



Indira Gandhi National Open University
School of Computer and Information
Sciences (SOCIS)

BCS-093

Introduction to Android



Block**2****ANDROID MOBILE APP DEVELOPMENT**

Unit 9	7
Testing and Debugging	
Unit 10	19
Integrating Multimedia	
Unit 11	33
Saving Data on Android Devices	
Unit 12	49
Locating and Sensing	
Unit 13	67
Connectivity and the Cloud	
Unit 14	75
Publish to Android Market	
Unit 15	81
Android App Performance	
Unit 16	97
Security Issues	

Course Coordinator

Prof.P.V.Suresh
SOCIS, IGNOU, New Delhi-110068

Copyright : CC-BY-SA

*The content is adopted to IGNOU CRC format. Wherever needed, content for
Check Your Progress is made (2021)*

Prof.P.V.Suresh
SOCIS, IGNOU, New Delhi-110068



CRC Preparation (2021)

Mr.Vishal Singh, Skilled Worker, SOCIS, IGNOU, New Delhi - 110068

Copyright

This course has been developed as part of the collaborative advanced ICT course development project of the Commonwealth of Learning (COL). COL is an intergovernmental organisation created by Commonwealth Heads of Government to promote the development and sharing of open learning and distance education knowledge, resources and technologies. Herewith, its acknowledged that the image portion in Cover Page is also adopted.

The Open University of Sri Lanka (OUSL) is the premier Open and Distance learning institution in the country where students can pursue their studies through Open and Distance Learning (ODL) methodologies. Degrees awarded by OUSL are treated as equivalent to the degrees awarded by other national universities in Sri Lanka by the University Grants Commission of Sri Lanka.



© 2017 by the Commonwealth of Learning and The Open University of Sri Lanka. Except where otherwise noted, *Introduction to Android* is made available under Creative Commons Attribution- ShareAlike 4.0 International (CC BY-SA 4.0) License: <https://creativecommons.org/licenses/by-sa/4.0/legalcode>.

For the avoidance of doubt, by applying this license the Commonwealth of Learning does not waive any privileges or immunities from claims that it may be entitled to assert, nor does the Commonwealth of Learning submit itself to the jurisdiction, courts, legal processes or laws of any jurisdiction. The ideas and opinions expressed in this publication are those of the author/s; they are not necessarily those of Commonwealth of Learning and do not commit the organisation.



The Open University of Sri Lanka
P. O. Box 10250,
Nawala,
Nugegoda,
Sri Lanka
Phone: +94 112881481
Fax: +94 112821285
Email: hdelect@ou.ac.lk
Website: www.ou.ac.lk

Commonwealth of Learning
4710 Kingsway, Suite 2500, Burnaby
V5H 4M2,
British Columbia,
Canada
Phone: +1 604 775 8200
Fax: +1 604 775 8210
Email: info@col.org
Website: www.col.org

Acknowledgements

Department of Electrical and Computer Engineering (ECE), The Open University of Sri Lanka (OUSL) wishes to thank those below for their contribution to this course material and accompanying Videos and Screencasts:

Chairperson of the Course team:

H.U.W. Ratnayake (Senior Lecturer, Dept. of ECE, OUSL)

Authors:

W.A.S.N. Perera (Lecturer, Dept. of ECE, OUSL)	Units 1 and 2
J. Nananyakkara (Lecturer, Dept. of ECE, OUSL)	Units 3 and 4
B.K. Werapitiya (Lecturer, Dept. of ECE, OUSL)	Units 5 and 6
S. Rajasingham (Lecturer, Dept. of ECE, OUSL)	Units 7 and 8
U.S. Premaratne (Lecturer, Dept. of ECE, OUSL)	Units 9 and 10
W.U. Erandaka (Lecturer, Dept. of Textile & Apparel Tech, OUSL)	Units 11 and 12
W.W.A.I.D. Wickramasinghe (Demonstrator, Dept. of ECE, OUSL)	Units 13 and 14
H.U.W. Ratnayake (Senior Lecturer, Dept. of ECE, OUSL)	Units 15 and 16

Content Editors:

C.W.S. Goonatilleke (Senior Software Engineer, WSO2 Inc)

G.S.N. Meedin (Lecturer, Dept. of ECE, OUSL)

Language Editor:

G.S.N. Meedin (Lecturer, Dept. of ECE, OUSL)

Reviewer:

D.G. U. Kulasekara (Senior Lecturer/Centre for Educational Technology & Media, OUSL)

Video Presenters:

U.S. Premartane (Lecturer, Dept. of ECE, OUSL)

S. Rajasingham (Lecturer, Dept. of ECE, OUSL)

Screen casters:

C.W.S. Goonatilleke (Senior Software Engineer, WSO2 Inc)

COURSE INTRODUCTION

This Course is developed by Open University of Sri Lanka and is adapted by Indira Gandhi National Open University. The copyright of the Course Material remains CC-BY-SA.

This Course comprises of 2 blocks. The blocks may be increased in future as per need.

This Course introduces Android and covers Android Architecture, Android Development Environment, Android APP fundamentals, Development in Android environment, Developing applications so that they run across Devices, Designing user interfaces, Testing and Debugging the applications, Including Multimedia, Saving Data on Android Devices, Making APPS location aware and using sensors, Publishing APP to Android Market, , Measuring the performance of APPs, and ensuring security of APS.



BLOCK INTRODUCTION

This Block primarily introduces features of Android. There are 8 units in this Block:

Unit 9 discusses about testing and debugging of an Android Application

Unit 10 discusses about integration of Multimedia

Unit 11 discusses Android storage options.

Unit 12 deals with location and sensors

Unit 13 focuses on connecting application wirelessly

Unit 14 elaborates on publication of Mobile Apps to Android Market

Unit 15 discusses performance related issues of Mobile APPs

Unit 16 deals with Security related issues



UNIT 9 TESTING AND DEBUGGING

- 9.0 Introduction
- 9.1 Objectives
- 9.2 What is Testing?
- 9.3 How to test Android application?
- 9.4 Unit Testing
 - 9.4.1 Check Your Progress
- 9.5 How to set up your Testing Environment?
 - 9.5.1 Video – V9: Android Unit Testing
 - 9.5.2 Check Your Progress
- 9.6 What is Debugging?
 - 9.6.1 Check Your Progress
- 9.7 What is Logcat?
- 9.8 Summary
- 9.9 Further readings

9.0 INTRODUCTION

This unit emphasizes the importance of the application testing and debugging. It provides you an opportunity to identify errors and faults in a developed application. It also introduces how to use testing tools and techniques to test Android Application and how to apply measures to rectify identified errors and faults.

9.1 OBJECTIVES

After studying this unit, you should be able to :



Outcomes

- differentiate testing and debugging an application
- set up the testing environment to develop Android applications
- write unit tests to test your Android programme
- perform debugging referring to log messages in Logcat



Terminology

- | | |
|------------------|--|
| error: | mistake in the program |
| fault: | usually a hardware problem happening at run time |
| emulator: | hardware or software that enables one computer system to behave like another |

9.2 WHAT IS TESTING?

Testing is one of the phases in software development life cycle that requires substantial amount of time before releasing software to users. To find the software bugs we use the process of executing a program or application. We have to test the software with test data to verify that a given set of input to a given function produces

the expected result. There are two testing approaches; static and dynamic testing. An introduction to static and dynamic testing is given in the next section

Static and Dynamic testing

Static testing is done basically to test the requirement specifications, test plan, user manual etc. They are not executed, but tested with the set of some tools and processes. Reviews, walkthroughs and inspections are some example processes. These processes are not discussed in this material.

Dynamic Testing is when execution is done on the software code as a technique to detect defects and to determine quality attributes of the code.

With dynamic testing methods software is executed using a set of inputs and its output and then compared with the expected results. There are various levels of dynamic testing techniques. Some of them are unit testing, integration testing, system testing and acceptance testing. Here we will be only focusing on how different dynamic testing techniques can be used in an Android application.

Now you know that techniques can be used in testing software. Let's learn how to test an Android application using above techniques.

9.3 HOW TO TEST ANDROID APPLICATION?

Testing an application on multiple physical devices at one place is not practical. Testing an application to run on different devices is referred to as device compatibility which is discussed in detail in a different unit. In order to avoid this practical barrier, Android SDK provides an emulator to test your application against all versions of Android on different devices. An emulator provides a virtual environment to test your application. It eliminates the requirement of a real physical device. This emulator uses an Android Virtual Device (AVD) to run an application. In order to do that it is required to create an AVD. Creating an AVD was discussed under an earlier section.

In addition, Amazon and various other companies maintain device farms to test applications with automation as well as manual testing.

An Android application should be tested for its functionality, user interfaces, performance etc. based on the generated test cases. It is application developers' responsibility to perform the unit tests and a separate testing team will be responsible of performing certain other types of testing such as functional testing, integration testing, acceptance testing etc.

9.4 UNIT TESTING

Unit tests are the fundamental tests in your app testing strategy. By creating and running unit tests against your code, you can easily verify that the logic of individual units is correct. Running unit tests after every build helps you to quickly catch and fix software regressions introduced by code changes to your app.

A unit test generally exercises the functionality of the smallest possible unit of code (which could be a method, class, or component) in a repeatable way. You should build unit tests when you need to verify the logic of specific code in your app. For example, if you are unit testing a class, your test might check that the class is in the right state.

Typically, the unit of code is tested in. After completing this section, you will be able to perform unit testing for your application.

Android unit tests are based on JUnit and to test Android apps, you typically create these types of automated unit tests:

- **Local tests:** Unit tests that run on your local machine only. These tests are compiled to run locally on the Java Virtual Machine (JVM) to minimize execution time. Use this approach to run unit tests that have no dependencies on the Android framework or have dependencies that can be filled by using mock objects.
- **Instrumented tests:** Unit tests that run on an Android device or emulator. These tests have access to instrumentation information, such as theContext for the app under test. Use this approach to run unit tests that have Android dependencies which cannot be easily filled by using mock objects.

You should write your unit or integration test class as a JUnit 4 test class in JUnit framework. This framework offers a convenient way to perform common setup, teardown, and assertion operations in your test.

9.4.1 Check Your Progress



Differentiate the use of local test from instrumented test when performing a unit test of an Android application.

9.5 HOW TO SET UP YOUR TESTING ENVIRONMENT?

In your Android Studio project, you must store the source files for instrumented tests at *module-name/src/androidTests/java/*. This directory already exists when you create a new project.

Before you begin, you should download the Android Testing Support Library Setup, which provides APIs that allow you to quickly build and run instrumented test code for your apps. The Testing Support Library includes a JUnit 4 test runner (AndroidJUnitRunner) and APIs for functional UI tests (Espresso and UI Automator).

You also need to configure the Android testing dependencies for your project to use the test runner and the rules APIs provided by the Testing Support Library. To simplify your test development, you should also include the Hamcrest library, which lets you create more flexible assertions using the Hamcrest matcher APIs.

In your app's top-level build.gradle file, you need to specify these libraries as dependencies:

```
dependencies {  
    androidTestCompile 'com.android.support:support-  
annotations:24.0.0'  
    androidTestCompile 'com.android.support.test:runner:0.5'  
    androidTestCompile 'com.android.support.test:rules:0.5'  
    // Optional -- Hamcrest library  
    androidTestCompile 'org.hamcrest:hamcrest-library:1.3'
```

```
// Optional -- UI testing with Espresso
androidTestCompile
'com.android.support.test.espresso:espresso-core:2.2.2'
// Optional -- UI testing with UI Automator
androidTestCompile
'com.android.support.test.uiautomator:uiautomator-v18:2.1.2'
}
```

To use JUnit 4 test classes, make sure to specify [AndroidJUnitRunner](#) as the default test instrumentation runner in your project by including the following setting in your app's module-level `build.gradle` file:

```
android {
    defaultConfig {
        testInstrumentationRunner
        "android.support.test.runner.AndroidJUnitRunner"
    }
}
```

A basic JUnit 4 test class is a Java class that contains one or more test methods. A test method begins with the `@Test` annotation and contains the code to exercise and verify a single functionality (that is, a logical unit) in the component that you want to test.

The code snippet below shows an example JUnit 4 integration test that uses the Espresso APIs to perform a click action on a UI element, and check whether an expected string is displayed.

```
@RunWith(AndroidJUnit4.class)
@LargeTest
public class MainActivityInstrumentationTest {

    @Rule
    public ActivityTestRule mActivityRule
    = new ActivityTestRule<>(MainActivity.class);

    @Test
    public void sayHello() {
        onView(withText("Sayhello!")).perform(click());

        onView(withId(R.id.textView)).check(matches(withText("Hello, World!")));
    }
}
```

In your JUnit 4 test class, you can call out sections in your test code for special processing by using the following annotations:

@Before: Use this annotation to specify a block of code that contains test setup operations. The test class invokes this code block before each test. You can have multiple `@Before` methods but the order in which the test class calls these methods is not guaranteed.

@After: This annotation specifies a block of code that contains test tear-down operations. The test class calls this code block after every test method. You can define

multiple `@After` operations in your test code. Use this annotation to release any resources from memory.

`@Test`: Use this annotation to mark a test method. A single test class can contain multiple test methods, each prefixed with this annotation.

`@Rule`: Rules allow you to flexibly add or redefine the behavior of each test method in a reusable way. In Android testing, use this annotation together with one of the test rule classes that the Android Testing Support Library provides, such as `ActivityTestRule` or `ServiceTestRule`.

`@BeforeClass`: Use this annotation to specify static methods for each test class to invoke only once. This testing step is useful for expensive operations such as connecting to a database.

`@AfterClass`: Use this annotation to specify static methods for the test class to invoke only after all tests in the class have run. This testing step is useful for releasing any resources allocated in the `@BeforeClass` block.

`@Test(timeout=)`: Some annotations support the ability to pass in elements for which you can set values. For example, you can specify a timeout period for the test. If the test starts but does not complete within the given timeout period, it automatically fails. You must specify the timeout period in milliseconds, for example:

`@Test(timeout=5000)`.

Instrumented unit tests

Unit tests that run on an Android device or emulator can take advantage of the Android framework APIs and supporting APIs, such as the Android Testing Support Library. You should create instrumented unit tests if your tests need access to instrumentation information (such as the target app's `Context`) or if they require the real implementation of an Android framework component (such as a `Parcelable` or `SharedPreferences` object). These tests have access to Instrumentation information, such as the `Context` of the app you are testing. Use these tests when your tests have Android dependencies that mock objects cannot satisfy.

Because instrumented tests are built into a stand-alone APK, they must have an `AndroidManifest.xml` file. However, Gradle automatically generates this file during the build so it is not visible in your project source set. You can add your own manifest file if necessary, such as to specify a different value for `minSdkVersion` or register run listeners just for your tests. When building your app, Gradle merges multiple manifest files into one manifest.

Create an Instrumented Unit Test Class

Your instrumented unit test class should be written as a JUnit 4 test class. To learn more about creating JUnit 4 test classes and using JUnit 4 assertions and annotations, see [Create a Local Unit Test Class](#).

To create an instrumented JUnit 4 test class, add the `@RunWith(AndroidJUnit4.class)` annotation at the beginning of your test class definition. You also need to specify the `AndroidJUnitRunner` class provided in the

Android Testing Support Library as your default test runner. This step is described in more detail in [Getting Started with Testing](#).

The following example shows how you might write an instrumented unit test to test that the `Parcelable` interface is implemented correctly for the `LogHistory` class:

```
import android.os.Parcel;
import android.support.test.runner.AndroidJUnit4;
import android.util.Pair;
import org.junit.Test;
import org.junit.runner.RunWith;
import java.util.List;
import static org.hamcrest.Matchers.is;
import static org.junit.Assert.assertThat;

@RunWith(AndroidJUnit4.class)
@SmallTest
public class LogHistoryAndroidUnitTest {

    public static final String TEST_STRING = "This is a string";
    public static final long TEST_LONG = 12345678L;
    private LogHistory mLogHistory;

    @Before
    public void createLogHistory() {
        mLogHistory = new LogHistory();
    }

    @Test
    public void logHistory_ParcelableWriteRead() {
        // Set up the Parcelable object to send and receive.
        mLogHistory.addEntry(TEST_STRING, TEST_LONG);

        // Write the data.
        Parcel parcel = Parcel.obtain();
        mLogHistory.writeToParcel(parcel,
mLogHistory.describeContents());

        // After you're done with writing, you need to reset
the parcel for reading.
        parcel.setDataPosition(0);

        // Read the data.
        LogHistory createdFromParcel
=LogHistory.CREATOR.createFromParcel(parcel);
        List<Pair<String, Long>> createdFromParcelData =
createdFromParcel.getData();

        // Verify that the received data is correct.
        assertThat(createdFromParcelData.size(), is(1));
        assertThat(createdFromParcelData.get(0).first, is(TEST_S
TRING));
        assertThat(createdFromParcelData.get(0).second, is(TEST_
LONG));
    }
}
```

Create a test suite

To organize the execution of your instrumented unit tests, you can group a collection of test classes in a *test suite* class and run these tests together. Test suites can be nested. That is your test suite can group other test suites and run all their component test classes together.

A test suite is contained in a test package, similar to the main application package. By convention, the test suite package name usually ends with the suite suffix (e.g. com.example.android.testing.mysample.suite).

To create a test suite for your unit tests, import the JUnit `RunWith` and `Suite` classes. In your test suite, add the `@RunWith(Suite.class)` and the `@Suite.SuiteClasses()` annotations. In the `@Suite.SuiteClasses()` annotation, list the individual test classes or test suites as arguments.





The following example shows how you might implement a test suite called `UnitTestSuite` that groups and runs the `CalculatorInstrumentationTest` and `CalculatorAddParameterizedTest` test classes together.

```
import
com.example.android.testing.mysample.CalculatorAddParameterized
Test;
import
com.example.android.testing.mysample.CalculatorInstrumentationT
est;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

// Runs all unit tests.
@RunWith(Suite.class)
@Suite.SuiteClasses({CalculatorInstrumentationTest.class,
    CalculatorAddParameterizedTest.class})
public class UnitTestSuite {}
```

Run Instrumented Unit Tests

To run your instrumented tests, follow these steps:

1. Be sure your project is synchronized with Gradle by clicking **Sync Project**  in the toolbar.
2. Run your test in one of the following ways:
 - o To run a single test, open the **Project** window, and then right-click a test and click **Run** .
 - o To test all methods in a class, right-click a class or method in the test file and click **Run** .
 - o To run all tests in a directory, right-click on the directory and select **Run tests** .


The Android Plugin for Gradle compiles the instrumented test code located in the default directory (`src/androidTest/java/`), builds a test APK and production APK, installs both APKs on the connected device or emulator, and runs the tests. Android Studio then displays the results of the instrumented test execution in the *Run* window.

Note: While running or debugging instrumented tests, Android Studio does not inject the additional methods required for Instant Run and turns the feature off.



By default, Android Studio sets up new projects to deploy to the Emulator or a physical device with just a few clicks. With Instant Run, you can push changes to methods and existing app resources to a running app without building a new APK, so code changes are visible almost instantly.

Instant Run

Introduced in Android Studio 2.0, Instant Run is a behavior for the Run  and

Debug  commands that significantly reduces the time between updates to your app. Although your first build may take longer to complete, Instant Run pushes subsequent updates to your app without building a new APK, so changes are visible much more quickly.

Instant Run is supported only when you deploy the debug build variant, use Android Plugin for Gradle version 2.0.0 or higher, and set `minSdkVersion` to 15 or higher in your app's module-level `build.gradle` file. For the best performance, set `minSdkVersion` to 21 or higher.

After deploying an app, a small, yellow thunderbolt icon appears within the Run  button (or Debug  button), indicating that Instant Run is ready to push updates the next time you click the button. Instead of building a new APK, it pushes just those new changes and, in some cases, the app doesn't even need to restart but immediately shows the effect of those code changes.

Instant Run pushes updated code and resources to your connected device or emulator by performing a hot swap, warm swap, or cold swap. It automatically determines the type of swap to perform based on the type of change you made.

9.5.1 Video – V9: Android Unit Testing



In this video you will be shown how to setup testing environment and how to write a unit test. You may watch this video and do activity 9.2.

URL: <https://tinyurl.com/yaatacnv>



9.5.2 Check Your Progress



Create an Android application “MyApp” with a class “ConversionUtil” to perform the given two functionalities.

- To convert centimeters into inches [write a method ConvertCmtoInch()]
- To convert inches into centimeters [write a method ConvertInchtoCm()]


Then write local unit tests to check whether the written functionalities provide the expected output. Use the values given as inputs and expected output to test the method.

