Handout#1

Problem Statement: Introduction to Mobile Application Development.

Introduction

Mobile Application Development

Mobile application development is a term used to denote the act or process by which application software is developed for mobile devices, such as personal digital assistants, enterprise digital assistants or mobile phones.

Mobility Panorama:

It is classified into

(1) Logical Landscape

(2) Physical Landscape

Logical Landscape:

It describes the rationale behind mobility for different stakeholders.

Consumer Mobility:

- Focused toward the end user.
- Comprise mobility solutions such as social networking, games, shopping, bidding & utilities.

Enterprise Mobility:

- Focused toward various stakeholders of an organization such as vendors, partners, suppliers, workforce and their end consumers.
- Mobility is enabling enterprises to increase productivity of their workforce.

Physical Landscape:

It portrays the infrastructure that enables mobility.

Mobile Components:

- Key mobility components are mobile devices, mobile platforms, and mobile app stores.
- Mobile devices are the centre piece of mobility, and available in different shapes and sizes such as smart phones, tablets, phablets and smart watches.
- Mobile platforms such as Android and Applie iOS, are software stacks that power mobile devices.
- Mobile app stores are online market places of mobile apps. Ex. Google Play, App Store

Enterprise Components:

- Comprises hosts of servers, such as database servers and application servers that cater to enterprise portion of mobility solutions.
- Also comprise enterprise solutions that cater to the requirements of
  - data security
  - data synchronization between mobile devices and enterprise servers and identity management

Middleware and Protocols:

- It acts as glue between mobility and enterprise components.
- Access mechanisms such as Wi-Fi, Bluetooth, Code Division Multiple Access (CDMA), General Packet Radio Service (GPRS), and GSM are some key components of this layer that allow mobile devices to communicate.
- Other key components are gateways such as WAP and SMS gateways that enable interaction between mobile devices and the Internet.

Mobile Platforms:

- Mobile OSs known as mobile platforms
- It is not just an OS but a software stack that typically comprises an OS, libraries and application development framework(s).
- The OS contributes to the core features of the platform such as

  - Memory management
  - Process management and
  - Various device drivers
- Libraries furnish the most required core functionality of the platform such as media libraries etc.
- The application development framework is the set of application programming interfaces (APIs) that in turn interact with the underlying libraries and are exposed to the developers for app development.
- Most popular mobile platforms are:
  - Android
  - Apple iOS
  - Black Berry
  - Windows phone

APP Development Approaches:

Three broad approaches:

1. Native
2. Web
3. Hybrid

Figure: Mobile App Development Approaches

(1) Native Approach:

- native app,
- developed for native platform using platform specific APIs
- distributed through online app stores
- preferred when app requires a native look and feel

(2) Web Approach:

- mobile web app,
- developed using web technologies such as HTML5, CSS3, and JavaScript
- not installed on mobile device, gets rendered in an mobile browser, over the network

- preferred when app requires to cater to diverse devices using a single codebase
- no native L&F and no high-end device capabilities

(3) Hybrid Approach:

- mixed approach (features of native + web approaches)
- developed using mobile cross platforms
- hybrid platforms do not power device
- facilitate multiplatform development
- same codebase of a mobile app translated to fit into any of the supported native platforms

[**Reference:** Book -"Mobile Apps Development" by Anubhav Pradhan, Anil V Deshpande]

Exercise:

1. List some of the mobile apps that you have used or familiar with & how mobile apps have changed the way you use your mobile phone.
2. What questions are raised in your mind about mobile app development?
3. What encompasses Mobility?
4. How mobile apps are similar or dissimilar when compared to enterprise mobility?
5. What do mean by Enterprise Mobility?
6. What is the basic difference between a native app, web app and hybrid app and across platform apps? Also highlight the advantages and disadvantages of one over the other.

CHAPTER

1

Overview

What is Android?

Android is an open source and Linux-based **Operating System** for mobile devices such as smartphones and tablet computers. Android was developed by the *Open Handset Alliance*, led by Google, and other companies.

Android offers a unified approach to application development for mobile devices which means developers need only develop for Android, and their applications should be able to run on different devices powered by Android.

The first beta version of the Android Software Development Kit (SDK) was released by Google in 2007 where as the first commercial version, Android 1.0, was released in September 2008.

On June 27, 2012, at the Google I/O conference, Google announced the next Android version, 4.1 **Jelly Bean**. Jelly Bean is an incremental update, with the primary aim of improving the user interface, both in terms of functionality and performance.

The source code for Android is available under free and open source software licenses. Google publishes most of the code under the Apache License version 2.0 and the rest, Linux kernel changes, under the GNU General Public License version 2.

Features of Android

Android is a powerful operating system competing with Apple 4GS and supports great features. Few of them are listed below:

Feature	Description
Beautiful UI	Android OS basic screen provides a beautiful and intuitive user interface.
Connectivity	GSM/EDGE, IDEN, CDMA, EV-DO, UMTS, Bluetooth, Wi-Fi, LTE, NFC and WiMAX.
Storage	SQLite, a lightweight relational database, is used for data storage purposes.
Media support	H.263, H.264, MPEG-4 SP, AMR, AMR-WB, AAC, HE-AAC, AAC 5.1, MP3, MIDI, Ogg Vorbis, WAV, JPEG, PNG, GIF, and BMP
Messaging	SMS and MMS

Web browser	Based on the open-source WebKit layout engine, coupled with Chrome's V8 JavaScript engine supporting HTML5 and CSS3.
Multi-touch	Android has native support for multi-touch which was initially made available in handsets such as the HTC Hero.
Multi-tasking	User can jump from one task to another and same time various application can run simultaneously.
Resizable widgets	Widgets are resizable, so users can expand them to show more content or shrink them to save space
Multi-Language	Supports single direction and bi-directional text.
GCM	Google Cloud Messaging (GCM) is a service that lets developers send short message data to their users on Android devices, without needing a proprietary sync solution.
Wi-Fi Direct	A technology that lets apps discover and pair directly, over a high-bandwidth peer-to-peer connection.
Android Beam	A popular NFC-based technology that lets users instantly share, just by touching two NFC-enabled phones together.

Android Applications

Android applications are usually developed in the Java language using the Android Software Development Kit.

Once developed, Android applications can be packaged easily and sold out either through a store such as **Google Play** or the **Amazon Appstore**.

Android powers hundreds of millions of mobile devices in more than 190 countries around the world. It's the largest installed base of any mobile platform and growing fast. Every day more than 1 million new Android devices are activated worldwide.

This tutorial has been written with an aim to teach you how to develop and package Android application. We will start from environment setup for Android application programming and then drill down to look into various aspects of Android applications.

