterScreenDown(InstrumentationTestCase test, Activity activity)	The method is invoked for simulating touching at the center of the screen and dragging it to one quarter of the way down.
dragQuarterScreenUp(InstrumentationTestCase test, Activity activity)	The method is invoked for simulating touching at the center of the screen and dragging it to one quarter of the way up.
dragViewTo(InstrumentationTestCase test, View v, int gravity, int toX, int toY)	The method is invoked for simulating touching a view to drag to a specified location such as dragViewToX or dragViewToY for viewing.
dragViewToBottom(InstrumentationTestCase test, Activity activity, View v)	The method is invoked for simulating touching the center of a view and drags it to the bottom of the screen.
dragViewToTop(InstrumentationTestCase test, View v)	The method is invoked for simulating touching the center of a view and drags it to the top of the screen.
longClickView(InstrumentationTestCase test, View v)	The method is invoked for simulating touching the center of a view, holding for a long press, and then releasing.

Method	Description
`scrollToTop((InstrumentationTestCase test, Activity activity, ViewGroup v)`	The method is invoked for simulating scroll a ViewGroup to the top by repeatedly invoking dragQuarterScreenDown.
`tapView(InstrumentationTestCase test, View v)`	The method is invoked for simulating touching the center of a view and releasing quickly (before the tap timeout).
`touchAndCancelView(InstrumentationTestCase test, View v)`	The method is invoked for simulating touching the center of a view to cancel.

Table 3.4: Methods of TouchUtils class

3.4 Using Mock Objects

Mock objects, as the name suggests are duplicate objects that are called instead of the real domain objects to allow testing of units in isolation. They isolate the tests by stubbing out or overriding them. Some of the mock objects available in Android are as follows: `Context` objects, `ContentProvider` objects, `ContentResolver` objects, `Service` objects, and mock `Intent` objects. Mock objects are used:

- to detach unit from dependencies.
- for repetitive testing.
- to monitor interactions.
- to run tests independently.

For successfully using mock objects in tests, there is a need to provide dependencies to compile tests. The Android testing frameworks supports a number of the mock object classes in the `android.test.mock` and the `android.test` packages.

3.4.1 Simple Mock Object Class

These simple mock object classes exhibit a simple and useful mock strategy. Their objective is to provide stubbed-out versions of the corresponding system object class. When implementing the methods, override only those methods that are to be used to provide mock dependencies. The methods throw an `UnsupportedOperationException` when invoked and needs to be overwritten. Table 3.5 lists the simple mock object classes.

Class	Description
MockApplication	The methods in MockApplication are non-functional and throw UnSupportedOperationException. This class contains methods such as onConfigurationChanged(), onCreate(), onTerminated().
MockContext	A mock Context class is used to force mocks or monitors in to the classes that are under testing.
MockContentProvider	A mock object for ContentProvider.
MockCursor	MockCursor class is used to separate the test code from the implementation of real cursor.
MockDialogInterface	It is a mock DialogInterface class implementation.
MockPackageManager	A mock implementation of PackageManager class.
MockResources	A mock Resources class.

Table 3.5: Simple Mock Objects Class

3.4.2 Resolver Mock Objects/ The MockContentResolver Class

MockContentResolver class isolates the test code by stubbing the normal system resolver framework. MockContentResolver uses its own internal table to find a content provider for a given authority string. All methods in the MockContentResolver class have a non-functional approach and throws the UnsupportedOperationException when used. The reason for this is that the MockContentResolver isolates the test code from the real content system.

Providers can be added to the MockContentResolver using the method addProvider(String, ContentProvider). Using the addProvider(String, ContentProvider) method, it is possible to perform the following:

- ➔ Add a mock content provider with an authority.
- ➔ Make a representative of the real provider but use test data in it.
- ➔ Set a provider with an authority to null.

3.4.3 Android Context Class

There are two Context classes in Android. They are, IsolatedContext and RenamingDelegatingContext classes present in the android.test package. Both these classes are used for testing. Table 3.6 shows the two Context class and their functions.

Class	Description	Function
`IsolatedContext`	Associates an isolated Context, file, directory, and database operations.	Uses the Context in a test area by creating a stub code to respond to system calls.
`RenamingDelegatingContext`	Associate a Context where most functions are run by an existing Context except for file and database functions which is taken care by the IsolatedContext.	Uses test directory to name special files and directories. Name can be assigned automatically by the constructor or can be controlled by the tester.

Table 3.6: Context Class

3.5 Using the TestCase Base Class

`TestCase` base class is the base class for all test cases in the JUnit framework. It provides the basic methods in the JUnit framework. It is seen in figure 3.5 that the test cases extend either directly from `TestCase` or from one of the subclasses of `TestCase`.

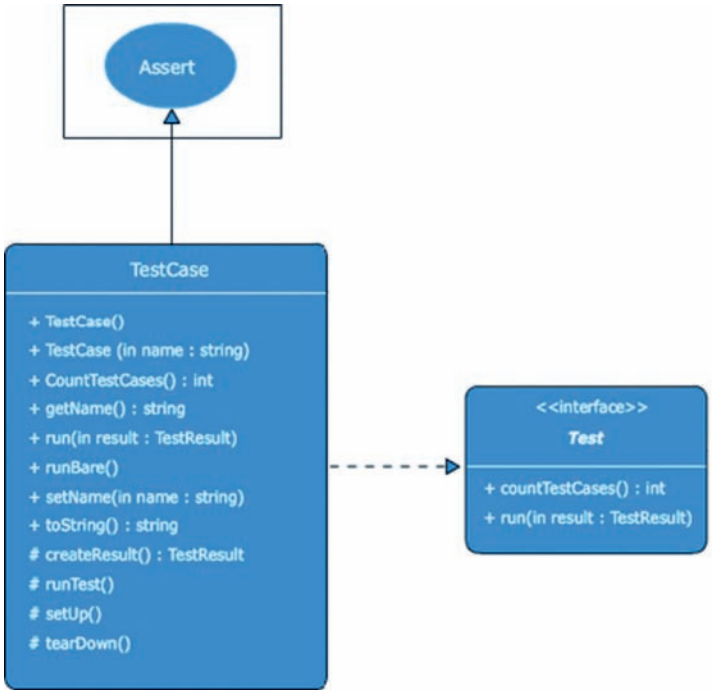


Figure 3.5: UML Diagram for TestCase Base Class

Sometimes, based on the test runner that is used, a constructor is invoked. The no argument constructor is the only default constructor that is required for all tests. The constructor is also used for serialization. Code snippet 7 demonstrates the use of the default constructor.

Code Snippet 7:

```
public class AddNumberTestCase extends TestCase {
     public AddNumberTestCase(){
         this("AddNumberTestCase Default Name");
     }
     public AddNumberTestCase(String name){
         super(name);
     }
}
```

The function of this constructor is to name itself as an argument to provide it to the other constructor. This constructor appears in test reports and is helpful to identify failed tests.

➔ `setName()` method

In cases where some classes extending the test case do not provide a given name to the constructor the `setName(String name)` method is used.

3.6. Methods of AndroidTestCase

As discussed in Session 2, the `AndroidTestCase` is the base class for general purpose Android test cases. Some of the methods of this class are discussed in detail.

➔ `assertActivityRequiresPermission()`

The `assertActivityRequiresPermission()` is an assertion method that checks whether the launch of a particular Activity is permission protected.

The syntax of this method is as follows:

Syntax:

```
public void assertActivityRequiresPermission (String packageName, String className, String permission)
```

where,

- `packageName` - represents a String that accepts the package name of the `Activity` to launch.
- `className` - refers to a String variable that accepts the classname of the `Activity` to launch.
- `permission` - represents a string variable that contains the name of the permission to query.

This method affirms that a particular permission protects the start of a given activity. Such action takes places when an activity is started and validated by invoking the exception, `SecurityException` that displays the permission in its error message.

Code snippet 8 checks the need for the `android.Manifest.permission.WRITE_EXTERNAL_STORAGE` permission which is required to write to external storage in the Activity class named `MyFirstProjectActivity`.

Code Snippet 8:

```
…
public void testActivityPermission(){
      final String AP = "com.example.myfirstproject";
      final String ACTION = AP + ".MyFirstProjectActivity";
      final String CONSENT =
          android.Manifest.permission.WRITE_EXTERNAL_STORAGE;
      assertActivityRequiresPermission(AP, ACTION, CONSENT);
}
…
```

➔ `assertReadingContentUriRequiresPermission()`

This assertion methods checks whether reading from a specific URI has the required permission as stated by the parameters.

The syntax of the method is:

Syntax:

```
public void assertReadingContentUriRequiresPermission (Uri uri, String permission)
```

where,

- `uri` - represents the URL where permission is required to query.
- `permission` - represents a string containing the permission to query.

The assertion method is validated when a `SecurityException` is thrown containing the specified permission.

Code snippet 9 demonstrates the use of this method.

Code Snippet 9:

```
...
public void testReadingContacts(){
     final Uri URI = ContactsContract.AUTHORITY_URI;
     final String CONSENT =
          android.Manifest.permission.READ_CONTACTS;
     assertReadingContentUriRequiresPermission(URI,CONSENT);
  }
…
```

In the code, the test method reads the contacts and verifies whether the correct `SecurityException` is generated.

➔ `assertWritingContentUriRequiresPermission()`

This assertion methods checks whether inserting into a specific URI has the required permission as stated by the parameters. The syntax of this method is as follows:

Syntax:

```
public void assertWritingContentUriRequiresPermission (Uri uri, String permission)
```

where,

- `uri` - represents the URI that requires the permission to query.
- `permission` - represents a string variable with the permission to query.

The assertion method is validated when a `SecurityException` is thrown containing the specified permission.

Code Snippet 10 demonstrates the use of the method.

Code Snippet 10:

```
…
public void testWritingContacts(){
     final Uri URI = ContactsContract.AUTHORITY_URI;
     final String CONSENT =
          android.Manifest.permission.WRITE_CONTACTS;
     assertWritingContentUriRequiresPermission(URI,CONSENT);
  }
…
```

In the code, a test method is created that writes to contacts and verifies whether the correct `SecurityException` is generated.

3.7 Component-specific Test Cases

The component-specific test classes are responsible for testing specific components with methods for fixture setup and teardown and component life cycle control. They are also required to provide methods for setting up of mock objects. The component-specific test classes are as follows:

➔ **Activity Testing:** This testing relies on the Android instrumentation framework. `InstrumentationTestCase` is the base class for activity testing. It supports the following functions:

a. Instrumentation helps in controlling the lifecycle of an activity being tested. It can start, pause, and destroy the activity using methods of the test case classes.

b. Instrumentation helps in controlling the test environment and separates it from the production systems by creating mock objects and using them to run the activity being tested. It is also possible to customize Intents and begin an activity using them.

c. Instrumentation can be used for user interface interaction by sending keystrokes or touch events directly to the UI of the activity being tested.

The two main subclasses of activity testing are as follows:

- `ActivityInstrumentationTestCase2` - is used for functional testing of one or more Activities in an application. It runs the activities in a normal system infrastructure. It uses the standard system context and runs the activities in a normal instance of the application under test.
- `ActivityUnitTestCase` - is used for testing a single activity in isolation. Use it to perform unit testing of methods that do not interact with Android.

  Along with these, the following are the different test cases:

  - SingleLaunchActivityTestCase, used to test a single activity in an environment that do not change from test to test.
  - Mock objects and activity testing, creates a hidden mock application object. This object is used as an application under test.
  - Assertions for activity testing, is used to verify the positions and alignments of view objects.

When there is a need to test an activity that runs in non-standard mode, the `SingleLaunchActivityTestCase` is invoked. This class is a convenience class that is used for testing a single activity where the test environment need not be reset from test to test.

➔ **Content Provider Testing:** Content providers are an essential part of the Android API. The main focus of the provider testing API is to supply an isolated test environment. Their function is to store and retrieve data and make them available across applications.

ProviderTestCase2 is the base test class for content providers. The primary function of this class is its initialization to create an isolated environment for testing. It uses the `IsolatedContext` and `MockContentResolver`, to complete its function.

- `IsolatedContext`: This object allows file and database operations and these operations take place in directory, local to the device.
- `MockContentResolver`: Uses as a resolver for the test.

➔ **Service Testing:** Service objects run in isolation in a separate testing framework provided by Android.

`ServiceTestCase` is the test case class for `Service` objects. It is a part of the JUnit `TestCase` class. It has methods to test application permissions and control the application and `Service` being tested.

`ServiceTestCase` delays initialization until the `ServiceTestCase.startService()` or `ServiceTestCase.bindService()` is invoked. By doing this, it sets up the mock objects before `Service` is started.

The `setUp()` method for `ServiceTestCase` is called before each test to set up the test fixture. It makes a copy of the current system Context before being inspected by any test methods. The methods `setApplication()` and `setContext(Context)` methods are used to set a mock Context or mock Application (or both) for the Service, to isolate the test from the rest of the system.

3.8 Function of the ApplicationTestCase

The `ApplicationTestCase` tests the setup and teardown of applications. The main function of this class of objects is to store the global information. All units in an application package use the stored information. Also, it verifies whether the `<application>` element in the manifest file is correctly set up.

3.9 The InstrumentationTestCase Class

The system initiates instrumentation before running any application code. Instrumentation examines the interaction between the system and the application. The interaction is specified in the **AndroidManifest.xml** file under the `<instrumentation>` tag.Code snippet 12 demonstrates the use of **AndroidManifest.xml** file in a test.

Code Snippet 12:

```
<instrumentation
android:targetPackage="com.example.helloandroidproject"
android:name="android.test.InstrumentationTestRunner"
     android:label="HelloAndroidProject Tests"/>
```

The `InstrumentationTestCase` class may be a direct or indirect base class for the different test cases that have an approach to Instrumentation. This class is present in the `android.test` package. This class extends from `junit.framework.TestCase` class which in turn extends from `junit.framework.Assert` class. Figure 3.6 displays the UML class diagram of the `InstrumentationTestCase` class.

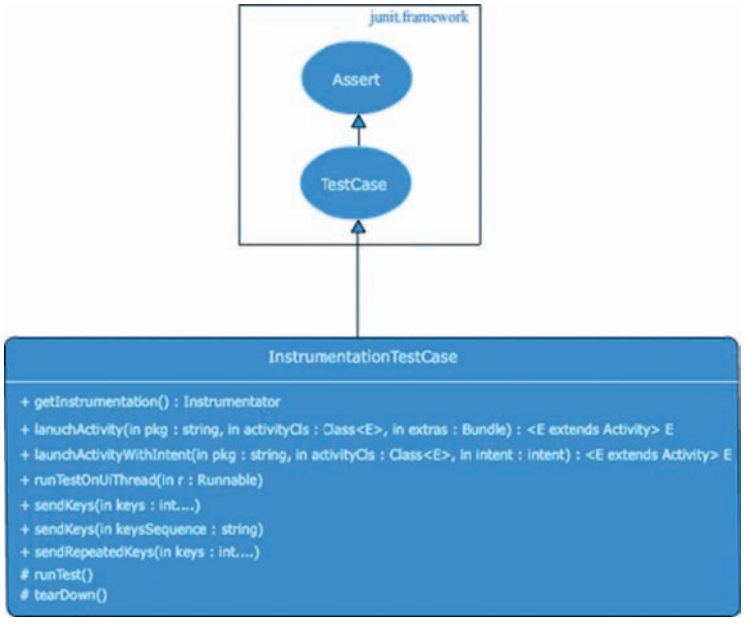


Figure 3.6: UML Class Diagram of InstrumentationTestCase

Related Classes

Some of the most important direct and indirect subclasses that have access to Instrumentation are as follows:

- ActivityTestCase
- `ProviderTestCase2<E extends ContentProvider>`
- `SingleLaunchActivityTestCase<E extends Activity>`

Concepts

- SyncBaseInstrumentation
- ActivityInstrumentationTestCase2<E extends Activity>
- ActivityUnitTestCase<E extends Activity>

3.9.1 Methods of InstrumentationTestCase Class

Some of the methods of the class are as follows:

- **launchActivity**

  This is an utility method that is used to launch an `Activity`. The syntax of this method, when an intent is not specified is as follows:

  Syntax:

```
public final T launchActivity( String pkg, Class<E> activityCls,
Bundle extras)
```

  where,

  `pkg` – refers to the package name hosting the activity that is to be launched which is mentioned in the **AndroidManifest.xml** file.

  `activityCls` – refers to the activity class name to be launched.

  `extra` – refers to the optional stuff to pass to the activity.

- **launchActivityWithIntent**

  This method will launch an `Activity` with a custom Intent. The syntax of this method, is as follows:

  Syntax:

```
public final T launchActivityWithIntent( String pkg, Class<E>
activityCls, Intent intent)
```

  where,

  `pkg` – refers to the package name hosting the activity that is to be launched which is mentioned in the **AndroidManifest.xml** file.

  `activityCls` – refers to the activity class name to be launched.

  `intent` – refers to the intent to launch with.

➔ sendKeys and sendRepeatedKeys

The `sendKeys` and `sendRepeatedKeys` are both utility methods that are used to interact with qwerty-based keyboards or DPAD buttons while testing the UI of the Activity class. They are used for the following functions:

- to send keys.
- to complete fields.
- to select shortcuts or.
- to navigate between different components.

Let us take an example of simple Calculator where developer uses `sendKeys` and `sendRepeatedKeys`.

Create an Android project named **Calculator** and test it through Android Test Project

To create an Android project, perform the following steps:

1. Create an Android Project named **`Calculator`**.

   Activity name: **`CalculatorActivity`**.

   Layout file: **`activity_calculator`**.

   Package: **`com.example.calculator`**.

2. Create a class named **`Calculator`** within the package, **com.example.calculator**.

3. Modify the code in the **`Calculator`** class as shown in code snippet 12.

   Code Snippet 12:

```
package com.example.calculator;


public class Calculator {


    /**
     * Left operand of the operation to be performed.
     */
    private double leftOperand;
    /**
     * Right operand of the operation to be performed.
     */
    private double rightOperand;
```

```
/**
 * Constructs a Calculator object by initializing the
 * leftOperand and rightOperand with the given values.
 *
 * @param leftOperand
 *        Left operand.
 * @param rightOperand
 *        Right operand.
 */
public Calculator(double leftOperand, double
rightOperand) {
    this.leftOperand = leftOperand;
    this.rightOperand = rightOperand;
}

/**
 * Adds the leftOperand to the rightOperand.
 *
 * @return The sum of the two operands.
 */
public double add() {
    return leftOperand + rightOperand;
}

/**
 * Subtracts the rightOperand from the leftOperand.
 *
 * @return The subtraction of the two operands.
 */
public double subtract() {
    return leftOperand - rightOperand;
}
```

```
    /**
     * Multiply the leftOperand to the rightOperand.
     *
     * @return The multiplication of the two operands.
     */
    public double multiply(){
        return leftOperand * rightOperand;
    }

    /**
     * Divides the leftOperand to the rightOperand.
     *
     * @return The division of the two operands.
     */
    public double divide(){
        if(rightOperand == 0){
            throw new ArithmeticException("right operand
should not be zero!");
        }
        return leftOperand / rightOperand;
      }

}
```

4. Modify the code in **activity_calculator.xml** file as shown in code snippet 13.

Code Snippet 13:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/
apk/res/android"
    android:id="@+id/linear"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
```

```
<TableLayout
    android:id="@+id/table"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:stretchColumns="1">

    <TableRow>

        <TextView
            android:id="@+id/leftOperand_label"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:padding="3dip"
            android:text="Left Operand"
            android:textStyle="bold">
        </TextView>

        <EditText
            android:id="@+id/leftOperand"
            android:nextFocusDown="@+id/rightOperand"
            android:numeric="decimal"
            android:padding="3dip"
            android:scrollHorizontally="true"
            android:singleLine="true">
        </EditText>
    </TableRow>

    <TableRow>

        <TextView
            android:id="@+id/rightOperand_label"
```

```
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:padding="3dip"
                android:text="Right Operand"
                android:textStyle="bold" >
            </TextView>

            <EditText
                android:id="@+id/rightOperand"
                android:nextFocusDown="@+id/plus"
                android:padding="3dip"
                android:scrollHorizontally="true"
                android:singleLine="true" >
            </EditText>
        </TableRow>
    </TableLayout>

    <LinearLayout
        android:id="@+id/buttons_linear"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >

        <Button
            android:id="@+id/plus"
            android:layout_width="0px"
            android:layout_height="wrap_content"
            android:layout_gravity="center_horizontal"
            android:layout_weight="1"
            android:nextFocusRight="@+id/minus"
            android:padding="6dip"
```

```
            android:text="+" >
        </Button>
        <Button
            android:id="@+id/minus"
            android:layout_width="0px"
            android:layout_height="wrap_content"
            android:layout_gravity="center_horizontal"
            android:layout_weight="1"
            android:nextFocusRight="@+id/multiply"
            android:padding="6dip"
            android:text="-" >
        </Button>
        <Button
            android:id="@+id/multiply"
            android:layout_width="0px"
            android:layout_height="wrap_content"
            android:layout_gravity="center_horizontal"
            android:layout_weight="1"
            android:nextFocusRight="@+id/divide"
            android:padding="6dip"
            android:text="*" >
        </Button>
        <Button
            android:id="@+id/divide"
            android:layout_width="0px"
            android:layout_height="wrap_content"
            android:layout_gravity="center_horizontal"
            android:layout_weight="1"
```

```
            android:nextFocusRight="@+id/divide"
            android:padding="6dip"
            android:text="/" >
        </Button>
        <Button
            android:id="@+id/clear"
            android:layout_width="0px"
            android:layout_height="wrap_content"
            android:layout_gravity="center_horizontal"
            android:layout_weight="1"
            android:nextFocusRight="@+id/divide"
            android:padding="6dip"
            android:text="C" >
        </Button>
    </LinearLayout>
    <TextView
        android:id="@+id/result"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:gravity="right"
        android:padding="20dip"
        android:text="Result"
        android:textStyle="bold" >
    </TextView>
</LinearLayout>
```

5. Modify the code in **`CalculatorActivity`** class as shown in code snippet 14.

Code Snippet 14:

```
package com.example.calculator;


import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;


public class CalculatorActivity extends Activity implements
OnClickListener {


    private EditText leftOperand;
    private EditText rightOperand;
    private TextView result;
    private Calculator calculator;


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_calculator);


        leftOperand = (EditText) findViewById(R.
        id.leftOperand);
        rightOperand = (EditText) findViewById(R.
        id.rightOperand);
```

```
        result = (TextView) findViewById(R.id.result);


        Button plus = (Button) findViewById(R.id.plus);
        plus.setOnClickListener(this);
        Button minus = (Button) findViewById(R.id.minus);
        minus.setOnClickListener(this);
        Button multiply = (Button) findViewById
(R.id.multiply);
        multiply.setOnClickListener(this);
        Button divide = (Button) findViewById
(R.id.divide);
        divide.setOnClickListener(this);
        Button clear = (Button) findViewById(R.id.clear);
        clear.setOnClickListener(this);
    }


    @Override
    public void onClick(View view) {
        double leftOp = Double.parseDouble(leftOperand.
getText().toString());
        double rightOp = Double.parseDouble(rightOperand.
getText().toString());
        calculator = new Calculator(leftOp, rightOp);


        if (view.getId() == R.id.plus) {
            result.setText("" + calculator.add());
        } else if (view.getId() == R.id.minus) {
            result.setText("" + calculator.subtract());
        } else if (view.getId() == R.id.multiply) {
            result.setText("" + calculator.multiply());
        } else if (view.getId() == R.id.divide) {
            result.setText("" + calculator.divide());
```

Concepts

```
        }else if(view.getId()==R.id.clear){
            leftOperand.setText("");
            rightOperand.setText("");
            result.setText("");
            leftOperand.requestFocus();
        }
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu){
        // Inflate the menu; this adds items to the action bar
if it is present.
        getMenuInflater().inflate(R.menu.calculator,
menu);
        return true;
    }

}
```

6. Clean the project to rebuild and execute it: right-click **Calculator** and select **Run As → Android Application**.

Figure 3.7 displays the output of the application.

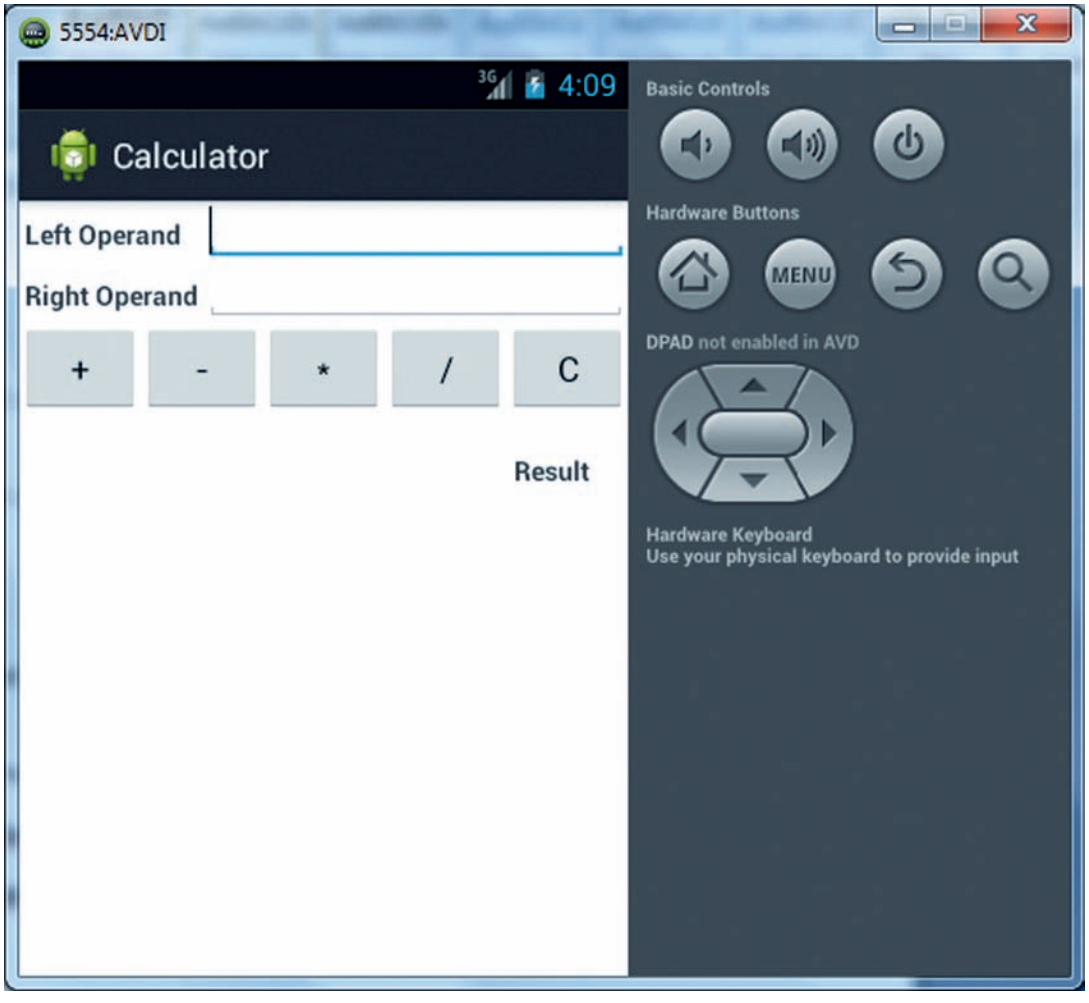


Figure 3.7: Calculator Application

Figure 3.8 displays the output after the values and the operand "+" have been clicked.

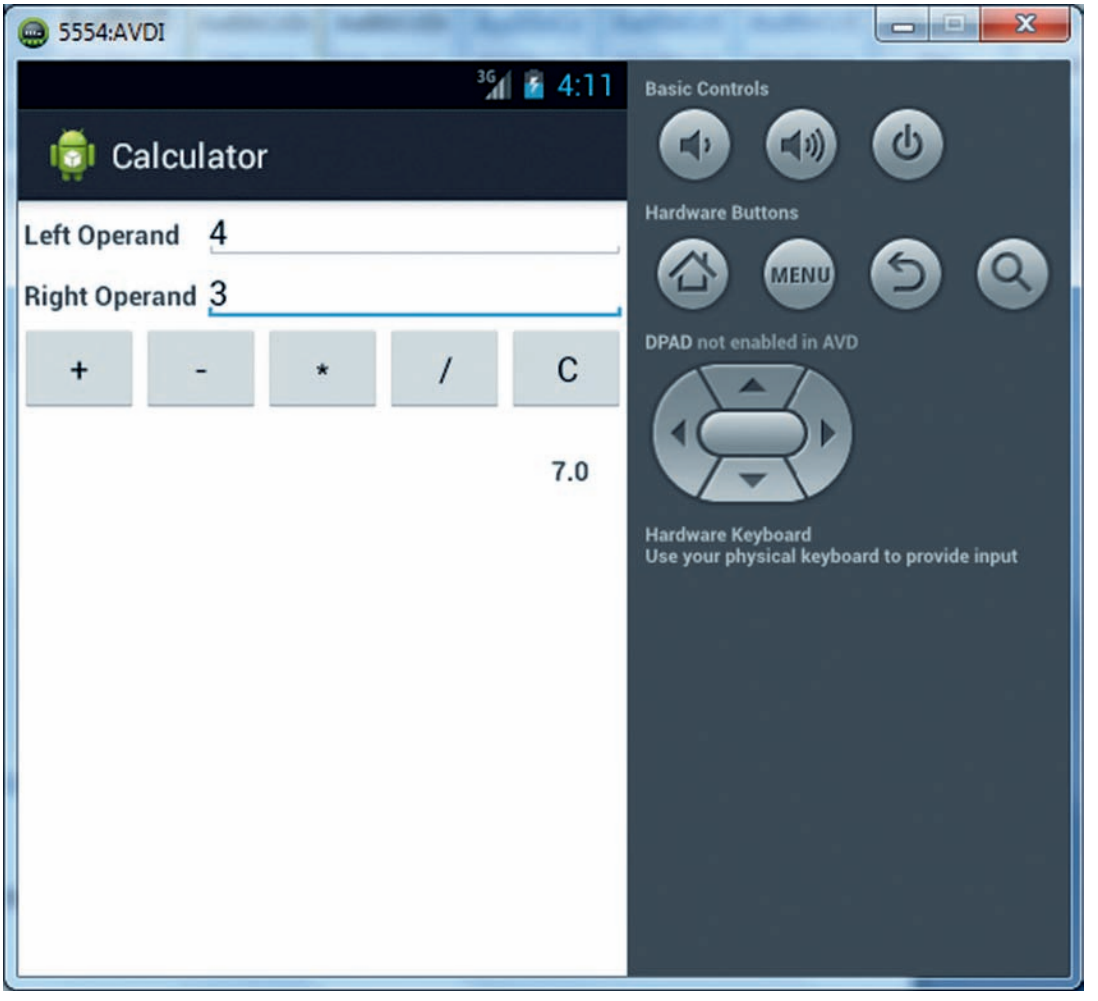


Figure 3.8: Data Output

To create the Android Test Project for **`Calculator`** application, perform the following steps:

1. Create an Android Test Project named **`CalculatorTest`**.

   Package: **`com.example.calculator.test`**

2. Create a test case class named **`CalculatorActivityTest`** within the package.

3. Modify the code in the **`CalculatorActivityTest`** class as shown in code snippet 15.

Concepts

Code Snippet 15:

```
package com.example.calculator.test;


import com.example.calculator.CalculatorActivity;
import android.test.ActivityInstrumentationTestCase2;
import android.test.TouchUtils;
import android.test.suitebuilder.annotation.SmallTest;
import android.view.KeyEvent;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;


public class CalculatorActivityTest extends
        ActivityInstrumentationTestCase2<CalculatorAct
        ivity> {
    private CalculatorActivity calculatorInstance;
    private TextView result;
    Button plus;
    EditText ed1, ed2;


    public CalculatorActivityTest() {
        super("com.example.calculator",
        CalculatorActivity.class);
    }


    @Override
    protected void setUp() throws Exception {
        // TODO Auto-generated method stub
        super.setUp();
        calculatorInstance = (CalculatorActivity)
        getActivity();
```

```
        result = (TextView) calculatorInstance
                .findViewById(com.example
                .calculator.R.id.result);
        plus = (Button) calculatorInstance
                .findViewById(com.example
                .calculator.R.id.plus);
        ed1 = (EditText) calculatorInstance
                .findViewById(com.example
                .calculator.R.id.leftOperand);
        ed2 = (EditText) calculatorInstance
                .findViewById(com.example
                .calculator.R.id.rightOperand);
    }

    /**
     * Test the addition operation of the CalculatorActivity
     *
     * @throws Throwable
     */
    public void testAdd() throws Throwable {

        // First field value
        sendKeys(KeyEvent.KEYCODE_3);
        sendKeys(KeyEvent.KEYCODE_PERIOD);
        sendKeys(KeyEvent.KEYCODE_5);

        // Move to the second field
        sendKeys(KeyEvent.KEYCODE_DPAD_DOWN);
        sendKeys(KeyEvent.KEYCODE_2);
        sendKeys(KeyEvent.KEYCODE_PERIOD);
        sendKeys(KeyEvent.KEYCODE_1);

        // Move to the '+' button
        sendKeys(KeyEvent.KEYCODE_DPAD_DOWN);
        sendKeys(KeyEvent.KEYCODE_DPAD_CENTER);
```

```
        //Wait for the activity to finish all of its
        processing.
        getInstrumentation().waitForIdleSync();


        //Use assertion to make sure the value is correct
        assertTrue(result.getText().toString()
        .equals("5.6"));
    }


    /** Testing Add Operation */
    @SmallTest
    public void testAddition(){
        String strNum1 = "100 PERIOD 5";
        String strNum2 = "50";
        float expected = 150.5f;


        /** Inputting Operand1 */
        TouchUtils.tapView(this, ed1);
        sendKeys(strNum1);


        /** Inputting Operand2 */
        TouchUtils.tapView(this, ed2);
        sendKeys(strNum2);


        /** Selects Add operator from the button Widget */
        TouchUtils.tapView(this, plus);
        this.sendRepeatedKeys(1, KeyEvent.KEYCODE_DPAD_
        CENTER, 1,
                KeyEvent.KEYCODE_DPAD_DOWN, 1, KeyEvent
                .KEYCODE_DPAD_CENTER);
```

```
        /** Performs calculation */
        try {
            runTestOnUiThread(new Runnable() {
                @Override
                public void run() {
                    plus.performClick();
                }
            });
        } catch (Throwable e1) {
            e1.printStackTrace();
        }

        /** Getting the result shown */
        float actual = Float.parseFloat(result.getText()
        .toString());
        /** Asserts, expected and actual are same */
        assertEquals(expected, actual, 0);
    }
}
```

In the code, a method named **`testAdd()`** has been created to test the addition of the numbers in the emulator. Similarly, you can create other methods, such as **`testSubstract()`**, **`testMultiply()`**, and **`testDivide()`** for testing the subtraction, multiplication, and division functionality of the **`Calculator`** class.

The method named **`testAddition()`** has been created to used `sendKeys()` and `sendRepeatedKeys()` method.

Perform the following steps to execute the project and test the **`Calculator`** application in the emulator.

4. Right-click **`CalculatorTest`** and select **Run → Run Configuration** to display the **Run Configurations** dialog box.
5. Right-click **Android Junit Test** in the left pane and select **New**.
6. Type **`CalculatorTest`** in the **Name** box.
7. Select **Run a single test**.
8. Type **`CalculatorTest`** in the **Project** box.

Concepts

9. Click **Search** and select Test class as **`CalculatorActivityTest`** to display **`com.example.calculator.test.CalculatorActivityTest`** in the **Test class** box.
10. Select **`android.test.InstrumentationTestRunner`** from the Instrumentation runner list.
11. Click **Run**.

Figure 3.9 displays the output after execution of the **`Test`** class.

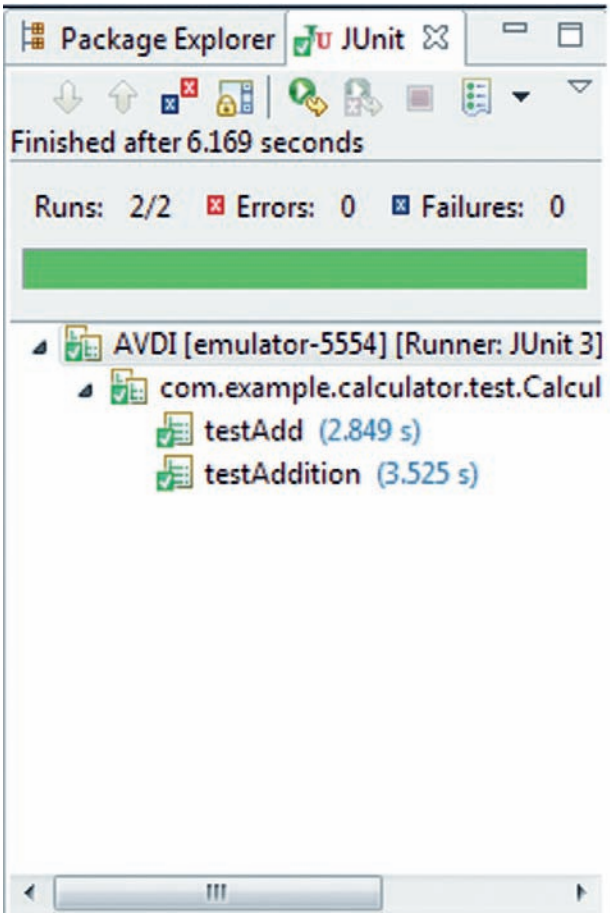


Figure 3.9: CalculatorTest Class - Output

➔ runTestOnUiThread helper method

This is a helper method that is used for running portions of a test on the UI thread.

The syntax of the method is:

Syntax

```
public void runTestOnUiThread (Runnable r)
```

where

`r` – represents the runnable containing the test code in the `run()` method.

Concepts

3.10 Check Your Progress

1. Match the following keywords with their respective description.

(A)	Assert Class	1.	Tests the setup and teardown of applications.
(B)	Mock objects	2.	Direct or indirect base class for the different test cases.
(C)	TouchUtils	3.	Base class for all test cases.
(D)	ApplicationTestCase	4.	Extends from the InstrumentationTestCase.
(E)	InstrumentationTestCase	5.	Duplicate real domain objects.

2. Which of the following class is the base class for all test cases in the JUnit framework?

(A)	TestCase class	(C)	Assert class
(B)	ApplicationTestCase class	(D)	JUnit tests

3. Which of the following assertion methods checks the required permission for the reading from a specific URI?

(A)	assertReadingContentUri RequiresPermission() method	(C)	assertWritingContentUri RequiresPermission() method
(B)	assertActivityRequires Permission() Method	(D)	CodeAccessPermission. Assert method

4. Which of the following option states the characteristics of mock objects?

(A)	support different parameter types	(C)	tapping on a View and releasing
(B)	set a provider with an authority to null	(D)	to detach unit from dependencies

5. _____ classes are advanced methods that are used for testing user interfaces.

(A)	ViewAssert	(C)	TouchUtils
(B)	MoreAssert	(D)	Context

3.10.1 Answers

1.	A-3; B-5; C-4; D-1; E-2
2.	A
3.	A
4.	D
5.	A

Summary

- The Assert class is the base class for all of the test case classes. It is a part of the JUnit API and consists of several assertion methods that are used for writing tests.
- The android.test.ViewAsserts test the views on screen and their absolute and relative positions on the screen. MoreAsserts are responsible for regular expression matching.
- TouchUtils class stimulates different kinds of touch events in the UI. It extends from the InstrumentationTestCase.
- Mock objects are duplicate objects that are called instead of the real domain objects to allow testing of units in isolation. Simple mock object classes stub out the corresponding system object class and override methods to provide mock dependencies. MockContentResolver isolates the test code.
- IsolatedContext and RenamingDelegatingContext classes are used for testing. IsolatedContext prevents user from talking to the rest of the devices. RenamingDelegatingContext allows replacing of the context used when calling file system methods.
- TestCase base class is the base class for all test cases in the JUnit framework. They provide the basic methods in the JUnit framework. AndroidTest case is the base class for general purpose Android test cases.
- Component-specific classes are responsible for testing specific components with methods for fixture setup and teardown and component life cycle control. They are also needed to provide methods to set mock objects.
- ApplicationTestCase tests the setup and teardown of applications.
- The InstrumentationTestCase class may be a direct or indirect base class for the different test cases that have an approach to Instrumentation.

Try it Yourself

1. Make an application calculator which add, sub, multiply, divide, and evaluate the result with the help of Assertion class in Android.

2. Modify the code showing the use of the **`testSubstract()`**, **`testMultiply()`**, *and* **`testDivide()`** *for testing the subtraction, multiplication, and division functionality of the* **`Calculator`** *class.*

“You must do the things
you think you cannot do”

Session - 4

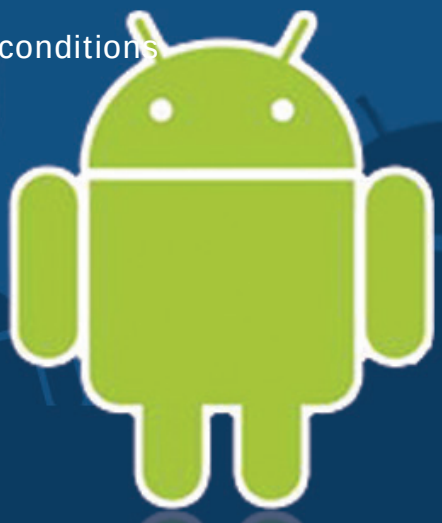
Android Testing Environment

Welcome to the session, **Android Testing Environment**.

This session discusses about the Android testing environment. It begins with creating the Android Virtual Device (AVD) that will pave the way for different conditions and configurations that are required to run tests with the available options. Finally, it proceeds to developing a test driven application that will simulate generated events and use it for testing.

In this session, you will learn to:

- Explain the process to create an AVD with AVD Manager
- Explain the process to run AVD from command line
- Explain how to simulate network conditions
- Explain a test driven application



4.1 Introduction

An essential part of creating Android applications is to run and test them. It is important to test the code and application on real devices to assess the drawbacks and rectify them. Learning to create an AVD, simulate network conditions, and develop a test driven application are some of the fundamental steps to build an Android testing environment.

4.2 Android Virtual Device (AVD)

An Android Virtual Device (AVD) is an emulator configuration. It is made up of the following components:

- Hardware that defines whether the device has a camera or the storage memory, use of a dialing pad or a physical QWERTY keyboard and so on.
- A mapping to the system image that detail on what Android version to run the virtual device. It also provides a choice of the version of the standard Android platform or the system image with the SDK add-on.
- It provides options to choose and specify the emulator skin, emulator SD card, screen dimensions, appearance, and so on.
- A dedicated storage area that stores the device's user data and emulated SD card.

The main objective of the AVD is to design a prototype of an actual device by specifying the hardware and software options to be emulated by the Android emulator.

4.2.1 Create AVD with AVD Manager

Using the AVD Manager is one of the simplest ways of creating an AVD. The AVD Manager has options to edit, delete, repair, and view details of each AVD.

It is possible to create many AVDs depending on the need and types of device that are to be modeled. It is, however, advised that to thoroughly test the application, it is necessary to test applications on all API levels higher than the target API level of the application.

To create an AVD, perform the following steps:

➔ Click **Window → AVD Manager** to launch the AVD Manager from Eclipse as shown in figure 4.1.

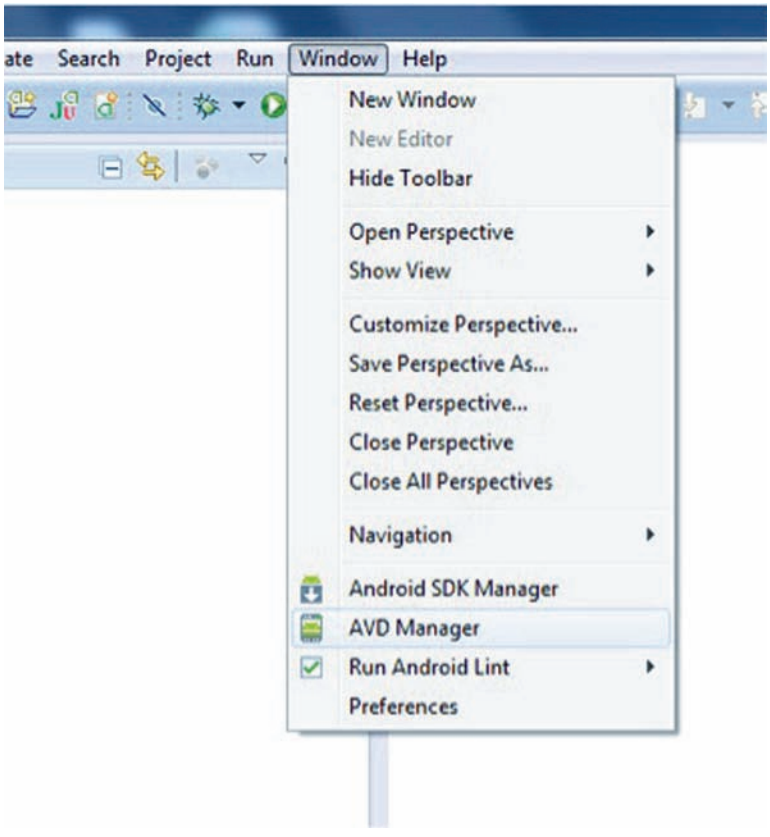


Figure 4.1: Launch AVD Manager from Eclipse

Click **AVD Manager** icon in the Eclipse toolbar as shown in figure 4.2.

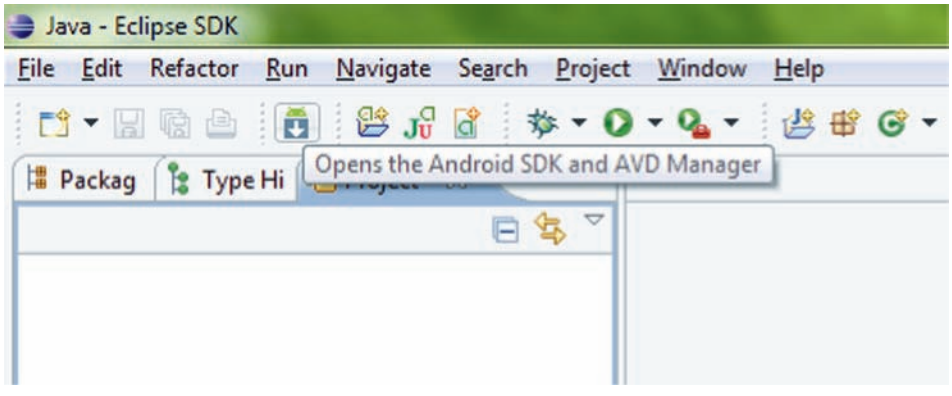


Figure 4.2: Launch AVD Manager Icon from the Eclipse Toolbar

Note - To start AVD Manager in other IDEs, navigate to SDK's **tools**/ directory and execute the android tool with no arguments.

Concepts

➔ To start the AVD Manager from the command line, go to the **sdk/tools/** directory and invoke the android tool with **avd** options.

➔ Virtual Devices panel of the **AVD Manager** lists the existing AVDs. To create a new AVD, click **New** as shown in figure 4.3.

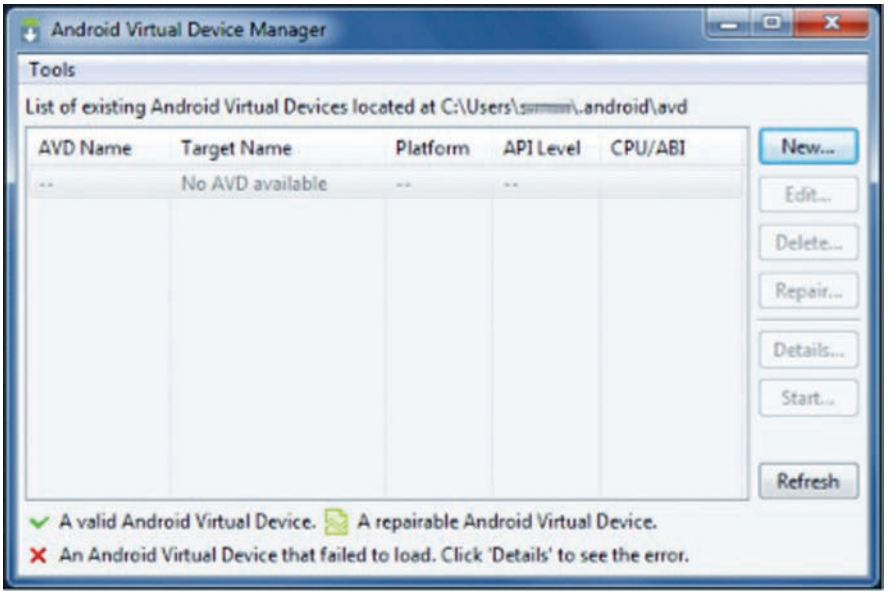


Figure 4.3: Virtual Devices Panel of the AVD Manager

The **Create new Android Virtual Device (AVD)** dialog box appears as shown in figure 4.4.

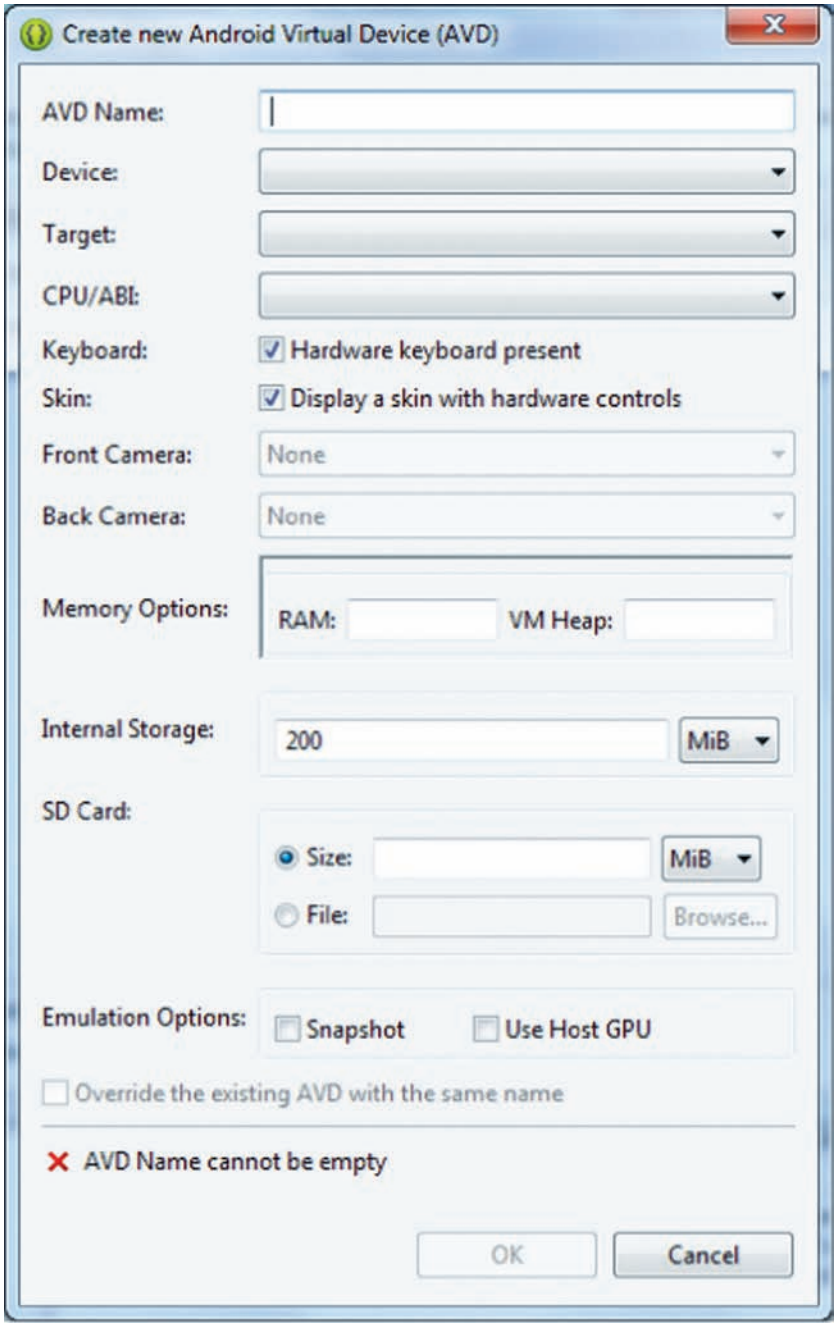


Figure 4.4: Create new Android Virtual Device Dialog Box

Concepts

➔ Enter details for AVD such as name, platform target, SD card size, and a skin as shown in figure 4.5. Give a descriptive name and use 200MB RAM for basic testing AVDs. Leave the rest of the options as default. Remember to define a target for the new AVD that meets the application's Build Target. The AVD platform target should be equal or greater than API Level of the application.

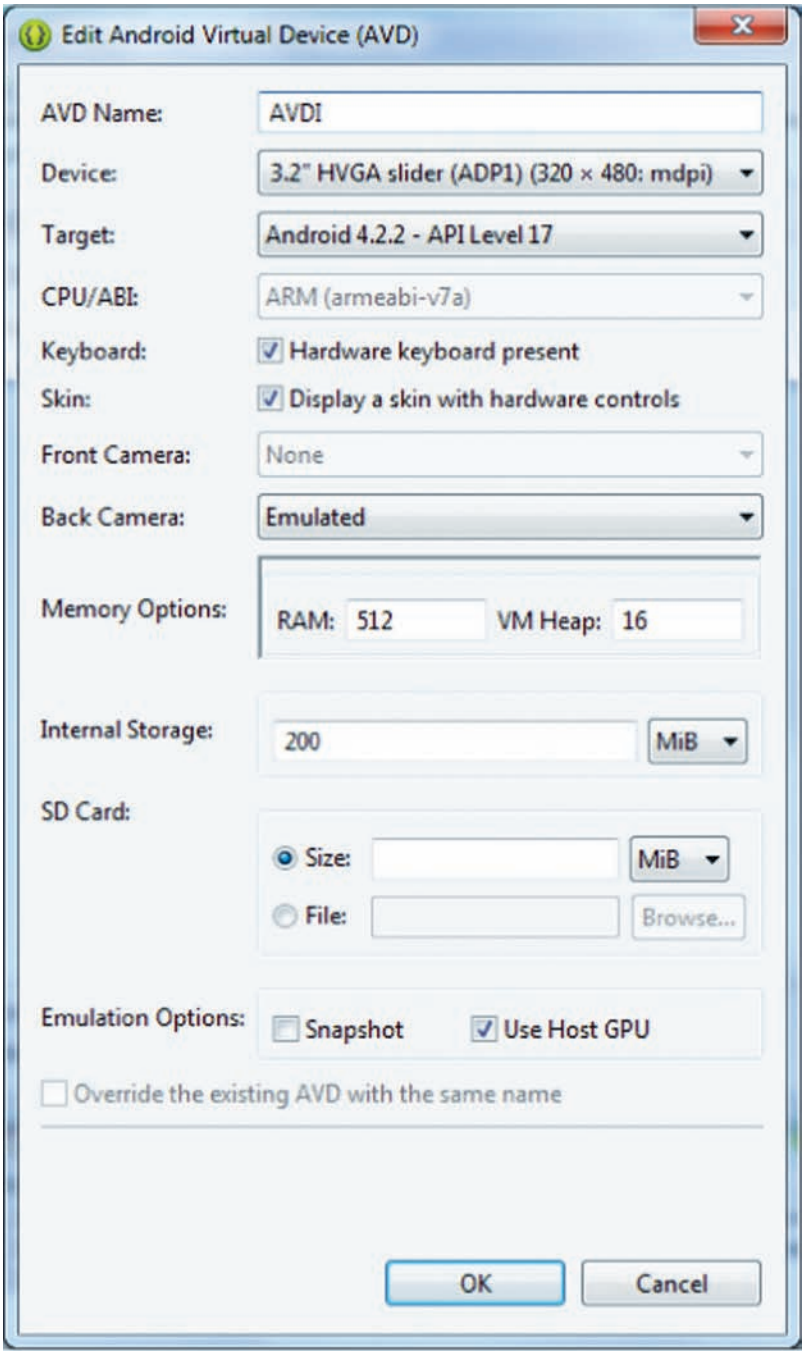


Figure 4.5: Enter Details for AVD

Concepts

➔ Click **OK** to create a new AVD. The Virtual Devices panel shows the newly created AVD as seen in figure 4.6. It is possible to create more AVDs using this method.

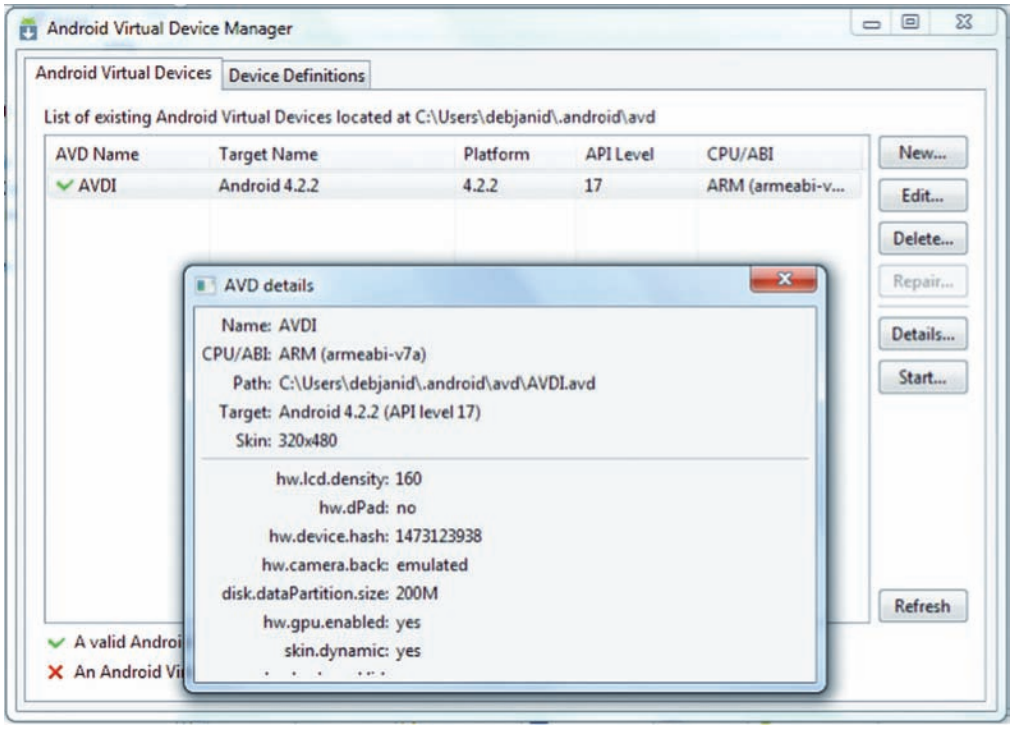


Figure 4.6: Newly Created AVD

Close the AVD Manager, or continue to create more AVDs, or launch an emulator. Follow the previous steps to create more AVDs.

➔ To launch an emulator with the AVD, select the AVD and click **Start**. It will launch the AVD by displaying the **Launch Options** dialog box as shown in figure 4.7. Your AVD is now ready and you can close the AVD Manager, create more AVDs, or launch an emulator with the AVD by selecting a device.

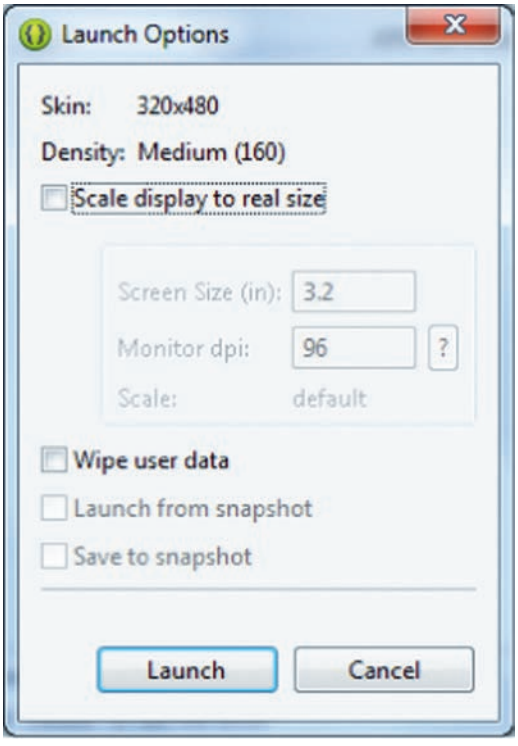


Figure 4.7: Launch Options Dialog Box

The **Launch Options** dialog box shows the skin type and density along with other launch options such as:

- **Scale display to real size:** refers to the resolution of the emulator's display scaled to the actual screen size of the physical Android device that is emulated. The scale display screen size is provided in terms of inches and dots per inch.
- **Wiper user data:** results in deletion of the AVD's user-data partition. The emulated device tends to store apps and state data across AVD when it restarts in a user-data partition.
- **Launch from snapshot:** refers to the emulated device being started from a previously saved snapshot. This checkbox is checked by default when an AVD is created with snapshot enabled. Snapshot refers to a stored file of emulator state.
- **Save to snapshot:** refers to saving a snapshot of the state of the emulated device on exit from the device. This checkbox is checked by default when an AVD is created. Launching an emulator is fast when it is launched from a snapshot.

Concepts

➔ Click **Launch** to see the emulator window as shown in figure 4.8. The emulator window has a device screen on the left and the phone controls on the right. The title bar of the emulator window shows 5554:AVDI. Value 5554 refers to the value of the console port that can be used to query and control the AVD's environment.

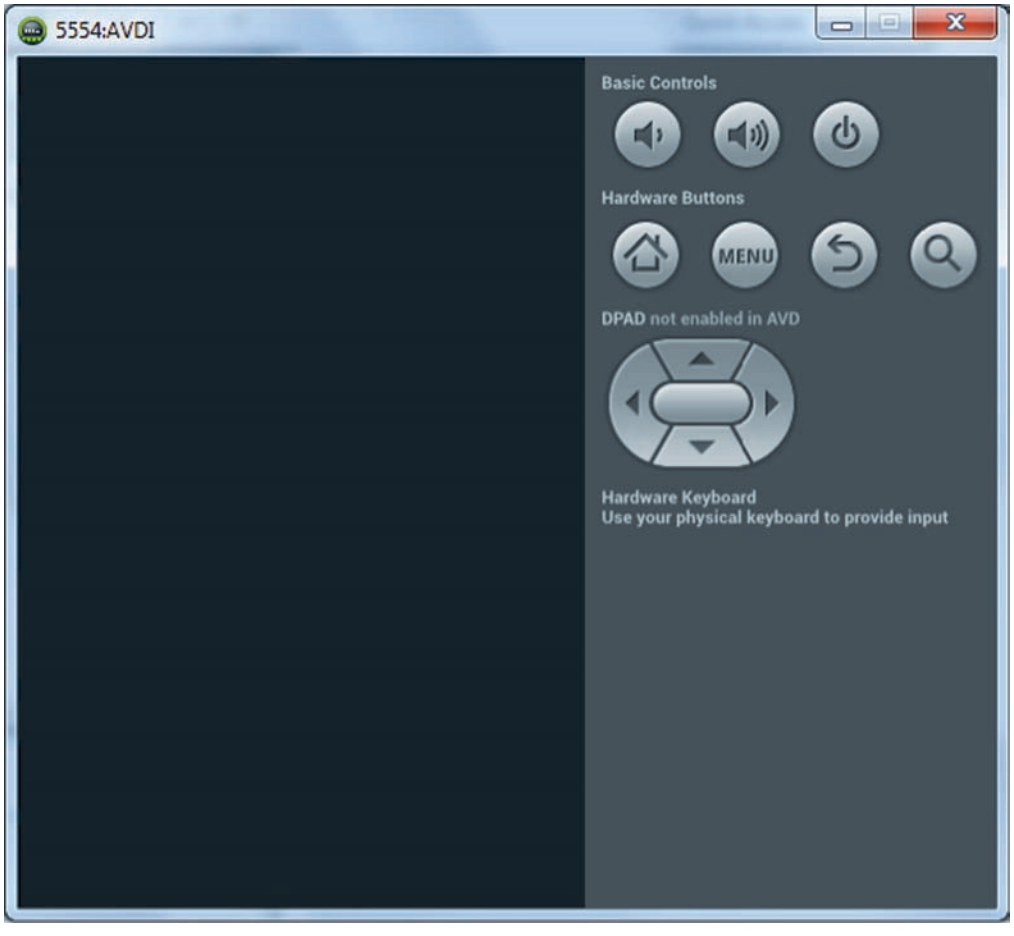


Figure 4.8: Emulator Window

The Android logo as shown in figure 4.9 is displayed few minutes later after the initial Android screen and is shown while the Android platform associated with the AVD is initializing.

Figure 4.9: Graphical Android Logo

After some time the home screen is displayed as shown in figure 4.10.

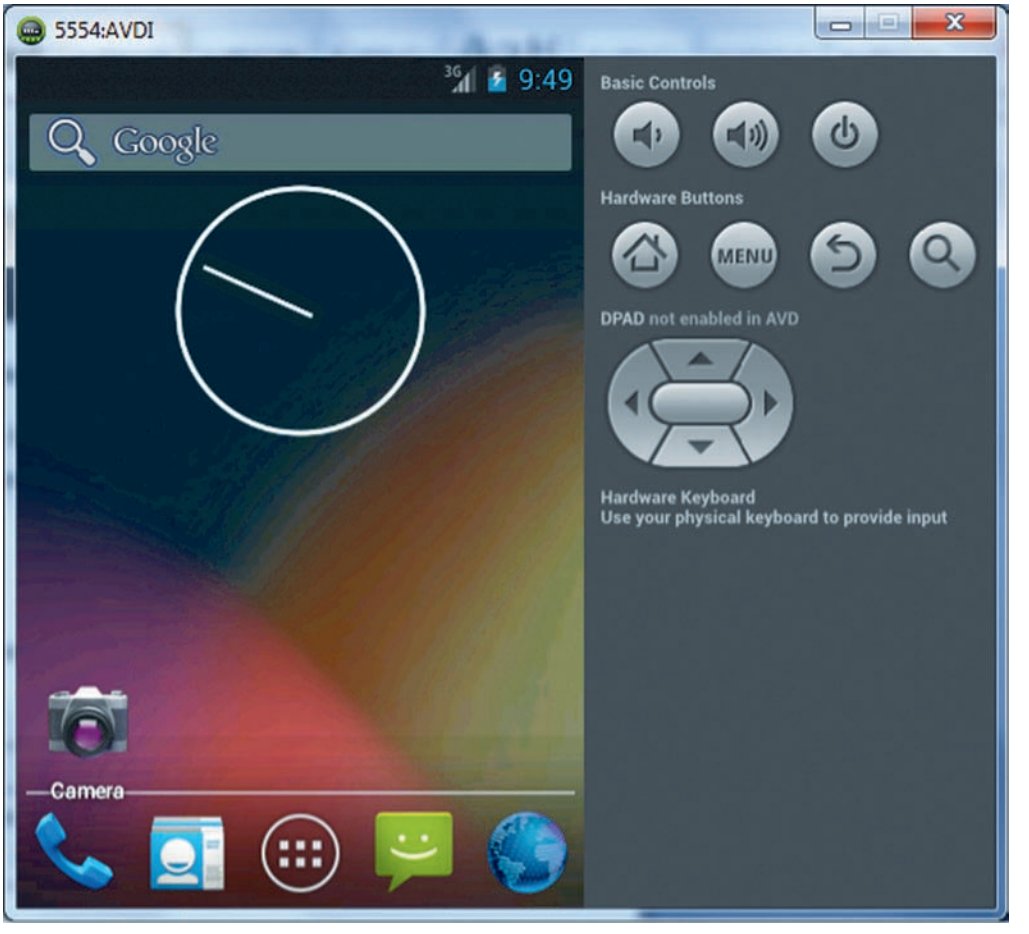


Figure 4.10: Android Home Screen

If the ANDROID logo is displayed for more than 15–30 minutes, then there may be an error. Restart the computer, start AVD Manager, delete the previous AVD, create a new one, and relaunch the new AVD.

4.2.2 Hardware Options

The different hardware properties that can be specified for the AVD to emulate are listed in table 4.1.

Characteristic	Description	Property	Type
Device RAM size	Physical RAM size on the device measured in megabytes. Default value is “96”.	hw.ramSize	integer
Touch-screen support	Presence/absence of a touch screen facility on the device. Default value is “yes”.	hw.touchScreen	boolean

Characteristic	Description	Property	Type
Trackball support	Presence/absence of a trackball on the device. Default value is “yes”.	hw.trackBall	boolean
Keyboard support	Presence of a QWERTY keyboard on device. Default value is “yes”.	hw.keyboard	boolean
DPad support	Presence of DPad keys on device. Default value is “yes”.	hw.dPad	boolean
GSM modem support	Presence of GSM modem in the device. Default value is “yes”.	hw.gsmModem	boolean
Camera support	Presence of camera in the device. Default value is “no”.	hw.camera	boolean
Maximum horizontal camera pixels	Default value is “640”.	hw.camera. maxHorizontalPixels	integer
Maximum vertical camera pixels	Default value is “480”.	hw.camera. maxVerticalPixels	integer
GPS support	Presence of a GPS in the device. Default value is “yes”.	hw.gps	boolean
Battery support	Whether the device can run on a battery. Default value is “yes”.	hw.battery	boolean
Accelerometer	Presence of an accelerometer in the device. Default value is “yes”.	hw.accelerometer	boolean
Audio recording support	Availability of audio record in the device. Default value is “yes”.	hw.audioInput	boolean
Audio playback support	Availability of audio play in the device. Default value is “yes”.	hw.audioOutput	boolean
SD Card support	Whether the device supports insertion/removal of virtual SD Cards. Default value is “yes”.	hw.sdCard	boolean
Cache partition support	Use a /cache partition on the device. Default value is “yes”.	disk.cachePartition	boolean
Cache partition size	Default value is “66MB”.	disk.cachePartition. size	integer
Abstracted LCD density	Sets the generalized density characteristic used by the AVD’s screen. Default value is “160”.	hw.lcd.density	boolean

Table 4.1: Hardware Property Specification for an AVD

4.3 Run AVD from Command Line

It is possible to run the AVD from the command line.

4.3.1 Headless Emulator

A headless (run without display) emulator is useful to run automated tests without any supervision or when the interaction between the test runner and application is too fast to decipher. However, it needs to be understood that sometimes observing the interaction is necessary to assess the reason for failure of tests.

Emulators are named based on the communication ports and its serial number. While running AVDs, their communication ports are provided at runtime, starting with 5554 and increasing by 2 from the last used port. So, an emulator using a port 5554 is given a name such as emulator-5554. Such naming is useful when AVDs are run during the development process as the port assignment can be ignored. However, it may be confusing when more than one emulator is running simultaneously. For such cases it is advisable to assign known ports to the communication ports to keep the specific AVD under control. Also, when running tests on more than one emulator at a time, it is best to avoid sound output as well as detach the window.

4.3.2 Disable Keyguard

Tests can executed using a standard approach such as a standard emulator launched from Eclipse. However, running them using the standard emulator may generate some test errors.

To disable the lock screen programmatically, the test-related code is required to be added in the application but needs to be removed or disabled when the application is ready to be shipped. To disable the lock screen programmatically, add the permission to the manifest file, **AndroidManifest.xml** and then disable the screen lock for the application under test. Code snippet 1 demonstrates the code to add permission in the manifest file.

Code Snippet 1:

```
<manifest>


<uses-permissionandroid:name=“android.permission.DISABLE
     KEYGUARD”/>


</manifest>
```

Code snippet 2 shows the code to be added to disable the key in the `onResume( )` method of the `Activity` class.

Code Snippet 2:

```
mKeyGuardManager =
 (KeyguardManager) getSystemService(KEYGUARD_SERVICE);
appLock = mKeyGuardManager.newKeyguardLock("com.example.tc");
appLock.disableKeyguard();
```

In the code, the `KeyGuardManager` is first retrieved, and then the `KeyguardLock` is obtained. Customize the package name to enable debugging. Disable the keyguard from displaying using `appLock.disableKeyguard()` method. The keyguard will be shown again when the `reenableKeyguard()`method is invoked.

4.3.3 Clean Up

Sometimes it is necessary to clean up services and processes. This prevents the results of the processes to be influenced by the ending conditions of the previous tests. Warm-boot the emulator and start from a known condition to free all used memory, stop services, reload resources, and restart processes. The command to open the emulator shell of the emulator and run the start and stop commands are as shown in code snippet 3.

Code Snippet 3:

```
$ adb -s emulator-5580 shell 'stop; sleep 5; start'
```

The `logcat` command can be used to monitor the evolution of the start and stop commands as shown in code snippet 4.

Code Snippet 4:

```
$ adb -s emulator-5580 logcat
```

4.3.4 Terminate the Emulator

Use the command as shown in code snippet 5 to kill the emulator, free used resources, and terminate the emulator process on the host computer.

Code Snippet 5:

```
$ adb -s emulator-5580 emu kill
```

4.4 Use Emulator Configurations

Sometimes it is necessary to test the AVD outside its set options. For example, when there is a need to test the application under different locales. Consider testing the application on a phone where the emulator language is set to Canadian French and country to Canada. The `–prop` command line option is used to introduce these properties as demonstrated in code snippet 6.

Code Snippet 6:

```
$emulator -avd test -no-window -no-audio -no-boot-anim -port 5580
-prop persist.sys.language=fr -prop persist.sys.country=CA
```

Use the `getprop` command to verify whether the settings are successful as shown in code snippet 7.

Code Snippet 7:

```
$adb -s emulator-5580 shell "getproppersist.sys.language"
fr
$adb –s emulator-5580 shell "getproppersist.sys.country"
CA
```

Use the command from the command line as shown in code snippet 8 to kill all user data after completing the task with persistent setting and starting the emulator afresh.

Code Snippet 8:

```
$adb -s emulator-5580 emu kill
$emulator -avd test -no-window -no-audio -no-boot-anim -port 5580\
-wipe-data
```

4.5 Simulate Network Conditions

Host network have different speed and latency. To ensure that the Android emulator supports network throttling it is important to test under different network conditions. For this, use the emulator command line options `–netspeed <speed>` and `–netdelay <delay>`.

The emulator uses default values for these options such as:

➔ Default network speed is 'full'.

➔ Default network latency is 'none' when values are not specified.

4.6 Develop Test Driven Applications

Test driven development unlike other approaches is a strategy to write tests prior to writing code in the development process. Initially a single test is added, followed by the code to satisfy the compilation of the test, and finally, the full set of test cases is written to verify the results. The advantages of using this method are as follows:

- to avoid writing test cases, especially when present at the end of application development.
- developers take responsibility for their work quality.

Test driven development, as shown in the UML figure 4.11 has a set of individual activities such as:

- understanding requirements
- writing a test case
- running all tests
- refactoring code

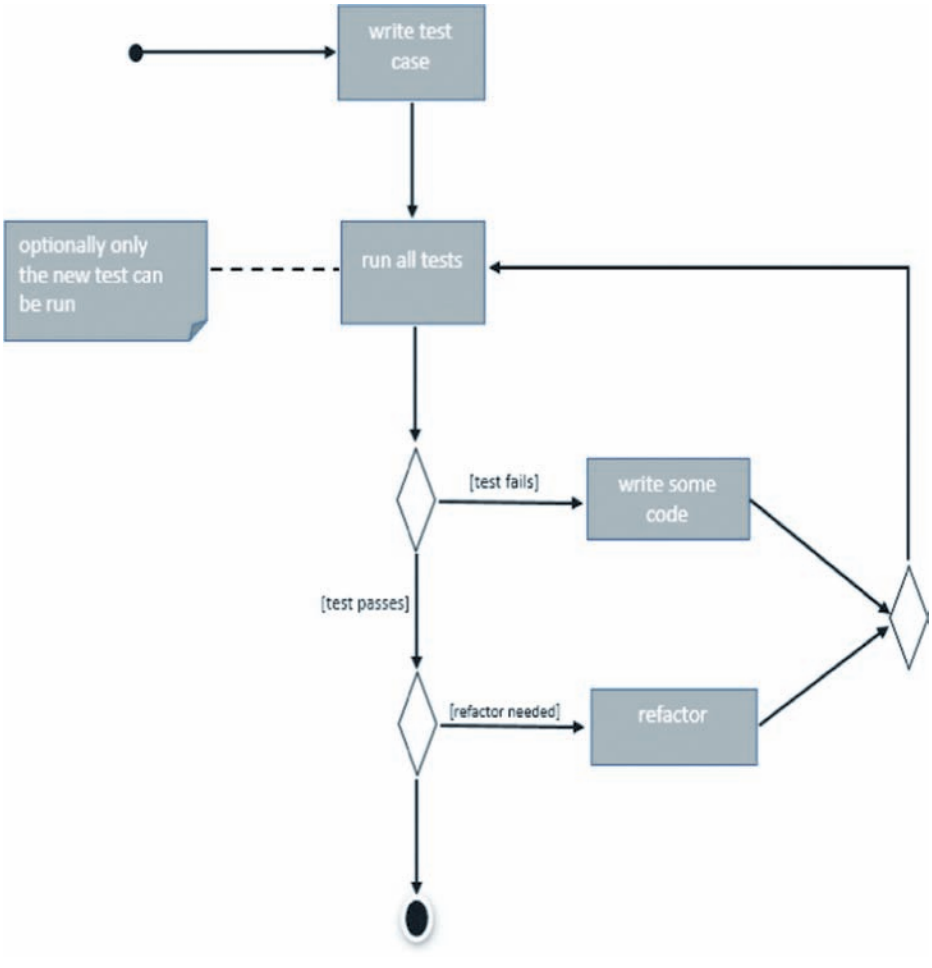


Figure 4.11: UML Diagram of the Test Driven Development Process

Concepts

4.6.1 Understand Requirements

One of the essential criteria for writing test cases is to understand the requirements. The process would be to first understand the subject under test, translate the requirements into tests, and finally to verify the tests against the requirements with an implementation and a verification process. Any change in the requirements will change the test cases verifying the requirements as well as change the implementation code to check that everything was clearly and correctly understood and mapped to the code.

4.6.2 Write a Test Case

It is necessary to understand the problem domain and its details before writing a code. The code is written based on the problem that is at hand.

4.6.3 Run All Tests

The next step is to run the tests along with the other tests that have been written until now. An IDE with a built-in support of the testing environment cuts the development time considerably. In this stage, it is expected that the test will fail since there is no code written. Therefore, to complete the test, additional codes are written and design decisions are taken into considerations. The additional code should be adequate to compile. The test is then compiled and executed. If the test fails again, then a minimum amount of required code is written to make the test successful. Also, execute the newly added test to save time before running the whole suite to verify all is still working properly.

4.6.4 Refactor the Code

When the test succeeds refactoring of the code that has been added is required to keep it tidy, clean, and minimal. Run all tests again to check if refactoring has not broken any sequence. When the test results are satisfactory, further refactoring is not necessary. Running tests after refactoring is a safe cover to verify every case used by the application and expressed as a test case.

4.6.5 Advantage of Test Driven Application

The advantages of test driven applications are as follows:

1. There is a reduction in the development time as the output is known.
2. There is a safety cover for every change of code. It is assured that the other parts of the system are not affected as long as every change of code is tested and verified.
3. Less bugs.
4. Few bugs found later during development results in less time spent on debugging.
5. Helps in development of modular code.

6. Uses stubs and mock objects.
7. Has an obvious approach to development process and does not require extensive prototyping or experimentation.

Disadvantage of Test Driven Application

1. Test driven development relying on unit tests does not perform sufficient testing in situations where full functional tests are required to determine the success or failure.
2. Developer is responsible for writing test cases and thus the tests may have some blind spots within the code.
3. New knowledge is required for writing test cases.
4. Tests become part of the maintenance overhead cost of a project.
5. Prototyping can be very difficult.
6. Testing abstraction changes from time to time.

4.7 Check Your Progress

1. Which one of the following command is used to start AVD Manager from Eclipse?

(A)	Window → AVD Manager	(C)	File → AVD Manager
(B)	Project → AVD Manager	(D)	Navigate → AVD Manager

2. In which dialog box an AVD with **wipe user data** can be enabled?

(A)	Virtual Device Dialog Box	(C)	Android Virtual Device Manager
(B)	Emulator Window	(D)	Launch Options Dialog Box

3. In which dialog box an AVD with a snapshot can be enabled?

(A)	Launch Options Dialog Box	(C)	Virtual Devices Dialog Box
(B)	Create new Android Virtual Device Dialog Box	(D)	Emulator Window

4. Which of the following commands is used to monitor the evolution of the start and stop commands?

(A)	start command	(C)	prop command
(B)	getprop command	(D)	logcat command

5. _____ permission is added to the manifest file to disable the lock screen programmatically.

(A)	AndroidManifest.xml	(C)	numLock
(B)	android.permission.DISABLE_KEYGUARD	(D)	disableKeyguard()

Concepts

4.7.1 Answers

1.	A
2.	D
3.	A
4.	D
5.	B

Summary

- An Android Virtual Device (AVD) is an emulator configuration.
- An AVD can be created with AVD Manager and from the command line.
- Different hardware options can be specified for the AVD to emulate.
- Use emulator configurations to test the AVD outside its set options.
- Test the Android emulator under different network conditions to ensure that it supports network throttling.
- In test driven development, tests are written, compiled, and executed prior to the development of full application code.

Try it Yourself

Demonstrate the steps to launch an emulator with an AVD.

“Practice is the best
of all instructors”

Session - 5

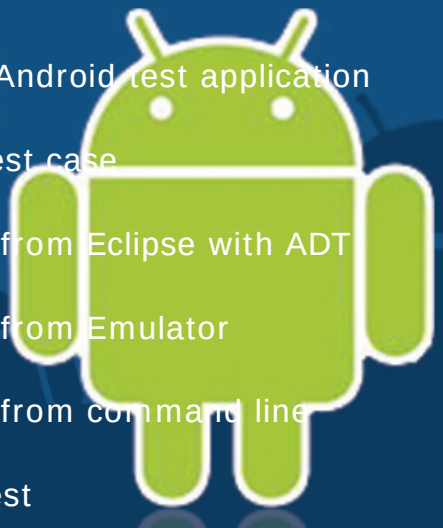
Testing Android Projects

Welcome to the session, **Testing Android Projects**.

This session introduces the various steps required for creating an Android application. After creating the application, the session will discuss on how to run and debug the tests. Finally, the session will provide a detailed guide on building a sample project. Here, the sample project under discussion is Currency Converter.

In this session, you will learn to:

- Explain the process to create an Android main project using Eclipse with ADT
- Explain the process to create an Android main project using command line tools
- Explain the process to create an Android test application
- Explain the process to create a test case
- Explain the process to run a test from Eclipse with ADT
- Explain the process to run a test from Emulator
- Explain the process to run a test from command line
- Explain the process to debug a test
- Explain the process to create a sample project



5.1 Introduction

An *Android* project has all the source codes required for the development of an Android application. With the help of Android SDK tools it is easy to start a new Android project with default project directories and files.

The session will explain how to create an Android project using Eclipse and command line tools. It further discusses the process of creating an Android test project and then running the tests from Eclipse and emulator. It will also describe how to debug tests. Later, the session will give a step-by-step guidance on creating a sample Android project.

5.2 How to Create an Android Main Application Project?

As mentioned earlier, all the files required for an Android application is stored in the Android project. A new project can be created using Eclipse with the ADT plugin or using the SDK tools from a command line. However, before beginning the application development the development environment is required to be set up. To work in Eclipse, Eclipse needs to be installed with the ADT plug-in. The Android SDK tools are also required to be installed to enable the developer for working from the command line.

5.2.1 Using Eclipse

To create an Android project using Eclipse, perform the following steps:

1. Start Eclipse.
2. Click **File → New → Project** as shown in figure 5.1. to display the **New Project** dialog box.

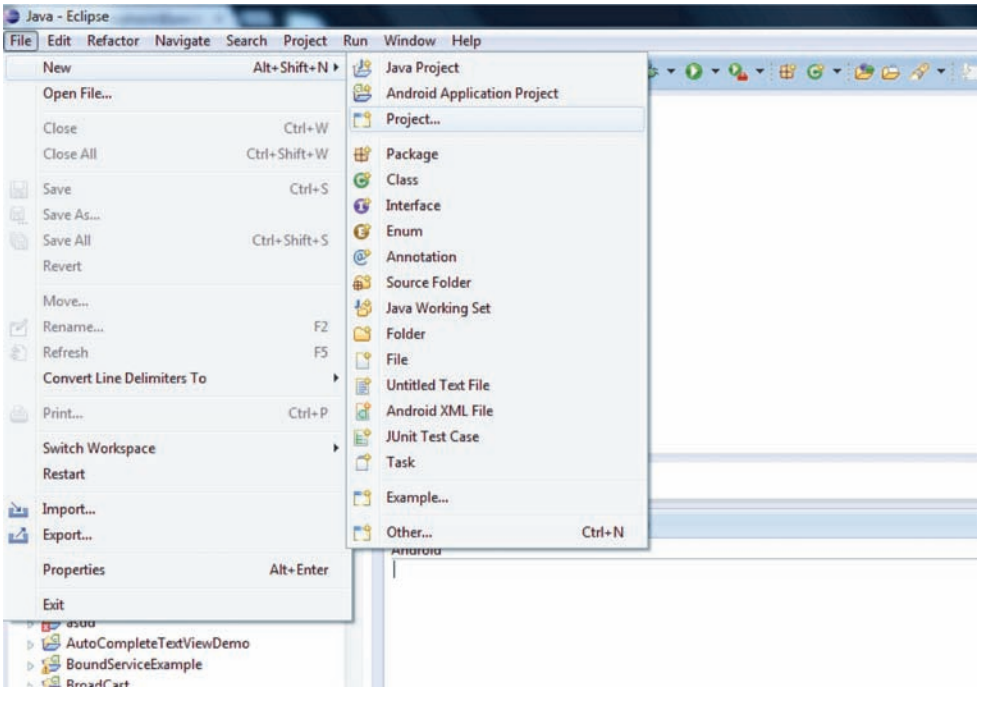


Figure 5.1: Navigating Using the File Menu

3. Navigate to **Android → New Android Project**.

4. Enter details in the **New Android Application** dialog box as shown in figure 5.2.

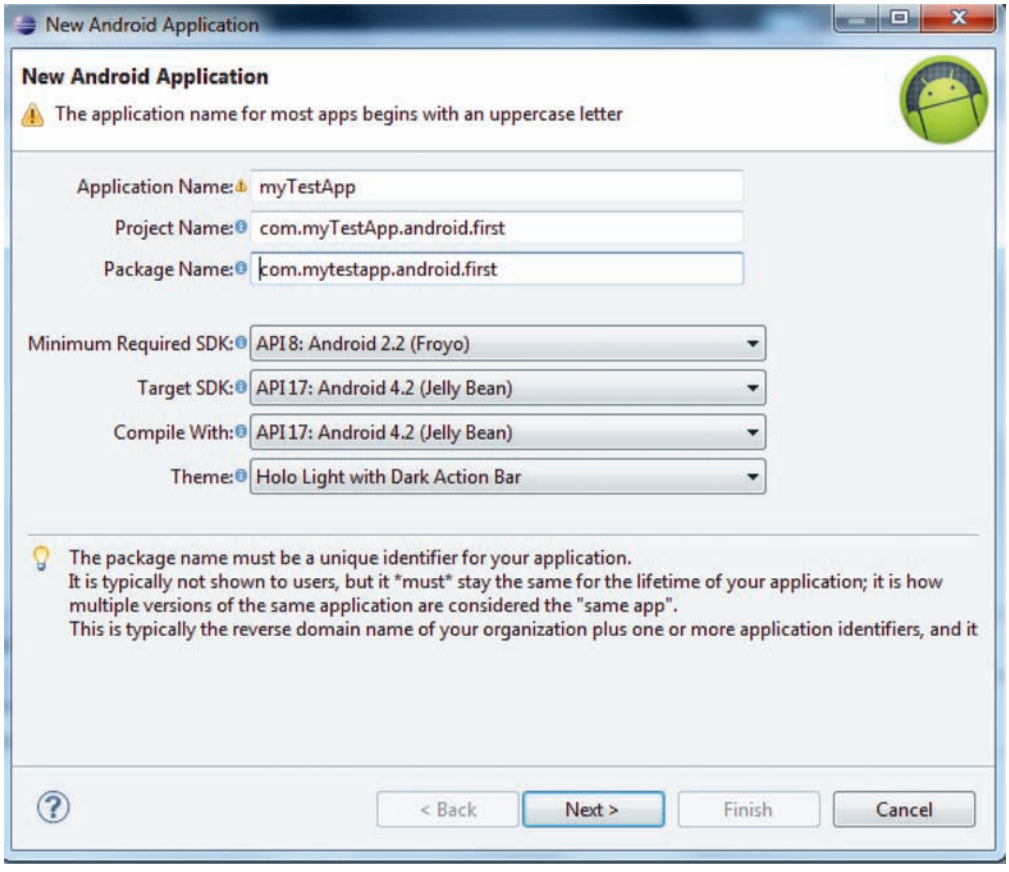


Figure 5.2: Entering Details in the New Android Application Dialog Box

Table 5.1 explains the options in details.

Parameters	Description
Application Name	Name seen by users for example, TestApp.
Project Name	Visible name in Eclipse project directory.
Package Name	Package represents the namespace of the application. To maintain the uniqueness of the name, the package name is usually given a reverse domain name. In this example, it is com.mytestApp.android.first. Such naming is followed as it is visible across all packages installed in the Android system. However, remember such namespace does not work on Google Play.
Minimum Required SDK	Lowest version of Android supporting the application. Expressed in terms of API level and is set by default. It is recommended to set this parameter to the lowest available version to enable the application to provide its core feature set. If any feature of the application is not part of the core feature set, then enable it to run only in versions that support it.

Parameters	Description
Target SDK	Highest version of Android that will test the application. It is best to test the new application in the latest versions.
Compile With	Platform version that compiles with the application. It is set by default to the latest version of Android available in the SDK. It is possible to set the parameter to older versions but at the loss of new features and lesser optimization of the application to user experience.
Theme	Indicates the Android UI style applied to the application. It is set to its default value.

Table 5.1: Parameters to Complete in New Android Application Dialog Box

5. Click **Next**.

6. Retain the default settings on the **Configure Project** pane as shown in figure 5.3.

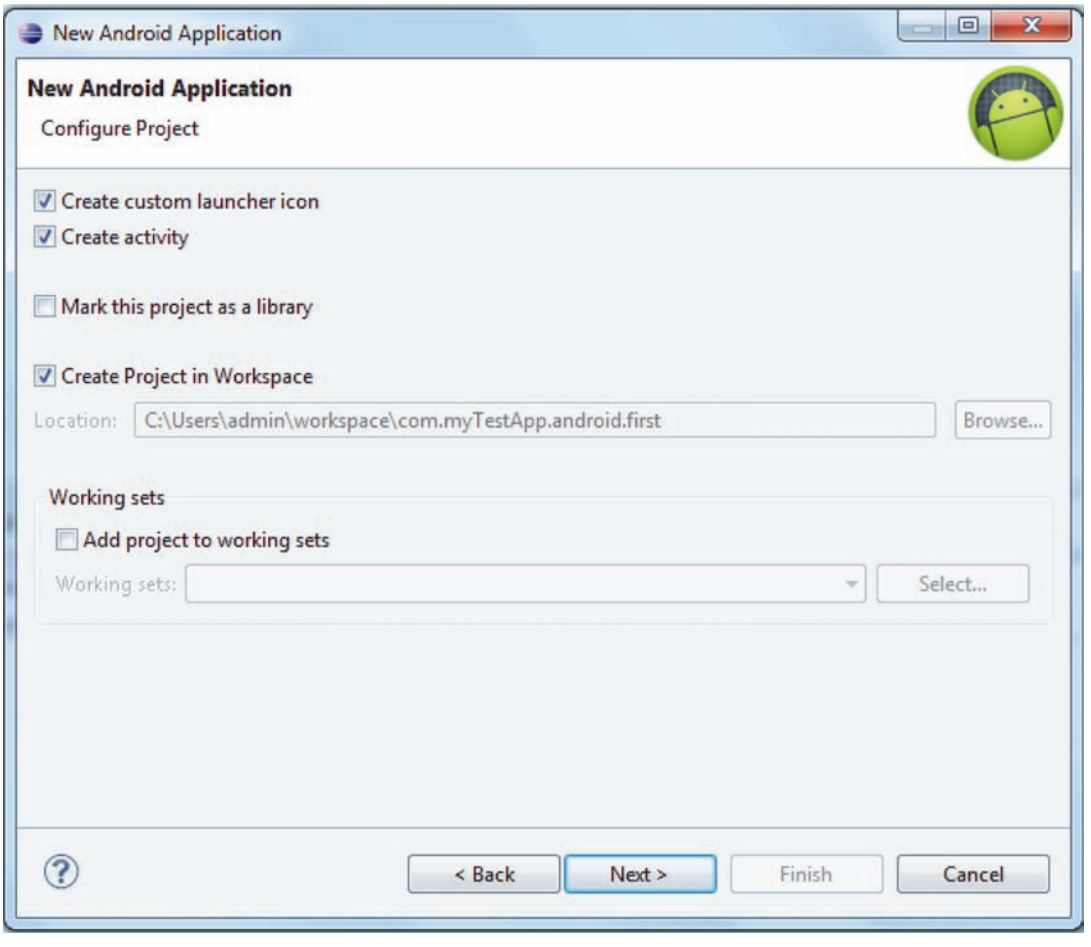


Figure 5.3: Default Settings in the Configure Project Window

7. Click **Next**.

8. Customize the application icon on the **Configure Launcher Icon** pane as shown in figure 5.4. Check whether it meets the specifications defined in the **Iconography** design guide. The icon is generated for all screen densities.

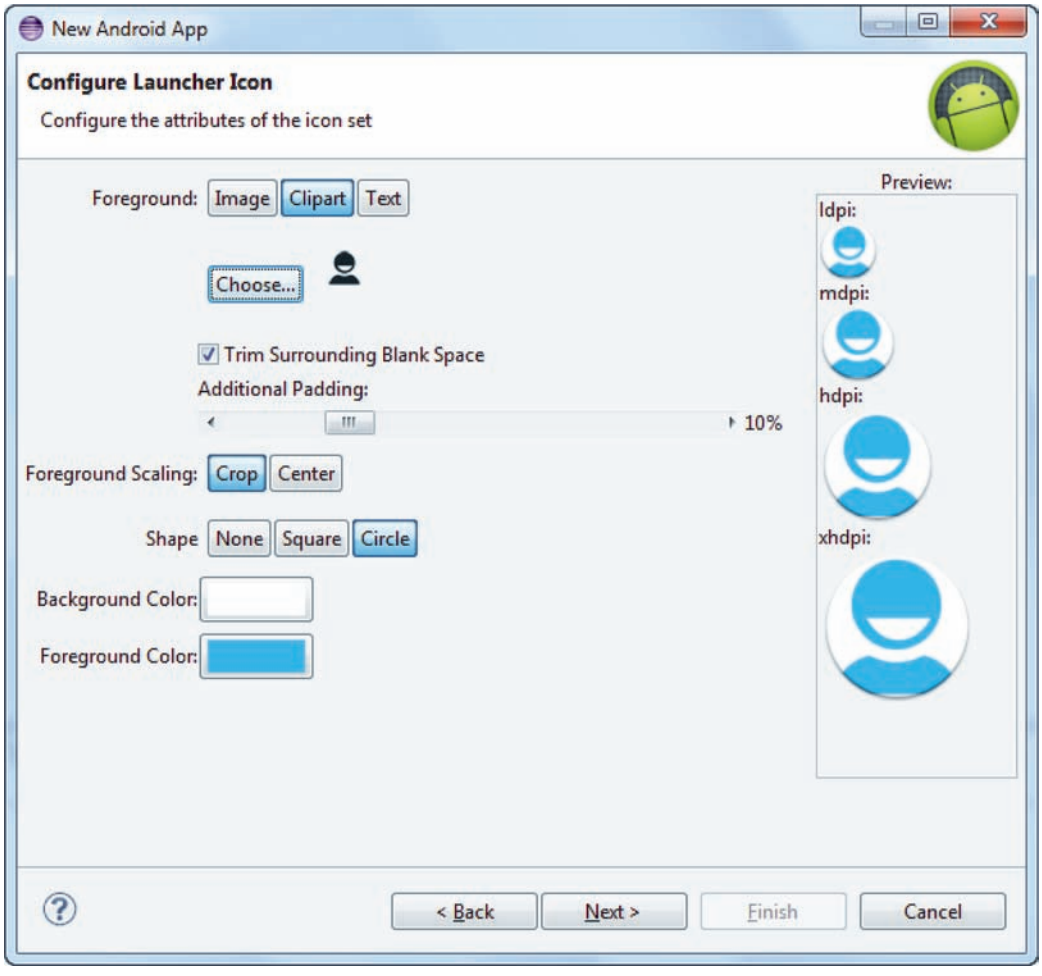


Figure 5.4: Customizing Launcher Icon

9. Click **Next**.

10. Select **BlankActivity** as the activity template on the **Create Activity** pane as shown in figure 5.5. The application will start with the selected template.

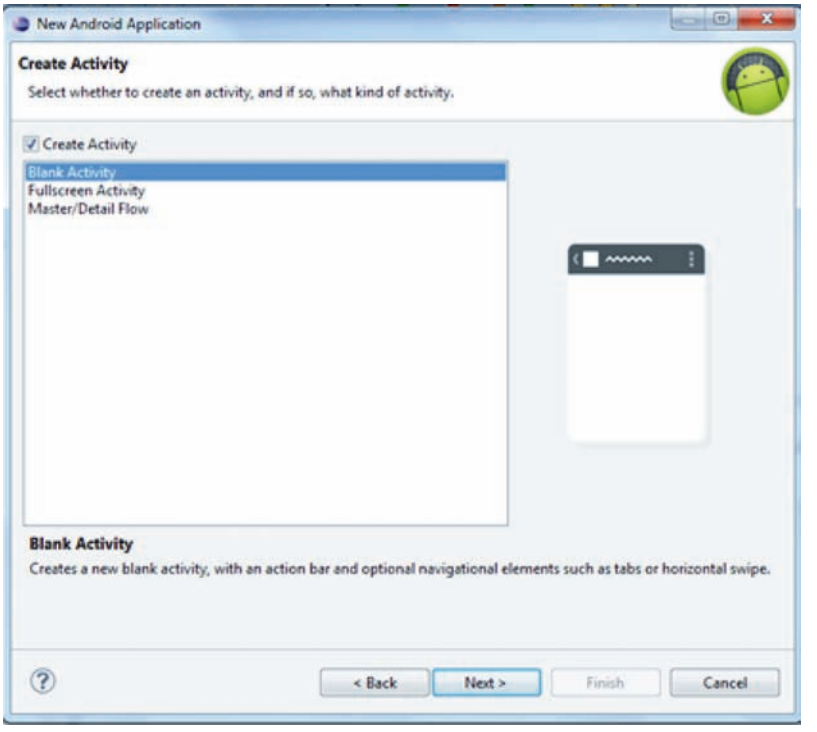


Figure 5.5: Create Activity Window

11. Click **Next**.

12. Retain the default setting for the **New Blank Activity** pane as shown in figure 5.6. Click **Finish**.

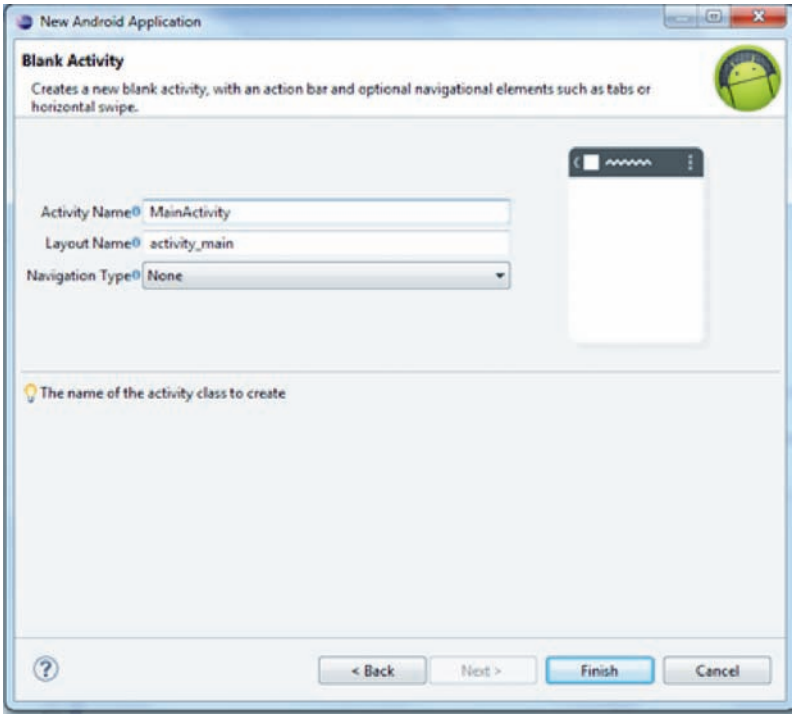


Figure 5.6: Settings in the New Blank Activity Window

The new Android application is created. Begin building the application.

5.2.2 Using Command Line Tools

To create an Android project using command line tools, perform the following steps:

1. Open the command prompt.
2. Set the path for tools and platform-tools directory as shown in figure 5.7.

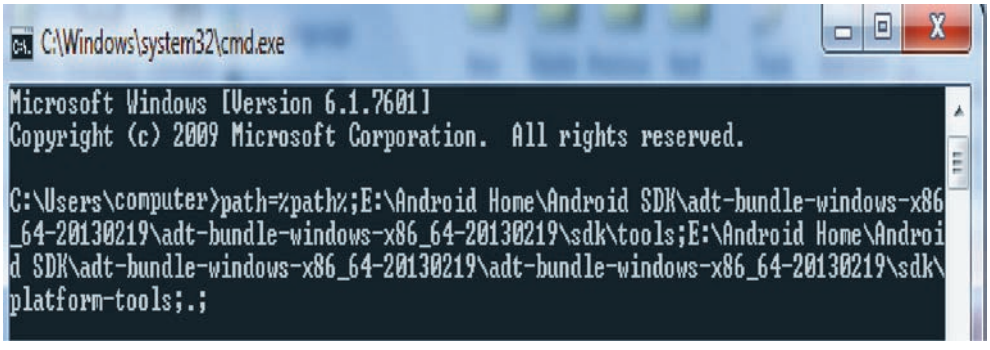


Figure 5.7: Android SDK Tools Path

3. Check the SDK versions installed in the system by executing the command as shown in code snippet 1.

Code Snippet 1:

```
android list targets
```

The command lists all the available Android platforms that are downloaded for the SDK as shown in figure 5.8.

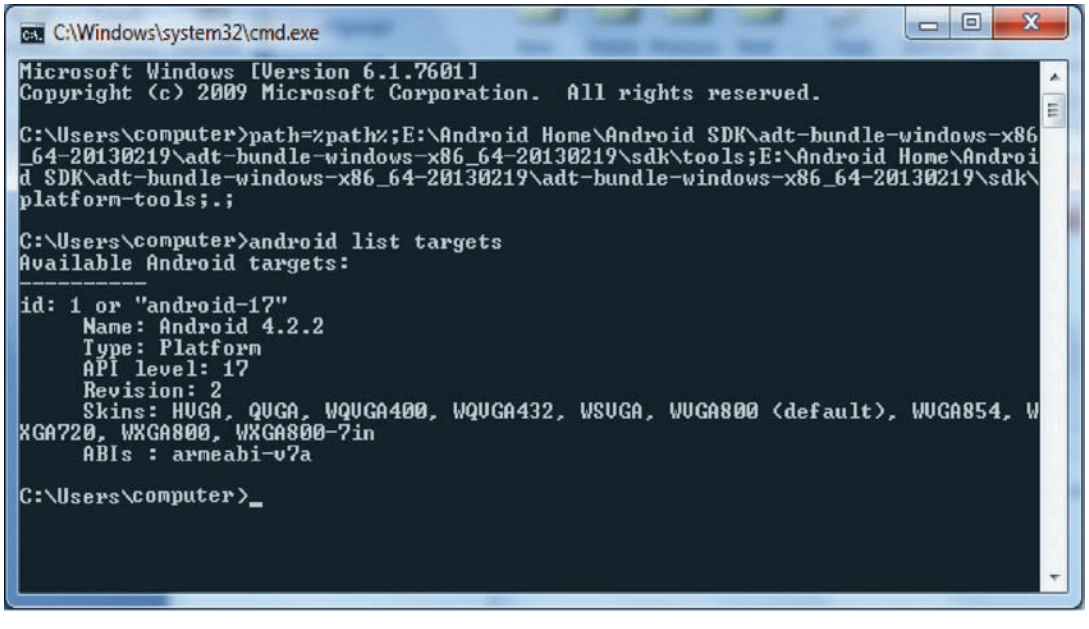


Figure 5.8: List of Available Android Platforms

Select the platform for compiling the application and note the target id. It is recommended to choose the highest version to optimize the application to latest devices. It is however, possible to select older versions to support the application but with the disadvantage of lesser optimization to new features and latest devices.

In case there are no targets listed, install a few using the Android SDK Manager tool.

4. With the build environment set, open command prompt, and type the code as shown in code snippet 2.

Code Snippet 2:

```
android create project --target <target-id> --name MyHelloApp
--path <path-to-workspace>/MyHelloAppProject --activity
MyHelloAppActivity --package com.example.myhelloapp
```

where,

- `<target-id>` will be the id selected from the list of available Android platforms
- `<path-to-workspace>` will be the location where the application is saved in Android projects.

The new Android application is created as shown in figure 5.9. Begin building the application.

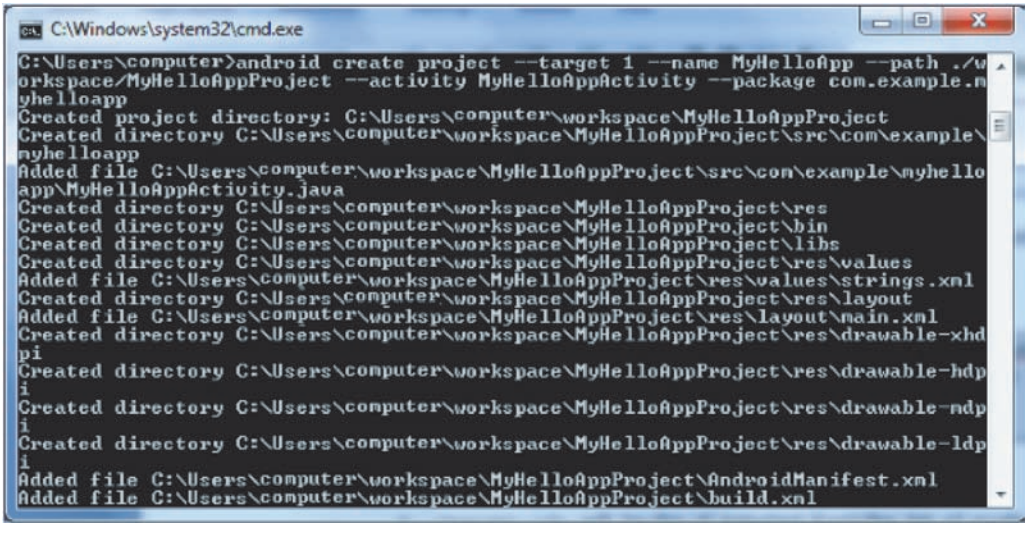


Figure 5.9: Command line – Project Creation

Concepts

5.3 How to Create an Android Test Application?

Before assembling the test environment, the basic processes to create and run Android applications with ADT are to be set up. The preliminary step in creating a test environment is to have a separate test project that stores the test code. A new test project stores the test code, resources, manifest file, and so on. The `<instrumentation>` element in the **AndroidManifest.xml** file connects the new test package to the application under test. The directory structure for the new project is same as any Android application.

Creating a test project in Eclipse with ADT has some advantages because ADT has several built-in features that help in setting up and managing the test environment successfully. The features provided by ADT include the following:

- Quick creation of the test project that links to the application being tested.
- Automatic insertion of the specific `<instrumentation>` element to the test package manifest file.
- Quick import, inspection, and analysis of the application classes that is being tested.
- Creation of run configurations of the test package.
- Insertion of flags in the run configurations to be recognized by the Android testing framework.
- Provision to run test package without leaving Eclipse.
- Automatic building of the application being tested and test package.
- Installation of the application being tested and test package on the device or emulator.
- Run the test package.
- Provision of a separate window in Eclipse to display results.

Concepts

To create a test project in Eclipse with ADT, perform the following steps:

1. Select **File → New → Other** in Eclipse as shown in figure 5.10.

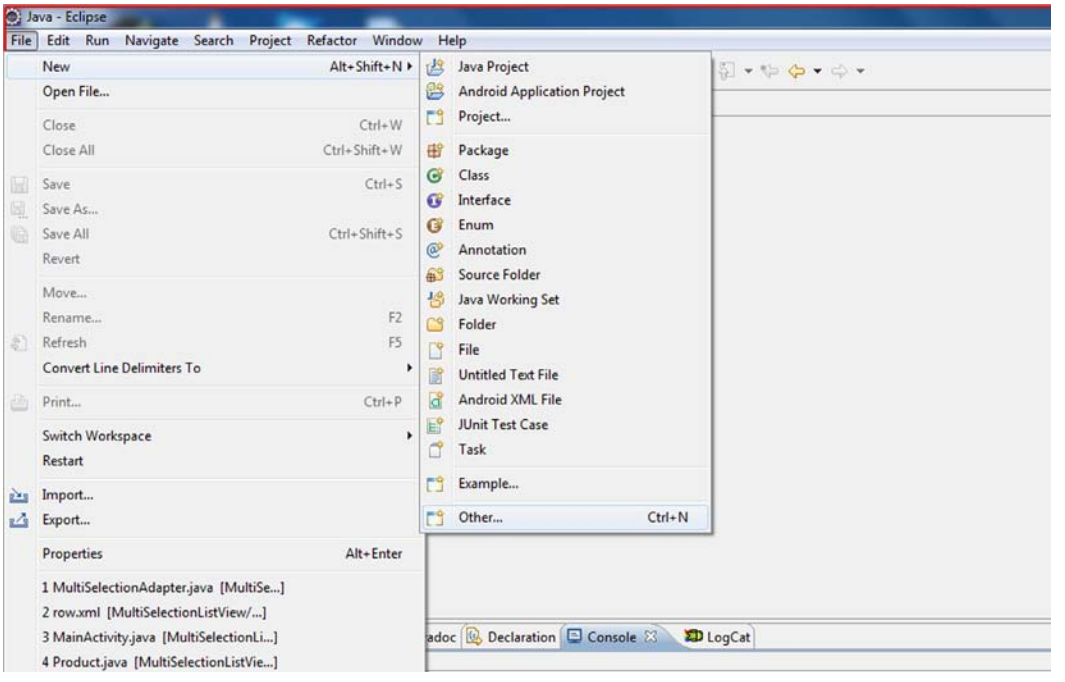


Figure 5.10: Navigating to Select a Wizard

2. In the **Select a wizard** pane of the **New** dialog box, scroll through the Wizards drop-down list to the **Android** folder.

3. Click the toggle to the left to open the list in the **Android** folder.

Concepts

4. Select **Android Test Project** as shown in figure 5.11.

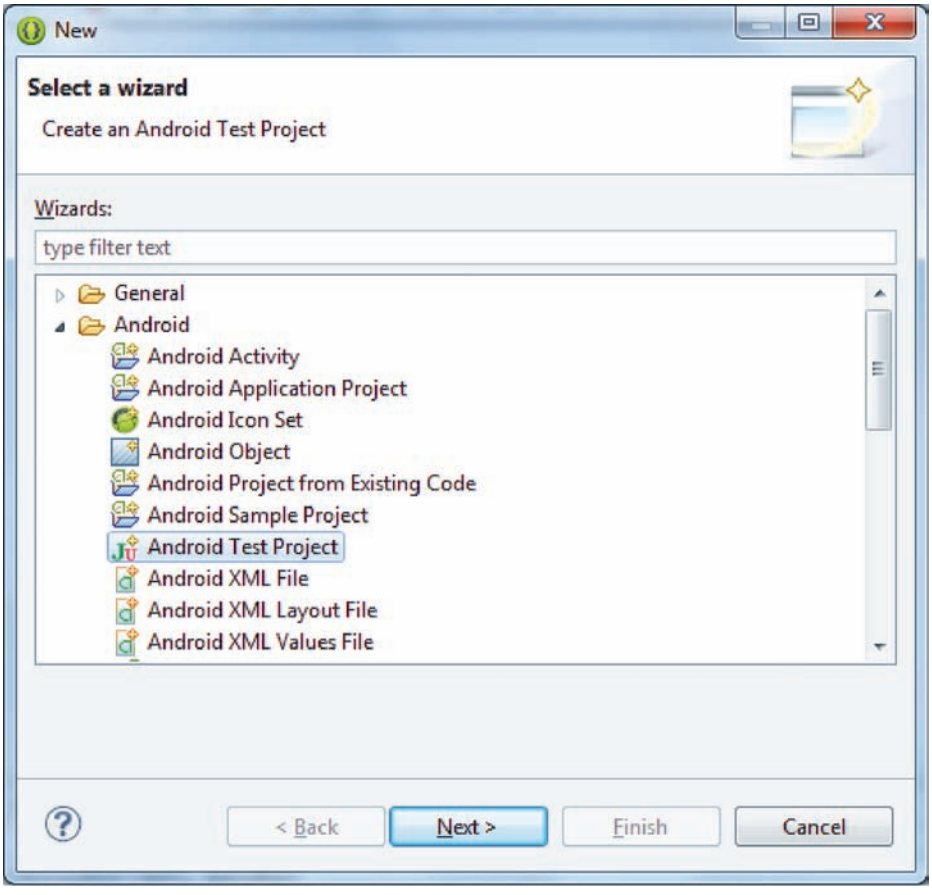


Figure 5.11: Selecting Android Test Project

5. Click **Next** to open the **New Android Test Project** wizard as shown in figure 5.12.

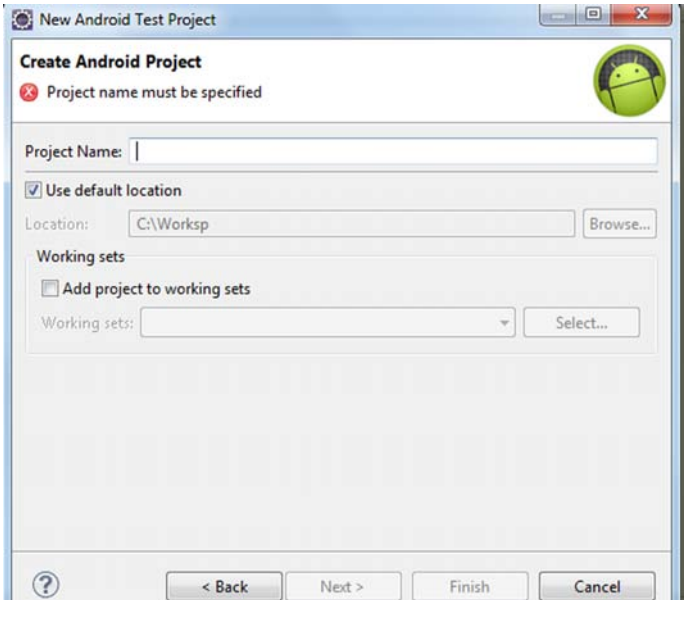


Figure 5.12: New Android Test Project Wizard

Concepts

Use the **New Android Test Project** to generate a new test project. The new project can be created any time and will have the proper structure, including the `<instrumentation>` element in the manifest file. The new project dialog box appears as soon as a new Android main application project is created. It can also be used to create a test project for an application that has been created earlier.

6. Type **`HelloAndroidTestApp`** in the **Project Name** box of the **New Android Test Project** dialog box as seen in figure 5.13.

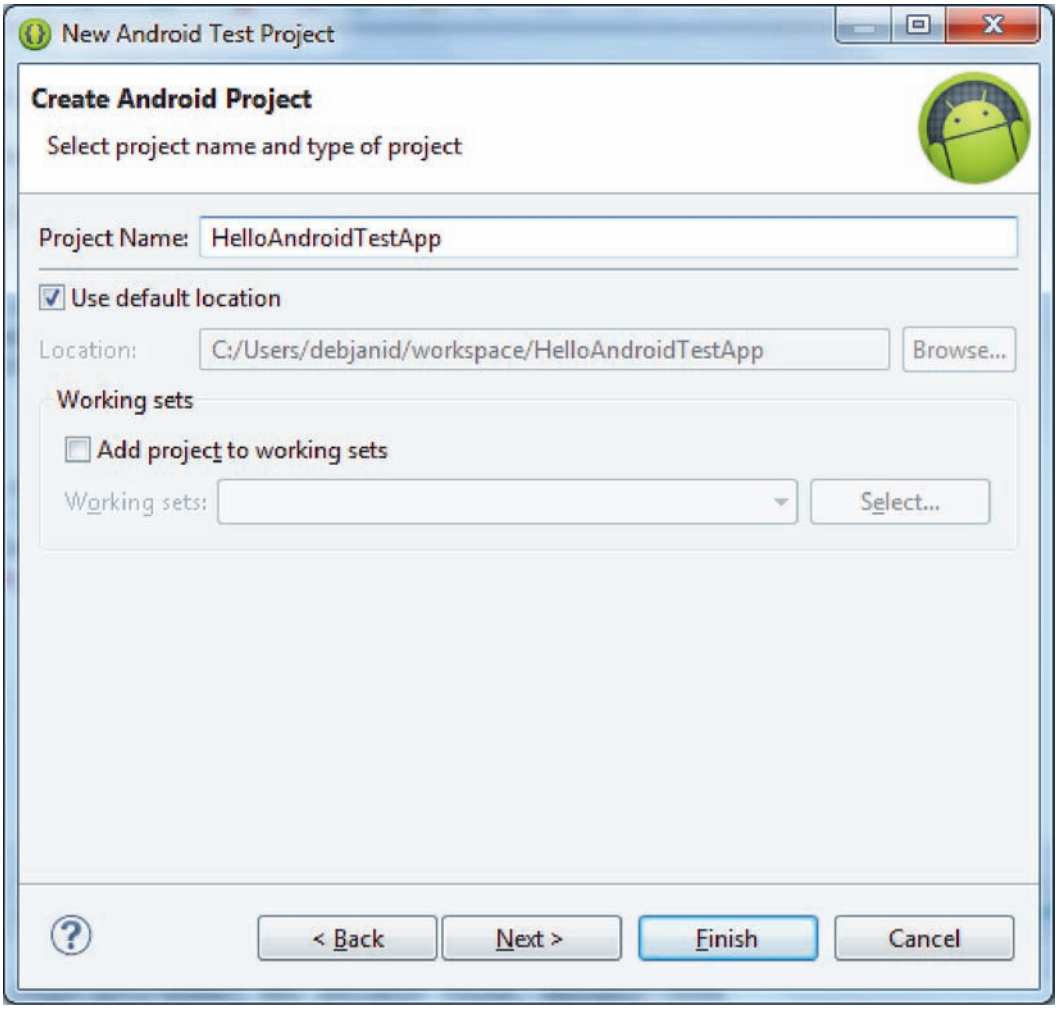


Figure 5.13: HelloAndroidTestApp

7. Click **Next**.

8. Select **An existing Android project** from **Select Test Target** pane.

9. Select the project as shown in figure 5.14.

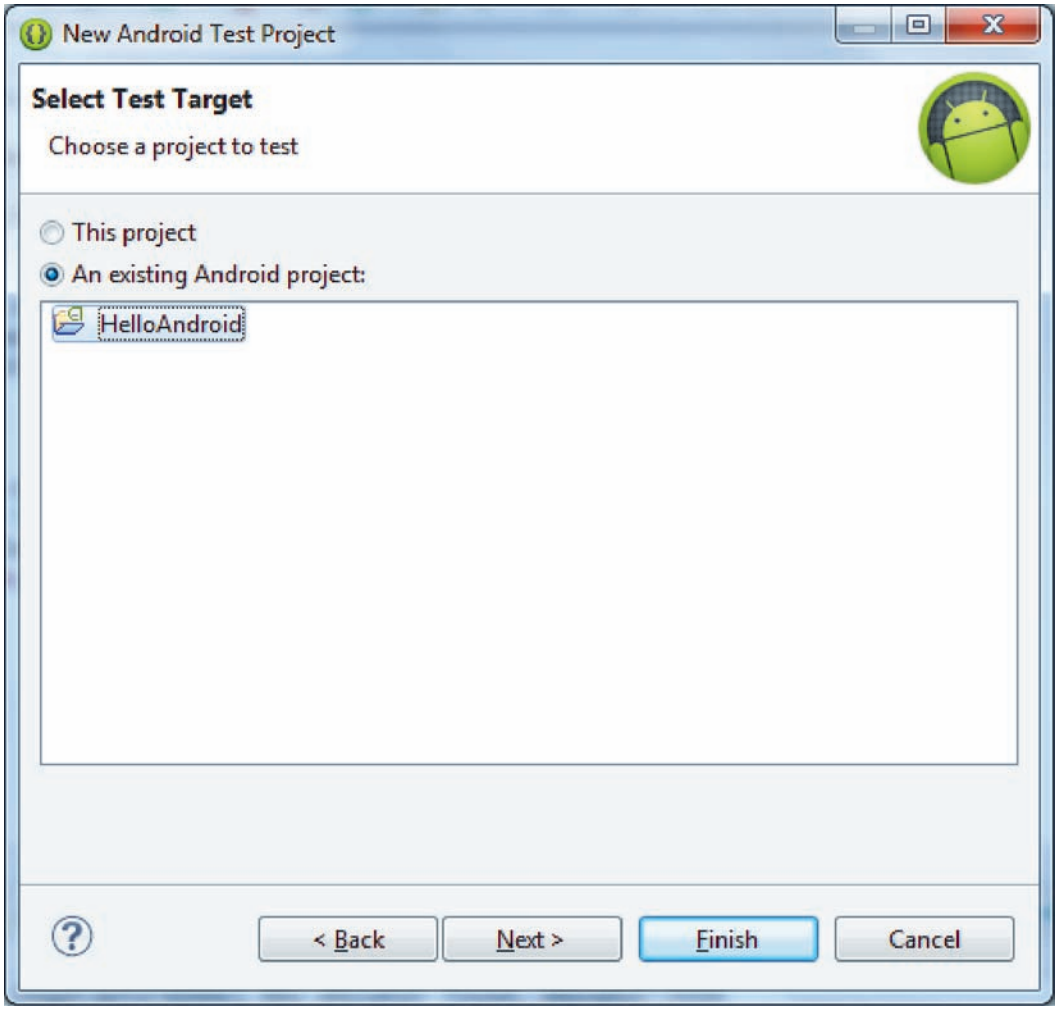


Figure 5.14: Select the project

10. Click **Finish**.

Thus, a test project is created for the existing **`HelloAndroid`** application.

5.4 How to Create a Test Case?

The main test case extends from the Android test case classes or JUnit classes because it provides the best testing features. When test packages are executed in Eclipse with ADT, its results appear in the JUnit view. Test cases can combine Activity classes, test case classes, or ordinary classes. To create a test package, use one of Android's test case classes defined in `android.test` package.

A test package does not require an `Activity`, although one can be defined. The Android test classes for Activity objects also provide instrumentation for testing an `Activity`.

To create a test case, the first step is to choose the Java package identifier used for the test case classes and the Android package name.

Concepts

To add a test case class to a test project, perform the following steps:

1. Open the test project in the **Package Explorer** pane.
2. Navigate to **test project → src** folder from the **Package Explorer** tab as shown in figure 5.15. Here it is **HelloAndroidTestApp → src → com.example.helloandroid.tests**.

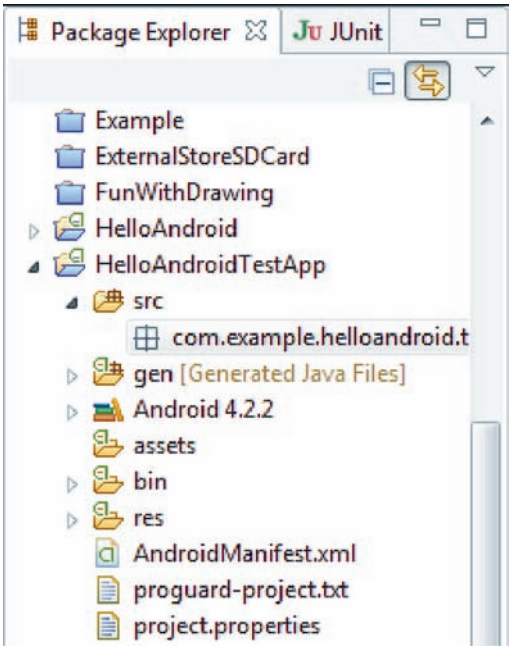


Figure 5.15: src Folder in the Project Explorer Tab

3. Right-click Java package identifier to display the context menu.
4. Select **New →** `JunitTestCase` to display the **New JUnit Test Case** dialog box.

5. Type **`HelloAndroidTestCase`** as shown in figure 5.16.

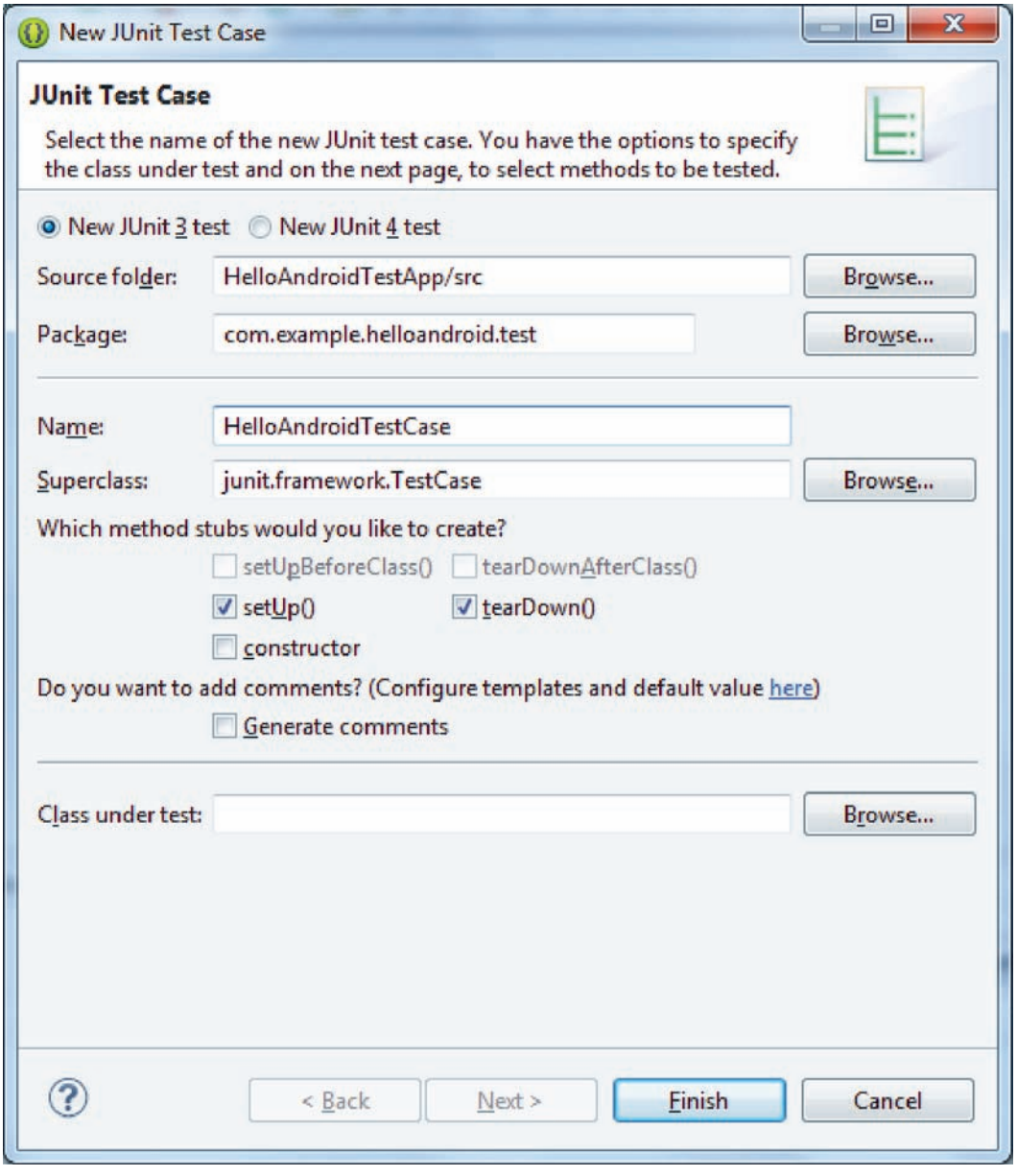


Figure 5.16: HelloAndroidTestCase

6. Click **Finish**.

7. Modify the code snippet of **HelloAndroidTestCase** as shown in code snippet 3.

Code Snippet 3:

```
package com.example.helloandroid.test;

import junit.framework.TestCase;

public class HelloAndroidTestCase extends TestCase {

    public HelloAndroidTestCase() {
        super();
        // TODO Auto-generated constructor stub
    }

    public HelloAndroidTestCase(String name) {
        super(name);
        // TODO Auto-generated constructor stub
    }

    protected void setUp() throws Exception {
        super.setUp();
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }

}
```

Ensure that the constructor for the class is set correctly with no arguments. Such a constructor is required by JUnit. Each base test case class has it unique constructor signature. Add a call to the constructor as the first statement.

To control the test environment, override the following methods:

- `setUp()`: Is the first method to be invoked in the class. Use it to perform the following functions:
  - Set up the environment for the test fixture.
  - Instantiate a new Intent with the action `ACTION _ MAIN`. Use this intent to start the Activity being tested.

- `tearDown()`: Is the last method to be invoked after all the test methods in the class. Use it to perform the following functions:
  - Waste collection.
  - Reset the test fixture.

- `testPreconditions()`: Is a method added to test class. Use it to perform the following function:
  - Correct initialization of the application being tested. The test failure of this method indicates that the initial conditions were not error-free. In this case, expect further test results, no matter whether the tests succeed or not.

5.5 How to Run the Test?

Test project can be executed from Eclipse, emulator, or the command line.

5.5.1 From Eclipse

The result of executing a test package in Eclipse with ADT is seen in the Eclipse JUnit view. Eclipse can run the entire package or a single test at a time. Remember to attach the device or use an emulator to run the test package. When using an emulator, use an AVD (Android Virtual Device) that uses the same target as the test package. When running the test package, Eclipse uses the adb command so it is similar to running tests from the command line.

There are two ways to run a test in Eclipse. The first method is used for running all tests and the second method is used for running a single test from Eclipse. These are explained as follows:

- Select **Run As** → **Android JUnit Test** from the test project context menu as shown in figure 5.17. The Android JUnit Test can also be selected from the main menu's **Run** option. Eclipse will run all the tests.

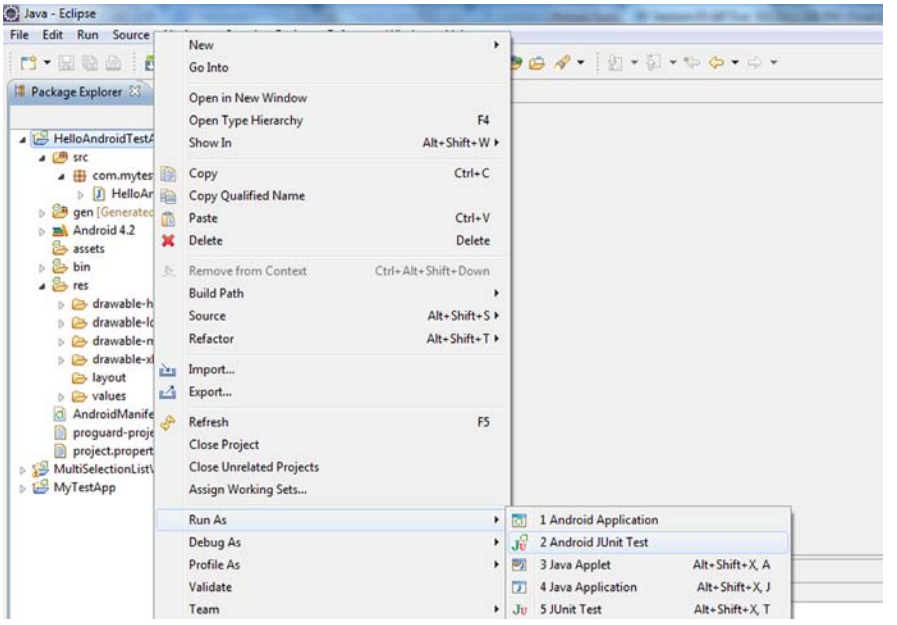


Figure 5.17: Running Hello Android

➔ Run the test by creating an Eclipse run configuration for the test project. This method is used when multiple test suites are to be tested, where each test suite consists of several selected tests from the project. Eclipse will run a single test case.

To run the test suite using run configuration, perform the following steps:

1. Select the test project in the **Package Explorer** pane.
2. Right-click the project to display the context menu and select **Run As → Run Configurations** as shown in figure 5.18.

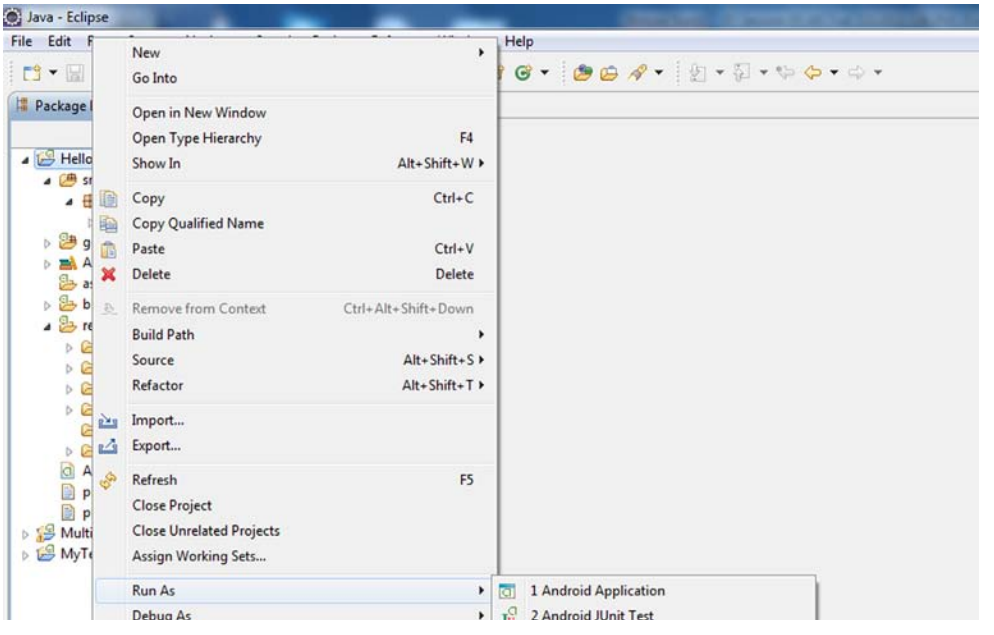


Figure 5.18: Running Test Package as Run Configurations

The **Run Configurations** dialog box is displayed.

3. Select **Android JUnit Test** entry in the left pane of the **Run Configurations** dialog box.

4. Click the **Test** tab in the right pane as shown in figure 5.19.

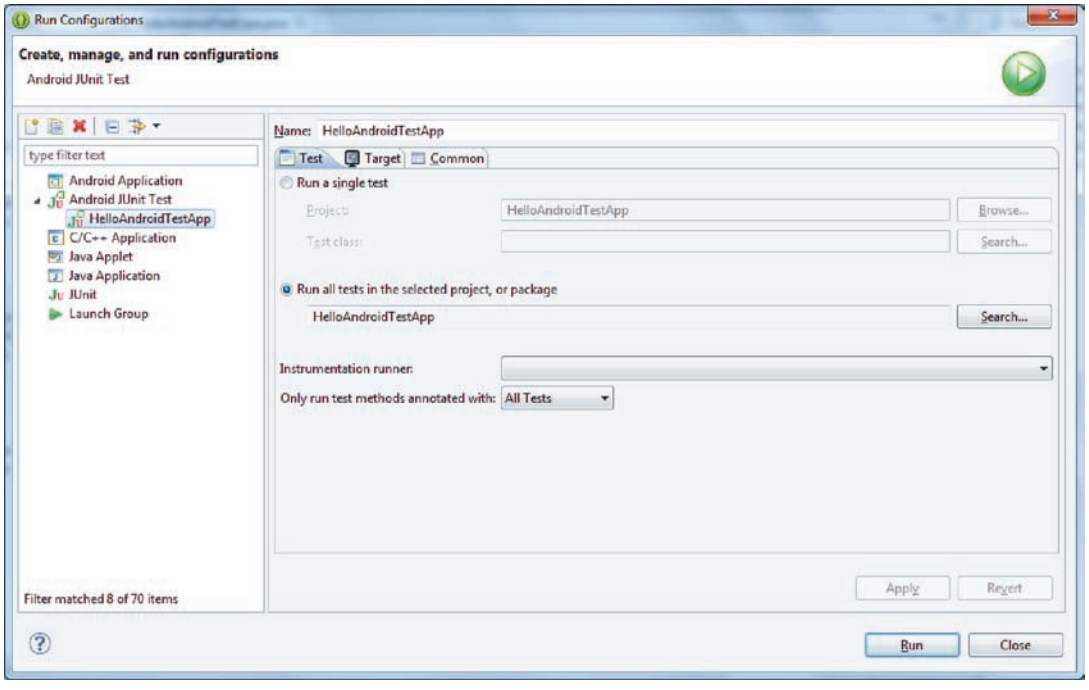


Figure 5.19: Entries in the Test Tab of the Run Configurations Dialog box

5. To run all test classes:

   - Click **Run all tests in the selected project or package**.
   - Enter the name of the project or package name in the box.
   - Select **android.test.InstrumentationTestRunner** from the **Instrumentation runner** list.

6. Click the **Target** tab in the right pane.

   - Click Automatic for deployment target selection.
   - Select an existing AVD from the AVD selection table as shown in figure 5.20. This step is optional and is performed, if using the emulator.

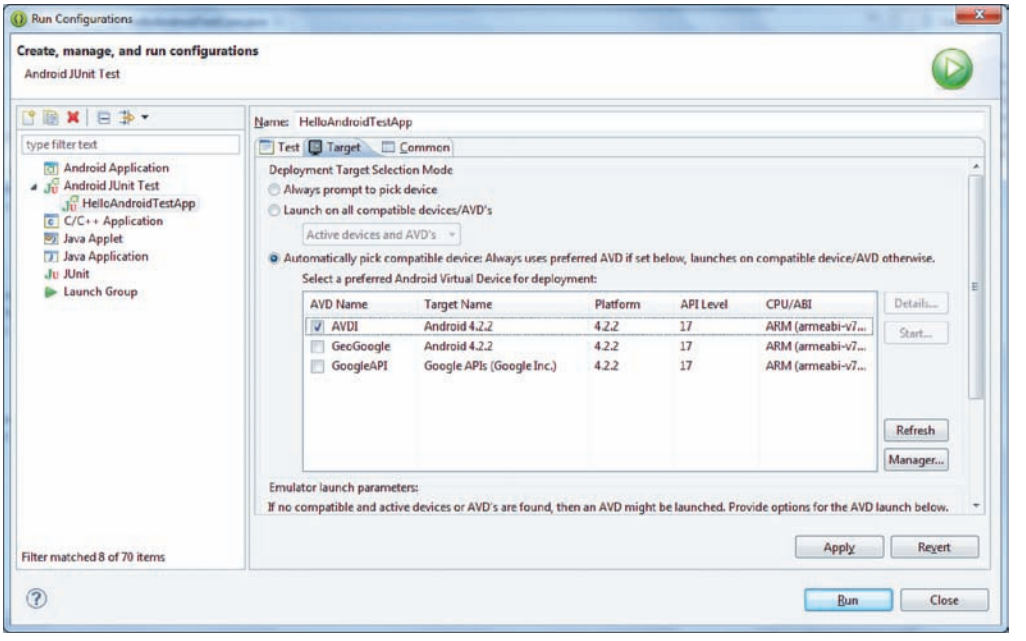


Figure 5.20: Deployment Target Selection Mode in the Target Tab

- Set the Android emulator flag to be used, in the **Emulator Launch Parameters** section as shown in figure 5.21.

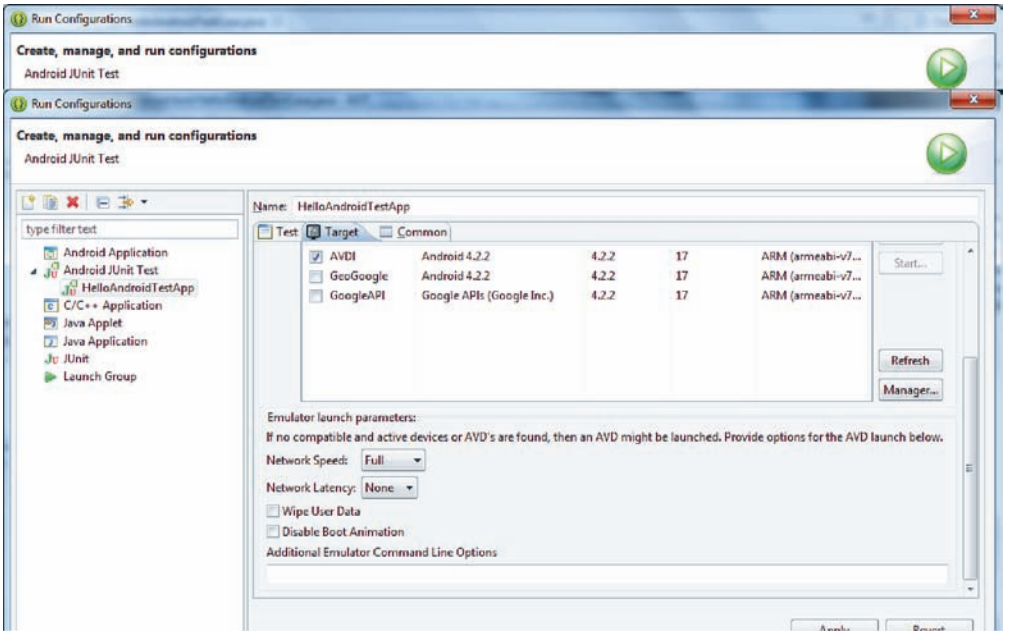


Figure 5.21: Emulator Launch Parameters in the Target Tab

Concepts

7. Click the **Common** tab in the right pane and select one of the following:
   - Click **Local** to save the configuration locally in the **Save As** pane.
   - Click **Shared** to save the configuration to another project.
8. Select the test run configuration to begin the test.

5.5.2 From Emulator

Instrumentation is the only means to run tests from the emulator. The emulator has a default system image which has the `Dev Tools` installed. The `Dev Tools` application lists the handy tools and settings.

1. Select **Instrumentation** from the `Dev Tools` as shown in figure 5.22.

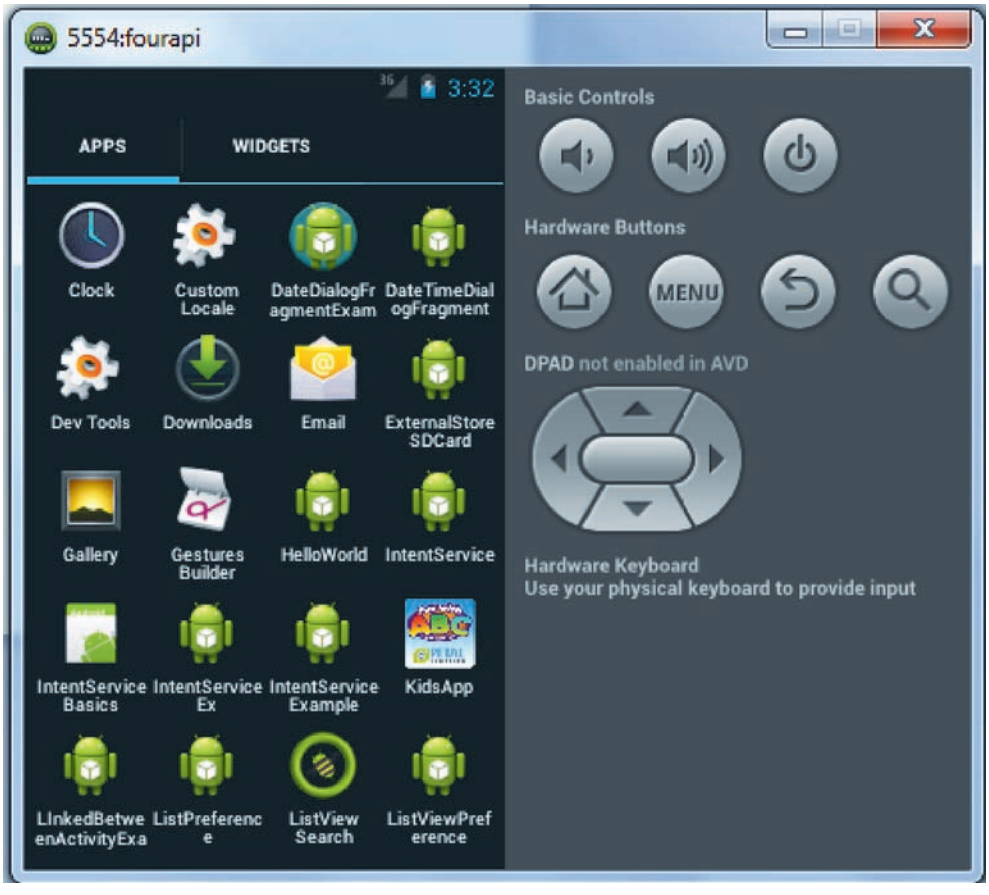


Figure 5.22: Dev Tools

2. Select **Instrumentation** as shown in figure 5.23.

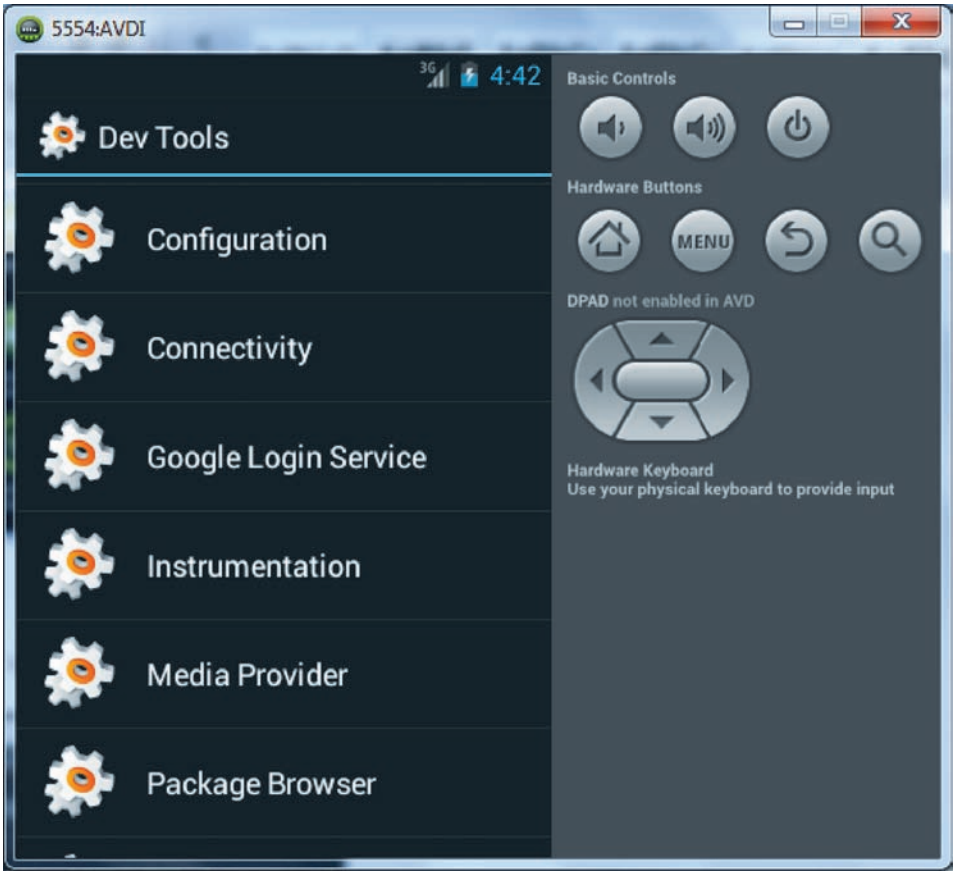


Figure 5.23: Dev Tools List

3. A list of installed packages having the `instrumentation` tag in their `AndroidManifest.xml` is listed as shown in figure 5.24.

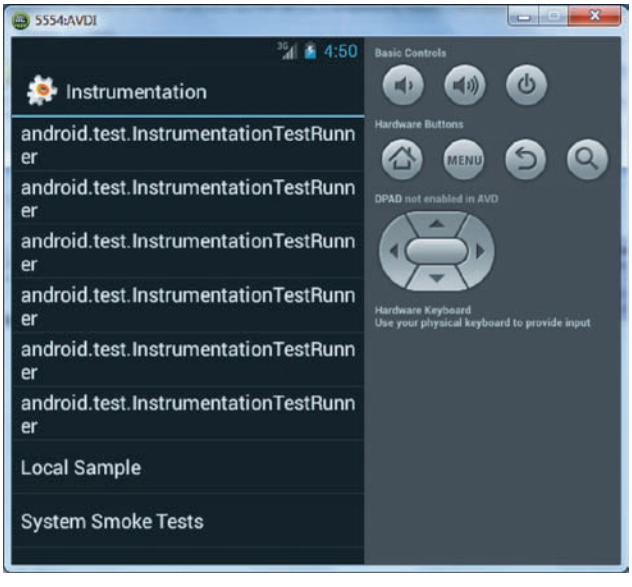


Figure 5.24: Instrumentation Tab

Set an optional label in the manifest, under the instrumentation tab, for easy identification of the test package. Remember that by default, packages are listed using `android.test.InstrumentationTestRunner` so recognizing the test package may be difficult when the list contains more than one test package.

4. Select the test with the new label after the Instrumentation list is re-displayed.

5. Run the test. The test results are displayed through `LogCat`.

5.5.3 From Command Line

There are two ways to run tests from the command line.

➔ Running with Ant

When a test project is created using the `android` tool, a `run-tests` target is created automatically. This target follows all the functions of re-building the main and test project, installing the test application to the current AVD, and then using ANT to run all test classes in the application. The test results are targeted to STDOUT.

For an existing project to use this feature, use the `update test-project` option in the `android` tool.

➔ Running with an Android Debug Bridge (adb) shell

To initiate a test run, use the `adb` shell command on the device or emulator. Next, run the `am instrument` command in the shell or command prompt. Use command-line flags to control `am` and the tests. The `adb` shell after opening on the device or emulator, and running the tests, produces the output. It, then returns to the command line on the system.

A faster way would be to use the command to start an `adb` shell, invoke `am instrument`, and specify command-line flags.

There are certain advantages of running tests from the command line using `adb` shell command. These are as follows:

- Provides more options for the tests to run than with any other method.
- Provides selection of individual methods, helps to filter tests according to their annotation or specifies testing options.
- Customizes testing with shell scripts because it provides complete control from a command line.

To run a test with `am instrument`, perform the following steps:

Use `am instrument` command to either run all test classes in the test package or all tests in a test case class. Before, beginning to run any of these tests, there are some preliminary steps that need to be followed. They are as follows:

1. Rebuild the main application and test package, if necessary.

2. Install the test package and the main application `.apk` files (Android package files) to the Android device or emulator that is to be used.

   The syntax for the `am instrument` for testing the project is as follows:

   Syntax:

```
$adb shell am instrument -w <test_package_name>/<runner_class>
```

   Table 5.2 displays the description of each parameter.

Parameter	Description
`<test_package_name>`	Android package name of the test application. Refers to the value specified in the `package` attribute of the `manifest` element in the manifest file (`AndroidManifest.xml`) of the test package.
`<runner_class>`	Android test runner class name that is to be used. Most often used runner class is `InstrumentationTestRunner`.

Table 5.2: Parameters in Running Tests from Command Line

3. Enter the code as shown in code snippet 4 to run all test classes in the test package.

   Code Snippet 4:

```
$adb shell am instrument -w com.example.helloandroid.tests/android.test.InstrumentationTestRunner
```

   Figure 5.25 displays the command and the output of execution of the command at the command prompt.

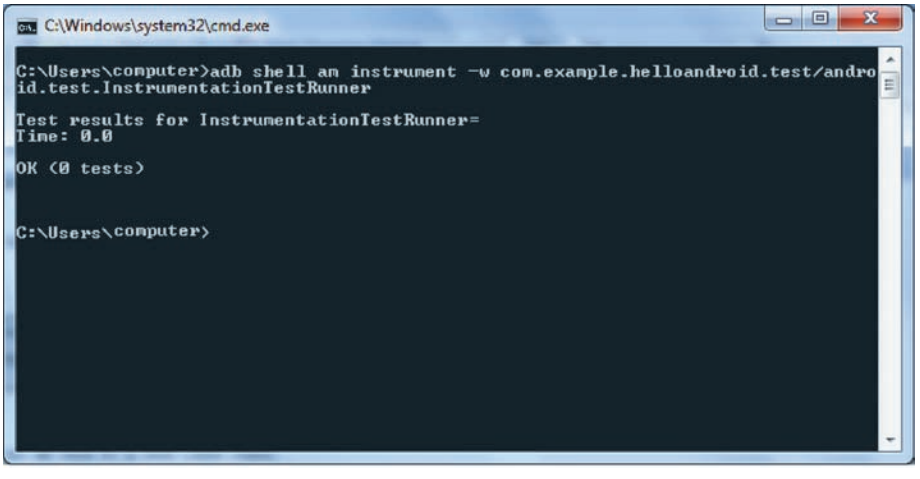


Figure 5.25: Command Line Output

   Observe the test results in STDOUT.

5.6 How to Debug the Test?

Debug techniques are implemented to remove bugs in tests. Debugging can be as simple as adding messages to `LogCat` or more complicated. Complex way of debugging is to attach the debugger to a test runner.

To debug a test, perform the following steps:

1. Select the test project.
2. Right-click the project folder and select **Debug As → Android JUnit Test** to set a break point as shown in figure 5.26.

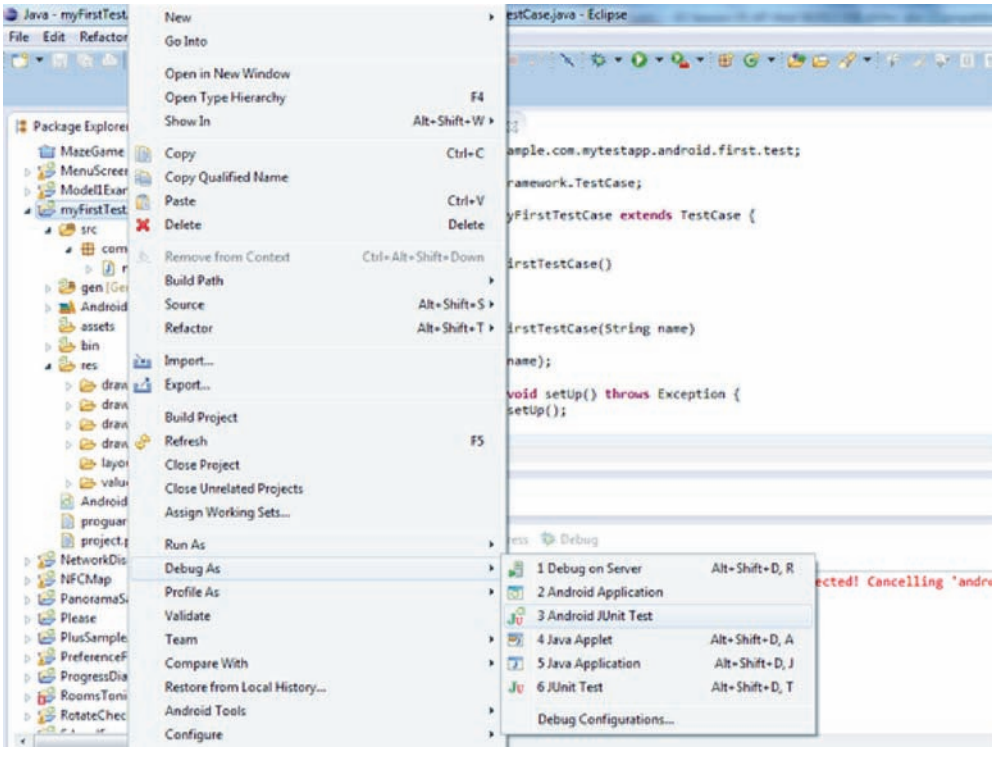


Figure 5.26: Debug a Test

3. Select the desired line in the editor where breakpoint is to be introduced.

Concepts

4. Use **Run → Toggle Line Breakpoint** to toggle a breakpoint as shown in figure 5.27 (or)

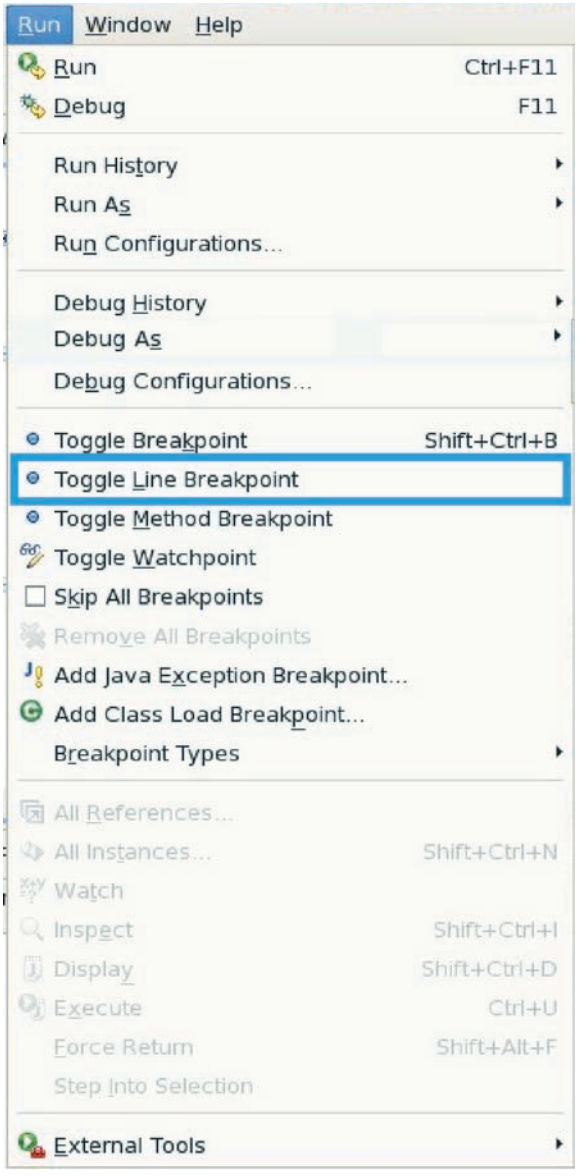


Figure 5.27: Toggle a Breakpoint

5. Modify the test codes as shown in code snippet 5 for the debugger connection. Add the code to the constructor at any location or add it to any other test method to debug.

Code Snippet 5:

```
public class HelloAndroidTestCase extends TestCase {
private static final boolean DEBUG = false;
...
     public HelloAndroidTestCase(String name) {
          super(name);
          if ( DEBUG ) {
               Debug.waitForDebugger();
               }

          // TODO Auto-generated constructor stub
...
     }
```

6. Start the tests by executing the code as shown in code snippet 6 at the command line prompt to debug the tests.

Code Snippet 6:

```
$ adb shell am instrument -w -e debug true com.example.
helloandroid.test/android.test.InstrumentationTestRunner
```

Figure 5.28 displays the output.

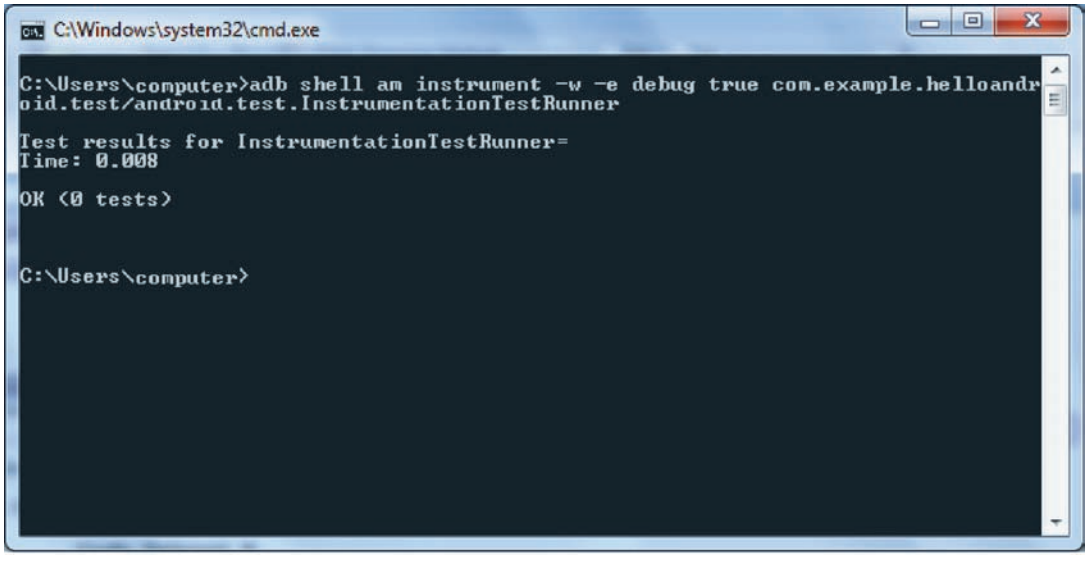


Figure 5.28: Output with Debugger

5.6.1 Example of Currency Converter

To create an Android project named **CurrencyConverter**, perform the following steps:

1. Create an Android Project named **CurrencyConverter**.

   Activity name: **MainActivity**

   Layout file: **activity_main.xml**

   Package: **com.example.currencyconverter**

2. Modify the code in **activity_main.xml** file as shown in code snippet 7.

   Code Snippet 7:

```
<RelativeLayout xmlns:android="http://schemas.android.com/
apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >

    <LinearLayout
        android:id="@+id/linearlayoutId"
        android:layout_width="200px"
        android:layout_height="280px"
        android:layout_alignParentTop="true"
        android:orientation="vertical" >

        <TextView
            android:id="@+id/TextView01"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Dollars"
            android:textStyle="bold" >
        </TextView>
```

```
        <EditText
            android:id="@+id/dollars"
            android:layout_width="100px"
            android:layout_height="50px"
            android:lines="1"
            android:numeric="integer" >
        </EditText>

        <TextView
            android:id="@+id/TextView02"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Euros"
            android:textStyle="bold" >
        </TextView>

        <EditText
            android:id="@+id/euros"
            android:layout_width="100px"
            android:layout_height="50px"
            android:lines="1"
            android:numeric="integer" >
        </EditText>
    </LinearLayout>

    <Button
        android:id="@+id/convert"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignRight="@+id/linearlayoutId"
        android:layout_below="@+id/linearlayoutId"
        android:text="Convert" >
    </Button>
```

```
    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/linearlayoutId"
        android:layout_marginLeft="24dp"
        android:layout_toLeftOf="@+id/convert"
        android:text="Clear" />

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBottom="@+id/linearlayoutId"
        android:layout_alignLeft="@+id/convert"
        android:layout_marginBottom="69dp"
        android:layout_marginLeft="40dp"
        android:text="Result is"
        android:textSize="14dp"
        android:textStyle="bold" />

</RelativeLayout>
```

3. Modify the code in **MainActivity** class as shown in code snippet 8.

Code Snippet 8:

```
package com.example.currencyconverter;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
```

```
import android.widget.RadioButton;
import android.widget.TextView;
import android.widget.Toast;

public class MainActivity extends Activity implements
OnClickListener {

    EditText dollars;
    EditText euros;

    Button convert;
    Button clear;
    TextView result;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        dollars = (EditText) findViewById(R.id.dollars);
        euros = (EditText) findViewById(R.id.euros);
        clear = (Button) findViewById(R.id.button1);
        convert = (Button) this.findViewById(R.id.convert);
        result = (TextView) findViewById(R.id.textView1);
        convert.setOnClickListener(this);

        clear.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {

                dollars.setText("");
                euros.setText("");
                result.setText("");
            }
        });
    }
```

```
@Override
public void onClick(View arg0) {
    // TODO Auto-generated method stub

    if (dollars.length() > 0) {
        convertDollarsToEuros();
    }
    if (euros.length() > 0) {
        convertEurosToDollars();
    }
    if (dollars.length() > 0 && euros.length() > 0) {
        Toast.makeText(getApplicationContext(),
            "Please filled only one filled", Toast.LENGTH_
            SHORT).show();
    }
}

protected void convertEurosToDollars() {

    if (euros.length() > 0) {
        // TODO Auto-generated method stub
        double val = Double.parseDouble(euros.getText().
        toString());
        // in a real app, we'd get this off the 'net
        result.setText(Double.toString(val / 0.67));

    } else {
        Toast.makeText(getApplicationContext(), "Please
        enter the value",Toast.LENGTH_SHORT).show();
    }
}
```

```
    protected void convertDollarsToEuros() {
        // TODO Auto-generated method stub
        if (dollars.length() > 0) {
            double val = Double.parseDouble(dollars.
            getText().toString());
            // in a real app, we'd get this off the 'net
            result.setText(Double.toString(val * 0.67));

        } else {
            Toast.makeText(getApplicationContext(), "Please
            enter the value", Toast.LENGTH_SHORT).show();
        }
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it
        is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

}
```

4. Right-click the **CurrencyConverter** project to display the context menu and select **Run As → Android Application**.

Figure 5.29 displays the output of the application.

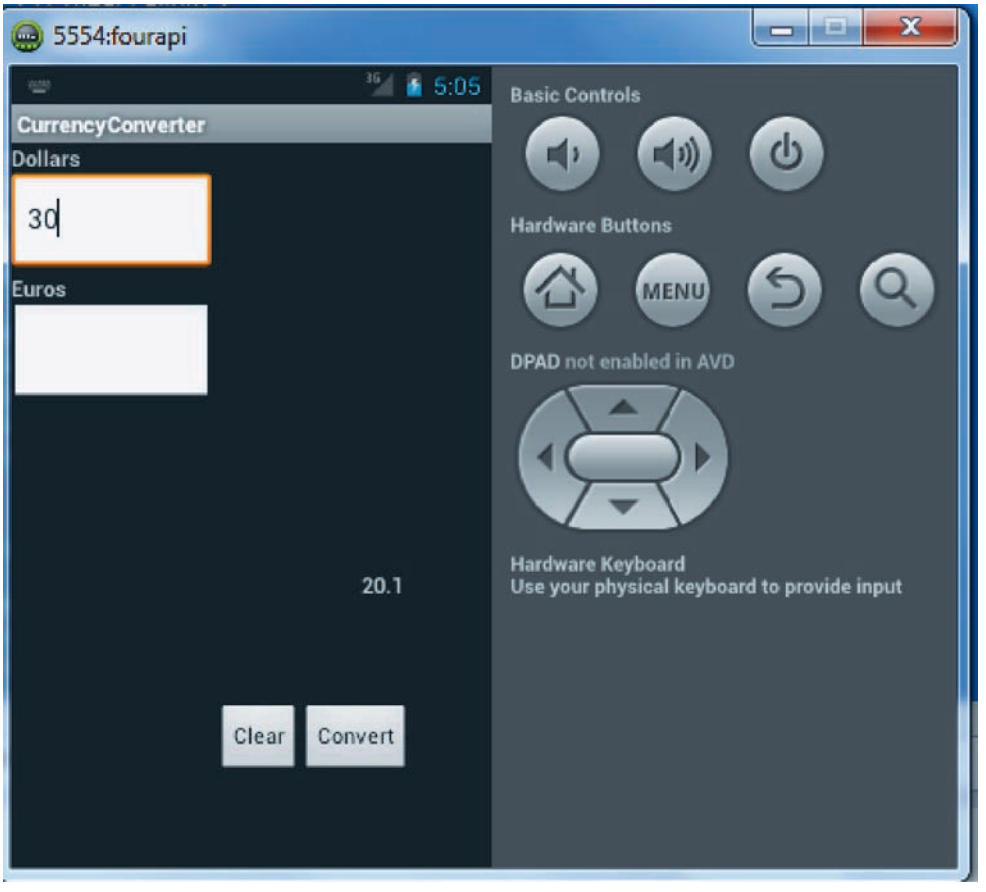


Figure 5.29: CurrencyConverter – Output

To create the Android Test Project for the **`CurrencyConverter`** application, perform the following steps:

1. Create an Android Test Project named **`CurrencyConverterTest`**.

   Package: `com.example.currencyconverter.test`

2. Create a test case class named **`CurrencyConverterActivityTest`** within the package.

3. Modify the code in the **`CurrencyConverterActivityTest`** class as shown in code snippet 9.

Concepts

Code Snippet 9:

```
package com.example.currencyconverter.test;

import com.example.currencyconverter.MainActivity;

import android.test.ActivityInstrumentationTestCase2;
import android.test.TouchUtils;
import android.test.suitebuilder.annotation.SmallTest;
import android.view.KeyEvent;
import android.widget.Button;
import android.widget.EditText;
import android.widget.RadioButton;
import android.widget.TextView;

public class CurrencyConverterActivityTest extends ActivityIns
trumentationTestCase2<MainActivity> {
    private MainActivity ConverterInstance;
    EditText dollars;
    EditText euros;
    Button convert;
    TextView result;

    @SuppressWarnings("deprecation")
    public CurrencyConverterActivityTest() {
        super("com.example.currencyconverter", MainActivity.
        class);
    }

    @Override
    protected void setUp() throws Exception {
        // TODO Auto-generated method stub
        super.setUp();
        ConverterInstance = (MainActivity) getActivity();
        result = (TextView) ConverterInstance
```

```
.findViewById(com.example.currencyconverter.R.id.textView1);
    dollars = (EditText) ConverterInstance

.findViewById(com.example.currencyconverter.R.id.dollars);
    euros = (EditText) ConverterInstance

.findViewById(com.example.currencyconverter.R.id.euros);
    convert = (Button) ConverterInstance

.findViewById(com.example.currencyconverter.R.id.convert);
}

/** Testing Conversion Operation */
@SmallTest
public void testAddition() {

    String strNum1 = "5";
    String strNum2 = "50";

    double expected = 3.35;

    /** Inputting Operand1 */
    TouchUtils.tapView(this, dollars);
    sendKeys(strNum1);

    /** Selects Add operator from the button Widget */
    TouchUtils.tapView(this, convert);
    this.sendRepeatedKeys(1, KeyEvent.KEYCODE_DPAD_
    CENTER, 1,
            KeyEvent.KEYCODE_DPAD_DOWN, 1, KeyEvent.
            KEYCODE_DPAD_CENTER);

    /** Performs calculation */
    try {
        runTestOnUiThread(new Runnable() {
```

```
                @Override
                public void run() {
                    convert.performClick();
                }
            });
        } catch (Throwable e1) {
            e1.printStackTrace();
        }