Functionality to test	Input	Output
Convert centimeters into inches	10 centimeters	3.93701 inches
Convert inches into centimeters	10 inches	25.4 centimeters

9.6 WHAT IS DEBUGGING?

It is the procedure of finding defects in a source code and removing them. Android Studio includes a debugger that allows you to debug apps running on the Android Emulator or a connected Android device. With the Android Studio debugger, you can:

- Select a device to debug your app on.
- Set breakpoints in your code.
- Examine variables and evaluate expressions at runtime.
- Capture screenshots and videos of your app.

To start debugging, click **Debug**  in the toolbar. Android Studio builds an APK, signs it with a debug key, installs it on your selected device, then runs it and opens the **Debug** window.

If no devices appear in the **Select Deployment Target** window after you click **Debug**, then you need to either connect a device or click **Create New Emulator** to setup the Android Emulator.

9.6.1 Check Your Progress



How to enable USB debugging in your device?

9.7 WHAT IS LOGCAT?

Logcat is a command-line tool that dumps a log of system messages, including stack traces when the device throws an error and messages that you have written from your app with the Log class.

This page is about the command-line logcat tool, but you can also view log messages from the Logcat window in Android Studio. For information about viewing and filtering logs from Android Studio

You can run logcat as an adb command or directly in a shell prompt of your emulator or connected device. To view log output using adb, navigate to your SDK platform-tools/ directory and execute:

```
$ adb logcat
```

You can create a shell connection to a device and execute:

```
$ adb shell  
# logcat
```

How to write Log Messages?

The Log class allows you to create log messages that appear in the logcat window. Generally, you should use the following log methods, listed in order from the highest to lowest priority (or, least to most verbose):

- Log.e → (error)
- Log.w → (warning)
- Log.i → (information)
- Log.d → (debug)
- Log.v → (verbose)

You should never compile verbose logs into your app, except during development. Debug logs are compiled in but stripped at runtime, while error, warning and info logs are always kept.

For each log method, the first parameter should be a unique tag and the second parameter is the message. The tag of a system log message is a short string indicating the system component from which the message originates (for example, ActivityManager). Your tag can be any string that you find helpful, such as the name of the current class.

A good convention is to declare a TAG constant in your class to use in the first parameter. For example, you might create an information log message as follows:

```
private static final String TAG = "MyActivity";  
...  
Log.i(TAG, "MyClass.getView() — get item number " + position);
```

Note: Tag names greater than 23 characters are truncated in the logcat output.

9.8 SUMMARY



Android Studio is designed to make testing simple. This unit explained how to set up a JUnit test that runs on the local JVM or an instrumented test that runs on a device.

9.9 FURTHER READINGS

<https://developer.android.com/studio/test?authuser=1>

<https://developer.android.com/training/testing>

<https://www.softwaretestinghelp.com/android-app-testing/>



ignou
THE PEOPLE'S
UNIVERSITY



UNIT 10 INTEGRATING MULTIMEDIA

- 10.0 Introduction
 - 10.1 Objectives
 - 10.2 Introduction to Multimedia
 - 10.3 Audio and Video Integration into Android Application Development
 - 10.3.1 Video-V10: Multimedia for Android Interactive Application Development
 - 10.3.2 Check Your Progress
 - 10.3.3 Check Your Progress
 - 10.4 Camera functions in Android Application Development
 - 10.4.1 Check Your Progress
 - 10.5 Supported Media Formats
 - 10.6 Summary
 - 10.7 Further readings
-

10.0 INTRODUCTION

This unit offers you with knowledge on how to integrate multimedia to Android applications. Further the unit discusses how to utilize multimedia to enhance the performance by selecting appropriate media formats for audio, video and images. You need to watch the provided video to get an insight of how different multimedia are being used in selected applications.

10.1 OBJECTIVES

After studying this unit, you should be able to:



Outcomes

- write a code to play audio and video depending on the functional requirements.
- implement camera functions to capture photos.
- select appropriate media codecs to maximize the compatibility and application performance.



Terminology

- | | |
|-------------------------|---|
| multimedia: | use of graphics, animations, video clippings, audio etc taken together |
| state diagram: | diagram depicting various states of an app |
| codecs: | a device or program that compresses data to enable faster transmission and decompresses received data |
| compatibility: | state of being two or more things are able to exist or work together in combination |
| streaming media: | Streaming media is multimedia that is constantly received by and presented to an end-user while being delivered by a provider |

10.2 INTRODUCTION TO MULTIMEDIA

Multimedia is an effective tool of communication. Let us spend some time to think in which forms we access the information. The simplest and the most common of these is the printed text. Examples include newspapers, web pages etc. In order to deliver information in a more attractive way, text materials are supported with graphics, still pictures, animations, video clippings, audio commentaries and so on. Use of different attractive formats to convey information in a meaningful manner is termed as multimedia. Television is a very good example of a multimedia broadcasting system. Information is distributed to the community using audio and video signals.

Android applications are developed to distribute different information to the community. Unlike in a broadcasting media, such as the television, in Android mobile applications the users directly interact with it. Multimedia is necessary to improve the user interactions in a mobile application. Integration of different media formats can significantly improve the user experience when interacting with the mobile application.

10.3 AUDIO AND VIDEO INTEGRATION INTO ANDROID APPLICATION DEVELOPMENT

MediaPlayer and AudioManager are two classes available to play sound and video in the Android framework. MediaPlayer class is the primary API for playing sound and audio while AudioManager class is used to manage audio sources and audio output on a device. MediaCodec and MediaExtractor classes are provided for building custom media players. The open source project, ExoPlayer, is a solution between these two options, providing a pre-built player that you can extend.

10.3.1 Video-V10: Multimedia for Android Interactive Application Development



By watching this video you will get an understanding how to incorporate multimedia to an Android application.

URL: <https://tinyurl.com/y7bj7mys>



Media Player

An object of this class can fetch, decode, and play both audio and video with minimal setup. It supports several different media sources including local resources, Internet URIs, external URIs. Here is an example of how to play audio that's available as a local raw resource saved in your application's res/raw/ directory:

```
MediaPlayer mediaPlayer = MediaPlayer.create(context,
R.raw.sound_file_1);
mediaPlayer.start();
```

Here is an example on how you might play from a URI available locally in the system (that you obtained through a Content Resolver, for instance):

```
Uri myUri = ....; // initialize Uri here
MediaPlayer mediaPlayer = new MediaPlayer();
mediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC
);
mediaPlayer.setDataSource(getApplicationContext(), myUri);
mediaPlayer.prepare();
mediaPlayer.start();
```

Playing from a remote URL via HTTP streaming looks like this:

```
String url = "http://....."; // your URL here
MediaPlayer mediaPlayer = new MediaPlayer();
mediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
mediaPlayer.setDataSource(url);
mediaPlayer.prepare(); // might take long! (for buffering, etc)
mediaPlayer.start();
```

Before using the MediaPlayer, it is necessary to make the appropriate declarations. Example is to request permissions to access Internet for streaming applications. MediaPlayer will not work as expected in certain scenarios. Why does this happen? The possible reasons can be explained by understanding the state based representation of the MediaPlayer class. MediaPlayer has specific “states” (Figure10.1) in which certain operations are only valid. If you perform an operation while in the wrong state, the system may throw an exception or cause other undesirable behaviors. Schematic of the state based representation is shown below.

The state based representation of the MediaPlayer is shown in Figure10.1. The state diagram clarifies which methods move the MediaPlayer from one state to another. For example, when you create a new MediaPlayer, it is in the *Idle* state. Then, you should initialize it by calling `setDataSource()`, bringing it to the *Initialized* state. Next, you have to prepare it using either the `prepare()` or `prepareAsync()` method. When the MediaPlayer is done preparing, it will then enter the *Prepared* state, which indicates that `start()` can be called to make it play the media. Then, you can move between the *Started*, *Paused* and *PlaybackCompleted* states by calling such methods as `start()`, `pause()` and `seekTo()`, amongst others. Once you call `stop()`, MediaPlayer cannot call `start()` again until you prepare it again.

MediaPlayer consumes valuable system resources. Therefore, you should always take extra precautions to make sure you are not hanging on to a MediaPlayer instance longer than necessary. When you are done with it, you should always call `release()` to make sure any system resources allocated to it are properly released.

```
mediaPlayer.release();  
mediaPlayer = null;
```

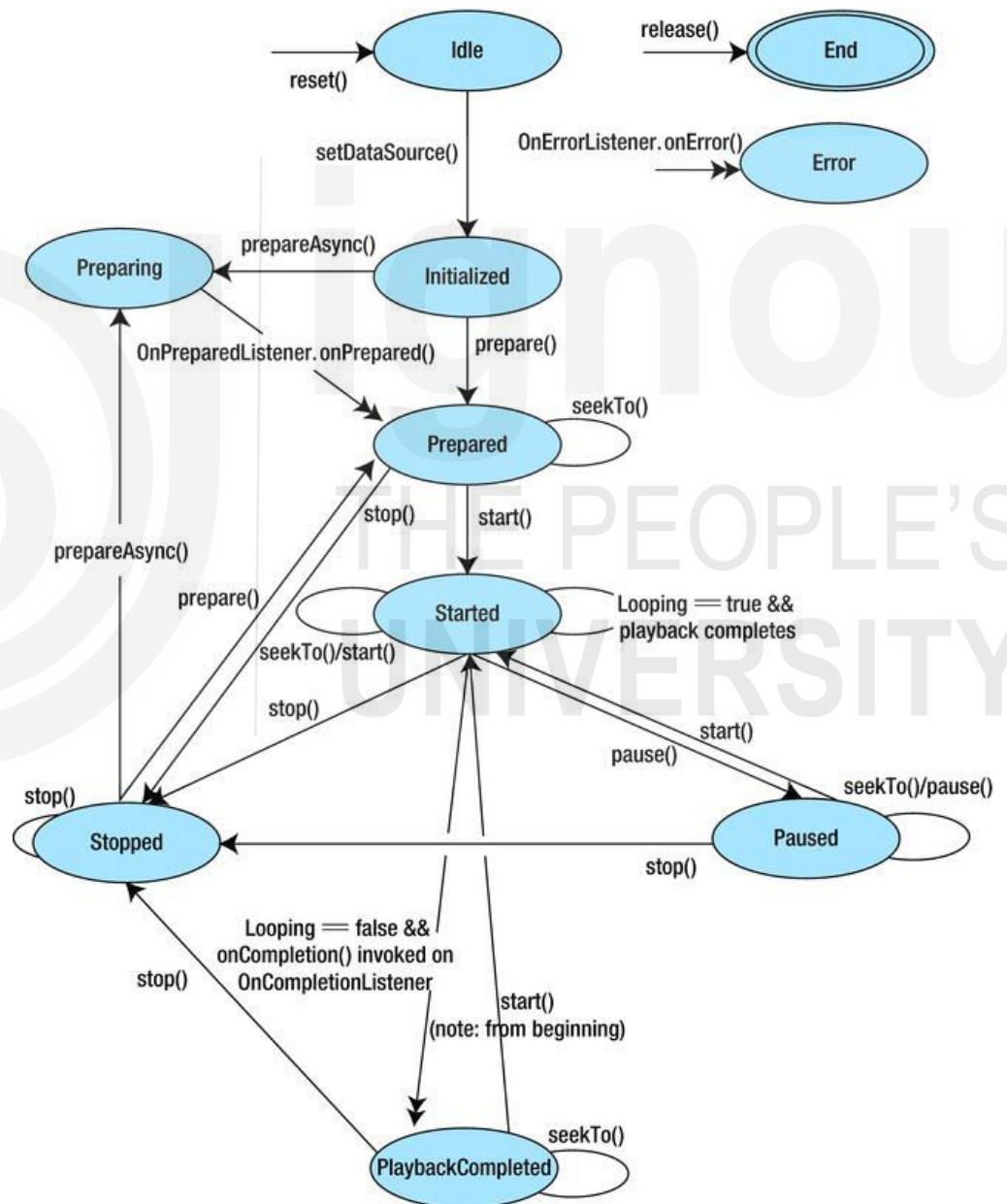


Figure10.1: State Diagram of MediaPlayer.
(Source: <https://developer.android.com/index.html>)

10.3.2 Check Your Progress



List five valid or invalid state transitions from the MediaPlayer by studying the state diagram shown in Figure10.1.

Volume and Playback Control

Depending on the user preference the loudness of the audio output should be manageable. Volume control should be available for the user by using the hardware or software volume controls of their device, bluetooth headset, or headphones. Apart from volume control, the user should also be able to have control on the playback videos. The control functions such as, the play, stop, pause, skip, and previous media playback keys should perform their respective actions on the audio stream used by the application you develop.

Audio streams - In order to control the audio output, the Android applications use different “streams”. An audio stream enables an application to independently and distinctly apply controls depending on the user preferences. The first step to creating a predictable audio experience is understanding which audio stream your app will use. For example, Android maintains a separate audio stream called `STREAM_MUSIC` for playing music, alarms, notifications, the incoming call ringer, system sounds, in-call volume, and DTMF tones. Most of the Android audio streams are restricted to system events.

Hardware Volume and Playback Controls

The hardware volume control button, generally adjust the ringer volume of the mobile phone. For example, you do not want the ringer volume to be adjusted by pressing hardware volume control key while playing a game, but the user only wants to increase the volume of the sounds played by the game. For this type of preferences, it is necessary to identify which audio stream to control. Android provides the `setVolumeControlStream()` method to direct volume key presses to the audio stream that is specified. Having identified the audio stream used in the application, this should be set as the volume stream target. This setting should be coded early in your application’s lifecycle as it is only needed to be called once, typically within the `onCreate()` method (of the Activity or Fragment that controls your media). This ensures that whenever your app is visible, the volume controls function as the user expects. The code snippet looks like this:

```
setVolumeControlStream(AudioManager.STREAM_MUSIC).
```

Once this is coded, when the user press the hardware volume keys on the device, the control affect the audio stream you specify whenever the target activity or fragment is visible.

On certain mobile devices, the hardware playback control buttons are available or even externally connected through wireless handsets. When the user presses one of these buttons, the Android application notifies it as a `ACTION_MEDIA_BUTTON`

action. In order to respond to these actions, as the receiving end, a BroadcastReceiver should be registered (or declared). The code snippet would look like this:

```
<receiver android:name=".RemoteControlReceiver">
    <intent-filter>
        <action
            android:name="android.intent.action.MEDIA_BUTTON" />
        </intent-filter>
    </receiver>
```

Once the receiver is defined, the appropriate response can only be generated if the receiver knows which playback button was pressed. To identify the button, Intent is used. Then, in order to provide the appropriate response action, KeyEvent class is used. An example is shown below.

```
public class RemoteControlReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if (Intent.ACTION_MEDIA_BUTTON.equals(intent.getAction())) {
            KeyEvent event =
                (KeyEvent)intent.getParcelableExtra(Intent.EXTRA_KEY_EVENT);
            if (KeyEvent.KEYCODE_MEDIA_PLAY == event.getKeyCode()) {
                // Handle key press.
            } } } }
```

Managing Audio Focus (**italic smaller font CHECK HEADINGS*)

To avoid every music app playing at the same time, Android uses audio focus to allow only apps that hold the audio focus to play audio. Before your app starts playing any audio, it should hold the audio focus for the stream it will be using. This is done with a call to requestAudioFocus(). AUDIOFOCUS_REQUEST_GRANTED is returned if the request is successful.

You must specify which stream you are using and whether you expect to require transient or permanent audio focus. Request transient focus when you expect to play audio for only a short time (for example when playing navigation instructions). Request permanent audio focus when you plan to play audio for the foreseeable future (for example, when playing music).

The following snippet requests permanent audio focus on the music audio stream. You should request the audio focus immediately before you begin playback, such as when the user presses play or the background music for the next game level begins.

```

    AudioManager am =
    mContext.getSystemService(Context.AUDIO_SERVICE);
    ...

    // Request audio focus for playback
    int result = am.requestAudioFocus(afChangeListener,
        // Use the music stream.
        AudioManager.STREAM_MUSIC,
        // Request permanent focus.
        AudioManager.AUDIOFOCUS_GAIN);

    if (result ==
    AudioManager.AUDIOFOCUS_REQUEST_GRANTED) {

    am.registerMediaButtonEventReceiver(RemoteControlReceiver);
        // Start playback.
    }

```

Once you have finished playback call `abandonAudioFocus()`. To notify the system that it is no longer require focus and unregisters the associated `OnAudioFocusChangeListener`. In the case of abandoning transient focus, this allows any interrupted app to continue playback.

```

// Abandon audio focus when playback complete
am.abandonAudioFocus(afChangeListener);

```

Ducking is the process of lowering your audio stream output volume to make transient audio from another app easier to hear without totally disrupting the audio from your own application. In the following code snippet lowers the volume on our media player object when we temporarily lose focus, then returns it to its previous level when we regain focus.

```

OnAudioFocusChangeListener afChangeListener = new
OnAudioFocusChangeListener() {
    public void onAudioFocusChange(int focusChange) {
        if (focusChange == AUDIOFOCUS_LOSS_TRANSIENT_CAN_DUCK)
        {
            // Lower the volume
        } else if (focusChange == AudioManager.AUDIOFOCUS_GAIN) {
            // Raise it back to normal
        }
    }
};

```

A loss of audio focus is the most important broadcast to react to. A temporary loss of audio focus should result in your app silencing its audio stream, but otherwise maintaining the same state. You should continue to monitor changes in audio focus and be prepared to resume playback where it was paused once you have regained the focus. If the audio focus loss is permanent, it is assumed that another application is now being used to listen to audio and your app should effectively end itself. In practical terms, that means stopping playback, removing media button listeners—allowing the new audio player to exclusively handle those events—and abandoning your audio focus. At that point, you would expect a user action (pressing play in your app) to be required before you resume playing audio.

When the playback functions need to be implemented, it is important for the application developers to know the existing protocols supported in the Android framework. The following are the network protocols are supported for audio and video playback in Android framework:

- RTSP (RTP, SDP)
- HTTP/HTTPS progressive streaming
- HTTP/HTTPS live streaming draft protocol:
 - MPEG-2 TS media files only
 - Protocol version 3 (Android 4.0 and above)
 - Protocol version 2 (Android 3.x)
 - Not supported before Android 3.0

10.3.3 Check Your Progress



Identify the audio and video multimedia functions used in one of the following Android apps: XfinityTV, Google Music, Ustream, Netflix. Consider the technical aspects we have covered in earlier sections.

Next, we will discuss the camera function usage in Android application development.

10.4 CAMERA FUNCTIONS IN ANDROID APPLICATION DEVELOPMENT

The Android framework includes support for various cameras and camera features available on mobile devices to capture pictures and videos in your applications.

Android class `Android.hardware.camera2` API primarily supports capturing images and videos. In addition, `Camera`, `Intent`, `SurfaceView`, `MediaRecorder`, classes are available. When developing an Android application it is important to declare the necessary permissions and features. A major advantage in declaring the features is to make sure that Google Play will only allow to install a specific application only on the mobile devices with those features of the camera. In addition to storage permission, depending on whether it is an external or internal memory should be specified during the application development. Also, if the application has a location tagging feature, in order to support this appropriate permissions to get the location information should be declared with associated permissions.

In an application, whether a camera is available on the device can be verified at runtime. Then, the camera is accessed through an instance of it. An example code is shown below.

```
private boolean checkCameraHardware(Context context) {
    if (context.getPackageManager().hasSystemFeature
        (PackageManager.FEATURE_CAMERA)){
        // this device has a camera
        return true;
    } else {
        // no camera on this device
        return false;
    }
}

/** A safe way to get an instance of the Camera object. */
public static Camera getCameraInstance(){
    Camera c = null;
    try {
        c = Camera.open(); // attempt to get a Camera instance
    }
    catch (Exception e){
        // Camera is not available (in use or does not exist)
    }
    return c; // returns null if camera is unavailable
}
```

Next, the camera features are loaded. Once the camera is ready to capture the picture or the video SurfaceView is used to obtain preview of the live image. Then, a layout is specified as a container for the preview through FrameLayout. Next, Camera.takePicture() method is used to capture a picture and MediaRecorder class is used to capture a video. Configuration of the MediaRecorder is vital. Example code with the steps for configuration is shown below. Also, note that MediaRecorder can be used to create videos from captured photos. Time lapse video allows users to create video clips that combine pictures taken a few seconds or minutes apart. This feature uses MediaRecorder to record the images for a time lapse sequence.

```
private boolean prepareVideoRecorder(){
    mCamera = getCameraInstance();
    mMediaRecorder = new MediaRecorder();

    // Step 1: Unlock and set camera to MediaRecorder
    mCamera.unlock();
    mMediaRecorder.setCamera(mCamera);

    // Step 2: Set sources
    mMediaRecorder.setAudioSource(MediaRecorder.AudioSource.CAMCORDER);
    mMediaRecorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);

    // Step 3: Set a CamcorderProfile (requires API Level 8 or higher)
    mMediaRecorder.setProfile(CamcorderProfile.get(CamcorderProfile.QUALITY_HIGH));

    // Step 4: Set output file
    mMediaRecorder.setOutputFile(getOutputMediaFile(MEDIA_TYPE_VIDEO).toString());

    // Step 5: Set the preview output
    mMediaRecorder.setPreviewDisplay(mPreview.getHolder().getSurface());

    // Step 6: Prepare configured MediaRecorder
    try {
        mMediaRecorder.prepare();
    } catch (IllegalStateException e) {
        Log.d(TAG, "IllegalStateException preparing MediaRecorder: " +
            e.getMessage());
        releaseMediaRecorder();
        return false;
    }
}
```

Camera is a resource that is shared by many applications on a device. Your application can make use of the camera after getting an instance of Camera, and you must be particularly careful to release the camera object when your application stops using it, and as soon as your application is paused (`Activity.onPause()`). If your application does not properly release the camera, all subsequent attempts to access the camera, including those by your own application, will fail and may cause your or other applications to shut down.