CHAPTER

2

Environment Setup

You will be glad to know that you can start your Android application development on either of the following operating systems:

- Microsoft Windows XP or later version.
- Mac OS X 10.5.8 or later version with Intel chip.
- Linux including GNU C Library 2.7 or later.

Second point is that all the required tools to develop Android applications are freely available and can be downloaded from the Web. Following is the list of software's you will need before you start your Android application programming.

- Java JDK5 or JDK6
- Android SDK
- Eclipse IDE for Java Developers (optional)
- Android Development Tools (ADT) Eclipse Plugin (optional)

Here last two components are optional and if you are working on Windows machine then these components make your life easy while doing Java based application development. So let us have a look how to proceed to set required environment.

Step 1 - Setup Java Development Kit (JDK)

You can download the latest version of Java JDK from Oracle's Java site: Java SE Downloads. You will find instructions for installing JDK in downloaded files, follow the given instructions to install and configure the setup. Finally set PATH and JAVA_HOME environment variables to refer to the directory that contains **java** and **javac**, typically java_install_dir/bin and java_install_dir respectively.

If you are running Windows and installed the JDK in C:\jdk1.6.0_15, you would have to put the following line in your C:\autoexec.bat file.

```
set PATH=C:\jdk1.6.0_15\bin;%PATH%
set JAVA_HOME=C:\jdk1.6.0_15
```

Alternatively, you could also right-click on *My Computer*, select *Properties*, then *Advanced*, then*Environment Variables*. Then, you would update the PATH value and press the OK button.

On Linux, if the SDK is installed in /usr/local/jdk1.6.0_15 and you use the C shell, you would put the following code into your **.cshrc** file.

```
setenv PATH /usr/local/jdk1.6.0_15/bin:$PATH
setenv JAVA_HOME /usr/local/jdk1.6.0_15
```

Alternatively, if you use an Integrated Development Environment (IDE) Eclipse, then it will know automatically where you have installed your Java.

Step 2 - Setup Android SDK

You can download the latest version of Android SDK from Android official website : Android SDK Downloads. If you are installing SDK on Windows machine, then you will find a *installer_rXX-windows.exe*, so just download and run this exe which will launch *Android SDK Tool Setup* wizard to guide you throughout of the installation, so just follow the instructions carefully. Finally you will have*Android SDK Tools* installed on your machine.
If you are installing SDK either on Mac OS or Linux, check the instructions provided along with the downloaded *android-sdk_rXX-macosx.zip* file for Mac OS and *android-sdk_rXX-linux.tgz* file for Linux. This tutorial will consider that you are going to setup your environment on Windows machine having Windows 7 operating system.

So let's launch *Android SDK Manager* using the option **All Programs > Android SDK Tools > SDK Manager**, this will give you following window:

Once you launched SDK manager, its time to install other required packages. By default it will list down total 7 packages to be installed, but I will suggest to de-select *Documentation for Android SDK* and*Samples for*

SDK packages to reduce installation time. Next click **Install 7 Packages** button to proceed, which will display following dialogue box:

If you agree to install all the packages, select **Accept All** radio button and proceed by clicking **Install**button. Now let SDK manager do its work and you go, pick up a cup of coffee and wait until all the packages are installed. It may take some time depending on your internet connection. Once all the packages are installed, you can close SDK manager using top-right cross button.

Step 3 - Setup Eclipse IDE

All the examples in this tutorial have been written using Eclipse IDE. So I would suggest you should have latest version of Eclipse installed on your machine.

To install Eclipse IDE, download the latest Eclipse binaries from http://www.eclipse.org/downloads/. Once you downloaded the installation, unpack the binary distribution into a convenient location. For example in C:\eclipse on windows, or /usr/local/eclipse on Linux and finally set PATH variable appropriately.

Eclipse can be started by executing the following commands on windows machine, or you can simply double click on eclipse.exe

```
%C:\eclipse\eclipse.exe
```

Eclipse can be started by executing the following commands on Linux machine:

```
$/usr/local/eclipse/eclipse
```

After a successful startup, if everything is fine then it should display following result:

Step 4 - Setup Android Development Tools (ADT) Plugin

This step will help you in setting Android Development Tool plugin for Eclipse. Let's start with launching Eclipse and then, choose **Help > Software Updates > Install New Software**. This will display the following dialogue box.

Now use **Add** button to add *ADT Plugin* as name and *https://dl-ssl.google.com/android/eclipse/* as the location. Then click OK to add this location, as soon as you will click OK button to add this location, Eclipse starts searching for the plug-in available the given location and finally lists down the found plugins.

Now select all the listed plug-ins using **Select All** button and click **Next** button which will guide you ahead to install Android Development Tools and other required plugins.

Step 5 - Create Android Virtual Device

To test your Android applications you will need a virtual Android device. So before we start writing our code, let us create an Android virtual device. Launch Android AVD Manager using Eclipse menu options**Window > AVD Manager>** which will launch Android AVD Manager. Use **New** button to create a new Android Virtual Device and enter the following information, before clicking **Create AVD** button.

If your AVD is created successfully it means your environment is ready for Android application development. If you like, you can close this window using top-right cross button. Better you re-start your machine and once you are done with this last step, you are ready to proceed for your first Android example but before that we will see few more important concepts related to Android Application Development.

CHAPTER

3

Architecture

Android operating system is a stack of software components which is roughly divided into five sections and four main layers as shown below in the architecture diagram.

Linux kernel

At the bottom of the layers is Linux - Linux 2.6 with approximately 115 patches. This provides basic system functionality like process management, memory management, device management like camera, keypad, display etc. Also, the kernel handles all the things that Linux is really good at such as networking and a vast array of device drivers, which take the pain out of interfacing to peripheral hardware.

Libraries

On top of Linux kernel there is a set of libraries including open-source Web browser engine WebKit, well known library libc, SQLite database which is a useful repository for storage and sharing of application data, libraries to play and record audio and video, SSL libraries responsible for Internet security etc.

Android Runtime

This is the third section of the architecture and available on the second layer from the bottom. This section provides a key component called **Dalvik Virtual Machine** which is a kind of Java Virtual Machine specially designed and optimized for Android.

The Dalvik VM makes use of Linux core features like memory management and multi-threading, which is intrinsic in the Java language. The Dalvik VM enables every Android application to run in its own process, with its own instance of the Dalvik virtual machine.

The Android runtime also provides a set of core libraries which enable Android application developers to write Android applications using standard Java programming language.

Application Framework

The Application Framework layer provides many higher-level services to applications in the form of Java classes. Application developers are allowed to make use of these services in their applications.