        /** Getting the result shown */
        Double actual = Double.parseDouble(result.getText().
        toString());

        assertEquals(expected, actual, 0);
    }

}
```

4. Right-click **CurrencyConverterTest** and select **Run → Run Configuration**.
5. Right-click **Android Junit Test** in the left pane and select **New**.
   - Name: **CurrencyConverterTest**.
   - Select **Run a single test**.
   - Project: **CurrencyConverterTest**.
   - Test class: **com.example.currencyconverter.test.CurrencyConverterActivityTest**.
   - Instrumentation runner: **android.test.InstrumentationTestRunner**.

Figure 5.30 displays the output of the application.

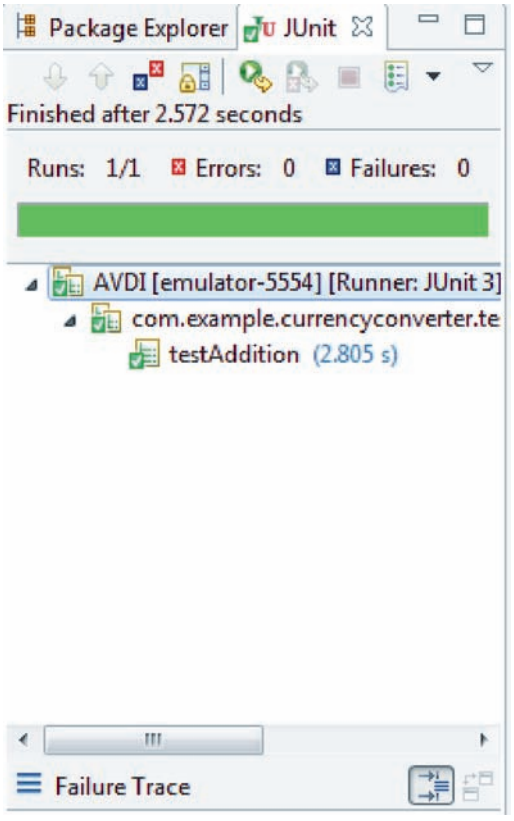


Figure 5.30: CurrencyConverterTest – Output

5.7 Check Your Progress

1. Match the following with their parameters.

(a)	Application Name	(1)	Visible name in Eclipse project directory.
(b)	Project Name	(2)	Name seen by users.
(c)	Package Name	(3)	Lowest version of Android supporting the application.
(d)	Minimum Required SDK	(4)	Highest version of Android tested with the application.
(e)	Target SDK	(5)	Reverse domain name.

(A)	a-2, b-1, c-5, d-3, e-4	(C)	a-3, b-4, c-5, d-1, e-2
(B)	a-1, b-2, c-3, d-5, e-4	(D)	a-4, b-5, c-3, d-2, e-1

2. Which of the following codes indicates the location where the application is created and saved?

(A)	`android list targets`	(C)	`$ adb shell am instrument -w \`
(B)	`android create project --target <target-id> --name myTestApp \ --path <path-to-workspace>/TestApp --activity MainActivity \ --package com.myTestApp.Android`	(D)	`$ adb shell am instrument -w com.android.demo.app.tests/android.test.InstrumentationTestRunner`

3. Where are the test results of running a test package in Eclipse with ADT is displayed?

(A)	adb shell	(C)	LogCat
(B)	Eclipse JUnit	(D)	STDOUT

4. Which of the following command is used to check the installation of SDK versions?

(A)	`list android targets`	(C)	`android targets list`
(B)	`targets android list`	(D)	`android list targets`

5. _____ target is created automatically using the `android` tool.

(A)	`run-tests`	(C)	`android.test.InstrumentationTestRunner`
(B)	`am instrument`	(D)	`LogCat`

5.7.1 Answers

1.	A
2.	B
3.	B
4.	D
5.	A

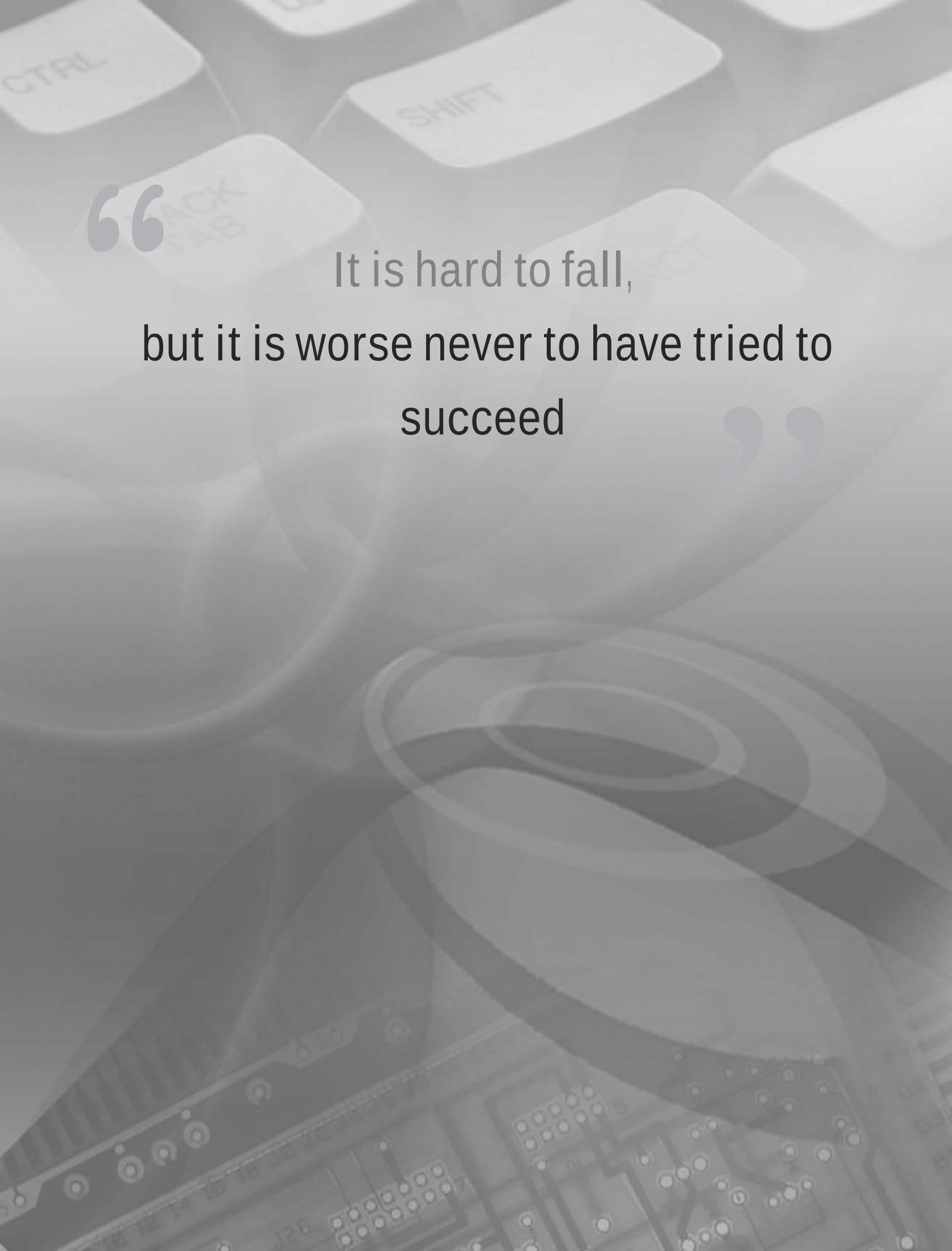
Summary

- A new Android main project can be created using Eclipse with the ADT plugin or using the SDK tools from a command line.
- A test project quickly created in Eclipse with ADT helps to link the application being tested and automatically inserts the specific <instrumentation> element to the test package manifest file.
- Main test case extends from the Android test case classes or JUnit classes as it provides the best testing features.
- Create a test package, using one of Android's test case classes defined in android test. To create a test case, the first step is to choose the Java package identifier used for the test case classes and the Android package name.
- Tests can be executed from Eclipse, emulator, or the command line. Eclipse can run the entire package or a single test at a time.
- Instrumentation is the only means to run tests from the emulator. The test results are displayed through LogCat.
- Tests from the command line is executed using ANT or with an Android Debug Bridge (adb) shell.
- Debugging can be as simple as adding messages to LogCat.

Try it Yourself

Samantha wants to create an application of weight conversion program where the user will enter the value in gram and output will be in Kilogram using JUnit Test Framework. Help Samantha to create the application and the test project. Also she should be able to perform the following activities:

1. Override the `setUp()` and `tearDown()` methods to control the test environment.
2. Demonstrate the steps to run a test suite using run configuration from Eclipse.
3. Demonstrate the steps to run the test project with `adb` shell command and also on an emulator.



“It is hard to fall,

but it is worse never to have tried to succeed”

Session - 6

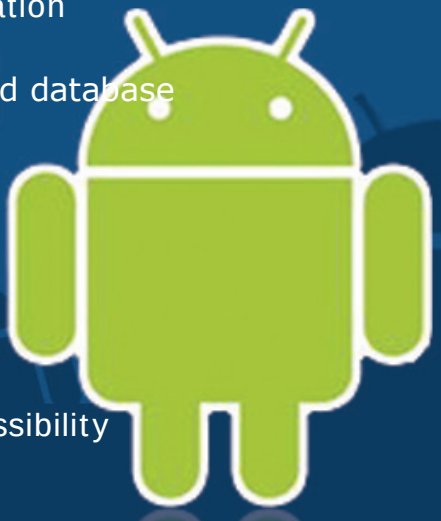
Testing for Different Situations

Welcome to the session, of **Testing for Different Situations**.

This session briefly describes the process of performing testing under different situations. Whenever you work on an application, you will encounter various situations that will need to be tested. Each situation needs to be tested in a specific way. Therefore, this session will introduce you to few such situations and also provide you necessary guidelines required to test those situations.

In this session, you will learn to:

- Describe different levels of testing units
- Explain testing an Android application
- Describe testing of file system and database
- Explain testing for exceptions
- Identify automated testing for UI
- Explain testing for memory leaks
- List the checklist for testing accessibility



6.1 Introduction

This session will provide a brief introduction of the different tests that might be required to be performed to ensure that the application works correctly. There will be few different areas under which a particular type of test would be required to be performed. For example, to ensure that the user is able to access all the different capabilities made available in an application, the developer must be testing the application under varying conditions. Testing the application in varying situations ensures that it functions under all possible situations.

6.2 Testing Units in Android

As discussed earlier, unit test refers to testing individual part of an application. While working with an Android application, sometimes it becomes essential to individually test the components of the application without disturbing the underlying system. To implement this type of testing, an Android application can be tested in three different levels. They are as follows:

- Testing Activities
- Testing Content Providers
- Testing Services

These are discussed in detail in the following section.

6.2.1 Testing Activities

An application uses `ActivityUnitTestCase<Activity>` base class to do the isolated testing of a single Activity. This kind of testing is primarily done to test the general behavior of an `Activity` class and also to provide flexibility and control over the `Activity` under test. Here, the activity to be tested is created with minimal number of connection to the underlying system. Care is also taken that the `Activity` will not interact with the other activities present in the system. In such cases, there is possibility to add mock objects and wrapped versions of the dependents of the `Activity` class. `ActivityInstrumentationTestCase2` is used to send mock Intents to the activity under test.

The application uses the `ActivityInstrumentationTestCase2` to test the functions of a single activity. In contrast to isolated testing of a single `Activity`, here the activity to be tested is created invoking the `InstrumentatonTestCase.lauchActivity()` method of the system infrastructure. In this, it is possible to directly manipulate the activity.

However, in both circumstances, the `setUp()` and the `teardown()` methods are called to automatically handle the work of an `Activity`. Overriding of the methods such as `setup()` and `teardown()` by extending the `TestCase` class has been explained in the previous session.

Concepts

The following validation checks can be performed on the `Activity` which is to be tested:

➔ **Validation of Input:** An activity can be validated whether it responds correctly to input values in an `EditText` View. Perform the following steps to test the validity of the input values of an activity.

1. Set up a keystroke sequence.
2. Send it to the activity being tested.
3. Use `findViewById(int)command` to examine the state of the View.

   To create the edittext by its id, the developer can perform the following steps:

   - Navigate to the **res/layout** folder in the android application.
   - Create the xml layout file by right-clicking the layout folder and selecting **New → Android XML file** or opening the existing layout xml file.
   - Open the layout xml file in graphical view.
   - Drag and drop the edittext view which is present in the text fields. The code generated will be as shown in code snippet 1.

     Code Snippet 1:

```
<EditText
android:id="@+id/myedittext"
android:layout_width="wrap_content"
android:layout_height="wrap_content" >
</EditText>
```

     Then in the `Activity` file the view object can be examined for its state by obtaining a reference to it using the command:

     `findViewById(R.id.myedittext)`

4. To check whether a valid keystroke sequence has been performed, the developer can create an **OK** button which will be enabled or disabled depending on the action performed. An invalid keystroke will send an error message which is visible in the View.

➔ **Lifecycle Events:** Lifecycle events trigger the `onCreate()` or `onClick()` callback methods either from the system or from the user. In other words, these are actions generated by the system or the user. When testing the activities of an application, check whether the lifecycle events are handled correctly by the activity being tested.

Concepts

- **Intents:** Each activity should correctly handle the intents which are listed in the intent filter. This intent filters are in turn specified in its **AndroidManifest.xml** file.

- **Runtime Configurations:** This includes that each activity is tested to show the desired output for all the possible changes that are performed in the device's configuration while your application is running. The changes can include a change to the device's orientation, a change to the current language, and so forth. The main goal is that any change in the device's configuration while the application is running should not affect the working of an activity.

- **Screen Sizes and Resolutions:** To enable the application to be available on different screen sizes and densities, the application needs to be tested on multiple sizes and densities using AVDs or testing it directly on the target device such as real time android phones.

6.2.2 Testing Content Providers

Content providers are supposed to be the main building blocks of the Android system as they can be viewed as data APIs that provides tables of data. The internals of these tables of data are hidden from view. These are generally responsible for guarding the data in an application from view. A content provider may have many public constants but no public variable. It may have a few public methods, if any. Thus, tests are written based only on provider's public members. The content providers are mainly used for sharing of data between one or more applications.

The `ProviderTestCase2` is the base class for content providers. It allows testing of the content provider in an isolated environment. Use the `ProviderTestCase2` to invoke the `MockContentResolver` to test the content provider. The `MockContentResolver` initiates testing by making queries to the content provider. The `setup()` and `tearDown()` methods are executed to start the initialization process before each test. Assertions are used to check retrieval of the correct data from the content provider.

Following are a few specific guidelines for testing content providers:

- Use a resolver object with the appropriate URI to test the provider. This guarantees that the provider is tested against the same interaction that an application would use.

- Test it as a contract to ensure that the provider is public and available to other applications. Ensure that testing is done for all valid URIs offered by the provider as well as invalid URIs. Also, test with constants that the provider publicly displays.

- Verify that all access methods namely query, insert, delete, update, getType, and `onCreate()` works correctly.

➔ In general, a content provider most likely does not have business logic because activities that modify the data could be implementing them. However, if the provider handles business logic such as invalid values, arithmetic calculations, and so on then they are to be tested.

6.2.3 Testing Services

The service objects in an Android system have a separate testing framework. For testing application permissions and for controlling the application and Service that is tested, Android uses the `ServiceTestCase` class. It initiates the testing process only with the assurance that mock objects are available for the tests. For this it invokes the method `ServiceTestCase.startService()` or `ServiceTestCase.bindService()`. Service is a very important component in android.

At the start of each test, the `setUp()` method for `ServiceTestCase` class is invoked to set up the test fixture. It makes a copy of the current system Context before being inspected by any of the test methods. The methods `setApplication()` and `setContext(Context)` are responsible for setting up a mock Context or mock Application (or both) for the service. Such setting isolates the test from the rest of the system.

By default, the test method named `testAndroidTestCaseSetupProperly()` is executed by `ServiceTestCase` class. This method checks whether the base test case class has successfully set up a Context before starting or not.

Follow these specific guidelines when testing services.

➔ Ensure that the methods listed in table 6.1 are invoked in response to their triggers.

Methods Invoked	Trigger Methods
`onCreate()`	`Context.startService()`or `Context.bindService()`
`onDestroy()`	`Context.stopService()`,`Context.unbindService()`,`stopSelf()`, or `stopSelfResult()`

Table 6.1: Methods Invoked while Testing Services

➔ With reference to calls, check whether the service correctly handles calls from `Context.startService()` method as listed in table 6.2.

No. Of calls	Trigger response
First call	`Service.onCreate()`
All calls	`Service.onStartCommand()`
Single call	`Context.stopService()` or `Service.stopSelf()` stops service under test

Table 6.2: Trigger Responses Invoked while Testing Services

➔ Testing of business logic present in the `Service` class which can be checked for invalid values, financial calculations, and so on.

6.3 Testing Application in Android

The base class of an Android application maintains the global state of the application whenever it deals with shared preferences. This action is undertaken with the assumption that the behavior of the real application remains unchanged though there may be some alteration in the preference values while testing.

A way to overcome this assumption is to mock a Context that mocks access to the `SharedPreferences`. The `RenamingDelegatingContext` though mocks database and files system access but it is of not much use here. So, to mock the `SharedPreferences` access, a specialized mock Context needs to be created.

As discussed in earlier session, mock `Context` are used to inject other dependencies, as well as mock or monitor other classes that are tested. All methods are non-functional and throw `UnsupportedOperationException`. The following example creates an empty mock Context that delegates to another Context, the `RenamingMockContext`. The new mock Context that is created uses a renaming prefix.

To develop the application, perform the following steps:

1. Start Eclipse.
2. Create a new project named **`MyApplication`** with the Activity name set to **`MockContextExamplesActivity`**.
3. Navigate to **src** → **com.example.myapplication** folder.
4. Modify the code in **`MockContextExamplesActivity`** file as shown in code snippet 2.

Code Snippet 2:

```
package com.example.myapplication;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import android.annotation.SuppressLint;
import android.graphics.Color;
```

Concepts

```
import android.widget.TextView;

public class MockContextExamplesActivity extends Activity {

    @SuppressLint("SdCardPath")
    public final static String FILE_NAME = "/sdcard/myfile.txt";

    private TextView tv;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_mock_context_
        examples);

        tv = (TextView) findViewById(R.id.textdisplay);

        final byte[] buffer = new byte[1024];
        try {
            File myFile = new File("/sdcard/myfile.txt");
            FileInputStream fIn = new FileInputStream(myFile);
            BufferedReader myReader = new BufferedReader(new
            InputStreamReader(
                    fIn));
            String aDataRow = "";
            String aBuffer = "";
            while ((aDataRow = myReader.readLine()) != null) {
                aBuffer += aDataRow + "\n";
            }
            tv.setText(aBuffer);
            myReader.close();

        } catch (Exception e) {

            tv.setText(e.toString());
```

Concepts

```
            tv.setTextColor(Color.RED);
        }

    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it
        is present.
        getMenuInflater().inflate(R.menu.mock_context_
        examples, menu);
        return true;
    }

    public String getText(){

        return tv.getText().toString();

    }
}
```

5. Navigate to **src** → **com.example.myapplication** folder.
6. Right-click the folder to display the context menu.
7. Select **New** → **Class** to create a new class named **`MyMockContext.java`**.
8. Modify the code as shown in code snippet 3.

Code Snippet 3:

```
package com.example.myapplication;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import android.content.Context;
import android.test.RenamingDelegatingContext;
import android.util.Log;
```

```
public class MyMockContext extends RenamingDelegatingContext {

    private static final String TAG = "MyMockContext";

    private static final String MOCK_FILE_PREFIX = "test.";

    public MyMockContext(Context context) {

        super(context, MOCK_FILE_PREFIX);
        makeExistingFilesAndDbsAccessible();

    }
    /*
     * (non-Javadoc)
     *
     * @see
     * android.test.RenamingDelegatingContext#openFileInput(
       java.lang.String)
     */

    @Override
    public FileInputStream openFileInput(String name)
        throws FileNotFoundException {

        Log.d(TAG, "actual location of " + name +
        " is " + getFileStreamPath(name));
        return super.openFileInput(name);

    }

    @Override
    public FileOutputStream openFileOutput(String name, int
    mode)
        throws FileNotFoundException {
```

```
        Log.d(TAG, "actual location of " + name +
        " is " + getFileStreamPath(name));
        return super.openFileOutput(name, mode);

    }

}
```

9. Navigate and open **AndroidManifest.xml** file.

10. Modify the code as shown in code snippet 4.

Code Snippet 4:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.example.myapplication"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.myapplication.
            MockContextExamplesActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.
                action.MAIN" />
```

```
                <category android:name="android.intent.
                category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
<uses-permission android:name="android.permission.READ_
EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.WRITE_
EXTERNAL_STORAGE"/>

</manifest>
```

11. Navigate to **res → layout** folder.

12. Modify the code in **activity_mock_context_examples.xml** file as shown in code snippet 5.

Code Snippet 5:

```
<RelativeLayout xmlns:android="http://schemas.android.com/
apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MockContextExamplesActivity" >

    <TextView
        android:id="@+id/textdisplay"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />

</RelativeLayout>
```

13. Select **Window → Android Virtual Device Manager** to display the **Android Virtual Device Manager** dialog box.

14. Select the AVD.
15. Click **Start** to display the **Launch** dialog box.
16. Click **Launch** from the **Launch** dialog box.

> **Note** - The emulator starts after some time.

17. Create a text file named **`myfile.txt`** with the content as **This is an example**.
18. Once the emulator has started move to **DDMS** perspective.
19. Click **File Explorer** tab in **DDMS** perspective.
20. Select the **emulator** from the **Device** pane on the left.
21. Select **sdcard** from **File Explorer** tab on the right pane.
22. Click **Push a file onto the device** icon to display the **Put File on Device** dialog box.
23. Navigate to the desired location and select the file **`myfile.txt`**.
24. Click **Open**.

The file will be placed in the **sdcard**. Select the application and execute the application to display the output as shown in figure 6.1.

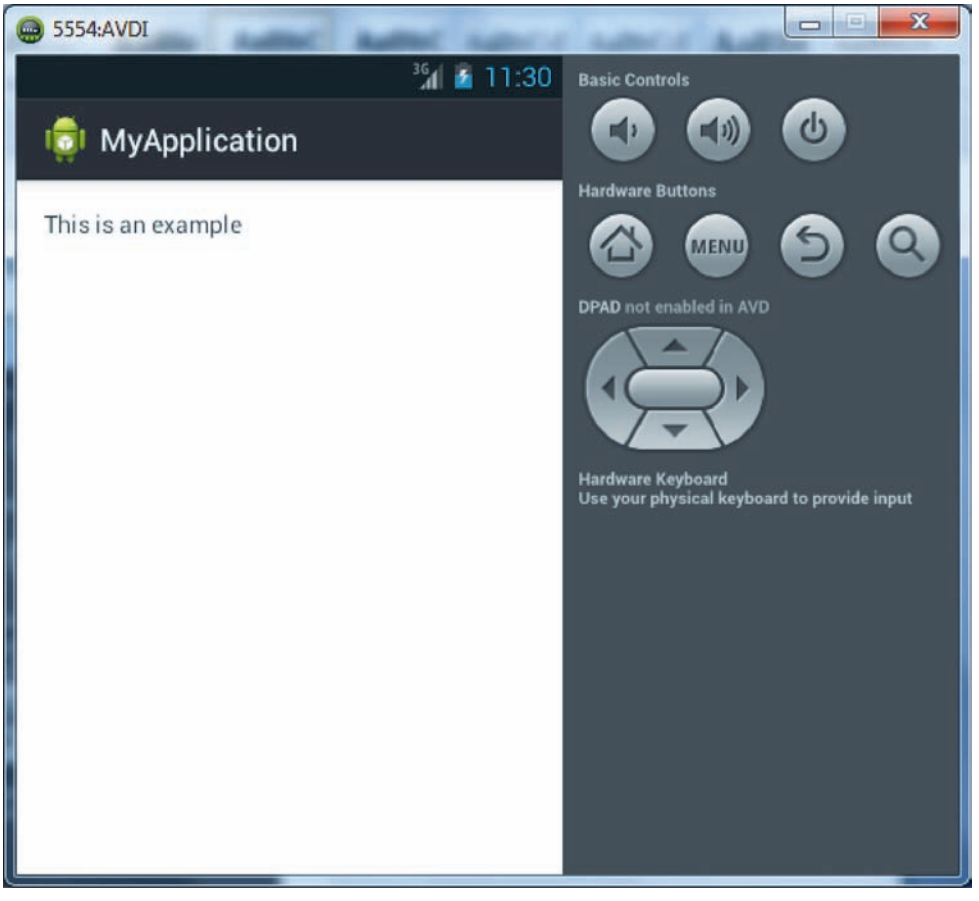


Figure 6.1: Read Data from the SD Card

To create the Android Test Project for the application created named **MyApplication**, perform the following steps:

1. Create an Android Test Project named **MyApplicationTest**.

   Package: **com.example.myapplicationtest.test**

2. Create a test case class named **MockContextExamplesTests** within the package.

3. Modify the code in the test case class as shown in code snippet 6.

   Code Snippet 6:

```
package com.example.myapplication.test;

import com.example.myapplication.MockContextExamplesActivity;
import com.example.myapplication.MyMockContext;

import android.content.Context;
import android.content.Intent;
import android.test.ActivityUnitTestCase;

public class MockContextExamplesTests extends
        ActivityUnitTestCase<MockContextExamplesActivity> {

    public MockContextExamplesTests() {
        super(MockContextExamplesActivity.class);
    }

    protected void setUp() throws Exception {
    }

    protected void tearDown() throws Exception {
    }

    public void testSampleTextDisplayed() {
        Context mockContext = new MyMockContext(getInstrumenta
        tion()
                .getTargetContext());
        setActivityContext(mockContext);
```

```
        startActivity(new Intent(), null, null);
        final MockContextExamplesActivity activity =
        getActivity();
        assertNotNull(activity);
        assertFalse("This is an Example".equals(activity.
        getText()));
    }

    public void testRealTextDisplayed(){
        //real context
        setActivityContext(getInstrumentation().
        getTargetContext());
        startActivity(new Intent(), null, null);
        final MockContextExamplesActivity activity =
        getActivity();
        assertNotNull(activity);
        assertFalse("This is an Example".equals(activity.
        getText()));
    }

    public void testDivisionByZero(){
        try{
            int n=2/0;
            fail("Division per zero.");
        }catch(ArithmeticException success){
            assertNotNull(success.getMessage());
        }
    }

}
```

Once you execute the code using JUnit Framework the output will be as shown in figure 6.2.

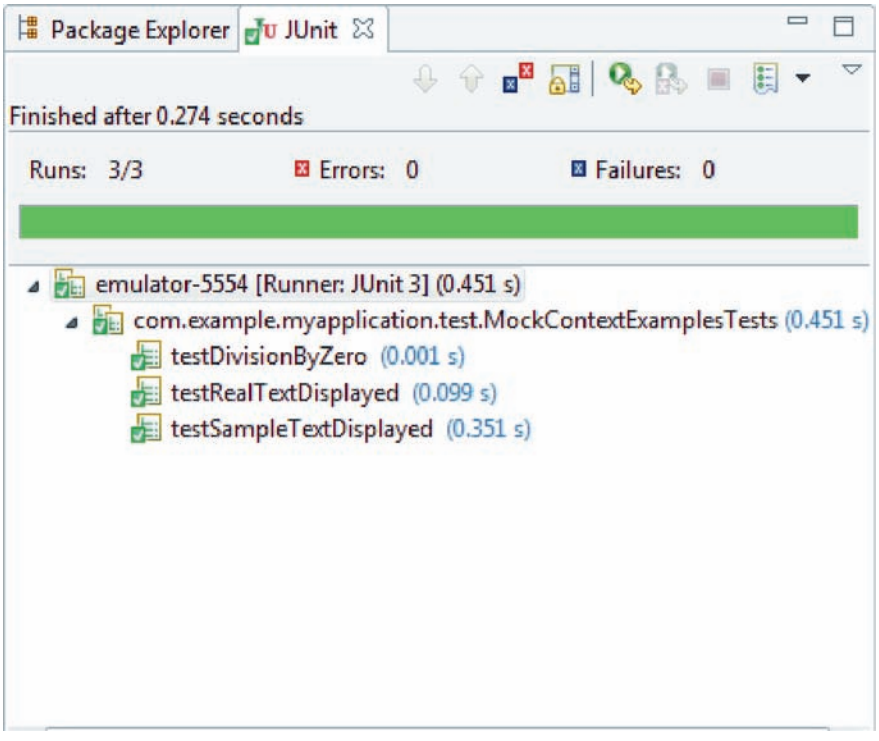


Figure 6.2: Test Case Application – Output

6.4 Testing Filesystem and Database in Android

Database and filesystem operations are tested so as to check whether they function correctly. Sometimes, when test cases exercise databases or content provider operations, it may be wise to test these operations either at a lower level in isolation or at a higher level through `ContentProviders` or from the application itself. For this Android mocks such operations to keep the `Activity` class under test unaltered.