To release an instance of the Camera object, use the `Camera.release()` method, as shown in the example code below.

```

public class CameraActivity extends Activity {
    private Camera mCamera;
    private SurfaceView mPreview;
    private MediaRecorder mMediaRecorder;
    ...
    @Override
    protected void onPause() {
        super.onPause();
        releaseMediaRecorder(); // if you are using MediaRecorder, release it
        releaseCamera();       // release the camera immediately on pause
    }
    private void releaseMediaRecorder(){
        if (mMediaRecorder != null) {
            mMediaRecorder.reset(); // clear recorder configuration
            mMediaRecorder.release(); // release the recorder object
            mMediaRecorder = null;
            mCamera.lock();         // lock camera for later use
        }
    }
    private void releaseCamera(){
        if (mCamera != null){

```

Using existing camera applications

It is also important to note that existing camera applications can be invoked through the Intent class to capture pictures and videos. The following example demonstrates how to construct an image capture intent and execute it.

```

private static final int CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE = 100;
private Uri fileUri;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    // create Intent to take a picture and return control to the calling application
    Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    fileUri = getOutputMediaFileUri(MEDIA_TYPE_IMAGE); // create a file to
save the image
    intent.putExtra(MediaStore.EXTRA_OUTPUT, fileUri); // set the image file
name
    // start the image capture Intent
    startActivityForResult(intent,
CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE);
}

```

The following example demonstrates how to construct a video capture intent and execute it.

```

private static final int
CAPTURE_VIDEO_ACTIVITY_REQUEST_CODE = 200;
private Uri fileUri;
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    //create new Intent
    Intent intent = new
Intent(MediaStore.ACTION_VIDEO_CAPTURE);
    fileUri = getOutputMediaFileUri(MEDIA_TYPE_VIDEO); //
create a file to save the video
    intent.putExtra(MediaStore.EXTRA_OUTPUT, fileUri); // set the
image file name
    intent.putExtra(MediaStore.EXTRA_VIDEO_QUALITY, 1); // set
the video quality to high
}

```

The general steps for creating a custom camera interface for your application are as follows: (**STEPS instead of bullets -> FLOW CHART)

- **Detect and Access Camera** - Create code to check for the existence of cameras and request access.
- **Create a Preview Class** - Create a camera preview class that extends SurfaceView and implements the SurfaceHolder interface. This class previews the live images from the camera.
- **Build a Preview Layout** - Once you have the camera preview class, create a view layout that incorporates the preview and the user interface controls you want.
- **Setup Listeners for Capture** - Connect listeners for your interface controls to start image or video capture in response to user actions, such as pressing a button.
- **Capture and Save Files** - Setup the code for capturing pictures or videos and saving the output.
- **Release the Camera** - After using the camera, your application must properly release it for use by other applications.

Camera hardware is a shared resource that must be carefully managed so your application does not collide with other applications that may also want to use it. The following sections discuss how to detect camera hardware, how to request access to a camera, how to capture pictures or video and how to release the camera when your application is done using it.

10.4.1 Check Your Progress



Write a code to play an audio file when a photo is open to view. Assume that the photo and the audio file are stored in the device memory.

Next, we will discuss the different formats/codes supported for images, audio and video files in the Android platform.

10.5 SUPPORTED MEDIA FORMATS

When working with multimedia for Android application development, it is important to understand the core file formats and codec support that is provided (or in-built) in the Android platform. MediaCodec class is useful to access low-level media codecs, which are the encoder/decoder components. This class is part of the Android low-level multimedia support infrastructure. A codec processes input data to generate output data. Input data can be compressed data, raw audio data and raw video data. These input data are processed asynchronously and use buffers to store the output. Once the output is consumed, the buffers are released back to the codec.

Different codecs are supported with encoding parameters. These parameters include, resolution, bit-rate, and type of channel. Here we have summarized (in Table10.1), the codec support provided for image, audio and video. It is important to note that some mobile devices may provide support for additional formats/codecs not listed explicitly in Table10.1.

Table10.1: Core Media Support in Android for Audio, Video and Images.
(Source: Content summarized based on the information given in <https://developer.android.com/guide/appendix/media-formats.html>)

Multimedia	Media CODEC
Audio	<ul style="list-style-type: none"> • AAC LC • HE-AACv1 (AAC+) • HE-AACv2 (enhanced AAC+) • AMR-NB • AMR-WB • FLAC • MP3 • MIDI • Vorbis • Opus • wavv
Video	<ul style="list-style-type: none"> • H.263 • H.264 AVC • H.265 HEVC • MPEG-4 SP • VP8 • VP9
Images	<ul style="list-style-type: none"> • JPEG • GIF • BMP • PNG • WebP

In this unit we have discussed how to incorporate multimedia functions when developing an Android application.

10.6 SUMMARY



This unit covered the topics on how to integrate multimedia in different Android applications. The unit discussed how to utilize multimedia with appropriate examples and code snippets. To further enhance your skills and understanding activities and supplementary video were provided.

10.7 FURTHER READINGS

- <https://developer.android.com/guide/topics/media?authuser=1>
- <https://developer.android.com/guide/topics/media/mediaplayer>
- <https://blog.contus.com/video-call-integration/>



ignou
THE PEOPLE'S
UNIVERSITY



UNIT 11 SAVING DATA ON ANDROID DEVICES

- 11.0 Introduction
 - 11.1 Objectives
 - 11.2 Android Storage Options
 - 11.3 Shared Preferences
 - 11.3.1 Check Your Progress
 - 11.4 Internal Storage
 - 11.5 External Storage
 - 11.5.1 Check Your Progress
 - 11.6 Saving Data in SQLite Databases
 - 11.6.1 Check Your Progress
 - 11.7 Summary
 - 11.8 Further readings
-

11.0 INTRODUCTION

In this unit you will learn about different methods to store data locally in Android. You will learn about how to use these different options and when to use them.

You will further learn about data management using SQLite. The videos available for this unit will guide you on these data related operations.

On the completion of this unit you will be able to integrate data management functionality to Android applications that you develop.

11.1 OBJECTIVES

After studying this unit, you should be able to:



Outcomes

- Compare and contrast the different methods of persisting data locally.
- Write a program to access data in internal file system of an Android device and a SD card
- Use SQLite to create, alter update data tables and manipulate data.



Terminology

- | | |
|-------------------------|---|
| Persisting data: | denotes information that is infrequently accessed and not likely to be modified |
| SD Card: | Secure Digital (SD) is a non-volatile memory card format |
| SQLite: | C library used for database software |

11.2 ANDROID STORAGE OPTIONS

Android provides several options for you to save persistent application data. The solution you choose depends on your specific needs, such as whether the data should be private to your application or accessible to other applications (and the user) and how much space your data requires.

Your data storage options are the following:

- i. Shared Preferences - Store private primitive data as key-value pairs.
- ii. Internal Storage - Store private data on the device memory.
- iii. External Storage - Store public data on the shared external storage.
- iv. SQLite Databases - Store structured data in a private database.
- v. Remote server- Store data on a remotely hosted server.

In the following sections, we will discuss about the first four storage options mentioned above. We will however, focus only on the first four options and will not discuss about saving persistent data on cloud or other network location.

11.3 SHARED PREFERENCES

The `SharedPreferences` class provides a general framework that allows you to save and retrieve persistent key-value pairs of primitive data types. You can use `SharedPreferences` to save any primitive data: booleans, floats, ints, longs, and strings. This data will persist across user units (even if your application is killed). These data will only be removed only by uninstalling the application from the device or clearing the Application data via Settings menu. Also, data saved in Shared Preferences are private to this application only and not accessible by anyway for any other application on the device. We will discuss how and when to use shared preferences below.

When to use Shared Preferences?

If you need to save simple data for your application, the simplest and straight forward method is to use Shared Preferences. It is generally used to save simple data like integer, double, boolean, and short text. As Example, it is used to save application settings or user login info.

Using Shared Preferences

To get a `SharedPreferences` object for your application, use one of two methods:

getSharedPreferences() - Use this method if you need multiple preferences files identified by name, which you specify with the first parameter.

getPreferences() - Use this method if you need only one preferences file for your Activity. Because this will be the only preferences file for your Activity, it is not required to provide a name.

Write to Shared Preferences

To write to a shared preferences file, create a '`SharedPreferences.Editor`' by calling *edit()* on your `SharedPreferences`.

Pass the keys and values you want to write with methods such as `putInt()` and `putString()`. Then call `commit()` to save the changes. For example:

```
SharedPreferences sharedPref =
getActivity().getPreferences (Context.MODE_PRIVATE) ;
SharedPreferences.Editor editor = sharedPref.edit() ;
editor.putInt (getString(R.string.saved_high_score) ,
newHighScore) ;
editor.commit() ;
```

Read from Shared Preferences

To retrieve values from a shared preferences file, call methods such as `getInt()` and `getString()`, providing the key for the value you want, and optionally a default value to return if the key isn't present. For example:

```
SharedPreferences sharedPref =
getActivity().getPreferences (Context.MODE_PRIVATE) ;
int defaultValue =
getResources().getInteger(R.string.saved_high_score_default) ;
long highScore =
sharedPref.getInt (getString(R.string.saved_high_score) ,
defaultValue) ;
```

Here is an example that saves a preference for silent *keypress* mode in a calculator:

```
public class Calc extends Activity {
    public static final String PREFS_NAME =
"MyPrefsFile";
    @Override
    protected void onCreate(Bundle state){
        super.onCreate(state);
        . . .
        // Restore preferences
        SharedPreferences settings =
getSharedPreferences (PREFS_NAME, 0);
        boolean silent = settings.getBoolean("silentMode",
false);
        setSilent(silent);
    }
    @Override
    protected void onStop(){
        super.onStop();

        // Need an Editor object to make preference changes.
        // All objects are from android.context.Context
        SharedPreferences settings =
getSharedPreferences (PREFS_NAME, 0);
        SharedPreferences.Editor editor = settings.edit();
        editor.putBoolean("silentMode", mSilentMode);

        // Commit the edits!
        editor.commit();
    }
}
```

Delete Shared Preference Data

You have two options to delete data persisted as Shared Preference;

- i. Delete a specific item
- ii. Delete all data

Delete a specific item

```
SharedPreferences sharedPref =  
getApplicationContext().getSharedPreferences("MyAppData", 0);  
  
Editor sharedPrefEditor = sharedPref.edit();  
  
  
sharedPrefEditor.remove(key); // key of the data you want to  
delete  
  
sharedPrefEditor.commit();
```

Delete all Data

```
SharedPreferences sharedPref =  
getApplicationContext().getSharedPreferences("MyAppData", 0);  
  
Editor sharedPrefEditor = sharedPref.edit();  
  
sharedPrefEditor.clear(); //clear all data inside MyAppData  
Shared Preference File  
  
sharedPrefEditor.commit();
```

The next section will discuss about the internal storage options available with Android.

11.3.1 Check Your Progress



Why do we need shared preferences to store persistent data?

11.4 INTERNAL STORAGE

You can save files directly on the device's internal storage. By default, files saved to the internal storage are private to your application and other applications cannot access them (nor can the user). When the user uninstalls your application, these files are removed. Data will be removed only by uninstalling the application from the device.

When saving a file to internal storage, you can acquire the appropriate directory as a File by calling one of two methods:

getFilesDir() - Returns a 'File' representing an internal directory for your app.

getCacheDir() - Returns a 'File' representing an internal directory for your app's temporary cache files. Be sure to delete each file once it is no longer needed and

implement a reasonable size limit for the amount of memory you use at any given time, such as 1MB. If the system begins running low on storage, it may delete your cache files without warning.

We will now discuss how and when to use internal storage.

When to use internal storage?

The amount of data in Internal Storage depends on the device. Therefore do not try to save a large persistent file because it may crash your application if there is not enough space available on the device. Preferably, keep any data under 1M such as text files or xml files.

Write files to internal storage

The following steps show how to create and write a private file to the internal storage:

Step 1 - Call `openFileOutput()` with the name of the file and the operating mode. This returns a `FileOutputStream`.

Step 2 - Write to the file with `write()`.

Step 3 - Close the stream with `close()`.

For example:

```
String FILENAME = "hello_file";
String string = "hello world!";

FileOutputStream fos = openFileOutput(FILENAME,
Context.MODE_PRIVATE);
fos.write(string.getBytes());
fos.close();
```

MODE_PRIVATE will create the file (or replace a file of the same name) and make it private to your application. Other modes available are: *MODE_APPEND*, *MODE_WORLD_READABLE*, and *MODE_WORLD_WRITEABLE*.

Read from Internal Storage

To read a file from internal storage:

Step 1 - Call `openFileInput()` and pass it the name of the file to read. This returns a `FileInputStream`.

Step 2 - Read bytes from the file with `read()`.

Step 3 - Then close the stream with `close()`.

Tip: If you want to save a static file in your application at compile time, save the file in your project `res/raw/` directory. You can open it with `openRawResource()`, passing the `R.raw.<filename>` resource ID. This method returns an `InputStream` that you can use to read the file (but you cannot write to the original file).

Delete Internal Storage Data

The following examples shows how to delete a file from the internal storage.

```
File fileDir = getFilesDir();  
  
File file = new File(fileDir, "fileName");  
  
file.delete();
```

Saving cache files

If you would like to cache some data, rather than store it persistently, you should use `getCacheDir()` to open a 'File' that represents the internal directory where your application should save temporary cache files.

When the device is low on internal storage space, Android may delete these cache files to recover space. However, you should not rely on the system to clean up these files for you. You should always maintain the cache files yourself and stay within a reasonable limit of space consumed, such as 1MB. When the user uninstalls your application, these files are removed.

Next, we will focus on external storage options available with Android.

11.5 EXTERNAL STORAGE

Every Android-compatible device supports a shared "external storage" that you can use to save files. This can be a removable storage media (such as an SD card) or an internal (non-removable) storage. Files saved to the external storage are world-readable and can be modified by the user when they enable USB mass storage to transfer files on a computer. We will see how and when to use the external storage along with checking the availability of the media device and managing the visibility your files to other apps.

When to use External Storage?

Use External Storage whenever you need to save large files such as audio or video files and can be retrieved by repeatedly. Also you can use this if want your files to be shared through different application like statistics files.

Getting access to external storage

In order to read or write files on the external storage, your app must acquire the `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE` system permissions.

```
<manifest ...>  
    <uses-permission  
        android:name="android.permission.WRITE_EXTERNAL_STORAGE" />  
    ...  
</manifest>
```

If you need to both read and write files, then you need to request only the `WRITE_EXTERNAL_STORAGE` permission, because it implicitly requires read access as well.

Caution: Currently, all apps have the ability to read the external storage without a special permission. However, this is going to be changed in a future release. If your app needs to read the external storage (but not write to it), then you will need to declare the *READ_EXTERNAL_STORAGE* permission. To ensure that your app continues to work as expected, you should declare this permission now, before the change takes effect.

```
<manifest ...>
    <uses-permission
android:name="android.permission.READ_EXTERNAL_STORAGE" />
    ...
</manifest>
```

However, if your app uses the *WRITE_EXTERNAL_STORAGE* permission, then it implicitly has permission to read the external storage as well.

You do not need any permissions to save files on the internal storage. Your application always has permission to read and write files in its internal storage directory.

Checking media availability

Before you do any work with the external storage, you should always call *getExternalStorageState()* to check whether a compatible media is available. The media might be mounted to a computer, read-only, or in some other state. For example, here are a couple methods you can use to check the availability:

```
/* Checks if external storage is available for read and write */
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}

/* Checks if external storage is available to at least read */
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

The *getExternalStorageState()* method returns other states that you might want to check, such as whether the media is being shared (connected to a computer), is missing entirely, has been removed badly, etc. You can use these to notify the user with more information when your application needs to access the media.

Query Free Space

If you know ahead of time how much data you are saving, you can find out whether sufficient space is available without causing an *IOException* by calling *getFreeSpace()* or *getTotalSpace()*. These methods provide the current available space and the total space in the storage volume, respectively.

Saving files that can be shared with other apps

You may require sharing files across other apps. Sometimes, you may want to hide your files from others. We will now see how to share with or hide your files from other apps.

Hiding your files from the Media Scanner

To hide your files, include an empty file named *.nomedia* in your external files directory (note the dot prefix in the filename). This prevents media scanner from reading your media files and providing them to other apps through the MediaStore content provider. However, if your files are truly private to your app, you should save them in an app-private directory.

Generally, new files that the user may acquire through your app should be saved to a "public" location on the device where other apps can access them and the user can easily copy them from the device. When doing so, you should use to one of the shared public directories, such as Music/, Pictures/, and Ringtones/.

Saving files that are public to the users

To get a 'File' representing the appropriate public directory, call *getExternalStoragePublicDirectory()*, passing it the type of directory you want, such as *DIRECTORY_MUSIC*, *DIRECTORY_PICTURES*, *DIRECTORY_RINGTONES*, or others. By saving your files to the corresponding media-type directory, the system's media scanner can properly categorize your files in the system (for instance, ringtones appear in system settings as ringtones, not as music).

For example, here is a method that creates a directory for a new photo album in the public pictures directory:

```
public File getAlbumStorageDir(String albumName) {
    // Get the directory for the user's public pictures directory.
    File file = new
    File(Environment.getExternalStoragePublicDirectory(
        Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

Saving files that are app-private

If you want to save files that are private to your app, you can acquire the appropriate directory by calling *getExternalFilesDir()* and 'passing' it a name indicating the type of directory you would like. Each directory created this way is added to a parent directory that encapsulates all your app's external storage files, which the system deletes when the user uninstalls your app. For example, here's a method you can use to create a directory for an individual photo album:

```
public File getAlbumStorageDir(Context context, String
albumName) {
    // Get the directory for the app's private pictures directory.

    File file = new File(context.getExternalFilesDir(
        Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

If none of the pre-defined sub-directory names suit your files, you can instead call `getExternalFilesDir()` and pass null. This returns the root directory for your app's private directory on the external storage.

Remember that `getExternalFilesDir()` creates a directory inside a directory that is deleted when the user uninstalls your app. If the files you are saving should remain available after the user uninstalls your app for instance when your app is a camera and the user will want to keep the photos. Instead use `getExternalStoragePublicDirectory()`.

Caution: Although the directories provided by `getExternalFilesDir()` and `getExternalFilesDirs()` are not accessible by the MediaStore content provider, other apps with the `READ_EXTERNAL_STORAGE` permission can access all files on the external storage, including these. If you need to completely restrict access for your files, you should instead write your files to the internal storage.

Saving cache files

To open a 'File' that represents the external storage directory where you should save cache files, call `getExternalCacheDir()`. If the user uninstalls your application, these files will be automatically deleted.

Similar to `ContextCompat.getExternalFilesDirs()`, mentioned above, you can also access a cache directory on a secondary external storage (if available) by calling `ContextCompat.getExternalCacheDirs()`.

Delete a File

You should always delete files that you no longer need. The most straightforward way to delete a file is to have the opened file reference call `delete()` on itself.