Applications

You will find all the Android application at the top layer. You will write your application to be installed on this layer only. Examples of such applications are Contacts Books, Browser, Games etc.

CHAPTER

4

Application Components

Application components are the essential building blocks of an Android application. These components are loosely coupled by the application manifest file *AndroidManifest.xml* that describes each component of the application and how they interact.

There are following four main components that can be used within an Android application:

Components	Description
Activities	They they dictate the UI and handle the user interaction to the smartphone screen
Services	They handle background processing associated with an application.
Broadcast Receivers	They handle communication between Android OS and applications.
Content Providers	They handle data and database management issues.

Activities

An activity represents a single screen with a user interface. For example, an email application might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. If an application has more than one activity, then one of them should be marked as the activity that is presented when the application is launched.

An activity is implemented as a subclass of **Activity** class as follows:

```
public class MainActivity extends Activity {

}
```

Services

A service is a component that runs in the background to perform long-running operations. For example, a service might play music in the background while the user is in a different application, or it might fetch data over the network without blocking user interaction with an activity.

A service is implemented as a subclass of **Service** class as follows:

```
public class MyService extends Service {
```

```
}
```

Broadcast Receivers

Broadcast Receivers simply respond to broadcast messages from other applications or from the system. For example, applications can also initiate broadcasts to let other applications know that some data has been downloaded to the device and is available for them to use, so this is broadcast receiver who will intercept this communication and will initiate appropriate action.

A broadcast receiver is implemented as a subclass of **BroadcastReceiver** class and each message is broadcasted as an **Intent** object.

```
public class MyReceiver  extends  BroadcastReceiver {

}
```

Content Providers

A content provider component supplies data from one application to others on request. Such requests are handled by the methods of the *ContentResolver* class. The data may be stored in the file system, the database or somewhere else entirely.

A content provider is implemented as a subclass of **ContentProvider** class and must implement a standard set of APIs that enable other applications to perform transactions.

```
public class MyContentProvider extends  ContentProvider {

}
```

We will go through these tags in detail while covering application components in individual chapters.

Additional Components

There are additional components which will be used in the construction of above mentioned entities, their logic, and wiring between them. These components are:

Components	Description
Fragments	Represents a behavior or a portion of user interface in an Activity.
Views	UI elements that are drawn onscreen including buttons, lists forms etc.
Layouts	View hierarchies that control screen format and appearance of the views.
Intents	Messages wiring components together.
Resources	External elements, such as strings, constants and drawables pictures.
Manifest	Configuration file for the application.

CHAPTER

5

Hello World Example

Let us start actual programming with Android Framework. Before you start writing your first example using Android SDK, you have to make sure that you have setup your Android development environment properly as explained in Android - Environment Setup tutorial. I also assume that you have a little bit working knowledge with Eclipse IDE.

So let us proceed to write a simple Android Application which will print "Hello World!".

Create Android Application

The first step is to create a simple Android Application using Eclipse IDE. Follow the option **File -> New -> Project** and finally select **Android New Application** wizard from the wizard list. Now name your application as **HelloWorld** using the wizard window as follows:

Next, follow the instructions provided and keep all other entries as default till the final step. Once your project is created successfully, you will have following project screen:

Anatomy of Android Application

Before you run your app, you should be aware of a few directories and files in the Android project:

S.N.	Folder, File & Description
1	**src** This contains the **.java** source files for your project. By default, it includes an *MainActivity.java*source file having an activity class that runs when your app is launched using the app icon.
2	**gen** This contains the **.R** file, a compiler-generated file that references all the resources found in your project. You should not modify this file.
3	**bin** This folder contains the Android package files **.apk** built by the ADT during the build process and everything

	else needed to run an Android application.
4	**res/drawable-hdpi** This is a directory for drawable objects that are designed for high-density screens.
5	**res/layout** This is a directory for files that define your app's user interface.
6	**res/values** This is a directory for other various XML files that contain a collection of resources, such as strings and colors definitions.
7	**AndroidManifest.xml** This is the manifest file which describes the fundamental characteristics of the app and defines each of its components.

Following section will give a brief overview few of the important application files.

The Main Activity File

The main activity code is a Java file **MainActivity.java**. This is the actual application file which ultimately gets converted to a Dalvik executable and runs your application. Following is the default code generated by the application wizard for *Hello World!* application:

```
package com.example.helloworld;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.view.MenuItem;
import android.support.v4.app.NavUtils;

public class MainActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.activity_main, menu);
        return true;
    }
}
```

Here, *R.layout.activity_main* refers to the *activity_main.xml* file located in the *res/layout* folder. The*onCreate()* method is one of many methods that are fi red when an activity is loaded.

The Manifest File

Whatever component you develop as a part of your application, you must declare all its components in a *manifest* file called **AndroidManifest.xml** which ressides at the root of the application project directory. This file works as an interface between Android OS and your application, so if you do not declare your component in this file, then it will not be considered by the OS. For example, a default manifest file will look like as following file:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.helloworld"
   android:versionCode="1"
```

```
    android:versionName="1.0" >
    <uses-sdk
       android:minSdkVersion="8"
       android:targetSdkVersion="15" />
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/title_activity_main" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Here <application>...</application> tags enclosed the components related to the application. Attribute*android:icon* will point to the application icon available under *res/drawable-hdpi*. The application uses the image named ic_launcher.png located in the drawable folders

The <activity> tag is used to specify an activity and *android:name* attribute specifies the fully qualified class name of the *Activity* subclass and the *android:label* attributes specifies a string to use as the label for the activity. You can specify multiple activities using <activity> tags.
The **action** for the intent filter is named *android.intent.action.MAIN* to indicate that this activity serves as the entry point for the application. The **category** for the intent-filter is named*android.intent.category.LAUNCHER* to indicate that the application can be launched from the device's launcher icon.
The *@string* refers to the *strings.xml* file explained below. Hence, *@string/app_name* refers to the*app_name* string defined in the strings.xml fi le, which is "HelloWorld". Similar way, other strings get populated in the application.