Consider these examples:

- Testing an application on a real device such as changed values might be shared by more than one application. Here, testing needs to be isolated (separated) so that it does not affect other normal operations performed on the device.

- Testing an activity where some files and databases need to be controlled to introduce a specialized content. The specialized content is required to perform and drive the tests but cannot be used in real files or database.

Both these cases can be solved using the `RenamingDelegatingContext`. This is a mock class that is a part of the `android.test` package. It's function is to mock files and database operations. This is done by adding a prefix to the target file and database

operations. The prefix is supplied in the constructor. Also, all other operations that are delegated to the delegating Context need to be specified in the constructor. It basically allows the developer to replace the context used when invoking the file system methods by re-directing the call to a mocked data file instead of the real file.

6.5 Testing for Exceptions in Android

During the execution of a test, some conditions or methods need to be tested for exceptions and wrong values. In Android, the assert method compares the expected and actual test results. It throws up exceptions when there is a difference between results. JUnit provides a JUnit class named `ExpectedExceptions` that can be used to test an exception that is expected.

6.6 Testing UIs in Android

It is important to test the User Interface (UI) of an application to ensure that the UI functions correctly. It needs to respond and provide the correct output in response to the user's actions. It can be a sequence of actions on the device which the user desires such as selecting menus, responding to dialogs, and so on.

Android has a built-in automated UI testing tool. The tool consists of the following two main components:

- **uiautomatorviewer GUI tool:** which scans and analyzes the GUI components of the application.
- **uiautomator Java library:** which provides the necessary APIs for automated, customized UI tests, and an execution engine to automate and run them.

These tools are available only if the following versions of the Android development tools are installed:

- Android SDK tools, Revision 21 or higher.
- Android SDK platform, API 16 or higher.

To check the platform availability in your android sdk, go to the toolbar in the eclipse and click the Android SDK Manager. Then one can check in the Android SDK Manager windows whether the correct platform has been installed in the status column. If not installed then select it and click install. It would take some time for it to be installed but it would be installed.

For automated testing of the UI, perform the following:

1. Ensure that the preliminary tasks are performed before the test can start. Such as:
   a. Installation of the application on a test device.
   b. Analyzation of the UI components of the application.

c. Checking the accessibility of the application by the test automation framework.

2. Create automated tests. These tests should simulate the specific user interaction of the application.
3. Compile test cases into a JAR file.
4. Install the file along with the application on the test device.
5. Run the tests.
6. View the test results.
7. Perform the bug fixing if any bugs are encountered.

6.7 Testing for Memory Leaks in Android

Checking the memory consumption by an application is important to test the working of an application which is often referred as the test target. Before checking for this, it is necessary to detect the memory used by different classes.

To check for memory usage in Android, perform the following steps:

1. Navigate to **Window → Open Perspective → Other** as shown in figure 6.3.

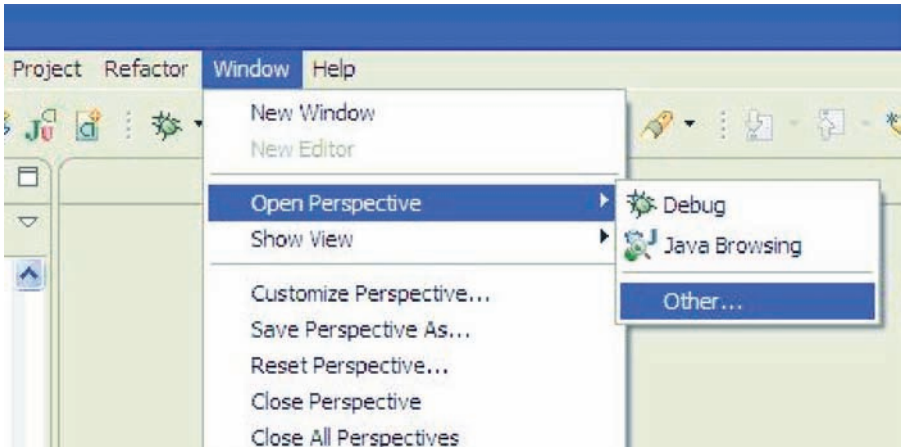


Figure 6.3: Navigating to Other Perspective

Another way to navigate to DDMS perspective is by selecting the icon next to **Java** in the top right corner of the toolbar in Eclipse. Once the icon is clicked, the **Open Perspective** dialog box is displayed and the user can click **DDMS**. After selecting DDMS, the user clicks **OK** to display the DDMS perspective.

2. Select **DDMS** from **Open Perspective** dialog box as shown in figure 6.4.

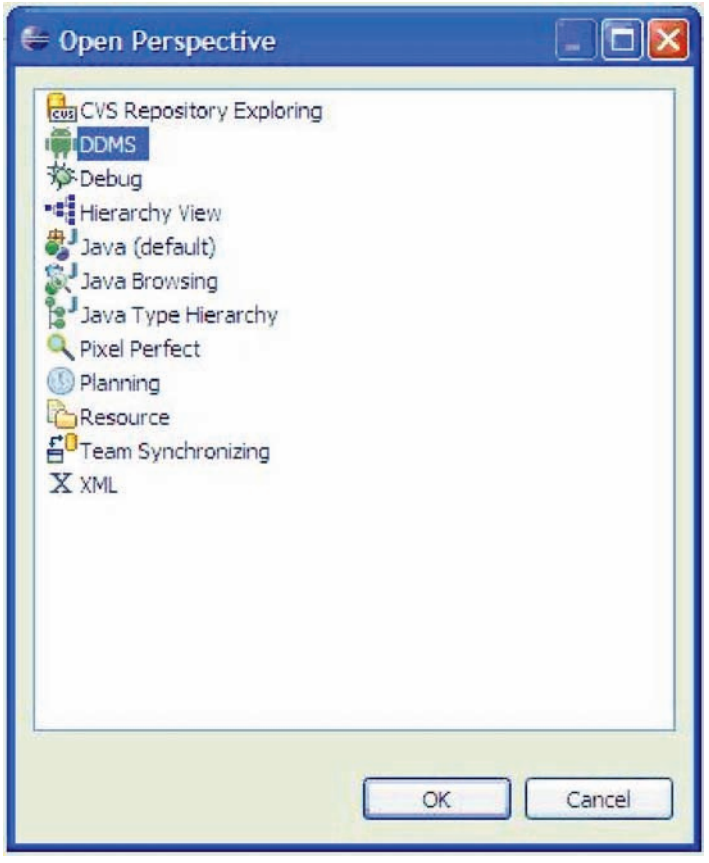


Figure 6.4: Other Perspective Dialog

3. Click **OK** to display the DDMS perspective as shown in figure 6.5.

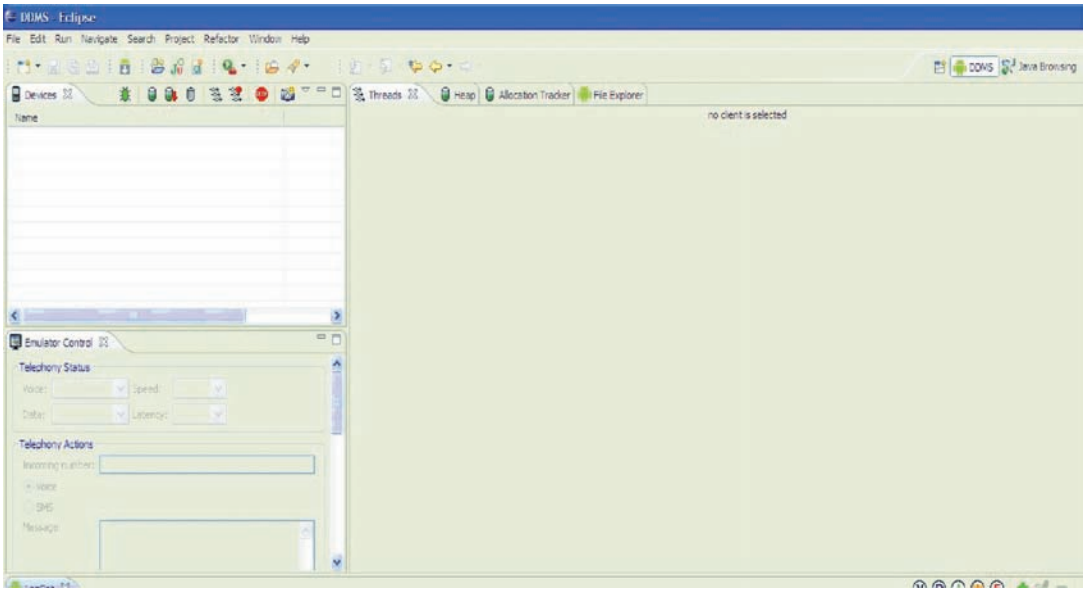


Figure 6.5: DDMS View

Concepts

4. Select the required process name in the left panel.
5. Click **Updates heap** icon from left pane as shown in figure 6.6.
6. Select **Heap** tab from the right pane as shown in figure 6.6.

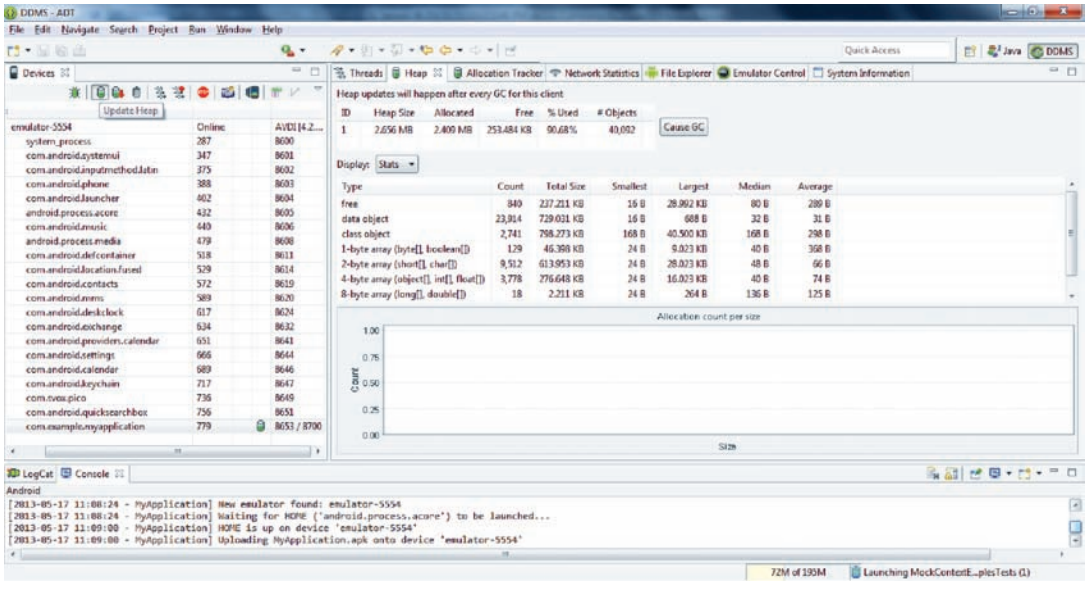


Figure 6.6: Selecting Heap

7. Click **Dump HPROF file** as shown in figure 6.7.

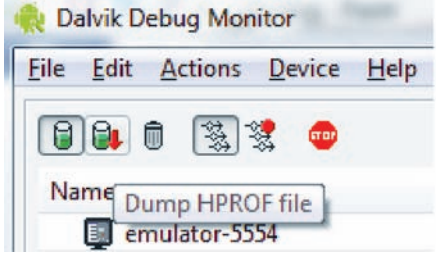


Figure 6.7: Dumping HPROF File

8. Save in desktop.
9. Open the command prompt.
10. Set the path to SDK tools folder.
11. Type in the command:

    cd C:\android-sdk\tools (or whichever directory that contains your Android sdk)

    or

    Set the path of **Android sdk tools** directory in the command prompt using the **`path`** command.

12. Assuming that the Dalvik file is present on the Desktop, type in the following command:

    hprof-conv "C:\Users\Desktop\heap-dump-tm-pid.hprof" "C:\Users\Desktop\4mat.hprof"

    This will generate the standard format of HPROF file that is named as **`4mat.hprof`** as shown in figure 6.8.

Figure 6.8: Convert Standard Format HRROF file

Note - The **hprof-conv** tool converts the HPROF file that is generated by the Android SDK tools to a standard format so that one can view the file in a profiling tool of your choice.

13. Download Eclipse MAT plugin from the following site:

    http://www.eclipse.org/mat/downloads.php

14. Open the saved HPROF file using MAT as shown in figure 6.9.

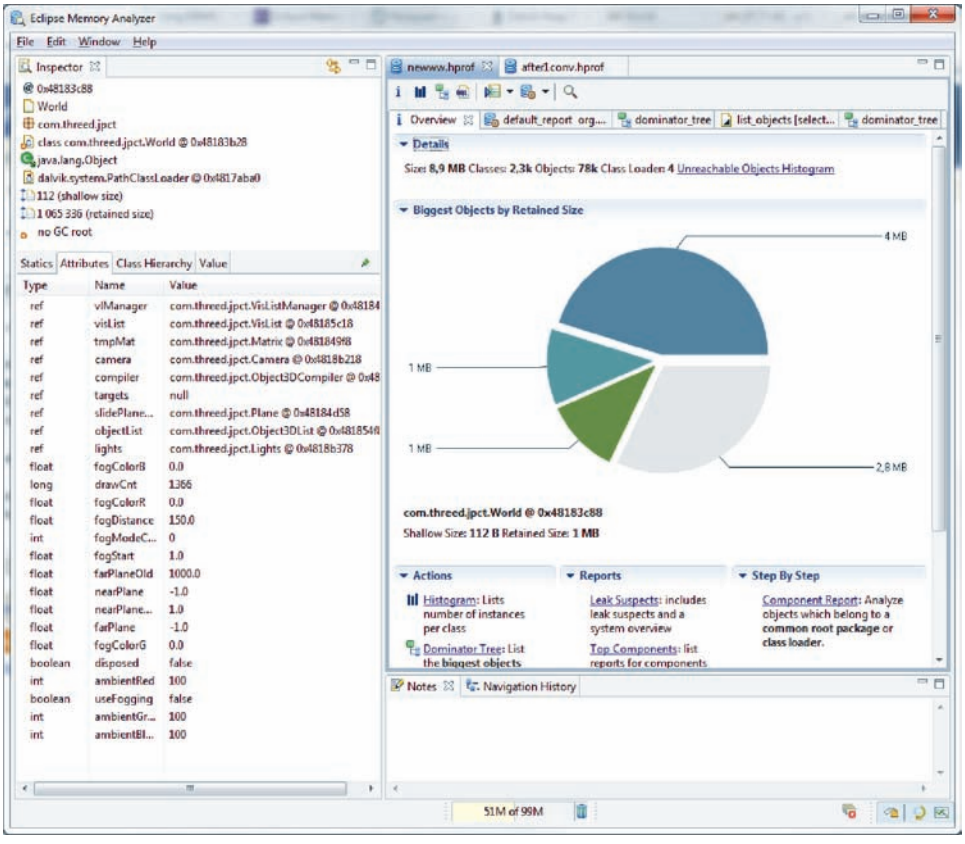


Figure 6.9: Opening Saved HPROF File in MAT

This will help the developer to view and detect the class using the memory most.

The following application displays leakage in memory. To develop the application, perform the following steps:

1. Start Eclipse.
2. Create a new project named **`AvodingMemoryLeaks`** with the Activity name set to **`LeakedActivity`**.
3. Navigate to **src → com.example.avoidingmemoryleak** folder.

Concepts

4. Modify the code in `LeakedActivity` file as shown in code snippet 7.

Code Snippet 7:

```
package com.example.avodingmemoryleaks;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.app.AlertDialog;
import android.content.DialogInterface;

public class LeakedActivity extends Activity {


    @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_leaked);

            AlertDialog.Builder builder = new AlertDialog.
            Builder(this);
            builder.setIcon(android.R.drawable.ic_dialog_
            alert);
            builder.setMessage("This dialog leaks!").
            setTitle("Leaky Dialog")
                .setCancelable(false)
                .setPositiveButton("Ok", new
                DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface
                    dialog, int which) {
                }
            });

        AlertDialog alert = builder.create();
        alert.show();
    }
```

```
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it
        is present.
        getMenuInflater().inflate(R.menu.leaked, menu);
        return true;
    }

}
```

Figure 6.10 displays the output when the code is executed.

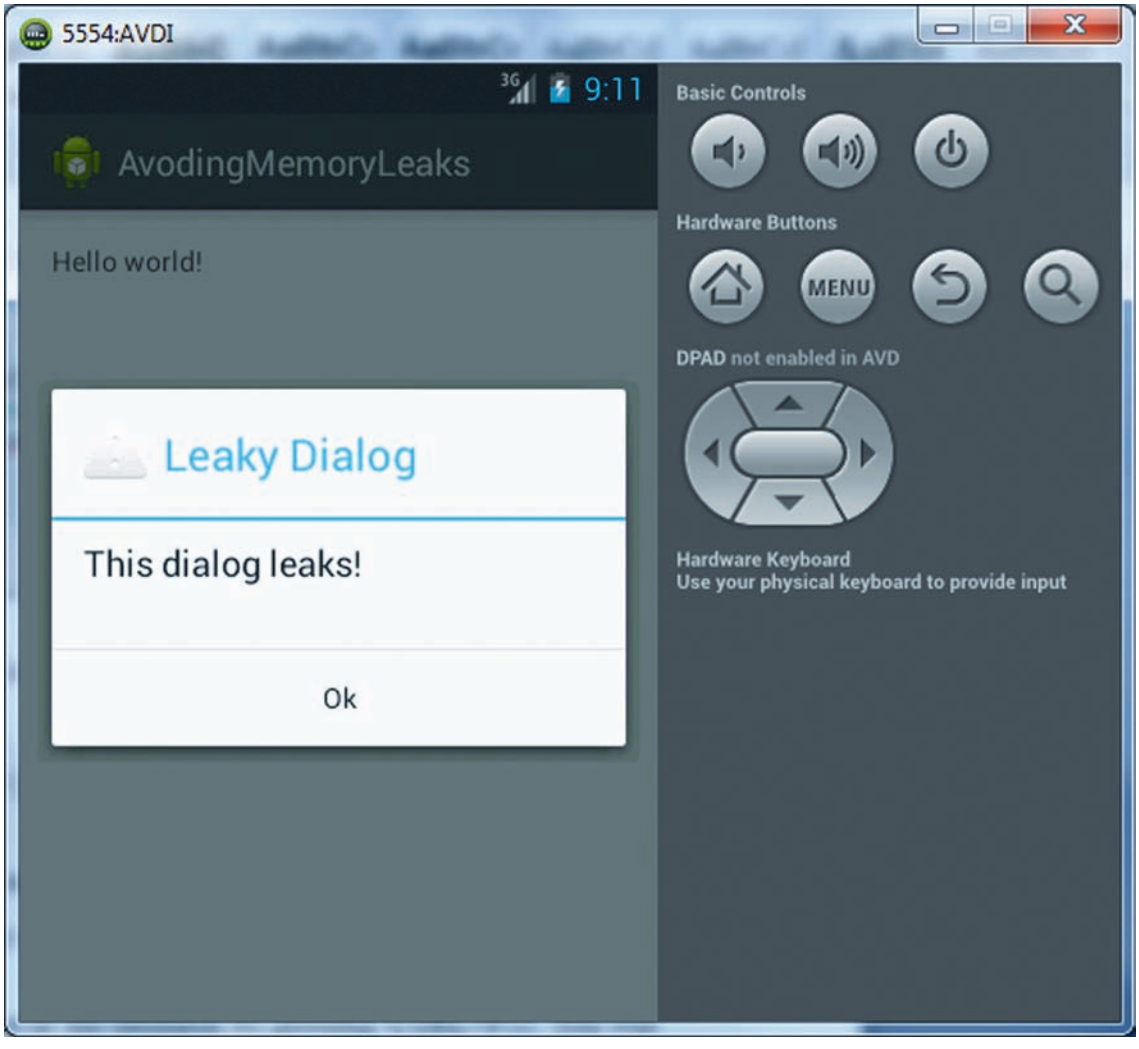


Figure 6.10: Leak Dialog

While this application is running, take a look at LogCat in Eclipse by clicking **Window → Show View → LogCat** from the menu. It will display messages in green color.

While the dialog is displayed, rotate the device or the emulator by pressing CTRL+F11. You will see red error text in LogCat as shown in figure 6.11.

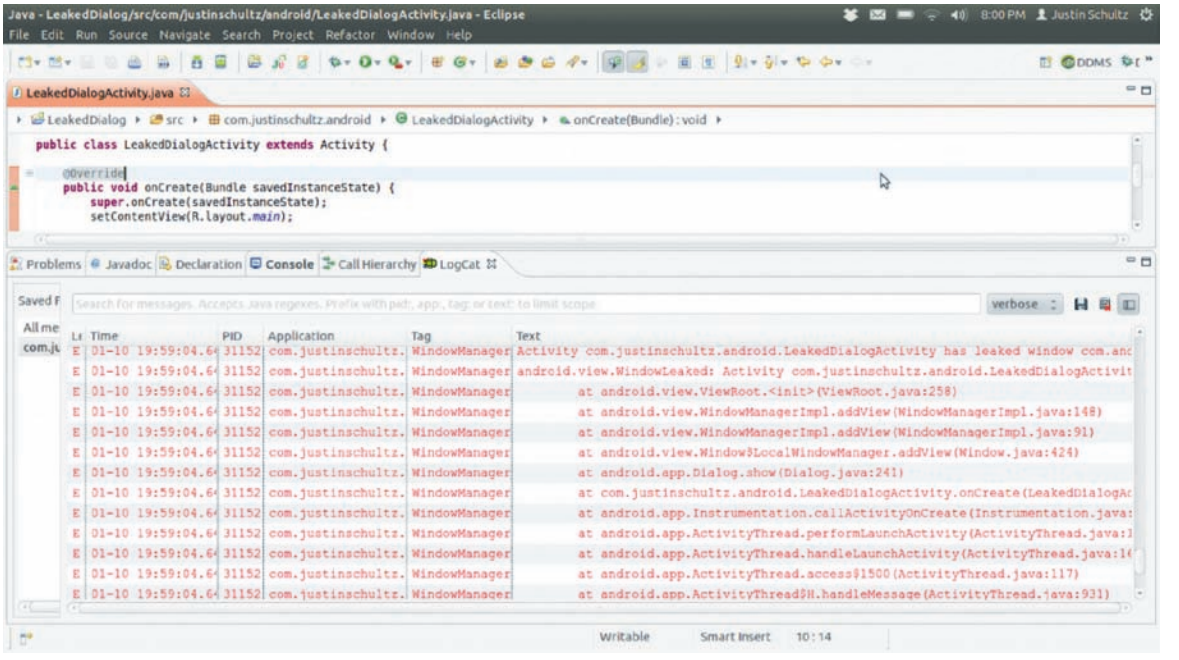


Figure 6.11: LogCat Leaked Window Error

Actually what happened?

When the device rotates, the onPause and onStop `Activity` events are invoked. Then, a new `Activity` is created having a landscape orientation, and its onCreate( ) method is invoked.

Since, the dialog is displayed and has a reference in the `Activity` class, it cannot be garbage collected, and is subsequently leaked. In other words, an object is created that cannot be garbage collected.

To fix this issue, let us make the dialog a member variable of `Activity` class and override the onPause( ) method to dismiss it as shown in code snippet 8.

Code Snippet 8:

```
AlertDialog _alert;

    @Override
    public void onPause(){
        super.onPause();
    if(_alert != null)
        _alert.dismiss();
}
```

This will dismiss the dialog each time the `Activity` is destroyed and recreated and the error message will disappear from LogCat.

The **`LeakedActivity`** class will contain the code as shown in code snippet 9.

Code Snippet 9:

```
package com.example.avodingmemoryleaks;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.app.AlertDialog;
import android.content.DialogInterface;

public class LeakedActivity extends Activity {

    AlertDialog _alert;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_leaked);

        AlertDialog.Builder builder = new AlertDialog.Builder(this);
        builder.setIcon(android.R.drawable.ic_dialog_alert);
        builder.setMessage("This dialog leaks!").setTitle("Leaky
        Dialog")
                .setCancelable(false)
                .setPositiveButton("Ok", new DialogInterface.
                OnClickListener() {
                    public void onClick(DialogInterface dialog,
                    int which) {
                    }
                });

        _alert = builder.create();
        _alert.show();
    }
```

```
        @Override
    public void onPause(){
        super.onPause();

        if(_alert != null)
            _alert.dismiss();
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu){
        // Inflate the menu; this adds items to the action bar if it is
        present.
        getMenuInflater().inflate(R.menu.leaked, menu);
        return true;
    }

}
```

6.8 Testing for Accessibility in Android

Testing the accessibility of an application for users with varying abilities is very important. Though an application is built following design and development guidelines, accessibility testing can showcase user interactions problems that may not be visible otherwise.

Accessibility testing, in general, should accomplish the following criteria in an application:

- Access to users with visual, physical, or age-related limitations.
- Use of only directional controls for navigation across all task workflows.
- Allow presentation of clear and specific feedback.
- Ensure that the following tests listed in table 6.3 are performed for accessibility testing of Android application.

Concepts

Tests	Action
Directional controls	Accomplish primary tasks using directional controls only.
Talkback audio prompts	Receive clear and accurate audio descriptions when TalkBack is enabled. Talkback, is a pre-installed screen reader which uses spoken feedback to describe the results of an action.
Explore by Touch prompts	Have appropriate audio description when Explore by Touch prompts is enabled. Explore by Touch is a feature that accompanies TalkBack that reads what is touched on the device screen via spoken feedback. This feature supports users with low vision.
Size of Touch controls	Minimum of 48dp (approx. 9mm) in length and width.
Accessibility settings	Change device's display and sound options.
Functioning of Gestures	Whether app-specific gesture function are used correctly or an alternative interface is required. App-specific functions include zooming images, scrolling lists, swiping between pages, navigating carousel controls etc.
No audio-only feedback	Secondary feedback mechanism, for example: sound alert accompanied by a visual alert to support users who are hard of hearing.
Repetitive audio prompts	Closely related controls not to repeat the same audio prompt.
Audio prompts overloading or underloading	Closely related controls provide appropriate level of audio information on a screen element for the user to understand and act.
Special cases and consideration	Check special cases and considerations that apply to the application and take proper action for them. Special cases and consideration include test field hints, custom control with high visual context, custom controls and click handling, controls that change function, prompts for related controls, video playback and captioning, supplemental accessibility audio feedback, custom controls with complex visual interactions, sets of small controls, decorative images and graphics, and so on.
Prompts for controls that change function	Controls that have multiple functions provide audio prompts appropriate to their current function.
Video playback and captioning	Indicate availability of captioning for a video play back controls and specify a clear way to enable them.

Table 6.3: Accessibility Testing Checklist

6.9 Check Your Progress

1. Android Testing Framework is based on?

(A)	Selinium	(C)	Sahi
(B)	JUint4	(D)	JUnit3

2. Which element in the AndroidManifest.xml file is used to specify the application to be tested?

(A)	activity	(C)	application
(B)	intent-filer	(D)	instrumentation

3. Which of the following option is used to test the Android TestCase class for correctness of <application> element in AndroidManifest.xml file?

(A)	InstrumentationTestCase	(C)	automatically assign InstrumentationTestRunner
(B)	build files and manifests files and directory structure	(D)	test each part of an application

4. Select a functional testing TestCase class available in Android Testing Framework?

(A)	InstrumentationTestRunner	(C)	ActivityUnitTestCase
(B)	ActivityTestCase	(D)	ActivityInstrumentationTestCase2

5. Which of the following class can be used to assert views of an Android Application?

(A)	android.text.TextUtils	(C)	android.test.ViewAsserts
(B)	android.test.MoreAsserts	(D)	ViewAsserts

6.9.1 Answers

1.	D
2.	D
3.	A
4.	D
5.	D

Summary

- An application uses `ActivityUnitTestCase<Activity>` base class for an isolated test of a single Activity. An application uses the `ActivityInstrumentationTestCase` to test the function of a single activity.
- The `ProviderTestCase2` tests content provider in an isolated environment.
- `ServiceTestCase` class is used for controlling the application and Services that is tested.
- The `RenamingDelegatingContext` is a mock class that mocks files and database operations.
- Android has a build-in automated UI testing tool that consists of the uiautomatorviewer GUI tool and uiautomator Java library.
- uiautomatorviewer GUI tool scans and analyzes the GUI components of the application and the uiautomator Java library provides the necessary APIs for automated, customized UI tests, and an execution engine to automate and run them.
- To detect memory consumption, use MAT view for the saved HPROF file.

Try it Yourself

Develop an application that will test for the NumberFormatException in Android.

“The future depends on what we do in the present”