```
myFile.delete();
```

If the file is saved on internal storage, you can also ask the Context to locate and delete a file by calling `deleteFile()`:

```
myContext.deleteFile(fileName);
```

Note: When the user uninstalls your app, the Android system deletes the following:

All files you saved on internal storage

All files you saved on external storage using `getExternalFilesDir()`.

However, you should manually delete all cached files created with *getCacheDir()* on a regular basis and also regularly delete other files you no longer need.

The next section will discuss about how to save data in SQLite databases, read the stored data when required and manipulating the stored data as per your requirement.

11.5.1 Check Your Progress



Compare and contrast data storage using internal storage and external storage options.

11.6 SAVING DATA IN SQLITE DATABASES

Saving data to a database is ideal for repeating or structured data, such as contact information. This class assumes helps you get started with SQLite databases on Android. The APIs you will need to use a database on Android are available in the `android.database.sqlite` package. We will discuss how and when to use databases as a storage option.

When to use SQLite DB?

You can use databases to store user data as per your app requirement. Otherwise, it does not have a specific reason to use a database.

Define a Schema and Contract

One of the main principles of SQL databases is the schema: a formal declaration of how the database is organized. The schema is reflected in the SQL statements that you use to create your database. You may find it helpful to create a companion class, known as a contract class, which explicitly specifies the layout of your schema in a systematic and self-documenting way.

A contract class is a container for constants that define names for URIs, tables, and columns. The contract class allows you to use the same constants across all the other classes in the same package. This let you change a column name in one place and have it propagate throughout your code.

A good way to organize a contract class is to put definitions that are global to your whole database in the root level of the class. Then create an inner class for each table that enumerates its columns.

Note: By implementing the `BaseColumns` interface, your inner class can inherit a primary key field called `_ID` that some Android classes such as cursor adaptors will expect it to have. It is not required, but this can help your database work harmoniously with the Android framework.

For example, this snippet defines the table name and column names for a single table:

```
public final class FeedReaderContract {
    // To prevent someone from accidentally instantiating the
    // contract class, give it an empty constructor.
    public FeedReaderContract() {}

    /* Inner class that defines the table contents */
    public static abstract class FeedEntry implements
    BaseColumns {
        public static final String TABLE_NAME = "entry";
        public static final String COLUMN_NAME_ENTRY_ID =
        "entryid";
        public static final String COLUMN_NAME_TITLE = "title";
        public static final String COLUMN_NAME_SUBTITLE =
        "subtitle";
        ...
    }
}
```

Create a Database Using a SQL Helper

Once you have defined how your database looks, you should implement methods that create and maintain the database and tables. Here are some typical statements that create and delete a table:

```
private static final String TEXT_TYPE = " TEXT";
private static final String COMMA_SEP = ",";
private static final String SQL_CREATE_ENTRIES =
    "CREATE TABLE " + FeedEntry.TABLE_NAME + " (" +
    FeedEntry._ID + " INTEGER PRIMARY KEY," +
    FeedEntry.COLUMN_NAME_ENTRY_ID + TEXT_TYPE + COMMA_SEP +
    FeedEntry.COLUMN_NAME_TITLE + TEXT_TYPE + COMMA_SEP +
    ... // Any other options for the CREATE command
    ")";

private static final String SQL_DELETE_ENTRIES =
    "DROP TABLE IF EXISTS " + FeedEntry.TABLE_NAME;
```

Just like files that you save on the device's internal storage, Android stores your database in private disk space associated with the application. Your data is secure, because by default this area is not accessible to other applications.

A useful set of APIs is available in the *SQLiteOpenHelper* class. When you use this class to obtain references to your database, the system performs the potentially long-running operations of creating and updating the database only when needed and not during app startup. All you need to do is call *getWritableDatabase()* or *getReadableDatabase()*.

Note: Because they can be long-running, be sure that you call *getWritableDatabase()* or *getReadableDatabase()* in a background thread, such as with *AsyncTask* or *IntentService*.

To use *SQLiteOpenHelper*, create a subclass that overrides the *onCreate()*, *onUpgrade()* and *onOpen()* callback methods. You may also want to implement *onDowngrade()*, but it's not required.

For example, here is an implementation of *SQLiteOpenHelper* that uses some of the commands shown above:

```
public class FeedReaderDbHelper extends SQLiteOpenHelper {
    // If you change the database schema, you must increment
    the database version.
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "FeedReader.db";

    public FeedReaderDbHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(SQL_CREATE_ENTRIES);
    }
    public void onUpgrade(SQLiteDatabase db, int oldVersion,
int newVersion) {
        // This database is only a cache for online data, so
        its upgrade policy is
        // to simply to discard the data and start over
        db.execSQL(SQL_DELETE_ENTRIES);
        onCreate(db);
    }
    public void onDowngrade(SQLiteDatabase db, int oldVersion,
int newVersion) {
        onUpgrade(db, oldVersion, newVersion);
    }
}
```

To access your database, instantiate your subclass of *SQLiteOpenHelper*:

```
FeedReaderDbHelper mDbHelper = new FeedReaderDbHelper(getContext());
```

Put Information into a Database

Insert data into the database by passing a *ContentValues* object to the *insert()* method:

```
// Gets the data repository in write mode
SQLiteDatabase db = mDbHelper.getWritableDatabase();

// Create a new map of values, where column names are the keys
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_ENTRY_ID, id);
values.put(FeedEntry.COLUMN_NAME_TITLE, title);
values.put(FeedEntry.COLUMN_NAME_CONTENT, content);

// Insert the new row, returning the primary key value of the new //row
long newRowId;
newRowId = db.insert(
    FeedEntry.TABLE_NAME,
    FeedEntry.COLUMN_NAME_NULLABLE,
    values);
```

The first argument for *insert()* is simply the table name. The second argument provides the name of a column in which the framework can insert NULL in the event

that the `ContentValues` is empty (if you instead set this to "null", then the framework will not insert a row when there are no values).

Read Information from a Database

To read from a database, use the `query()` method, passing it your selection criteria and desired columns. The method combines elements of `insert()` and `update()`, except the column list defines the data you want to fetch, rather than the data to insert. The results of the query are returned to you in a `Cursor` object.

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// Define a projection that specifies which columns from the
// database
// you will actually use after this query.
String[] projection = {
    FeedEntry._ID,
    FeedEntry.COLUMN_NAME_TITLE,
    FeedEntry.COLUMN_NAME_UPDATED,
    ...
};

// How you want the results sorted in the resulting Cursor
String sortOrder =
    FeedEntry.COLUMN_NAME_UPDATED + " DESC";

Cursor c = db.query(
    FeedEntry.TABLE_NAME,    // The table to query
    projection,              // The columns to return
    selection,               // The columns for the WHERE clause
    selectionArgs,           // The values for the WHERE clause
    null,                   // don't group the rows
    null,                   // don't filter by row groups
    sortOrder               // The sort order
);
```

To look at a row in the cursor, use one of the `Cursor` move methods, which you must always call before you begin reading values. Generally, you should start by calling `moveToFirst()`, which places the "read position" on the first entry in the results. For each row, you can read a column's value by calling one of the `Cursor` get methods, such as `getString()` or `getLong()`. For each of the get methods, you must pass the index position of the column you desire, which you can get by calling `getColumnIndex()` or `getColumnIndexOrThrow()`.

For example:

```
cursor.moveToFirst();
long itemId = cursor.getLong(
    cursor.getColumnIndexOrThrow(FeedEntry._ID)
);
```

Note: You can execute `SQLite` queries using the `SQLiteDatabase query()` methods, which accept various query parameters, such as the table to query, the projection, selection, columns, grouping, and others. For complex queries, such as those that

require column aliases, you should use *SQLiteQueryBuilder*, which provides several convenient methods for building queries.

Delete Information from a Database

To delete rows from a table, you need to provide selection criteria that identify the rows. The database API provides a mechanism for creating selection criteria that protects against SQL injection. The mechanism divides the selection specification into a selection clause and selection arguments. The clause defines the columns to look at, and also allows you to combine column tests. The arguments are values to test against that are bound into the clause. Because the result is not handled the same as a regular SQL statement, it is immune to SQL injection.

```
// Define 'where' part of query.
String selection = FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
// Specify arguments in placeholder order.
String[] selectionArgs = { String.valueOf(rowId) };
// Issue SQL statement.
db.delete(table_name, selection, selectionArgs);
```

Update a Database

When you need to modify a subset of your database values, use the *update()* method.

Updating the table combines the content values syntax of *insert()* with the where syntax of *delete()*.

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// New value for one column
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_TITLE, title);

// Which row to update, based on the ID
String selection = FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
String[] selectionArgs = { String.valueOf(rowId) };

int count = db.update(
    FeedReaderDbHelper.FeedEntry.TABLE_NAME,
    values,
    selection,
    selectionArgs);
```

With that we have come to the end of this unit and now you are in a position to use data management on your apps.

11.6.1 Check Your Progress



Choose a popular game app and discuss how each of the storage options you learnt is used in that app.

11.7 SUMMARY



In this unit we discussed about the four of the five methods used to persist data on Android devices. They are namely; Shared preferences, Internal storage, external storage and external database. We also learnt about when to use each of these options and how we can use them in apps. The important things to keep in your mind is that the privacy level of the files/data and the size of the data.

11.8 FURTHER READINGS

<https://developer.android.com/training/location/index.html>

<https://developer.android.com/training/data-storage>

<https://www.sqlite.org/index.html>

ignou
THE PEOPLE'S
UNIVERSITY



ignou
THE PEOPLE'S
UNIVERSITY

UNIT 12 LOCATING AND SENSING

- 12.0 Introduction
- 12.1 Objectives
- 12.2 Introduction to Sensors
- 12.3 Android Sensor Framework
- 12.4 Identifying Sensors and sensor Capabilities
 - 12.4.1 Check Your Progress
- 12.5 Monitoring Sensor Events
- 12.6 Sensor Coordinate System
- 12.7 Best Practices for Accessing and Using Sensors
- 12.8 Commonly Used Sensors
 - 12.8.1 Check Your Progress
- 12.9 Making Your App Location-Aware
- 12.10 Getting the Last Known Location
- 12.11 Changing Location Settings
- 12.12 Receiving Location Updates
- 12.13 Adding Google Maps to Your App
 - 12.13.1 Check Your Progress
- 12.14 Summary
- 12.15 Further readings

12.0 INTRODUCTION

In this unit you will be introduced to various sensors available in an Android device. The availability of a particular sensor depends on the device you use. You will learn how to use these sensors accessing relevant API's to obtain data and use them in applications.

You will also learn about Google Maps and how location awareness can be achieved in applications.

A video is incorporated in this unit to help you to understand more about sensors and Google map.

12.1 OBJECTIVES

After studying this unit, you should be able to :



Outcomes

- identify functionality of the common sensors available in Android devices.
- illustrate how sensors can be used in apps with examples.
- develop a small app to demonstrate the context awareness of the users with location.



Terminology

- | | |
|---------------------------|---|
| sensors: | measure motion, orientation, position, and various environmental conditions |
| context awareness: | refers to a general class of mobile systems that can sense their physical environment |

12.2 INTRODUCTION TO SENSORS

Most Android devices have built-in sensors that measure motion, orientation, position, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy, and are useful if you want to monitor three-dimensional device movement or positioning, or you want to monitor changes in the ambient environment near a device.

Some of these sensors are hardware-based and some are software-based. Hardware-based sensors are physical components built into a handset or tablet device. They derive their data by directly measuring specific environmental properties, such as acceleration, geomagnetic field strength, or angular change. Software-based sensors are not physical devices, although they mimic hardware-based sensors. Software-based sensors derive their data from one or more of the hardware-based sensors and are sometimes called virtual sensors or synthetic sensors. The linear acceleration sensor and the gravity sensor are examples of software-based sensors. Appendix 12.1 summarizes the sensor types that are supported by the Android platform.

The Android platform supports three broad categories of sensors:

- Motion sensors - These sensors measure acceleration forces and rotational forces along three axes (x,y,and z). List of Motion sensors that are supported on the Android platform is given in reference 12.2
- Environmental sensors - These sensors measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity. Appendix 12.3 lists the environment sensors that are supported on the Android platform
- Position sensors - These sensors measure the physical position of a device. Appendix 12.4 lists the position sensors that are supported on the Android platform.

In the next section, we will discuss about the Android sensor framework.

12.3 ANDROID SENSOR FRAMEWORK

The sensor framework provides several classes and interfaces that help you perform a wide variety of sensor-related tasks. For example, you can use the sensor framework to do the following:

- Determine which sensors are available on a device.
- Determine an individual sensor's capabilities, such as its maximum range, manufacturer, power requirements, and resolution.
- Acquire raw sensor data and define the minimum rate at which you acquire sensor data.
- Register and unregister sensor event listeners that monitor sensor changes.

Now, we will see the classes and interfaces available in the sensor framework.

Classes and Interfaces Available in the Sensor Framework

The sensor framework is part of Android. Hardware package includes the following classes and interfaces:

Sensor Manager

You can use this class to create an instance of the sensor service. This class provides various methods for accessing and listing sensors, registering and unregistering sensor event listeners, and acquiring orientation information. This class also provides several sensor constants that are used to report sensor accuracy, set data acquisition rates, and calibrate sensors.

Sensor

You can use this class to create an instance of a specific sensor. This class provides various methods that let you determine a sensor's capabilities.

Sensor Event

The system uses this class to create a sensor event object, which provides information about a sensor event. A sensor event object includes the following information: the raw sensor data, the type of sensor that generated the event, the accuracy of the data, and the timestamp for the event.

Sensor Event Listener

You can use this interface to create two callback methods that receive notifications (sensor events) when sensor values change or when sensor accuracy changes.

In the next section we will learn how to identify different sensors available on a device and the capabilities of each sensor.

12.4 IDENTIFYING SENSORS AND SENSOR CAPABILITIES

Identifying sensors and sensor capabilities at runtime is useful if your application has features that rely on specific sensor types or capabilities. For example, you may want to identify all of the sensors that are present on a device and disable any application features that rely on sensors that are not present. Likewise, you may want to identify all of the sensors of a given type so you can choose the sensor implementation that has the optimum performance for your application.

To identify the sensors that are on a device you first need to get a reference to the sensor service. To do this, you create an instance of the 'SensorManager' class by calling the *getSystemService()* method and passing in the *SENSOR_SERVICE* argument.

```
private SensorManager mSensorManager;  
...  
mSensorManager = (SensorManager)  
getSystemService(Context.SENSOR_SERVICE);
```

Next, you can get a listing of every sensor on a device by calling the *getSensorList()* method and using the *TYPE_ALL* constant.

```
List<Sensor> deviceSensors =  
mSensorManager.getSensorList(Sensor.TYPE_ALL);
```

If you want to list all of the sensors of a given type, you could use another constant instead of `TYPE_ALL` such as `TYPE_GYROSCOPE`, `TYPE_LINEAR_ACCELERATION`, or `TYPE_GRAVITY`.

You can also determine whether a specific type of sensor exists on a device by using the `getDefaultSensor()` method. If a default sensor does not exist for a given type of sensor, the method call returns null, which means the device does not have that type of sensor. The following code checks whether there's a magnetometer on a device:

```
private SensorManager mSensorManager;
...
mSensorManager = (SensorManager)
getSystemService(Context.SENSOR_SERVICE);
if (mSensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD)
!= null) {
    // Success! There's a magnetometer.
}
else {
    // Failure! No magnetometer.
}
```

In addition to listing the sensors that are on a device, you can use the public methods of the `Sensor` class to determine the capabilities and attributes of individual sensors. For example, you can use the `getResolution()` and `getMaximumRange()` methods to obtain a sensor's resolution and maximum range of measurement. You can also use the `getPower()` method to obtain a sensor's power requirements.

Now that we have learnt how to identify sensors and their capabilities, it is time to learn how to monitor sensor events. The next section will discuss about monitoring sensor events.

12.4.1 Check Your Progress



Find and list all the sensors available in your Android device.

12.5 MONITORING SENSOR EVENTS

A sensor event occurs every time a sensor detects a change in the parameters it is measuring. A sensor event provides you with four pieces of information: the name of the sensor that triggered the event, the timestamp for the event, the accuracy of the event, and the raw sensor data that triggered the event.

To monitor raw sensor data you need to implement two callback methods that are exposed through the `SensorEventListener` interface: `onAccuracyChanged()` and `onSensorChanged()`. The Android system calls these methods whenever the following occurs:

A sensor's accuracy changes.

In this case the system invokes the *onAccuracyChanged()* method, providing you with a reference to the *Sensor* object that changed and the new accuracy of the sensor.

A sensor reports a new value.

In this case the system invokes the *onSensorChanged()* method, providing you with a *SensorEvent* object. A *SensorEvent* object contains information about the new sensor data, including: the accuracy of the data, the sensor that generated the data, the timestamp at which the data was generated, and the new data that the sensor recorded.

The following code shows how to use the *onSensorChanged()* method to monitor data from the light sensor. This example displays the raw sensor data in a *TextView* that is defined in the *main.xml* file as *sensor_data*.

```
public class SensorActivity extends Activity implements
SensorEventListener {
    private SensorManager mSensorManager;
    private Sensor mLight;

    @Override
    public final void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        mSensorManager = (SensorManager)
getSystemService(Context.SENSOR_SERVICE);
        mLight =
mSensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
    }
    @Override
    public final void onAccuracyChanged(Sensor sensor, int
accuracy) {
        // Do something here if sensor accuracy changes.
    }

    @Override
    public final void onSensorChanged(SensorEvent event) {
        // The light sensor returns a single value.
        // Many sensors return 3 values, one for each axis.
        float lux = event.values[0];
        // Do something with this sensor value.
    }

    @Override
    protected void onResume() {
        super.onResume();
        mSensorManager.registerListener(this, mLight,
SensorManager.SENSOR_DELAY_NORMAL);
    }

    @Override
    protected void onPause() {
        super.onPause();
        mSensorManager.unregisterListener(this);
    }
}
```

In this example, the default data delay (`SENSOR_DELAY_NORMAL`) is specified when the `registerListener()` method is invoked. The data delay (or sampling rate) controls the interval at which sensor events are sent to your application via the `onSensorChanged()` callback method.

Next, we will discuss about the sensor coordinate system.

12.6 SENSOR COORDINATE SYSTEM

In general, the sensor framework uses a standard 3-axis coordinate system to express data values. For most sensors, the coordinate system is defined relative to the device's screen when the device is held in its default orientation (Figure 12.1).

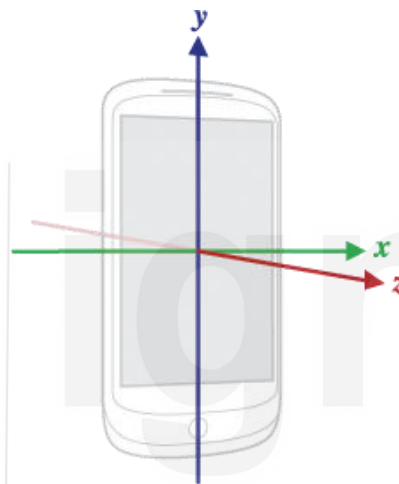


Figure 12.1- Coordinate system used by the Sensor API

The most important point to understand about this coordinate system is that the axes are not swapped when the device's screen orientation changes—that is, the sensor's coordinate system never changes as the device moves.

Another point to understand is that your application must not assume that a device's natural (default) orientation is portrait. The natural orientation for many tablet devices is landscape. And the sensor coordinate system is always based on the natural orientation of a device.

In the next section, we will discuss about the best practices for accessing and using sensors.

12.7 BEST PRACTICES FOR ACCESSING AND USING SENSORS

Under this topic we will be explaining some of the best practices that you should follow when accessing and using sensors.

Unregister sensor listeners

Be sure to unregister a sensor's listener when you have finished using the sensor or when the sensor activity pauses. If a sensor listener is registered and its activity is paused, the sensor will continue to acquire data and use battery resources unless you unregister the sensor. Also there could be some exception the application runtime, if activity get destroyed without unregistering the sensor manager. The following code snippet shows how to use the *onPause()* method to unregister a listener:

```
private SensorManager mSensorManager;  
...  
@Override  
protected void onPause() {  
    super.onPause();  
    mSensorManager.unregisterListener(this);  
}
```

Do not test your code on the emulator

The default Android emulator cannot emulate sensors. You should test your sensor code on a physical device or emulator which capable to emulate the sensors as well. There are, however, sensor simulators that you can use to simulate sensor output.

Do not block the *onSensorChanged()* method

Sensor data can change at a high rate, which means the system may call the *onSensorChanged(SensorEvent)* method quite often. As a best practice, you should do as little as possible within the *onSensorChanged(SensorEvent)* method so you don't block it. If your application requires you to do any data filtering or reduction of sensor data, you should perform that work outside of the *onSensorChanged(SensorEvent)* method.

Verify sensors before you use them

Always verify that a sensor exists on a device before you attempt to acquire data from it. Don't assume that a sensor exists simply because it's a frequently-used sensor. Device manufacturers are not required to provide any particular sensors in their devices.

Choose sensor delays carefully

When you register a sensor with the *registerListener()* method, be sure you choose a delivery rate that is suitable for your application or use-case. Sensors can provide data at very high rates. Allowing the system to send extra data that you don't need wastes system resources and uses battery power.

Next, we will discuss about some of the commonly used sensors.

12.8 COMMONLY USED SENSORS

There are many different types of sensors. Under this topic we will be explaining some of the commonly used sensors accelerometer, Gravity Sensor, Gyroscope and Proximity Sensor.

Accelerometer

An acceleration sensor measures the acceleration applied to the device, including the force of gravity. The following code shows you how to get an instance of the default acceleration sensor:

```
private SensorManager mSensorManager;
private Sensor mSensor;

...
mSensorManager = (SensorManager)
getSystemService(Context.SENSOR_SERVICE);
mSensor =
mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
```

It should be noted that the force of gravity(g) is always influencing the measured acceleration. For this reason, when the device is sitting on a table (and not accelerating), the accelerometer reads a magnitude of $g = 9.81 \text{ m/s}^2$. Similarly, when the device is in free fall and therefore rapidly accelerating toward the ground at 9.81 m/s^2 , its accelerometer reads a magnitude of $g = 0 \text{ m/s}^2$. Therefore, to measure the real acceleration of the device, the contribution of the force of gravity must be removed from the accelerometer data. This can be achieved by applying a high-pass filter. Conversely, a low-pass filter can be used to isolate the force of gravity. The following example shows how you can do this:

```
public void onSensorChanged(SensorEvent event) {
    // In this example, alpha is calculated as  $t / (t + \text{dT})$ ,
    // where  $t$  is the low-pass filter's time-constant and
    //  $\text{dT}$  is the event delivery rate.

    final float alpha = 0.8;

    // Isolate the force of gravity with the low-pass filter.
    gravity[0] = alpha * gravity[0] + (1 - alpha) *
event.values[0];
    gravity[1] = alpha * gravity[1] + (1 - alpha) *
event.values[1];
    gravity[2] = alpha * gravity[2] + (1 - alpha) *
event.values[2];

    // Remove the gravity contribution with the high-pass filter.
    linear_acceleration[0] = event.values[0] - gravity[0];
    linear_acceleration[1] = event.values[1] - gravity[1];
    linear_acceleration[2] = event.values[2] - gravity[2];
}
```

In general, the accelerometer is a good sensor to use if you are monitoring device motion. Almost every Android-powered handset and tablet has an accelerometer, and it uses about 10 times less power than the other motion sensors. One drawback is that you might have to implement low-pass and high-pass filters to eliminate gravitational forces and reduce noise.

Gravity Sensor

The gravity sensor provides a three dimensional vector indicating the direction and magnitude of gravity. The following code shows you how to get an instance of the default gravity sensor:

```
private SensorManager mSensorManager;
private Sensor mSensor;
...
mSensorManager = (SensorManager)
getSystemService(Context.SENSOR_SERVICE);
mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_GRAVITY);
```

The units are the same as those used by the acceleration sensor (m/s²), and the coordinate system is the same as the one used by the acceleration sensor

Gyroscope

The gyroscope measures the rate of rotation in rad/s around a device's x, y, and z axis. The following code shows you how to get an instance of the default gyroscope:

```
private SensorManager mSensorManager;
private Sensor mSensor;
...
mSensorManager = (SensorManager)
getSystemService(Context.SENSOR_SERVICE);
mSensor =
mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE);
```

The sensor's coordinate system is the same as the one used for the acceleration sensor. Rotation is positive in the counter-clockwise direction.

Standard gyroscopes provide raw rotational data without any filtering or correction for noise and drift (bias). In practice, gyroscope noise and drift will introduce errors that need to be compensated for. You usually determine the drift (bias) and noise by monitoring other sensors, such as the gravity sensor or accelerometer.

Proximity Sensor

The proximity sensor lets you determine how far away an object is from a device. The following code shows you how to get an instance of the default proximity sensor:

```
private SensorManager mSensorManager;
private Sensor mSensor;
...
mSensorManager = (SensorManager)
getSystemService(Context.SENSOR_SERVICE);
mSensor =
mSensorManager.getDefaultSensor(Sensor.TYPE_PROXIMITY);
```

The proximity sensor is usually used to determine how far away a person's head is from the face of a handset device (for example, when a user is making or receiving a phone call). Most proximity sensors return the absolute distance, in cm, but some return only near and far values. The following code shows you how to use the proximity sensor:

```
public class SensorActivity extends Activity implements
SensorEventListener {
    private SensorManager mSensorManager;
    private Sensor mProximity;
```

```

@Override
public final void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    // Get an instance of the sensor service, and use that to
    get an instance of
    // a particular sensor.
    mSensorManager = (SensorManager)
    getSystemService(Context.SENSOR_SERVICE);
    mProximity =
    mSensorManager.getDefaultSensor(Sensor.TYPE_PROXIMITY);
}

@Override
public final void onAccuracyChanged(Sensor sensor, int
accuracy) {
    // Do something here if sensor accuracy changes.
}

@Override
public final void onSensorChanged(SensorEvent event) {
    float distance = event.values[0];
    // Do something with this sensor data.
}

@Override
protected void onResume() {
    // Register a listener for the sensor.
    super.onResume();
    mSensorManager.registerListener(this, mProximity,
    SensorManager.SENSOR_DELAY_NORMAL);
}

@Override
protected void onPause() {
    // Be sure to unregister the sensor when the activity
    pauses.
    super.onPause();
    mSensorManager.unregisterListener(this);
}
}

```

Next we will discuss about how to make your apps location-aware.

12.8.1 Check Your Progress



What are the sensors/location details required in following category of apps?

- ✓ Car racing game
- ✓ Weather predictor
- ✓ Navigation app
- ✓ Video chat app

12.9 MAKING YOUR APP LOCATION-AWARE

One of the unique features of mobile applications is location awareness. Mobile users take their devices with them everywhere, and adding location awareness to your app offers users a more contextual experience. The location APIs available in Google Play services facilitate adding location awareness to your app with automated location tracking, geofencing (a facility that monitors whether a device is near to a location of interest), and activity recognition.

Android offers two ways to add location awareness to your apps; one through the Google Play services location APIs and the other is through the Android framework location API. However, the former is preferred over the latter as a way of adding location awareness to your app.

In this section, we will discuss how to use the Google Play services location APIs in your app to get the current location, and get periodic location updates. In addition the same API's can be used to maintain addresses and geofences. However, we will not discuss about addresses and geofences here.

Specify App Permissions

Apps that use location services must request location permissions. Android offers two location permissions: `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`. The permission you choose determines the accuracy of the location returned by the API. If you specify `ACCESS_COARSE_LOCATION`, the API returns a location with an accuracy approximately equivalent to a city block.

Connect to Google Play Services

To connect to the API, you need to create an instance of the Google Play services API client. For details about using the client, see the guide to Accessing Google APIs.

In your activity's `onCreate()` method, create an instance of Google API Client, using the `GoogleApiClient.Builder` class to add the `LocationServices` API, as the following code snippet shows.

```
// Create an instance of GoogleApiClient.
if (mGoogleApiClient == null) {
    mGoogleApiClient = new GoogleApiClient.Builder(this)
        .addConnectionCallbacks(this)
        .addOnConnectionFailedListener(this)
        .addApi(LocationServices.API)
        .build();
}
```

To connect, call `connect()` from the activity's `onStart()` method. To disconnect, call `disconnect()` from the activity's `onStop()` method. The following snippet shows an example of how to use both of these methods.

```
protected void onStart() {
    mGoogleApiClient.connect();
    super.onStart();
}

protected void onStop() {
    mGoogleApiClient.disconnect();
    super.onStop();
}
```

In the next section we will discuss about how to obtain the last known location of a particular Android device.

12.10 GETTING THE LAST KNOWN LOCATION

Use the fused location provider to retrieve the device's last known location. The fused location provider manages the underlying location technology and provides a simple API so that you can specify requirements at a high level, like high accuracy or low power. It also optimizes the device's use of battery power.

This service requires only coarse location. Request this permission with the `uses-permission` element in your app manifest, as the following code snippet shows:

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
package="com.google.android.gms.location.sample.basiclocations
ample" >

<uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION"/>
</manifest>
```

To request the last known location, call the `getLastLocation()` method, passing it your instance of the `GoogleApiClient` object. Do this in the `onConnected()` callback provided by Google API Client, which is called when the client is ready. The following code snippet illustrates the request and a simple handling of the response:

```
public class MainActivity extends ActionBarActivity implements
    ConnectionCallbacks, OnConnectionFailedListener {
    ...
    @Override
    public void onConnected(Bundle connectionHint) {
        mLastLocation =
        LocationServices.FusedLocationApi.getLastLocation(
            mGoogleApiClient);
        if (mLastLocation != null) {
            mLatitudeText.setText(String.valueOf(mLastLocation.
            getLatitude()));
            mLongitudeText.setText(String.valueOf(mLastLocation
            .getLongitude()));
        }
    }
}
```

The `getLastLocation()` method returns a `Location` object from which you can retrieve the latitude and longitude coordinates of a geographic location.

Now that we are aware of how to get the location details, we can discuss about changing location settings. Next section will discuss about changing location settings.

12.11 CHANGING LOCATION SETTINGS

To change location settings, coarse location detection is sufficient. Request this permission with the `uses-permission` element in your app manifest, as shown in the following example:

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
package="com.google.android.gms.location.sample.locationupdates" >

<uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION"/>

</manifest>
```

Set Up a Location Request

To store parameters for requests to the fused location provider, create a `LocationRequest`. The parameters determine the level of accuracy for location requests. Here you will learn to set the update interval, fastest update interval, and priority, as described below:

Update interval

`setInterval()` - This method sets the rate in milliseconds at which your app prefers to receive location updates.

Fastest update interval

`setFastestInterval()` - This method sets the fastest rate in milliseconds at which your app can handle location updates. You need to set this rate because other apps also affect the rate at which updates are sent. The Google Play services location APIs send out updates at the fastest rate that any app has requested with `setInterval()`. If this rate is faster than your app can handle, you may encounter problems with UI flicker or data overflow. To prevent this, call `setFastestInterval()` to set an upper limit to the update rate.

Priority

`setPriority()` - This method sets the priority of the request, which gives the Google Play services location services a strong hint about which location sources to use. The following values are supported:

- **PRIORITY_BALANCED_POWER_ACCURACY** - Use this setting to request location precision to within a city block (approximately 100 meters). This is considered a coarse level of accuracy, and is likely to consume less power. With this setting, the location services are likely to use WiFi and cell tower positioning.
- **PRIORITY_HIGH_ACCURACY** - Use this setting to request the most precise location possible. With this setting, the location services are more likely to use GPS to determine the location.

- **PRIORITY_LOW_POWER** - Use this setting to request city-level precision, which is an accuracy of approximately 10 kilometers. This is considered a coarse level of accuracy, and is likely to consume less power.
- **PRIORITY_NO_POWER** - Use this setting if you need negligible impact on power consumption, but want to receive location updates when available. With this setting, your app does not trigger any location updates, but receives locations triggered by other apps.
- Create the location request and set the parameters as shown in this code sample:

```
protected void createLocationRequest() {
    LocationRequest mLocationRequest = new LocationRequest();
    mLocationRequest.setInterval(10000);
    mLocationRequest.setFastestInterval(5000);
    mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_
    ACCURACY);
}
```

Get Current Location Settings

To do this, create a *LocationSettingsRequest.Builder*, and add one or more location requests. The following code snippet shows how to add the location request that was created in the previous step:

```
LocationSettingsRequest.Builder builder = new
LocationSettingsRequest.Builder()
    .addLocationRequest(mLocationRequest);

Next check whether the current location settings are satisfied:

PendingResult<LocationSettingsResult> result
= LocationServices.SettingsApi.checkLocationSettings(mG
oogleClient,builder.build());
```

When the *PendingResult* returns, your app can check the location settings by looking at the status code from the *LocationSettingsResult* object. To get even more details about the the current state of the relevant location settings, your app can call the *LocationSettingsResult* object's *getLocationSettingsStates()* method.

Prompt the User to Change Location Settings

To determine whether the location settings are appropriate for the location request, check the status code from the *LocationSettingsResult* object. A status code of **RESOLUTION_REQUIRED** indicates that the settings must be changed. To prompt the user for permission to modify the location settings, call *startResolutionForResult(Activity, int)*. This method brings up a dialog asking for the user's permission to modify location settings. The following code snippet shows how to check the location settings, and how to call *startResolutionForResult(Activity, int)*.

```
result.setResultCallback(new
ResultCallback<LocationSettingsResult>()) {
    @Override
    public void onResult(LocationSettingsResult result) {
        final Status status = result.getStatus();
```

```

        final LocationSettingsStates =
result.getLocationSettingsStates();
        switch (status.getStatusCode()) {
            case LocationSettingsStatusCodes.SUCCESS:
                // All location settings are satisfied. The
client can
                // initialize location requests here.
                ...
                break;
            case
LocationSettingsStatusCodes.RESOLUTION_REQUIRED:
                // Location settings are not satisfied, but
this can be fixed
                // by showing the user a dialog.
                try {
                    // Show the dialog by calling
startResolutionForResult(),
                    // and check the result in
onActivityResult().

                    status.startResolutionForResult(
                        OuterClass.this,
                        REQUEST_CHECK_SETTINGS);
                } catch (SendIntentException e) {
                    // Ignore the error.
                }
                break;
            case
LocationSettingsStatusCodes.SETTINGS_CHANGE_UNAVAILABLE:
                // Location settings are not satisfied.
However, we have no way
                // to fix the settings so we won't show the
dialog.
                ...
                break;
        }
    }
});

```

Next we will discuss how to receive location updates on a device.

12.12 RECEIVING LOCATION UPDATES

If your app can continuously track location, it can deliver more relevant information to the user. For example, if your app helps the user find their way while walking or driving, or if your app tracks the location of assets, it needs to get the location of the device at regular intervals. As well as the geographical location (latitude and longitude), you may want to give the user further information such as the bearing (horizontal direction of travel), altitude, or velocity of the device. This information, and more, is available in the Location object that your app can retrieve from the fused location provider.

While you can get a device's location with `getLastLocation()`, a more direct approach is to request periodic updates from the fused location provider. In response, the API

updates your app periodically with the best available location, based on the currently-available location providers such as WiFi and GPS (Global Positioning System). The accuracy of the location is determined by the providers, the location permissions you've requested, and the options you set in the location request.

The starting point of location update would be to get the last known location of the device. This also ensures that the app has a known location before starting the periodic location updates. The snippets in the following sections assume that your app has already retrieved the last known location and stored it as a `Location` object in the global variable `mCurrentLocation`.

For this service you require fine location detection, so that your app can get as precise a location as possible from the available location providers. Request this permission with the `uses-permission` element in your app manifest, as shown in the following example:

```
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.google.android.gms.location.sample.locationupdates"
  >

  <uses-permission
    android:name="android.permission.ACCESS_FINE_LOCATION"/>
</manifest>
```

Request Location Updates

Before requesting location updates, your app must connect to location services and make a location request. Once a location request is in place you can start the regular updates by calling `requestLocationUpdates()`. Do this in the `onConnected()` callback provided by Google API Client, which is called when the client is ready.

Depending on the form of the request, the fused location provider either invokes the `LocationListener.onLocationChanged()` callback method and passes it a `Location` object, or issues a `PendingIntent` that contains the location in its extended data. The accuracy and frequency of the updates are affected by the location permissions you've requested and the options you set in the location request object.

This section shows you how to get the update using the `LocationListener` callback approach. Call `requestLocationUpdates()`, passing it your instance of the `GoogleApiClient`, the `LocationRequest` object, and a `LocationListener`. Define a `startLocationUpdates()` method, called from the `onConnected()` callback, as shown in the following code sample:

```
@Override
public void onConnected(Bundle connectionHint) {
    ...
    if (mRequestingLocationUpdates) {
        startLocationUpdates();
    }
}

protected void startLocationUpdates() {
    LocationServices.FusedLocationApi.requestLocationUpdates (
```

```

        mGoogleApiClient, mLocationRequest, this);
    }

```

Notice that the above code snippet refers to a boolean flag, *mRequestingLocationUpdates*, used to track whether the user has turned location updates on or off.

Define the Location Update Callback

The fused location provider invokes the *LocationListener.onLocationChanged()* callback method. The incoming argument is a *Location* object containing the location's latitude and longitude. The following snippet shows how to implement the *LocationListener* interface and define the method, then get the timestamp of the location update and display the latitude, longitude and timestamp on your app's user interface:

```

public class MainActivity extends ActionBarActivity implements
    ConnectionCallbacks, OnConnectionFailedListener,
    LocationListener {
    ...
    @Override
    public void onLocationChanged(Location location) {
        mCurrentLocation = location;
        mLastUpdateTime =
            DateFormat.getInstance().format(new Date());
        updateUI();
    }

    private void updateUI()
    {
        mLatitudeTextView.setText(String.valueOf(mCurrentLocat
            ion.getLatitude()));
        mLongitudeTextView.setText(String.v
            alueOf(mCurrentLocation.getLongitude()));
        mLastUpdateTimeTextView.setText(mLastUpdateTime);
    }
}

```

Stop Location Updates

Consider whether you want to stop the location updates when the activity is no longer in focus, such as when the user switches to another app or to a different activity in the same app. This can be handy to reduce power consumption, provided the app doesn't need to collect information even when it's running in the background. This section shows how you can stop the updates in the activity's *onPause()* method.

To stop location updates, call *removeLocationUpdates()*, passing it your instance of the *GoogleApiClient* object and a *LocationListener*, as shown in the following code sample:

```

@Override
protected void onPause() {
    super.onPause();
    stopLocationUpdates();
}

protected void stopLocationUpdates() {

```

```

        LocationServices.FusedLocationApi.removeLocationUpdates(
            mGoogleApiClient, this);
    }

```

Use a boolean, *mRequestingLocationUpdates*, to track whether location updates are currently turned on. In the activity's *onResume()* method, check whether location updates are currently active, and activate them if not:

```

@Override
public void onResume() {
    super.onResume();
    if (mGoogleApiClient.isConnected()
        && !mRequestingLocationUpdates) {
        startLocationUpdates();
    }
}

```

12.13 ADDING GOOGLE MAPS TO YOUR APP

Google maps are the most popular method of displaying maps. You can easily incorporate Google maps to your app. The following link shows you a step by step approach to incorporate Google maps to your app.

12.13.1 Check Your Progress



Develop a small app that uses GPS information and your current location (in terms of latitude and longitude)

12.14 SUMMARY



In this unit, first we discussed about the various sensors available with Android framework. We also learnt about some important sensors and how to use them in an application. You will have to get familiar with the sensor coordinate system and the best practices when you develop apps. It will be useful to make your applications more efficient.

Thereafter, we discussed about how to incorporate location awareness to an app. We discussed about getting the current location of a device and also updating a location. Also we learnt about how to incorporate Google maps to your app. Sensor types supported by the Android platform are at the end of this unit as an Annexure.

12.15 FURTHER READINGS

https://developer.android.com/guide/topics/sensors/sensors_overview?authuser=1

<https://source.android.com/devices/sensors/sensor-stack>

<https://www.javatpoint.com/android-sensor-tutorial>

UNIT 13 CONNECTIVITY AND THE CLOUD

- 13.0 Introduction
 - 13.1 Objectives
 - 13.2 Connecting devices wirelessly
 - 13.3 Performing network operations
 - 13.3.1 Check Your Progress
 - 13.4 Considerations when transferring data
 - 13.5 Syncing to the cloud with information delivery models
 - 13.5.1 Video – V11: Connectivity and the cloud
 - 13.6 Push notification
 - 13.6.1 Check Your Progress
 - 13.7 Summary
 - 13.8 Further Readings
-

13.0 INTRODUCTION

This unit will help you to understand the process of connecting your application to the world beyond the user's device. You will learn how to connect your application to other devices in the area and to the Internet, how to take backup and sync your applications data and how to deal with Push notifications in this unit.

13.1 OBJECTIVES

After studying this unit, you should be able to:



Outcomes

- explain the modes of connecting an Android application with different devices wirelessly
- write a program to connect the app with outside world
- discuss the advantages of sending push notifications over polling method
- write a program to send Push notifications from the Android application



Terminology

- | | |
|----------------------|--|
| cloud: | remote servers hosted on the Internet to store, manage, and process data |
| polling: | actively sampling the status of an external device by a client program as a synchronous activity |
| connectivity: | being connected to Internet |
-

13.2 CONNECTING DEVICES WIRELESSLY

First of all we will see what a wireless network is. It is any type of computer network that uses wireless data connections for connecting network nodes. Besides enabling communication with the cloud, Android's wireless APIs also enable communication with other devices on the same local network, and even devices which are not on a

network, but are physically nearby. The addition of Network Service Discovery (NSD) takes this further by allowing an application to seek out a nearby device running services with which it can communicate. Integrating this functionality into your application helps you provide a wide range of features, such as playing games with users in the same room, pulling images from a networked NSD-enabled webcam, or remotely logging into other machines on the same network.

Using Network Service Discovery

Adding Network Service Discovery (NSD) to your app allows your users to identify other devices on the local network that support the services your application requests. This is useful for a variety of peer-to-peer applications such as file sharing or multi-player gaming. Android's NSD APIs simplify the effort required for you to implement such features.

Creating peer-to-peer (P2P) Connections with Wi-Fi

The Wi-Fi P2P APIs allow applications to connect to nearby devices without needing to connect to a network or hotspot. Wi-Fi P2P allows your application to quickly find and interact with nearby devices, at a range beyond the capabilities of Bluetooth. Also it provides fast data transfer with more security than the Bluetooth.

Using Wi-Fi P2P for Service Discovery

Let's see how to discover services published by nearby devices without being on the same network using Wi-Fi P2P. Using Wi-Fi Peer-to-Peer (P2P) Service Discovery allows you to discover the services of nearby devices directly without being connected to a network. You can also advertise the services running on your device. These capabilities help you communicate between apps, even when no local network or hotspot is available. While this set of APIs is similar in purpose to the Network Service Discovery APIs outlined in a previous section, implementing them in code is very different.

13.3 PERFORMING NETWORK OPERATIONS

This section explains the basic tasks involved in connecting to the network, monitoring the network connection (including connection changes), and giving users control over an app's network usage. It also describes how to parse and consume XML data.

These fundamental building blocks will enable you to create Android applications that download content and parse data efficiently, while minimizing network traffic.

13.3.1 Check Your Progress



State the dependencies and prerequisites when performing network operations.

Connecting to the Network

To perform the network operations, your application manifest must include the following permissions:

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

Managing Network Usage

If your application performs a lot of network operations, you should provide user settings that allow users to control your app's data habits, such as how often your app syncs data, whether to perform uploads/downloads only when on Wi-Fi, whether to use data while roaming, and so on. With these controls available to them, users are much less likely to disable your app's access to background data when they approach their limits, because they can instead precisely control how much data your app uses.

Optimizing Network Data Usage

Over the life of a smartphone, the cost of a cellular data plan can easily exceed the cost of the device itself. From Android 7.0 (API level 24), users can enable Data Saver on a device-wide basis in order to optimize their device's data usage, and use less data. This ability is especially useful when roaming, near the end of the billing cycle, or for a small prepaid data pack. Though this has been introduced as an OS feature with Android 7.0, various vendors have introduced their own data saving mechanisms like ultra-data saving mode, on other Android versions which support VPN.

When a user enables Data Saver in Settings and the device is on a metered network, the system blocks background data usage and signals apps to use less data in the foreground wherever possible. Users can white-list specific apps to allow background metered data usage even when Data Saver is turned on.

Parsing XML Data

Extensible Markup Language (XML) is a set of rules for encoding documents in machine-readable form. XML is a popular format for sharing data on the internet. Websites that frequently update their content, such as news sites or blogs, often provide an XML feed so that external programs can keep abreast of content changes. Uploading and parsing XML data is a common task for network-connected apps.

13.4 CONSIDERATIONS WHEN TRANSFERRING DATA

In mobile devices battery and memory is a limited resource. For your app to be considered 'good', it should seek to limit its impact on the battery life of its device.

By taking steps such as batching network requests, disabling background service updates when you lose connectivity, or reducing the rate of such updates when the battery level is low, you can ensure that the impact of your app on battery life is minimized, without compromising the user experience.

Optimizing Downloads for Efficient Network Access

Using the wireless radio to transfer data is potentially one of your app's most significant sources of battery drain. To minimize the battery drain associated with

network activity, it's critical that you understand how your connectivity model will affect the underlying radio hardware. It goes on to propose ways to minimize your data connections, use prefetching, and bundle your transfers in order to minimize the battery drain associated with your data transfers

Minimizing the effect of regular updates

The optimal frequency of regular updates will vary based on device state, network connectivity, user behavior, and explicit user preferences.

Optimizing battery life discusses how to build battery-efficient apps that modify their refresh frequency based on the state of the host device. That includes disabling background service updates when you lose connectivity and reducing the rate of updates when the battery level is low.

How your refresh frequency can be varied to best mitigate the effect of background updates on the underlying wireless radio state machine.

Redundant downloads are redundant

The most fundamental way to reduce your downloads is to download only what you need. In terms of data, that means implementing REST APIs that allow you to specify query criteria that limit the returned data by using parameters such as the time of your last update.

Similarly, when downloading images, it is a good practice to reduce the size of the images server-side, rather than downloading full-sized images that are reduced on the client.

Modifying your download patterns based on the connectivity Type

When it comes to impact on battery life, not all connection types are created equal. Not only does the Wi-Fi radio use significantly less battery than its wireless radio counterparts, but the radios used in different wireless radio technologies have different battery implications

13.5 SYNCING TO THE CLOUD WITH INFORMATION DELIVERY MODELS

The Wearable Data Layer API, which is part of Google Play services, provides a communication channel for your handheld and wearable apps. The API consists of a set of data objects that the system can send and synchronize over the wire and listeners that notify your apps of important events with the data layer

13.5.1 Video – V11: Connectivity and the cloud



In this video you will be learning how to integrate cloud based services to your application.



URL: <https://tinyurl.com/y86wfhhw>

13.6 PUSH NOTIFICATION

A notification is a message you can display to the user outside of your application's normal UI. When you tell the system to issue a notification, it first appears as an icon in the notification area. To see the details of the notification, the user opens the notification drawer. Both the notification area and the notification drawer are system-controlled areas that the user can view at any time.

Polling happens when the phone goes to the server with a certain interval and asks if there are any messages to process. Basically there are two ways to get messages to the phone:

1. Push messages (the server contacts the phone and tells it there are messages waiting)
2. Polling service (the phone contacts the server and ask for messages)

Creating a notification

You can specify the UI information and actions for a notification in a `NotificationCompat.Builder` object. To create the notification you call `NotificationCompat.Builder.build()`, which returns a `Notification` object containing your specifications. To issue the notification, you pass the `Notification` object to the system by calling `NotificationManager.notify()`

The following snippet illustrates a simple notification that specifies an activity to open when the user clicks the notification. Notice that the code creates a `TaskStackBuilder` object and uses it to create the `PendingIntent` for the action. This pattern is explained in more detail in the section *Preserving Navigation when Starting an Activity*:

```
NotificationCompat.Builder mBuilder =
    new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.notification_icon)
        .setContentTitle("My notification")
        .setContentText("Hello World!");

// Creates an explicit intent for an Activity in your app
Intent resultIntent = new Intent(this, ResultActivity.class);
// The stack builder object will contain an artificial back //stack for
// the started Activity.
// This ensures that navigating backward from the Activity leads //out of your
// application to the Home //screen.
TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
// Adds the back stack for the Intent (but not the Intent //itself)
stackBuilder.addParentStack(ResultActivity.class);
// Adds the Intent that starts the Activity to the top of the //stack
stackBuilder.addNextIntent(resultIntent);
PendingIntent resultPendingIntent =
    stackBuilder.getPendingIntent(
        0,
        PendingIntent.FLAG_UPDATE_CURRENT
```