Following is the list of tags which you will use in your manifest file to specify different Android application components:

- <activity>elements for activities
- <service> elements for services
- <receiver> elements for broadcast receivers
- <provider> elements for content providers

The Strings File

The **strings.xml** file is located in the *res/values* folder and it contains all the text that your application uses. For example, the names of buttons, labels, default text, and similar types of strings go into this file. This file is responsible for their textual content. For example, a default strings file will look like as following file:

```
<resources>
    <string name="app_name">HelloWorld</string>
    <string name="hello_world">Hello world!</string>
    <string name="menu_settings">Settings</string>
    <string name="title_activity_main">MainActivity</string>
</resources>
```

The R File

The **gen/com.example.helloworld/R.java** file is the glue between the activity Java files like*MainActivity.java* and the resources like *strings.xml*. It is an automatically generated file and you should not modify the content of the R.java file. Following is a sample of R.java file:

```
/* AUTO-GENERATED FILE.  DO NOT MODIFY.
 *
 * This class was automatically generated by the
 * aapt tool from the resource data it found.  It
 * should not be modified by hand.
 */

package com.example.helloworld;

public final class R {
    public static final class attr {
    }
    public static final class dimen {
        public static final int padding_large=0x7f040002;
        public static final int padding_medium=0x7f040001;
        public static final int padding_small=0x7f040000;
    }
    public static final class drawable {
        public static final int ic_action_search=0x7f020000;
        public static final int ic_launcher=0x7f020001;
    }
    public static final class id {
        public static final int menu_settings=0x7f080000;
    }
    public static final class layout {
        public static final int activity_main=0x7f030000;
    }
    public static final class menu {
        public static final int activity_main=0x7f070000;
    }
    public static final class string {
        public static final int app_name=0x7f050000;
        public static final int hello_world=0x7f050001;
        public static final int menu_settings=0x7f050002;
        public static final int title_activity_main=0x7f050003;
    }
    public static final class style {
        public static final int AppTheme=0x7f060000;
    }
}
```

The Layout File

The **activity_main.xml** is a layout file available in *res/layout* directory, that is referenced by your application when building its interface. You will modify this file very frequently to change the layout of your application. For your "Hello World!" application, this file will have following content related to default layout:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent" >

   <TextView
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:padding="@dimen/padding_medium"
        android:text="@string/hello_world"
        tools:context=".MainActivity" />

</RelativeLayout>
```

This is an example of simple *RelativeLayout* which we will study in a separate chapter. The *TextView* is an Android control used to build the GUI and it have various attribuites like *android:layout_width*,*android:layout_height* etc which are being used to set its width and height etc. The *@string* refers to the strings.xml file located in the res/values folder. Hence, @string/hello_world refers to the hello string defined in the strings.xml fi le, which is "Hello World!".

Running the Application

Let's try to run our **Hello World!** application we just created. I assume you had created your **AVD** while doing environment setup. To run the app from Eclipse, open one of your project's activity files and click Run icon from the toolbar. Eclipse installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display following Emulator window:

Congratulations!!! you have developed your first Android Application and now just keep following rest of the tutorial step by step to become a great Android Developer. All the very best.

CHAPTER

6

Resources Organizing & Accessing

There are many more items which you use to build a good Android application. Apart from coding for the application, you take care of various other **resources** like static content that your code uses, such as bitmaps, colors, layout definitions, user interface strings, animation instructions, and more. These resources are always maintained separately in various sub-directories under **res/** directory of the project.

This tutorial will explain you how you can organize your application resources, specify alternative resources and access them in your applications.

Organize Resources

You should place each type of resource in a specific subdirectory of your project's **res/** directory. For example, here's the file hierarchy for a simple project:

```
MyProject/
    src/
        MyActivity.java
    res/
        drawable/
            icon.png
        layout/
            activity_main.xml
            info.xml
        values/
            strings.xml
```

The **res/** directory contains all the resources in various subdirectories. Here we have an image resource, two layout resources, and a string resource file. Following table gives a detail about the resource directories supported inside project res/ directory.

Directory	Resource Type
anim/	XML files that define property animations. They are saved in res/anim/ folder and accessed from the **R.anim** class.
color/	XML files that define a state list of colors. They are saved in res/color/ and accessed

	from the **R.color** class.
drawable/	Image files like .png, .jpg, .gif or XML files that are compiled into bitmaps, state lists, shapes, animation drawables. They are saved in res/drawable/ and accessed from the **R.drawable** class.
layout/	XML files that define a user interface layout. They are saved in res/layout/ and accessed from the **R.layout** class.
menu/	XML files that define application menus, such as an Options Menu, Context Menu, or Sub Menu. They are saved in res/menu/ and accessed from the**R.menu** class.
raw/	Arbitrary files to save in their raw form. You need to call*Resources.openRawResource()* with the resource ID, which is *R.raw.filename* to open such raw files.
values/	XML files that contain simple values, such as strings, integers, and colors. For example, here are some filename conventions for resources you can create in this directory: arrays.xml for resource arrays, and accessed from the **R.array** class. integers.xml for resource integers, and accessed from the **R.integer**class. bools.xml for resource boolean, and accessed from the **R.bool** class. colors.xml for color values, and accessed from the **R.color** class. dimens.xml for dimension values, and accessed from the **R.dimen** class. strings.xml for string values, and accessed from the **R.string** class. styles.xml for styles, and accessed from the **R.style** class.
xml/	Arbitrary XML files that can be read at runtime by calling *Resources.getXML()*. You can save various configuration files here which will be used at run time.

Alternative Resources

Your application should provide alternative resources to support specific device configurations. For example, you should include alternative drawable resources (ie.images) for different screen resolution and alternative string resources for different languages. At runtime, Android detects the current device configuration and loads the appropriate resources for your application.

To specify configuration-specific alternatives for a set of resources, follow the following steps:

- Create a new directory in res/ named in the form **<resources_name>-<config_qualifier>**. Here**resources_name** will be any of the resources mentioned in the above table, like layout, drawable etc. The **qualifier** will specify an individual configuration for which these resources are to be used. You can check official documentation for a complete list of qualifiers for different type of resources.

- Save the respective alternative resources in this new directory. The resource files must be named exactly the same as the default resource files as shown in the below example, but these files will have content specific to the alternative. For example though image file name will be same but for high resolution screen, its resolution will be high.

Below is an example which specifies images for a default screen and alternative images for high resolution screen.

```
MyProject/
    src/
        MyActivity.java
    res/
        drawable/
            icon.png
            background.png
```

```
        drawable-hdpi/
            icon.png
            background.png
        layout/
            activity_main.xml
            info.xml
        values/
            strings.xml
```

Below is another example which specifies layout for a default language and alternative layout for arabic language.

```
MyProject/
    src/
        MyActivity.java
    res/
        drawable/
            icon.png
            background.png
        drawable-hdpi/
            icon.png
            background.png
        layout/
            activity_main.xml
            info.xml
        layout-ar/
            main.xml
        values/
            strings.xml
```

Accessing Resources

During your application development you will need to access defined resources either in your code, or in your layout XML files. Following section explains how to access your resources in both the scenarios:

ACCESSING RESOURCES IN CODE

When your Android application is compiled, a **R** class gets generated, which contains resource IDs for all the resources available in your **res/** directory. You can use R class to access that resource using sub-directory and resource name or directly resource ID.

EXAMPLE:

To access *res/drawable/myimage.png* and set an ImageView you will use following code:

```
ImageView imageView = (ImageView) findViewById(R.id.myimageview);
imageView.setImageResource(R.drawable.myimage);
```

Here first line of the code make use of *R.id.myimageview* to get ImageView defined with id*myimageview* in a Layout file. Second line of code makes use of *R.drawable.myimage* to get an image with name **myimage** available in drawable sub-directory under ***/*res**.

EXAMPLE:

Consider next example where *res/values/strings.xml* has following definition:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string  name="hello">Hello, World!</string>
```

```
</resources>
```

Now you can set the text on a TextView object with ID msg using a resource ID as follows:

```
TextView msgTextView = (TextView) findViewById(R.id.msg);
msgTextView.setText(R.string.hello);
```

EXAMPLE:

Consider a layout *res/layout/activity_main.xml* with the following definition:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:orientation="vertical" >
<TextView android:id="@+id/text"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Hello, I am a TextView" />
<Button android:id="@+id/button"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Hello, I am a Button" />
</LinearLayout>
```

This application code will load this layout for an Activity, in the onCreate() method as follows:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_activity);
}
```

ACCESSING RESOURCES IN XML

Consider the following resource XML *res/values/strings.xml* file that includes a color resource and a string resource:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <color name="opaque_red">#f00</color>
   <string name="hello">Hello!</string>
</resources>
```

Now you can use these resources in the following layout file to set the text color and text string as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<EditText xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:textColor="@color/opaque_red"
    android:text="@string/hello" />
```

Now if you will go through previous chapter once again where I have explained **Hello World!** example, and I'm sure you will have better understanding on all the concepts explained in this chapter. So I highly recommend to check previous chapter for working example and check how I have used various resources at very basic level.

CHAPTER

7

Activities

An activity represents a single screen with a user interface. For example, an email application might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. If an application has more than one activity, then one of them should be marked as the activity that is presented when the application is launched.

If you have worked with C, C++ or Java programming language then you must have seen that your program starts from **main()** function. Very similar way, Android system initiates its program with in an**Activity** starting with a call on *onCreate()* callback method. There is a sequence of callback methods that start up an activity and a sequence of callback methods that tear down an activity as shown in the below Activity lifecycle diagram: (*image courtesy : android.com*)

The Activity class defines the following callbacks i.e. events. You don't need to implement all the callbacks methods. However, it's important that you understand each one and implement those that ensure your app behaves the way users expect.

Callback	Description
onCreate()	This is the first callback and called when the activity is first created.
onStart()	This callback is called when the activity becomes visible to the user.
onResume()	This is called when the user starts interacting with the application.

onPause()	The paused activity does not receive user input and cannot execute any code and called when the current activity is being paused and the previous activity is being resumed.
onStop()	This callback is called when the activity is no longer visible.
onDestroy()	This callback is called before the activity is destroyed by the system.
onRestart()	This callback is called when the activity restarts after stopping it.

Example

This example will take you through simple steps to show Android application activity life cycle. Follow the following steps to modify the Android application we created in *Hello World Example* chapter:

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *HelloWorld* under a package *com.example.helloworld* as explained in the *Hello World Example* chapter.
2	Modify main activity file *MainActivity.java* as explained below. Keep rest of the files unchanged.
3	Run the application to launch Android emulator and verify the result of the changes done in the aplication.

Following is the content of the modified main activity file**src/com.example.helloworld/MainActivity.java**. This file includes each of the fundamental lifecycle methods. The **Log.d()** method has been used to generate log messages:

```
package com.example.helloworld;

import android.os.Bundle;
import android.app.Activity;
import android.util.Log;

public class MainActivity extends Activity {
   String msg = "Android : ";

   /** Called when the activity is first created. */
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      Log.d(msg, "The onCreate() event");
   }

   /** Called when the activity is about to become visible. */
   @Override
   protected void onStart() {
      super.onStart();
      Log.d(msg, "The onStart() event");
   }

   /** Called when the activity has become visible. */
   @Override
   protected void onResume() {
      super.onResume();
      Log.d(msg, "The onResume() event");
   }

   /** Called when another activity is taking focus. */
   @Override
```

```
   protected void onPause() {
      super.onPause();
      Log.d(msg, "The onPause() event");
   }

   /** Called when the activity is no longer visible. */
   @Override
   protected void onStop() {
      super.onStop();
      Log.d(msg, "The onStop() event");
   }

   /** Called just before the activity is destroyed. */
   @Override
   public void onDestroy() {
      super.onDestroy();
      Log.d(msg, "The onDestroy() event");
   }
}
```

An activity class loads all the UI component using the XML file available in *res/layout* folder of the project. Following statement loads UI components from *res/layout/activity_main.xml file*:

```
setContentView(R.layout.activity_main);
```

An application can have one or more activities without any restrictions. Every activity you define for your application must be declared in your *AndroidManifest.xml* file and the main activity for your app must be declared in the manifest with an <intent-filter> that includes the MAIN action and LAUNCHER category as follows:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.helloworld"
   android:versionCode="1"
   android:versionName="1.0" >
   <uses-sdk
      android:minSdkVersion="8"
      android:targetSdkVersion="15" />
   <application
       android:icon="@drawable/ic_launcher"
       android:label="@string/app_name"
       android:theme="@style/AppTheme" >
       <activity
           android:name=".MainActivity"
           android:label="@string/title_activity_main" >
           <intent-filter>
               <action android:name="android.intent.action.MAIN" />
               <category android:name="android.intent.category.LAUNCHER"/>
           </intent-filter>
       </activity>
   </application>
</manifest>
```

If either the MAIN action or LAUNCHER category are not declared for one of your activities, then your app icon will not appear in the Home screen's list of apps.

Let's try to run our modified **Hello World!** application we just modified. I assume you had created your **AVD** while doing environment setup. To run the app from Eclipse, open one of your project's activity files and click Run icon from the toolbar. Eclipse installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display Emulator window and you should see following log messages in **LogCat** window in Eclipse IDE:

```
07-19 15:00:43.405: D/Android :(866): The onCreate() event
```

```
07-19 15:00:43.405: D/Android :(866): The onStart() event
07-19 15:00:43.415: D/Android :(866): The onResume() event
```

Let us try to click Red button on the Android emulator and it will generate following events messages in **LogCat** window in Eclipse IDE:

```
07-19 15:01:10.995: D/Android :(866): The onPause() event
07-19 15:01:12.705: D/Android :(866): The onStop() event
```

Let us again try to click Menu button on the Android emulator and it will generate following events messages in **LogCat** window in Eclipse IDE:

```
07-19 15:01:13.995: D/Android :(866): The onStart() event
07-19 15:01:14.705: D/Android :(866): The onResume() event
```

Next, let us again try to click Back button on the Android emulator and it will generate following events messages in **LogCat** window in Eclipse IDE and this completes the Acitivity Life Cycle for an Android Application.