```
);  
mBuilder.setContentIntent(resultPendingIntent);  
NotificationManager mNotificationManager =  
    (NotificationManager)  
    getSystemService(Context.NOTIFICATION_SERVICE);  
// mId allows you to update the notification later on.  
mNotificationManager.notify(mId, mBuilder.build());
```

Google Cloud Messaging as an alternative to polling

Every time an app polls a server to check if an update is required, you activate the wireless radio, drawing power unnecessarily, for up to 20 seconds on a typical 3G connection.

Compared to polling, where your app must regularly ping the server to query for new data, Google cloud messaging model allows your app to create a new connection only when it knows there is data to download.

The result is a reduction in unnecessary connections, and a reduced latency for updated data within your application.

Google Cloud Messaging for Android (GCM) is a lightweight mechanism used to transmit data from a server to a particular app instance. Using GCM, your server can notify your app running on a particular device that there is new data available for it.

GCM is implemented using a persistent TCP/IP connection. While it's possible to implement your own push service, it's best practice to use GCM. This minimizes the number of persistent connections and allows the platform to optimize bandwidth and minimize the associated impact on battery life

Google Cloud Messaging (GCM) is a free service that enables developers to send messages between servers and client apps. This includes downstream messages from servers to client apps, and upstream messages from client apps to servers.

For example, a lightweight downstream message could inform a client app that there is new data to be fetched from the server, as in the case of a "new email" notification. For use cases such as instant messaging, a GCM message can transfer up to 4kb of payload to the client app. The GCM service handles all aspects of queuing of messages and delivery to and from the target client app.

A GCM implementation includes a Google connection server, an app server in your environment that interacts with the connection server via HTTP or XMPP protocol, and a client app.

13.6.1 Check Your Progress



Select a commonly used Android application and identify how the application connect with other devices, with internet, how the application syncs and backup data.

13.7 SUMMARY



Summary

This unit provided you an insight of how to connect different devices to your application wirelessly. Furthermore how to transfer data without draining the battery and how to get started with syncing to the cloud using information delivery models were discussed. At the end of the unit the drawbacks of using polling methods over push notifications and the Android code snippets used for sending push notifications were explained.

13.8 FURTHER READINGS

<https://developer.android.com/training/connect-devices-wirelessly#:~:text=Besides%20enabling%20communication%20with%20the,network%2C%20but%20are%20physically%20nearby.>
<https://developer.android.com/training/basics/network-ops>
<https://www.netapp.com/hybrid-cloud/what-is-hybrid-cloud/>



UNIT 14 PUBLISH TO ANDROID MARKET

- 14.0 Introduction
- 14.1 Objectives
- 14.2 How can you obtain an Android application?
- 14.3 App Stores
 - 14.3.1 Check Your Progress
- 14.4 Revenue Models
 - 14.4.1 Check Your Progress
- 14.5 Google Play Store
- 14.6 Process of Publishing an Android Application
 - 14.6.1 Video – V12: Publish to Android Market
 - 14.6.2 Check Your Progress
- 14.7 Summary
- 14.8 Further readings

14.0 INTRODUCTION

By following this unit you will gain theoretical knowledge on the revenue and distribution models, process of launching an Android application in a distributed environment. Knowledge gained from this unit will help you to make an informed choice of the business models to cater to specific requirements when launching a developed Android application to the market.

The provided video will demonstrate you how to publish an Android application to the Google Play.

14.1 OBJECTIVES

After studying this unit, you should be able to :



Outcomes

- translate the appropriate distribution model to reach target audience
- describe available business models
- select appropriate business models for the applications
- demonstrate how to deploy the developed application in Android Market



Terminology

Localization: the process of making something local in character or restricting it to a particular place

14.2 HOW CAN YOU OBTAIN AN ANDROID APPLICATION?

As you already learnt in the previous units, Android mobile applications are software programs that may be installed on portable computing devices such as smartphones, tablets, some digital set-top boxes, laptops etc. Mobile applications market is the place where the buyers (consumers/users) and sellers (app developers) of mobile applications meet. These mobile application markets are dedicated retail platforms known as AppStores which can be accessible through the consumer's device. The main objective of an AppStore is to serve as a host to initially source online to distribute to potential users through a data connection and accessing device.

An end-user can obtain an Android application in two main ways. Applications may be pre-installed or downloaded on-demand. Pre-installed mobile applications are selected by device manufacturers and usually include the following.

- utilities (for example, calendars, alarm clocks, camera apps)
- services (for example, weather apps, Google maps, a compass, world clock)
- entertainment (for example, music, video, games)
- communications applications (for example, chat applications)

Mobile applications may be downloaded and installed by consumers in several ways:

- Via the device - end-user can directly access the device manufacturer's app store through a menu on the device, and download and install a mobile application. Access may be enabled through 3G or Wi fi networks (in Wifi-enabled devices).
- Via the Internet directly onto the mobile device - the end-user can access the manufacturer's app store via the web browser on his/her device.
- Via the Internet and transfer to the mobile device - the end-user can access the manufacturer's app store via the web browser on a personal computer and then transfer over Bluetooth or an external memory to install in the mobile device.

14.3 APP STORES

Mobile applications are commonly made available through aggregators with online stores. Mobile applications aggregators are not new to the communications industry. Online stores offering mobile phone ringtones, themes and other applications have existed since the late 1990s.

App stores may be categorized as:

Device manufacturers—including Apple's App Store, Nokia's Ovi, and Blackberry's App World. These stores can be used only by consumers with the appropriate manufacturer's device and proprietary software.

Operating system developer—including Android Market and Microsoft Windows Mobile. These stores can be accessed by consumers with devices from multiple handset manufacturers via the proprietary operating system software (OS). For

example, Android mobile applications can be used on Motorola, HTC and Samsung devices etc. which have Android Operating system.

Mobile network operator—including Telstra, Verizon and Optus. These stores can only be accessed by consumers with service contracts with the network operator. Consumers can use multiple handset brands to access these stores.

Independent—including app stores operated as independent commercial concerns, or by developers such as GetJar and Mobango. Access to these stores is not dependent on the brand of device used, service provider or proprietary software.

14.3.1 Check Your Progress



Based on the discussion in Sections 14.1 and 14.2, compare the AppStore classifications for publishing the following two applications.

AppA - health diary application to update calorie intake, number of hours of exercise etc for Hutch mobile users.

AppB - advanced colouring app which can edit and use existing art for children

14.4 REVENUE MODELS

Revenue models are different ways that entrepreneurs can earn money from their mobile applications. Under this topic we will be explaining three revenue models: Free Model, Paid model and Paymium model.

Free Model

Free model is also referred to as freemium model. In this model, users don't pay to download or use an application. The main advantage of removing the barrier of price increases the likelihood that users in the target market will download and try. This initiation can help increase awareness for the application and grow the distribution among the target user base.

The main source of revenue for freemium model is through advertisements. Applications can display advertisements and in return generate "ad revenue". It is also important to select appropriate advertisements to suit the application and its target market. Inappropriate advertisements can reduce the interactions with the users and their retention to continue usage.

Paid Model

In this model, users pay once to download an application. Then, use the functionalities provided by that application without any further payments. The paid model is suitable for users who prefer to pay once to get the full application experience, without in-app purchases. We will discuss the in-app model in the next section. Often the application developers need to pay careful attention to include premium experiences through attractive design and functionality for applications intend to be distributed using paid model. Since there is only a one-time payment it is important to drive the marketing strategy to acquire more users. It is advisable that the developers make sure that their

app's title, icon, description, preview, screenshots, and other marketing communications effectively showcase the premium nature of the application.

Paymium Model

Paymium model is essentially a combination of the paid and freemium models. Users pay to download the application and have the option to buy additional features, content, or services as they continue to use the application. Paymium model is suitable for users who would like to engage interacting with the application step-by-step depending on their requirements. Unlike in the paid model, this model allows a user to decide whether to acquire more features as and when needed rather than to pay a hefty price. In-app purchases can reduce the chances of disappointments in paid model where the user is not fully satisfied with the functionalities and features offered by an application. Another important aspect is the developers have the flexibility to add more customizable features.

Offering Subscriptions

This model offer an easy experience for digital subscriptions. In-app purchase APIs provide a simple, standardized way to implement auto-renewable and non-renewing subscriptions to content or services. With in-app purchase subscriptions, it is possible to the price and duration of subscriptions. Duration of subscriptions may be seven days, one month, two months, three months, six months, or one year. Often AppStores allow the users to manage the subscription of their applications. For example, in Apple AppStore a user can manage the subscriptions through the Apple user ID account.

Auto-Renewable Subscriptions: This subscription type gives users access to content that is regularly updated. At the end of each subscription duration, the subscription will renew automatically until a user chooses to turn off auto-renewal. Often free trials are offered for such applications using this business model. The length of the subscription determines how long the free trial can be. For example, for a monthly subscription you can offer users a 7-day or 1-month free trial. When users sign up for a subscription with a free trial, their subscription will begin immediately but they won't be billed until the free trial period is over. They will then continue to be billed on a recurring basis, unless the users turn off auto-renewal.

Non-Renewing Subscriptions: This subscription type give users access to content or services for a limited duration Non-renewing subscriptions require users to renew each time a subscription ends and may notify them when subscriptions is due to expire with a prompt to purchase a new subscription. Often free trials are offered for such applications using this business model. When users sign up for a subscription they won't be billed until the free trial period is over. Then the subscription will be billed to a pre-defined period and not on a recurring basis as there is no auto-renewal.

14.4.1 Check Your Progress



Identify the revenue model for the Equalizer Android application. Include the knowledge gained from the previous sections.

14.5 GOOGLE PLAY STORE

The Google Play Store or Google Play is a digital distribution service operated and developed by Google. It serves as the official app store for the Android operating system. Google Play allows users to browse and download applications developed with the Android SDK and published through Google. In Android devices, Google Play Store is an official pre-installed application. This pre-installed application provides access to the Google Play store for users to browse and download music, books, magazines, movies, television programs, and other applications from Google Play.

The Devices segment of Google Play is not accessible through the Play Store. The Play Store application is not open source. Only Android devices that comply with Google's compatibility requirements may install and access Google's closed-source Play Store application, subject to entering into a free-of-charge licensing agreement with Google.

Applications are available through Google Play use freemium or paid business models. They can be downloaded directly to an Android or [Google TV](#) device through the Play Store [mobile app](#), or by [deploying](#) the application to a device from the Google Play website. Many applications can be targeted to specific users based on a particular hardware attribute of their device, such as a motion sensor (for motion-dependent games) or a front-facing camera (for online video calling). Such specific application are allowed to download onto devices with appropriate in-built hardware or sufficient capabilities.

14.6 PROCESS OF PUBLISHING AN ANDROID APPLICATION

Once the application is developed, the next process is to make it available for the users to download and use it. To reliably distribute an Android application for the users to download, an AppStore is required. The steps of the publishing process can be summarized as follows.

1. Select an appropriate AppStore
2. Read and understand the policies and agreements of the selected AppStore
3. Quality test
4. Determine the content rating for the Android application
5. Determine the country (or the countries) to distribute
6. Confirm the overall size, platform and the screen compatibility ranges
7. Decide the revenue model
8. Decide how to bill or collect the revenue (e.g. In-App or using Google Pay)
9. Set the price (or prices)
10. Localization
11. Prepare promotional graphics, videos and screencasts
12. Build and upload
13. Plan for Beta release
14. Complete AppStore listing
15. Support users after launch

14.6.1 Video – V12: Publish to Android Market



This video shows the steps to follow when building your application to release and releasing the application to users.



URL: <https://tinyurl.com/Publish-to-Android-Market>

14.6.2 Check Your Progress



Based on the content covered in Section 14.3, 14.4 and 14.5, in your opinion what are the most significant steps that can have a significant impact (in terms of popularity and revenue loss/gain) if the revenue model is changed during the publishing process of the Android application.

14.7 SUMMARY



This unit covered the topics on revenue and distribution models, process of launching an Android application in a distributed environment. To further enhance your understanding on these topics activities and a supplementary video were provided.

14.8 FURTHER READINGS

- <https://www.businessofapps.com/guide/app-stores-list/>
- <https://support.google.com/googleplay/answer/113409?hl=en>
- <https://themanifest.com/mobile-apps/how-publish-app-google-play-step-step-guide>
- <https://developer.android.com/studio/publish>

UNIT 15 ANDROID APP PERFORMANCE

- 15.0 Introduction
- 15.1 Objectives
- 15.2 Performance Profiling
- 15.3 Android Monitor Overview
 - 15.3.1 Video -V13: Performance Profiling
 - 15.3.2 Check Your Progress
- 15.4 Android Monitor Basics
- 15.5 Profiling a Running App in Android Monitor
- 15.6 How Android Manages Memory
 - 15.6.1 Check Your Progress
- 15.7 Battery Analysis
 - 15.7.1 Video – V14: Battery Analysis
- 15.8 Optimizing Battery Life
 - 15.8.1 Check Your Progress
- 15.9 Summary
- 15.10 Further readings

15.0 INTRODUCTION

In this unit you will learn to analyze the performance of an Android application. First you will identify and use tools to visualize performance which would enable you to analyze performance in various aspects of an Android application. Then you will learn how to improve the performance by optimizing memory usage and minimizing the power consumption by selecting appropriate techniques.

15.1 OBJECTIVES

After studying this unit, you should be able to :



Outcomes

- identify tools to visualize performance of an Android app
- analyse performance of a developed application to identify the drawbacks
- apply techniques to reduce memory usage when programming for Android
- analyse battery life and apply techniques to optimize battery usage



Terminology

profiling: analyse performance using a tool

optimize: make the most effective use of something

15.2 PERFORMANCE PROFILING

With Android applications, it is possible to perform various things on devices such as running tasks in background, playing music or videos, or connecting with different networks like, Wi-Fi, 4G, and Bluetooth. So applications running in the device may consume its resources in many ways. For example displaying pixels on the screen involves four primary pieces of hardware. In simple terms the Central Processing Unit (CPU) computes display lists, the Graphical Processing Unit (GPU) renders images to the display, the memory stores images and data, and the battery provides electrical power. Each of these pieces of hardware has constraints; pushing or exceeding those constraints causes your app to be slow, have bad display performance, or exhaust the battery. Performance profiling means investigating and analyzing a mobile application's runtime behavior to decide how to optimize the performance the programs involved. In the next section, we will discuss a popular profiling tool and its sub-tools.

Performance Profiling Tools

To discover what causes your specific performance problems, you need to investigate in depth, i.e. use tools to collect data about your app execution behavior, organize that data in lists and graphics, and analyze what you see. Your mobile device together with Android Studio provides profiling tools to record and visualize the rendering, do computations, analyze memory and battery usage of your app. Android Monitor, which is introduced below is one such tool.

15.3 ANDROID MONITOR OVERVIEW

Android Monitor provides various sub-tools that you can use to profile the performance of an app so that you can optimize, debug, and improve them. It lets you monitor the following aspects of your apps from a hardware device or the [Android Emulator](#):

- [Log](#) messages, either system or user defined
- Memory, CPU, and GPU usage
- Network traffic (hardware device only)

It lets you capture data as your app runs and stores it in a file that you can analyze in various viewers. You can also capture screenshots and videos of your app as it runs.

[Android Monitor](#) has a main window that contains performance monitors such as logcat, Memory, CPU, GPU, and Network Monitors. From this window, you can select a device and app process to work with, terminate an app, collect *dumpsys* system information, and create screenshots and videos of the running app. *dumpsys* is an Android tool that runs on the device and dumps information about the status of system services.

Usage of these performance monitors will be described in the screen cast on 'Performance monitors'. A brief description of each performance monitor with a screen shot is given below for you to see what it looks like.

Those screen shots are taken from the weather app shown in Figure 15.1 below.



Figure 15.1 weather app

15.3.1 Video -V13: Performance Profiling



In this video you will see how different profiling tools in Android Monitor are used. These tools are, CPU monitor, GPU monitor, memory monitor, network monitor and logcat.

URL: <https://tinyurl.com/Performance-Profiling>



Figure 15.2 given below is a screen shot from logcat.

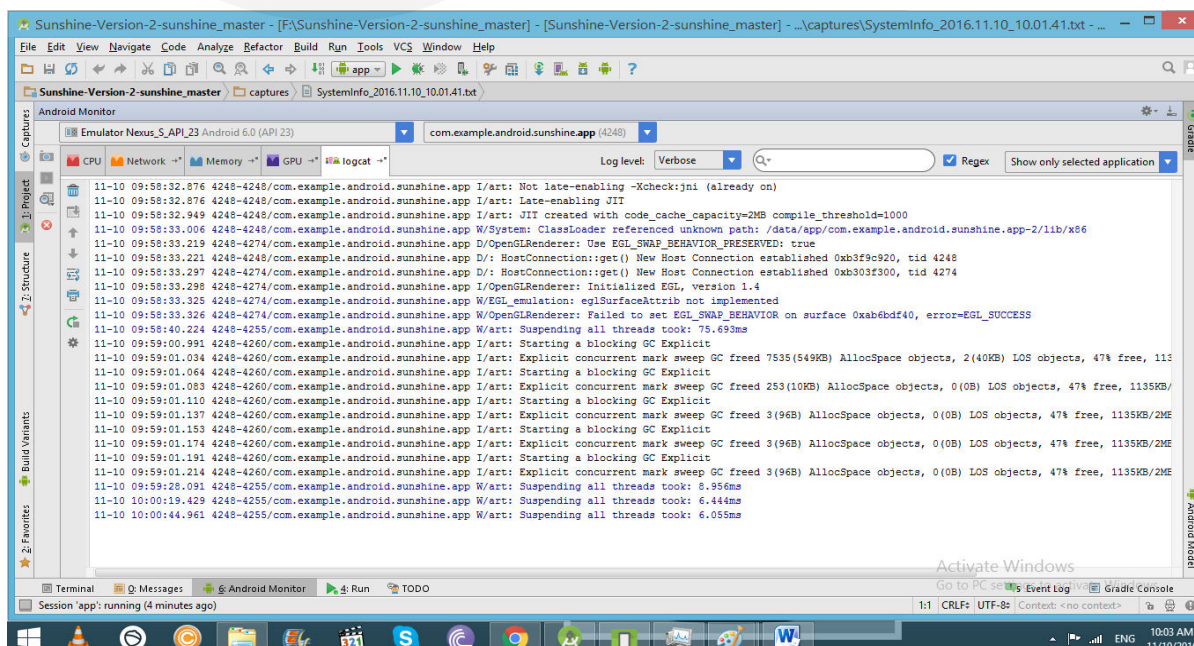


Figure 15.2 Logcat

Memory Monitor – We use the Memory Monitor an in figure 15.3 to evaluate memory usage and find de-allocated objects, locate memory leaks, and track the amount of memory that the connected device is using.

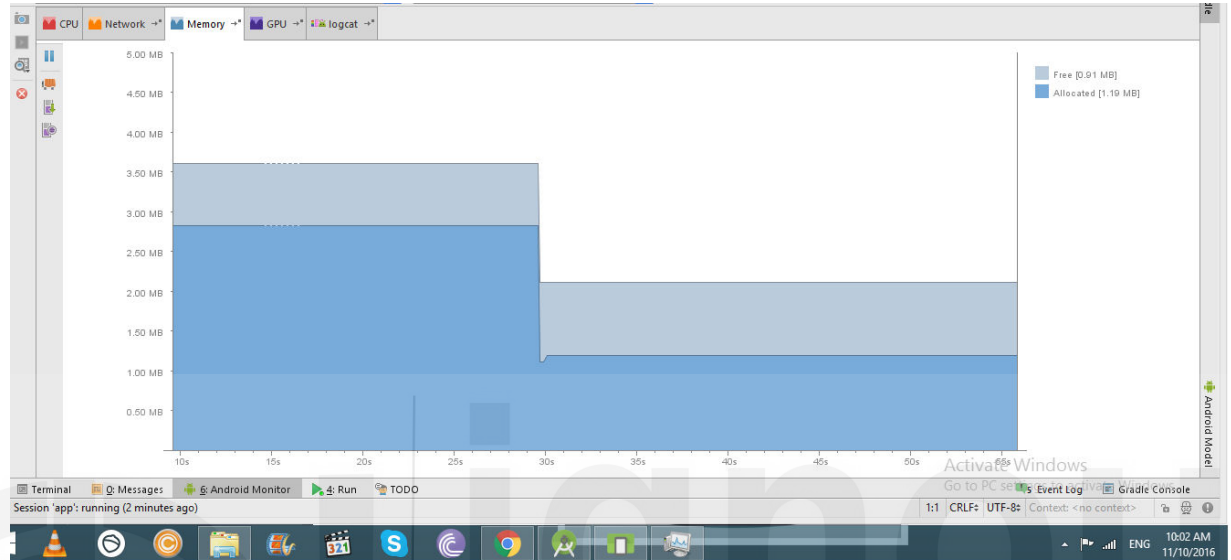


Figure 15.3 Android Monitor – Memory Footprint by running Weather app

Applications Memory Usage (kB):

Uptime: 269104 Realtime: 269104

**** MEMINFO in pid 4248 [com.example.android.sunshine.app] ****

Pss	Private	Private	Swapped	Heap	Heap			
		Total	Dirty	Clean	Dirty	Size	Alloc	Free
		-----	-----	-----	-----	-----	-----	-----
Native Heap		3361	3276	0	0	14336	13865	470
Dalvik Heap		1634	1524	0	0	3684	2920	764
Dalvik Other		341	288	0	0			
Stack		120	120	0	0			
Cursor		4	4	0	0			
Ashmem	2		0	0	0			
Other dev		4	0	4	0			
.so mmap		1109	300	0	0			
.apk mmap		168	0	8	0			
.ttf mmap		113	0	72	0			
.dex mmap		2112	4	2108	0			
.oat mmap		1057	0	152	0			
.art mmap		834	436	0	0			
Other mmap		38	8	4	0			
Unknown		169	168	0	0			
TOTAL		11066	6128	2348	0	18020	16785	1234

App Summary

Pss (KB)

Java Heap:	1960
Native Heap:	3276
Code:	2644
Stack:	120
Graphics:	0
Private Other:	476
System:	2590

TOTAL: 11066 TOTAL SWAP (KB): 0

CPU Monitor—As in Figure 15.4, CPU Monitor can be used to display CPU usage in real time and the percentage of total CPU time (including all cores) used in user and kernel mode.

Android App Performance

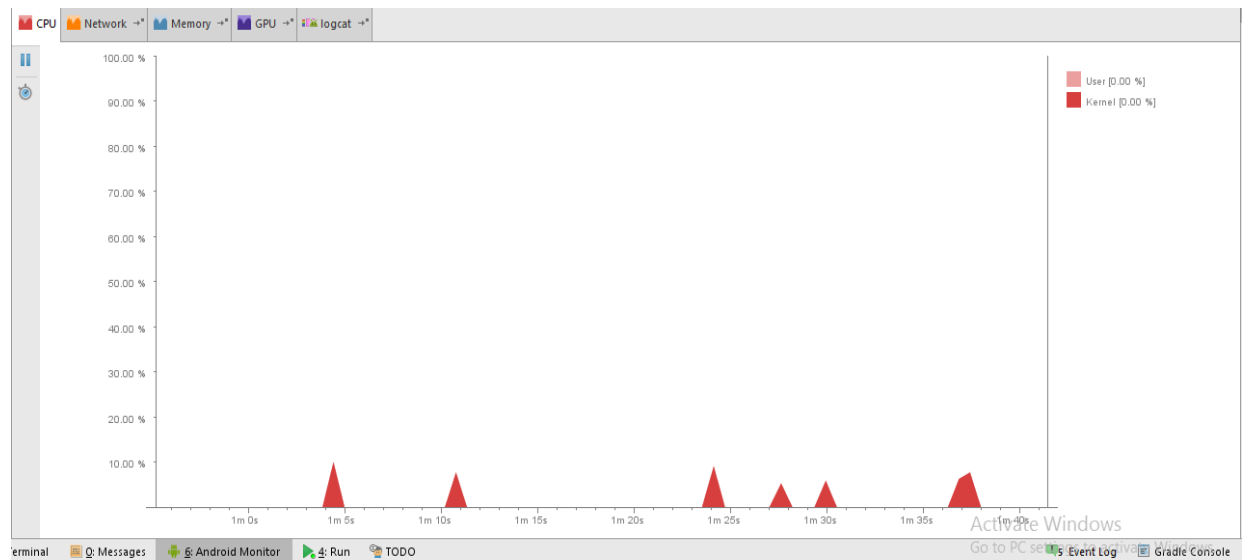


Figure 15.4 CPU Monitor

GPU Monitor – You can use the GPU Monitor as in figure 15.5 for a visual representation of how much time it takes to render the frames of a UI window. This information can be used to optimize the code that displays graphics and conserve memory.



Figure 15.5 GPU monitor

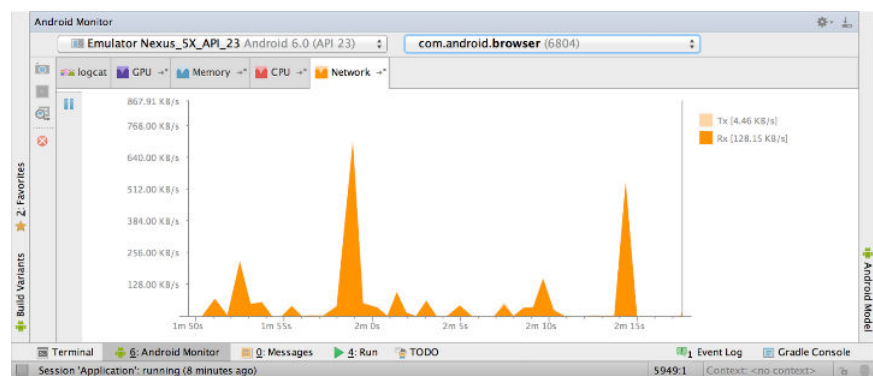


Figure 15.6 Android

15.3.2 Check Your Progress



What are the sub-tools provided by the Android Monitor to analyse the performance of an app?



Data Analysis

Android Monitor also lets you capture various types of data about your app while it is running and stores it in a file, which you can access later. It lists these files in the *Captures* window and you may observe how it works in the screen cast provided for this unit.

15.4 ANDROID MONITOR BASICS

In this section you will be guided how to use the Android Monitor step by step. The screen cast on Android Monitor Basics will demonstrate this activity.

Android Monitor is integrated into the Android Studio main window:

- To display Android Monitor, click  **Android Monitor**, which by default is at the bottom of the main window.
- To hide Android Monitor, click  **Android Monitor** again.
- Or select **View>Tool Windows>Android Monitor**.

Note: If you don't see the sidebar buttons, you can display them by selecting **View>Tool Buttons**.

A screen shot of Android Monitor is given below in Figure 15.7:

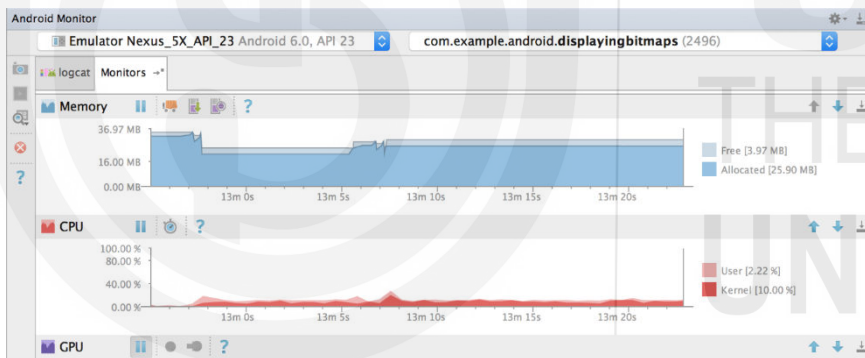


Figure 15.7 Android monitor

Before start using Android Monitor, you need to set up the environment, as well as the hardware device or emulator. All of the monitors require the following:

- If you want to run your app on a hardware device (as opposed to the emulator), connect it to the USB port. Make sure your development computer [detects your device](#), which often happens automatically when you connect it.
- Enable ADB integration by selecting **Tools>Android>Enable ADB Integration**. **Enable ADB Integration** should have a check mark next to it in the menu to indicate it's enabled.

All but the logcat Monitor have these additional requirements:

- [Enable USB debugging](#) in **Developer Options** on the device or emulator. In your app, set the debuggable property to true in the manifest or build.gradle file (it's initially set by default).

The GPU Monitor has this requirement as well:

- For Android 5.0 (API level 21) and Android 5.1 (API level 22), in **Developer Options** on the device or emulator, set **Profile GPU rendering** to '**In adb shell dumpsys gfxinfo**'.

The Network Monitor and the Video Capture tool work with a hardware device only, not with the emulator.

15.5 PROFILING A RUNNING APP IN ANDROID MONITOR

After you have met the prerequisites and connected a hardware device, you are ready to profile an app in Android Monitor. To start;

1. Open an app project and [run the app](#) on a device or emulator.
2. Display Android Monitor and click the tab for the monitor you want to view.
3. follow the instructions about using the monitor:
 - o [logcat Monitor](#)
 - o [Memory Monitor](#)
 - o [CPU Monitor](#)
 - o [GPU Monitor](#)
 - o [Network Monitor](#)

Switching between Devices and Apps

By default, Android Monitor displays data for your most recently run app. You can switch to another device and app as needed. In addition to currently running apps, you can view information about apps that are no longer running so you can continue to see any information about them that you gathered previously.

At the top of the Android Monitor main window, there are two menus listing devices and processes. To switch to another device, process, or both:

1. Select the device or emulator.

The Device menu lists the devices and emulators that are running or have run during your current unit. There are various status messages that can appear in the Device menu:

- o **DISCONNECTED** - You closed an emulator or unplugged a device from the computer.
- o **UNAUTHORIZED** - A device needs you to accept the incoming computer connection. For example, if the connected device displays an *Allow USB Debugging* dialog, click **OK** to allow the connection.
- o **OFFLINE** - Android Monitor can't communicate with a device, even though it has detected that device.

2. Select the process.

The Process menu lists the processes that are running or have run during your current unit. If a process is no longer running, the menu displays status of **DEAD**.

Terminating an App and removing it from a Device

To stop an app you have run from Android Studio, [select the device and the process](#) in the Android Monitor menus and click 'Terminate Application'. The process status changes to **DEAD** in the Processes menu. The emulator or device may continue to run, but the app closes. Any running monitors in Android Monitor would stop.

To remove an app from a device you use for development, use the normal uninstall procedure on the device.

If you run a new version of an app from Android Studio that's been already installed on a hardware device, the device displays an *Application Installation Failed* dialog. Click **OK** to install the new version of the app.

From the Android Monitor main window, you can also do the following:

- [Examine dumpsys system information.](#)
- [Take a screen capture of the device.](#)
- [Record a video from the screen.](#)

15.6 HOW ANDROID MANAGES MEMORY

Random-access memory (RAM) is a valuable resource in any software development environment, but it is even more valuable on a mobile operating system where physical memory is often constrained. Although Android's Dalvik virtual machine or Android Runtime (Introduced later, with KitKat) performs routine garbage collection, this does not allow you to ignore when and where your app allocates and releases memory. Techniques such as sharing memory, allocating and reclaiming app memory, restricting app memory and switching apps have been implemented in Android. In this course these techniques are not discussed in detail.

You may read the following pages found in Android Developer Forum <https://developer.android.com/training/articles/memory.html> to understand how Android manages app processes and memory allocation, and how you can proactively reduce memory usage while developing for Android.