```
07-19 15:33:15.687: D/Android :(992): The onPause() event
07-19 15:33:15.525: D/Android :(992): The onStop() event
07-19 15:33:15.525: D/Android :(992): The onDestroy() event
```

CHAPTER

8

Services

A service is a component that runs in the background to perform long-running operations without needing to interact with the user. For example, a service might play music in the background while the user is in a different application, or it might fetch data over the network without blocking user interaction with an activity. A service can essentially take two states:

State	Description
Started	A service is **started** when an application component, such as an activity, starts it by calling *startService()*. Once started, a service can run in the background indefinitely, even if the component that started it is destroyed.
Bound	A service is **bound** when an application component binds to it by calling*bindService()*. A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC).

A service has lifecycle callback methods that you can implement to monitor changes in the service's state and you can perform work at the appropriate stage. The following diagram on the left shows the lifecycle when the service is created with startService() and the diagram on the right shows the lifecycle when the service is created with bindService(): *(image courtesy : android.com)*

To create an service, you create a Java class that extends the Service base class or one of its existing subclasses. The **Service** base class defines various callback methods and the most important are given below. You don't need to implement all the callbacks methods. However, it's important that you understand each one and implement those that ensure your app behaves the way users expect.

Callback	Description
onStartCommand()	The system calls this method when another component, such as an activity, requests that the service be started, by calling *startService()*. If you implement this method, it is your responsibility to stop the service when its work is done, by calling *stopSelf()* or *stopService()* methods.
onBind()	The system calls this method when another component wants to bind with the service by calling *bindService()*. If you implement this method, you must provide an interface that clients use to communicate with the service, by returning an *IBinder* object. You must always implement this method, but if you don't want to allow binding, then you should return *null*.
onUnbind()	The system calls this method when all clients have disconnected from a particular interface published by the service.
onRebind()	The system calls this method when new clients have connected to the service, after it had previously been notified that all had disconnected in its*onUnbind(Intent)*.

onCreate()	The system calls this method when the service is first created using*onStartCommand()* or *onBind()*. This call is required to perform one-time setup.
onDestroy()	The system calls this method when the service is no longer used and is being destroyed. Your service should implement this to clean up any resources such as threads, registered listeners, receivers, etc.

The following skeleton service demonstrates each of the lifecycle methods:

```
package com.tutorialspoint;

import android.app.Service;
import android.os.IBinder;
import android.content.Intent;
import android.os.Bundle;

public class HelloService extends Service {

   /** indicates how to behave if the service is killed */
   int mStartMode;
   /** interface for clients that bind */
   IBinder mBinder;
   /** indicates whether onRebind should be used */
   boolean mAllowRebind;

   /** Called when the service is being created. */
   @Override
   public void onCreate() {

   }

   /** The service is starting, due to a call to startService() */
   @Override
   public int onStartCommand(Intent intent, int flags, int startId) {
      return mStartMode;
   }

   /** A client is binding to the service with bindService() */
   @Override
   public IBinder onBind(Intent intent) {
      return mBinder;
   }

   /** Called when all clients have unbound with unbindService() */
   @Override
   public boolean onUnbind(Intent intent) {
      return mAllowRebind;
   }

   /** Called when a client is binding to the service with bindService()*/
   @Override
   public void onRebind(Intent intent) {

   }

   /** Called when The service is no longer used and is being destroyed */
   @Override
   public void onDestroy() {

   }
```

```
}
```

Example

This example will take you through simple steps to show how to create your own Android Service. Follow the following steps to modify the Android application we created in *Hello World Example*chapter:

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *HelloWorld* under a package *com.example.helloworld* as explained in the *Hello World Example* chapter.
2	Modify main activity file *MainActivity.java* to add *startService()* and *stopService()* methods.
3	Create a new java file *MyService.java* under the package *com.example.helloworld*. This file will have implementation of Android service related methods.
4	Define your service in *AndroidManifest.xml* file using <service.../> tag. An application can have one or more services without any restrictions.
5	Modify the detault content of *res/layout/activity_main.xml* file to include two buttons in linear layout.
6	Define two constants *start_service* and *stop_service* in *res/values/strings.xml* file
7	Run the application to launch Android emulator and verify the result of the changes done in the aplication.

Following is the content of the modified main activity file**src/com.example.helloworld/MainActivity.java**. This file can include each of the fundamental lifecycle methods. We have added *startService()* and *stopService()* methods to start and stop the service.

```
package com.example.helloworld;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.content.Intent;
import android.view.View;

public class MainActivity extends Activity {

   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
   }
   @Override
   public boolean onCreateOptionsMenu(Menu menu) {
      getMenuInflater().inflate(R.menu.activity_main, menu);
      return true;
   }

   // Method to start the service
   public void startService(View view) {
      startService(new Intent(getBaseContext(), MyService.class));
   }

   // Method to stop the service
   public void stopService(View view) {
      stopService(new Intent(getBaseContext(), MyService.class));
   }
```

```
}
```

Following is the content of **src/com.example.helloworld/MyService.java**. This file can have implementation of one or more methods associated with Service based on requirements. For now we are going to implement only two methods *onStartCommand()* and *onDestroy()*:

```
package com.example.helloworld;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.widget.Toast;

public class MyService extends Service {
   @Override
   public IBinder onBind(Intent arg0) {
      return null;
   }

   @Override
   public int onStartCommand(Intent intent, int flags, int startId) {
      // Let it continue running until it is stopped.
      Toast.makeText(this, "Service Started", Toast.LENGTH_LONG).show();
      return START_STICKY;
   }
   @Override
   public void onDestroy() {
      super.onDestroy();
      Toast.makeText(this, "Service Destroyed", Toast.LENGTH_LONG).show();
   }
}
```

Following will the modified content of *AndroidManifest.xml* file. Here we have added <service.../> tag to include our service:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.helloworld"
   android:versionCode="1"
   android:versionName="1.0" >
   <uses-sdk
      android:minSdkVersion="8"
      android:targetSdkVersion="15" />
   <application
       android:icon="@drawable/ic_launcher"
       android:label="@string/app_name"
       android:theme="@style/AppTheme" >
       <activity
           android:name=".MainActivity"
           android:label="@string/title_activity_main" >
           <intent-filter>
               <action android:name="android.intent.action.MAIN" />
               <category android:name="android.intent.category.LAUNCHER"/>
           </intent-filter>
       </activity>
       <service android:name=".MyService" />
   </application>
</manifest>
```

Following will be the content of **res/layout/activity_main.xml** file to include two buttons:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:orientation="vertical" >

   <Button android:id="@+id/btnStartService"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:text="@string/start_service"
   android:onClick="startService"/>

   <Button android:id="@+id/btnStopService"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:text="@string/stop_service"
   android:onClick="stopService" />

</LinearLayout>
```

Following will be the content of **res/values/strings.xml** to define two new constants:

```
<resources>

    <string name="app_name">HelloWorld</string>
    <string name="hello_world">Hello world!</string>
    <string name="menu_settings">Settings</string>
    <string name="title_activity_main">MainActivity</string>
    <string name="start_service">Start Service</string>
    <string name="stop_service">Stop Service</string>

</resources>
```

Let's try to run our modified **Hello World!** application we just modified. I assume you had created your **AVD** while doing environment setup. To run the app from Eclipse, open one of your project's activity files and click Run icon from the toolbar. Eclipse installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display following Emulator window:

Now to start your service, let's click on **Start Service** button, this will start the service and as per our programming in *onStartCommand()* method, a message *Service Started* will appear on the bottom of the the simulator as follows:

To stop the service, you can click the Stop Service button.

CHAPTER

9

Broadcast Recievers

Broadcast Receivers simply respond to broadcast messages from other applications or from the system itself. These messages are sometime called events or intents. For example, applications can also initiate broadcasts to let other applications know that some data has been downloaded to the device and is available for them to use, so this is broadcast receiver who will intercept this communication and will initiate appropriate action.

There are following two important steps to make BroadcastReceiver works for the systen broadcasted intents:

- Creating the Broadcast Receiver.
- Registering Broadcast Receiver

There is one additional steps in case you are going to implement your custom intents then you will have to create and broadcast those intents.

Creating the Broadcast Receiver

A broadcast receiver is implemented as a subclass of **BroadcastReceiver** class and overriding the onReceive() method where each message is received as a **Intent** object parameter.

```
public class MyReceiver extends BroadcastReceiver {

   @Override
   public void onReceive(Context context, Intent intent) {
      Toast.makeText(context, "Intent Detected.",
Toast.LENGTH_LONG).show();
   }

}
```

Registering Broadcast Receiver

An application listens for specific broadcast intents by registering a broadcast receiver in*AndroidManifest.xml* file. Consider we are going to register *MyReceiver* for system generated event ACTION_BOOT_COMPLETED which is fired by the system once the Android system has completed the boot process.

```
<application
   android:icon="@drawable/ic_launcher"
```

```
   android:label="@string/app_name"
   android:theme="@style/AppTheme" >

   <receiver android:name="MyReceiver">
      <intent-filter>
         <action android:name="android.intent.action.BOOT_COMPLETED">
      </action>
      </intent-filter>
   </receiver>

</application>
```

Now whenever your Android device gets booted, it will be intercepted by BroadcastReceiver *MyReceiver*and implemented logic inside *onReceive()* will be executed.

There are several system generated events defined as final static fields in the **Intent** class. The following table lists a few important system events.

Event Constant	Description
android.intent.action.BATTERY_CHANGED	Sticky broadcast containing the charging state, level, and other information about the battery.
android.intent.action.BATTERY_LOW	Indicates low battery condition on the device.
android.intent.action.BATTERY_OKAY	Indicates the battery is now okay after being low.
android.intent.action.BOOT_COMPLETED	This is broadcast once, after the system has finished booting.
android.intent.action.BUG_REPORT	Show activity for reporting a bug.
android.intent.action.CALL	Perform a call to someone specified by the data.
android.intent.action.CALL_BUTTON	The user pressed the "call" button to go to the dialer or other appropriate UI for placing a call.
android.intent.action.DATE_CHANGED	The date has changed.
android.intent.action.REBOOT	Have the device reboot.

Broadcasting Custom Intents

If you want your application itself should generate and send custom intents then you will have to create and send those intents by using the *sendBroadcast()* method inside your activity class. If you use the*sendStickyBroadcast(Intent)* method, the Intent is **sticky**, meaning the *Intent* you are sending stays around after the broadcast is complete.

```
public void broadcastIntent(View view)
{
   Intent intent = new Intent();
   intent.setAction("com.tutorialspoint.CUSTOM_INTENT");
   sendBroadcast(intent);
}
```

This intent *com.tutorialspoint.CUSTOM_INTENT* can also be regsitered in similar way as we have regsitered system generated intent.

```
<application
   android:icon="@drawable/ic_launcher"
   android:label="@string/app_name"
   android:theme="@style/AppTheme" >
```

```
   <receiver android:name="MyReceiver">
      <intent-filter>
         <action android:name="com.tutorialspoint.CUSTOM_INTENT">
      </action>
      </intent-filter>
   </receiver>

</application>
```

Example

This example will explain you how to create *BroadcastReceiver* to intercept custom intent. Once you are familiar with custom intent, then you can program your application to intercept system generated intents. So let's follow the following steps to modify the Android application we created in *Hello World Example* chapter:

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *HelloWorld* under a package *com.example.helloworld* as explained in the *Hello World Example* chapter.
2	Modify main activity file *MainActivity.java* to add *broadcastIntent()* method.
3	Create a new java file called *MyReceiver.java* under the package *com.example.helloworld* to define a BroadcastReceiver.
4	An application can handle one or more custom and system intents without any restrictions. Every indent you want to intercept must be registered in your *AndroidManifest.xml* file using <receiver.../> tag
5	Modify the detault content of *res/layout/activity_main.xml* file to include a button to broadcast intent.
6	Define a constant *broadcast_intent* in *res/values/strings.xml* file
7	Run the application to launch Android emulator and verify the result of the changes done in the aplication.

Following is the content of the modified main activity file**src/com.example.helloworld/MainActivity.java**. This file can include each of the fundamental lifecycle methods. We have added *broadcastIntent()* method to broadcast a custom intent.