You should consider RAM constraints throughout all phases of development, especially during app design. There are many ways you can design and write code that lead to more efficient results, through aggregation of the same techniques applied over and over.

Here are few techniques that you can apply while designing and implementing your app to make it more memory efficient.

Use services sparingly

If your app needs a [service](#) to perform work in the background, do not keep it running unless it is actively performing a job. Be careful to never leak your service by failing to stop it when its work is over.

Release memory when your user interface becomes hidden

When the user navigates to a different app and your user interface (UI) is no longer visible, you should release any resources that are used by only your UI. Releasing UI

resources at this time can significantly increase the system's capacity for cached processes, which has a direct impact on the quality of the user experience.

Release memory as memory becomes tight

During any stage of your app's lifecycle, the [`onTrimMemory\(\)`](#) callback also tells you when the overall device memory is getting low. You should respond by further releasing resources based on the memory levels delivered by [`onTrimMemory\(\)`](#).

Because the [`onTrimMemory\(\)`](#) callback was added in API level 14, you can use the [`onLowMemory\(\)`](#) callback as a fallback for older versions.

Check how much memory you should use

As mentioned earlier, each Android-powered device has a different amount of RAM available to the system and thus provides a different heap limit for each app. You can call [`getMemoryClass\(\)`](#) to get an estimate of your app's available heap in megabytes. If your app tries to allocate more memory than is available here, it will receive an [`OutOfMemoryError`](#).

Large heap size is not the same on all devices so that when running on devices that have limited RAM, the large heap size may be exactly the same as the regular heap size. So even if you request the large heap size, you should call [`getMemoryClass\(\)`](#) to check the regular heap size and strive to stay below that limit always.

Avoid wasting memory with bitmaps

When you load a bitmap, keep it in RAM only at the resolution you need for the current device's screen. Scale it down if the original bitmap is a higher resolution. You may keep in mind that an increase in bitmap resolution results in a corresponding in-memory needed, because both the X and Y dimension increase.

Use optimized data containers

Take advantage of optimized containers in the Android framework, such as [`SparseArray`](#), [`SparseBooleanArray`](#), and [`LongSparseArray`](#). The generic [`HashMap`](#) implementation can be quite memory inefficient because it needs a separate entry object for every mapping.

Be aware of memory overhead

Be knowledgeable about the cost and overhead of the language and libraries you are using, and keep this information in mind when you design your app, from start to finish. Often, things on the surface that look trivial may in fact have a large amount of overhead.

Be careful with code abstractions

Often, developers use abstractions simply as a "good programming practice," because abstractions can improve code flexibility and maintenance. However, generally they require a fair amount more code that needs to be executed, requiring more time and more RAM for that code to be mapped into memory. So you may avoid abstractions if they are not giving a significant benefit.

Be careful about using external libraries

External library code is often not written for mobile environments and can be inefficient when used for work on a mobile client. At the very least, when you decide

to use an external library, you should assume you are taking on a significant porting and maintenance burden to optimize the library for mobile. Plan for that work up-front and analyse the library in terms of code size and RAM footprint before deciding to use it at all.

Optimize overall performance

Some of the actions can be taken to optimize your app's performance in various ways to improve its responsiveness and battery efficiency are given below.

- Avoid creating unnecessary objects
- Prefer static over virtual
- Use static final for constants
- Avoid internal getters/setters
- Use enhanced for loop syntax
- Consider package instead of private access with private inner classes
- Avoid using floating-point
- Use native methods carefully
- measure your existing performance

Further details on implementing these techniques are given in the Android Developer Forum in <https://developer.android.com/training/best-performance.html>

Use zipalign on your final Android Application Package (APK)

Android Application Package (APK) is the packaging format used by Android operating system in distribution and installation of Android apps. *Zipalign* is a tool that optimizes the way an application is packaged.

If you do any post-processing of an APK generated by a build system, then you must run *zipalign* on it to have it re-aligned. Failing to do so can cause your app to require significantly more RAM, because things like resources can no longer be mapped from the APK.

Use multiple processes

If it is appropriate for your app, an advanced technique that may help you manage your app's memory is dividing components of your app into multiple processes. This technique must always be used carefully and **most apps should not run multiple processes**, as it can easily increase—rather than decrease—your RAM footprint if done incorrectly. It is primarily useful to apps that may run significant work in the background as well as the foreground and can manage those operations separately.

15.6.1 Check Your Progress



List down what techniques you used to optimize memory in the app you developed. Briefly describe them in the on-line discussion forum under *managing app memory*. You should give feedback to what your peers have written and also invite them to give feedback to you.

15.7 BATTERY ANALYSIS

The battery-life impact of performing application updates depends on the battery level and charging state of the device. The impact of performing updates while the device is

charging over AC is negligible, so in most cases you can maximize your refresh rate whenever the device is connected to a wall charger. Conversely, if the device is discharging, reducing your update rate helps prolong the battery life.

Similarly, you can check the battery charge level, potentially reducing the frequency of or even stopping the updates when the battery charge is nearly exhausted.

Battery Historian Walkthrough

This walkthrough shows the basic usage and workflow for the Batterystats tool and the Battmery Historian script.

Batterystats collects battery data from your device, and Battery Historian converts that data into an HTML visualization that you can view in your Browser. Batterystats is part of the Android framework, and Battery Historian script is open-sourced and available on GitHub at <https://github.com/google/battery-historian>.

It is good for:

- Showing you where and how processes are drawing current from the battery and
- Identifying tasks in your app that could be deferred or even removed to improve battery life.

15.7.1 Video – V14: Battery Analysis



This video shows you how to analyse battery power and generate a report.



URL: <https://tinyurl.com/ya2lsmko>

Battery Historian Charts

The Battery Historian chart graphically illustrate power-relevant events over time.

Each row shows a colored bar segment when a system component is active and thus drawing current from the battery. The chart does *not* show *how much* battery was used by the component, only that the app was active.

Charts are organized by category.

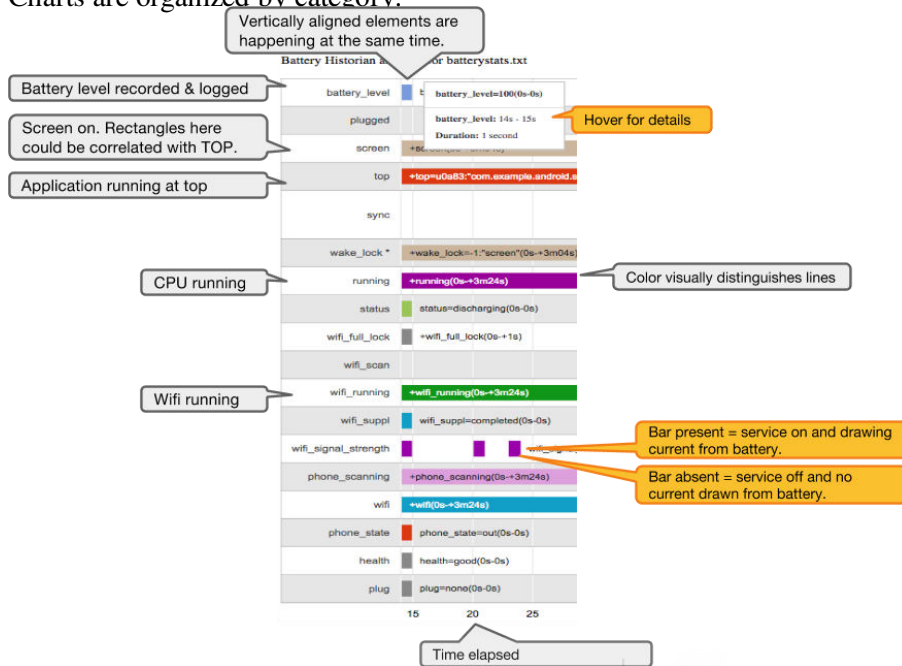


Figure 6 Battery Historian Chart

Battery usage categories

Given below is a list of battery usage categories.

- **battery_level:** When the battery level was recorded and logged. Reported in percent, where 093 is 93%. Provides an overall measure of how fast the battery is draining.
- **top:** The application running at the top; usually, this is the application that is visible to the user. If you want to measure battery drain while your app is active, make sure it is the top app. If you want to measure battery drain while your app is in the background, make sure it's *not* the top app.
- **wifi_running:** Shows that the Wi-Fi network connection was active.
- **screen:** Screen is turned on.
- **phone_in_call:** Recorded when the phone is in a call.
- **wake_lock:** App wakes up, grabs a lock, does small work, then goes back to sleep. This is one of the most important pieces of information. Waking up the phone is expensive, so if you see lots of short bars here, that might be a problem.
- **running:** Shows when the CPU is awake. Check whether it is awake and asleep when you expect it to be.
- **wake_reason:** The last thing that caused the kernel to wake up. If it's your app, determine whether it was necessary.
- **mobile_radio:** Shows when the radio was on. Starting the radio is battery expensive. Many narrow bars close to each other can indicate opportunities for batching and other optimizations.
- **gps:** Indicates when the GPS was on. Make sure this is what you expect.
- **sync:** Shows when an app was syncing with a backend. The sync bar also shows which app did the syncing. For users, this can show apps where they might turn syncing off to save battery. Developers should sync as little as possible and only as often as necessary.

Working with Batterystats and Battery Historian

You should watch the screen cast and do the following steps to learn how to use Batterystats and Battery Historian tools.

For more details you may go to following links in Android Developer Forum.

- <https://developer.android.com/studio/profile/battery-historian.html>
- <https://developer.android.com/training/.../battery-monitoring.html>

15.8 OPTIMIZING BATTERY LIFE

For your app to be ‘good’, it should seek to limit its impact on the battery life of its device. After studying this section, you will be able to build apps that modify their functionality and behavior based on the state of its device.

By taking steps such as batching network requests, disabling background service updates when you lose connectivity, or reducing the rate of such updates when the battery level is low, you can ensure that the impact of your app on battery life is minimized, without compromising the user experience.

Here we briefly describe the steps you can take to optimize battery life. You should go to Android Developer Studio for more details.

(<https://developer.android.com/training/monitoring-device-state/index.html>) What is described in detail in this page are summarized below.

1. [Reducing Network Battery Drain](#)

Requests that your app makes to the network are a major cause of battery drain because they turn on the power-hungry mobile or Wi-Fi radios. Beyond the power needed to send and receive packets, these radios spend extra power just turning on and keeping awake. Something as simple as a network request every 15 seconds can keep the mobile radio on continuously and quickly use up battery power.

2. [Optimizing for Doze and App Standby](#)

Starting from Android 6.0 (API level 23), Android introduces two power-saving features that extend battery life for users by managing how apps behave when a device is not connected to a power source. *Doze* reduces battery consumption by deferring background CPU and network activity for apps when the device is unused for long periods of time. *App Standby* defers background network activity for apps with which the user has not recently interacted.

3. [Monitoring the Battery Level and Charging State](#)

When you are altering the frequency of your background updates to reduce the effect of those updates on battery life, check the current battery level. You can stop your updates when the battery charge is nearly exhausted.

4. [Determining and Monitoring the Docking State and Type](#)

Android devices can be docked into several different kinds of docks. These include car or home docks and digital versus analog docks. The dock-state is typically closely linked to the charging state as many docks provide power to docked devices.

5. [Determining and Monitoring the Connectivity Status](#)

Some of the most common uses for repeating alarms and background services is to schedule regular updates of application data from Internet resources, cache data, or execute long running downloads. But if you aren't connected to the Internet, or the connection is too slow to complete a download there is no use waking the device to schedule the update.

You can use the [ConnectivityManager](#) to check the Internet connectivity.

6. [Manipulating Broadcast Receivers On Demand](#)

The simplest way to monitor device state changes is to create a [BroadcastReceiver](#) for each state you are monitoring and register each of them in your application manifest. Then within each of these receivers you simply reschedule your recurring alarms based on the current device state.

A side-effect of this approach is that your app will wake the device each time any of these receivers is triggered—potentially much more frequently than required.

15.8.1 Check Your Progress



List down names of the techniques listed above that can be used in the app you developed.

How can you change your app to optimize the battery life?

15.9 SUMMARY



There are five main performance monitors provided by the tool, Android Monitor - and they can make you visualize the behavior and performance of your app. There are many ways that you can improve the performance of your application but optimizing memory and battery life are very important. Several techniques for optimizing memory and battery life were discussed in this unit.

15.10 FURTHER READINGS

- <https://developer.android.com/studio/profile>
- <https://developer.android.com/games/sdk/performance-tuner/unity/run-monitor-app>
- <https://developer.android.com/topic/performance/memory-overview>



UNIT 16 SECURITY ISSUES

- 16.0 Introduction
 - 16.1 Objectives
 - 16.2 Security Concerns of an Android Application
 - 16.3 Security Provided by the OS
 - 16.3.1 Check Your Progress
 - 16.4 Information Leakage
 - 16.4.1 Check Your Progress
 - 16.5 Device management policies
 - 16.5.1 Check Your Progress
 - 16.6 Summary
 - 16.7 Further readings
-

16.0 INTRODUCTION

Android is considered to be the most widely used mobile operating system in the world today. Being Open Source it is very much vulnerable for security breaches if the security is not managed properly.

In this unit, first you will learn about security provided by the operating system and how to analyse an application to understand what security features are built into it. Then you will study device management policies, which will enable you to design and develop applications for devices that enforce security policies.

16.1 OBJECTIVES

After studying this unit, you should be able to :



Outcomes

- identify possible security concerns of an Android application
- identify the device management policies implemented in different setups
- identify how security can be enhanced by using device management policies
- design and develop applications that enforces security policies on devices



Terminology

malicious app: software that brings harm to the mobile device

cryptography: coding or decoding messages to keep them secure

16.2 SECURITY CONCERNS OF AN ANDROID APPLICATION

There are many security concerns regarding Android applications. One major security threat is over the limit access permission given and requested by apps. When an app is downloaded from Google Play, users ignore the extent of permission this app should have on their devices. Very often, app developers also do not have a clear understanding as to what permissions a mobile application actually needs and request overzealous and irrelevant permission. In any operating system, this kind of situations expose the user to a source of potential risks.

However, risks increase when users download apps from unidentified sources to avoid paying the fee. Anyone can create a malicious app and upload it on the Internet. This can result in downloading a malicious app or one that has been modified to automatically install a virus on Android devices.

Fact that Android is Open Source makes it more vulnerable for malware and malicious software attacks. However, being Open Source there is a large community of experts reviewing it and developing patches. Anyway, users are being hacked without them knowing that they are hacked. Another major security threat faced by Android platform due to the option of customizing the operating system. Device manufacturers can modify the OS to make it function optimally on their device. Moreover, users also can modify the OS, integrating customization layers or launchers. These practices leads to more security breaches.

Another major issue with Android is fragmentation. It means that there exist multiple versions of Android, even on latest devices. Since some devices are never updated to the latest version they will not have the latest security updates. It is also difficult to take appropriate security measures or educate the users about potential vulnerabilities because user experience on each device is different.

16.3 SECURITY PROVIDED BY THE OS

Android has many security features built into the operating system that significantly reduce the frequency and impact of application security issues we already discussed. Latest versions of the operating system is designed in a way that you can build your apps with default system and file permissions without worrying too much about security.

Some of the core security features provided by Android OS are:

- Android Application Sandbox that isolates your app data and code execution from other apps by running each application other than system apps with a different user.
- Permission reflects Linux permissions groups and corresponding user related to app will be assign to particular user group to assign permissions. So, access restrictions guaranteed even in the kernel level.
- An application framework which provides common security functionality such as cryptography, permissions, and secure inter process communication
- Techniques to mitigate risks associated with common memory management errors
- An encrypted file system that can be enabled to protect data on lost or stolen devices
- User-granted permissions to restrict access to system features and user data
- Application-defined permissions to control application data on a per-app basis.

- Android 6.0 and higher versions ask from user to allow or deny individual permissions for application dynamically when needed.

However, it is important for you to be familiar with the Android security best practices as given in <https://developer.android.com/training/best-security.html>. This web page describe best practices to be followed when storing data, using permissions, using networks, validating inputs, handling user data, using web view, using cryptography, using inter process communication, and dynamically loading code. Following these practices will reduce many security issues that may adversely affect the users.

16.3.1 Check Your Progress



Briefly describe different methods recommended as Android best practices to store data.

Security in a virtual machine

Unlike in many other Virtual Machine environments the Dalvik VM in Android does not provide a security boundary. The application sandbox is implemented at the OS level, so Dalvik can interoperate with native code in the same application without any security constraints.

As there is limited storage on mobile devices, it is common for developers to build modular applications and use dynamic class loading. When doing this, you must consider both the source where you retrieve your application logic and where you store it locally. Dynamic class loading from sources that are not verified should not be done as that code might be modified to include malicious behavior.

Security in native code

It is recommended to use the Android SDK for application development, rather than using native development kit. Applications built with native code are more complex, less portable, and more like to include common memory-corruption errors such as buffer overflows.

16.4 INFORMATION LEAKAGE

It is widely known that many mobile applications share data with third parties without the knowledge of users. There are many reports and research papers about in-app advertising that leak potentially sensitive personal information on millions of mobile phone users. These data may include how much money users make, whether or not they have kids, and what their political affiliations are.

Following is a list of situations how information leakage in mobile apps take place.

- Mobile app developers choose to accept in-app advertisements inside their app
- Advertisement networks pay a fee to app developers in order to show advertisements and monitor user activity. User data could be device models, geolocations, etc. This information is provided to advertisers select where to place ads

- Advertisers instruct ad networks to show their ads based on topic targeting (such as “movies”), interest targeting (such as usage patterns and previous click throughs), and demographic targeting (such as estimated age range)
- The ad network displays ads to appropriate mobile app users and receives payment from advertisers for successful views or click throughs by the recipients
- In-app ads are displayed unencrypted as part of the app’s GUI. So that mobile app developers can access the targeted ad content delivered to its own app users and then reverse-engineer that data to construct a profile of their app customer

In one research [1], to test what is being leaked, researchers had created a custom-built Android app that they installed on more than 200 participants’ phones. From that they have been able to review the accuracy of personalized advertisements served to test subjects from the Google mobile ad network, AdMob, based on users’ personal interests and demographic profiles.

Many researchers [1] [2] have found that the root cause of the privacy leakage is the lack of isolation between the advertisements and mobile apps. It is reported that adopting HTTPS would not do anything to protect the advertisement traffic.

16.4.1 Check Your Progress



This activity is to be done in LMS for this course in the given discussion forum.

Briefly describe main findings of a research paper or a report on information leakage from mobile devices. Please give the reference and access date and time with URL, if it was accessed on-line.

16.5 DEVICE MANAGEMENT POLICIES

Device management policies control features in mobile devices and computers. To use them, first you have to define the type of policy to support at the functional level. Policies may cover screen-lock password strength, expiration timeout, encryption, etc. Any operating system would have its own device management policies. For mobile devices, these policies can be created by using templates that contain recommended or custom settings, and then deploying them to devices. Since Android 2.2 (API level 8), the Android platform offers system-level device management capabilities through the Device Administration APIs so that the application can be configured to ensure a strong screen-lock password before displaying restricted content to the user.

Device management policies implemented in different set-ups

‘Android for Work’ is a program for supporting enterprise use of Android. You can develop apps for Android for Work to take advantages of built in security and management features in Android. Enterprise mobility management (EMM) providers and enterprise application developers can accessed it via APIs.

Apps built on Android for Work include data security, app security and device security as explained below.

- **Data security**—Business data is separated in a work profile and protected device-wide on work-managed devices. So data leakage prevention policies can easily be applied.
- **Apps security**—Work apps are deployed through ‘Google Play for Work’ preventing installation of apps from unknown sources and apply app configurations.
- **Device security**— ‘Android for Work’ devices are protected with disk encryption, lockscreen, remote attestation services, and hardware-backed key store.

Using ‘Android for Work’, organizations can choose what devices, APIs, and framework they want to use to develop apps which may enable;

- Building apps to help employees be more productive in scenarios such as Bring Your Own Device (BYOD), Corporate Owned Personally Enabled (COPE) devices, and Corporate-Owned, Single-Use (COSU) devices.
- Connecting with leading enterprise mobility management (EMM) partners to help integrate Android in your business.

You may use [Android for Work Developer Guide, https://developer.android.com/work/guide.html](https://developer.android.com/work/guide.html) to learn to create apps that best utilize features in Android.

Moreover, ‘Android for Work’ offers a partner program for developers through the [Android for Work DevHub \(https://enterprise.google.com/android/developers/applyDevHub/\)](https://enterprise.google.com/android/developers/applyDevHub/). It provides exclusive access to beta features and developer events, along with access to a community of Android developers making enterprise apps.

Designing and developing applications that enforces security policies on devices

In this section, you will learn how to create a security-aware application that manages access to its content by enforcing device management policies. To do that we will discuss how to,

1. [define and declare your policy](#)
2. [create a device administration receiver](#)
3. [activate the device administrator](#)
4. [implement the device policy controller](#)

Define and declare your policy

First, you need to define the kinds of policy to support at the functional level such as screen-lock password strength, expiration timeout, encryption, etc.

Then, you must declare the selected policy set, which will be enforced by the application, in the res/xml/device_admin.xml file. The Android manifest should also reference the declared policy set.

Each declared policy corresponds to some number of related device policy methods in [DevicePolicyManager](#) (e.g. defining minimum password length and minimum number of uppercase characters). If an application attempts to invoke methods whose corresponding policy is not declared in the XML, this will result in a [SecurityException](#) at runtime.

Create a device administration receiver

You must create a Device Administration broadcast receiver, which gets notification of events related to the policies you have declared to support. An application can selectively override callback methods.

Activate the device administrator

Before enforcing any policies, the user needs to manually activate the application as a device administrator. It is a good practice to include the explanatory text to highlight to users why the application is requesting to be a device administrator. It can be done by specifying the [EXTRA_ADD_EXPLANATION](#) extra in the intent.

Implement the device policy controller

After the device administrator is activated successfully, the application then configures Device Policy Manager with the requested policy. However, new policies are being added to Android with each release. It is appropriate to perform version checks in your application if using new policies while supporting older versions of the platform.

More details and necessary code snippets can be found at <https://developer.android.com/work/device-management-policy.html>.

16.5.1 Check Your Progress



What is the importance of Network security configuration?

16.6 SUMMARY



In this unit we discussed importance of security in Android apps, the security provided by the operating system, problem of information leakage and how to develop secure apps by adhering to the security policies.

16.7 FURTHER READINGS

- <https://developer.android.com/training/articles/security-tips?authuser=1>
- [https://developers.google.com/android/management/create-policy#:~:text=policies%20\(also%20called%20a%20policy,policies.](https://developers.google.com/android/management/create-policy#:~:text=policies%20(also%20called%20a%20policy,policies.)
- <https://support.google.com/a/users/answer/9453213?hl=en>
- <https://developer.android.com/work/device-management-policy>