```
package com.example.helloworld;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.content.Intent;
import android.view.View;

public class MainActivity extends Activity {

   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
   }
   @Override
   public boolean onCreateOptionsMenu(Menu menu) {
      getMenuInflater().inflate(R.menu.activity_main, menu);
      return true;
   }
   // broadcast a custom intent.
   public void broadcastIntent(View view)
```

```
    {
       Intent intent = new Intent();
       intent.setAction("com.tutorialspoint.CUSTOM_INTENT");
       sendBroadcast(intent);
    }
}
```

Following is the content of **src/com.example.helloworld/MyReceiver.java**:

```
package com.example.helloworld;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.widget.Toast;

public class MyReceiver extends BroadcastReceiver {

   @Override
   public void onReceive(Context context, Intent intent) {
      Toast.makeText(context, "Intent Detected.", Toast.LENGTH_LONG).show();
   }

}
```

Following will the modified content of *AndroidManifest.xml* file. Here we have added <service.../> tag to include our service:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.helloworld"
   android:versionCode="1"
   android:versionName="1.0" >
   <uses-sdk
      android:minSdkVersion="8"
      android:targetSdkVersion="15" />
   <application
       android:icon="@drawable/ic_launcher"
       android:label="@string/app_name"
       android:theme="@style/AppTheme" >
       <activity
           android:name=".MainActivity"
           android:label="@string/title_activity_main" >
           <intent-filter>
               <action android:name="android.intent.action.MAIN" />
               <category android:name="android.intent.category.LAUNCHER"/>
           </intent-filter>
       </activity>
       <receiver android:name="MyReceiver">
          <intent-filter>
          <action android:name="com.tutorialspoint.CUSTOM_INTENT">
          </action>
          </intent-filter>
      </receiver>
   </application>
</manifest>
```

Following will be the content of **res/layout/activity_main.xml** file to include a button to broadcast our custom intent:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
```

```
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:orientation="vertical" >

   <Button android:id="@+id/btnStartService"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:text="@string/broadcast_intent"
   android:onClick="broadcastIntent"/>

</LinearLayout>
```

Following will be the content of **res/values/strings.xml** to define two new constants:

```
<resources>

    <string name="app_name">HelloWorld</string>
    <string name="hello_world">Hello world!</string>
    <string name="menu_settings">Settings</string>
    <string name="title_activity_main">MainActivity</string>
    <string name="broadcast_intent">Broadcast Intent</string>

</resources>
```

Let's try to run our modified **Hello World!** application we just modified. I assume you had created your **AVD** while doing environment setup. To run the app from Eclipse, open one of your project's activity files and click Run icon from the toolbar. Eclipse installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display following Emulator window:

Now to broadcast our custom intent, let's click on **Broadcast Intent** button, this will broadcast our custom intent *"com.tutorialspoint.CUSTOM_INTENT"* which will be intercepted by our registered BroadcastReceiver ie. MyReceiver and as per our implemented logic a toast will appear on the bottom of the the simulator as follows:

You can try implementing other BroadcastReceiver to intercept system generated intents like system bootup, date changed, low battery etc.

CHAPTER

10

Content Providers

A content provider component supplies data from one application to others on request. Such requests are handled by the methods of the ContentResolver class. A content provider can use different ways to store its data and the data can be stored in a database, in files, or even over a network.

Each Android applications runs in its own process with its own permissions which keeps an application data hidden from another application. But sometimes it is required to share data across applications. This is where content providers become very useful.

Content providers let you centralize content in one place and have many different applications access it as needed. A content provider behaves very much like a database where you can query it, edit its content, as well as add or delete content usingg insert(), update(), delete(), and query() methods. In most cases this data is stored in an **SQlite** database.

A content provider is implemented as a subclass of **ContentProvider** class and must implement a standard set of APIs that enable other applications to perform transactions.

```
public class MyContentProvider extends  ContentProvider {

}
```

Content URIs

To query a content provider, you specify the query string in the form of a URI which has following format:

```
<prefix>://<authority>/<data_type>/<id>
```

Here is the detaial of various parts of the URI:

Part	Description
prefix	This is always set to content://
authority	This specifies the name of the content provider, for example *contacts*, *browser* etc. For third-party content providers, this could be the fully qualified name, such as*com.tutorialspoint.statusprovider*
data_type	This indicates the type of data that this particular provider provides. For example, if you are getting all the contacts from the *Contacts* content provider, then the data path would be *people* and URI would look like this *content://contacts/people*
Id	This specifies the specific record requested. For example, if you are looking for contact number 5 in the Contacts content provider then URI would look like this*content://contacts/people/5*.

Create Content Provider

This involves number of simple steps to create your own content provider.

- First of all you need to create a Content Provider class that extends the *ContentProviderbase*class.
- Second, you need to define your content provider URI address which will be used to access the content.
- Next you will need to create your own database to keep the content. Usually, Android uses SQLite database and framework needs to override *onCreate()* method which will use SQLite Open Helper method to create or open the provider's databse. When your application is launched, the*onCreate()* handler of each of its Content Providers is called on the main application thread.
- Next you will have to implement Content Provider queries to perform different database specific operations.
- Finally register your Content Provider in your acitivity file using <provider> tag.

Here is the list of methods which you need to override in Content Provider class to have your Content Provider working:

- **onCreate()** This method is called when the provider is started.
- **query()** This method receives a request from a client. The result is returned as a Cursor object.
- **insert()**This method inserts a new record into the content provider.
- **delete()** This method deletes an existing record from the content provider.
- **update()** This method updates an existing record from the content provider.
- **getType()** This method returns the MIME type of the data at the given URI.

Example

This example will explain you how to create your own *ContentProvider*. So let's follow the following steps to similar to what we followed while creating *Hello World Example*:

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *MyContentProvider*under a package *com.example.mycontentprovider*, with blank Activity.
2	Modify main activity file *MainActivity.java* to add two new methods *onClickAddName()* and*onClickRetrieveStudents()*.
3	Create a new java file called *StudentsProvider.java* under the package*com.example.mycontentprovider* to define your actual provider and associated methods.
4	Register your content provider in your *AndroidManifest.xml* file using <provider.../> tag
5	Modify the detault content of *res/layout/activity_main.xml* file to include a small GUI to add sudents records.
6	Define required constants in *res/values/strings.xml* file
7	Run the application to launch Android emulator and verify the result of the changes done in the aplication.

Following is the content of the modified main activity file**src/com.example.mycontentprovider/MainActivity.java**. This file can include each of the fundamental lifecycle methods. We have added two new methods *onClickAddName()* and*onClickRetrieveStudents()* to handle user interaction with the application.

```
package com.example.mycontentprovider;
```

```java
import android.net.Uri;
import android.os.Bundle;
import android.app.Activity;
import android.content.ContentValues;
import android.content.CursorLoader;
import android.database.Cursor;
import android.view.Menu;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;

public class MainActivity extends Activity {

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

   @Override
   public boolean onCreateOptionsMenu(Menu menu) {
      getMenuInflater().inflate(R.menu.main, menu);
      return true;
   }

   public void onClickAddName(View view) {
      // Add a new student record
      ContentValues values = new ContentValues();

      values.put(StudentsProvider.NAME,
      ((EditText)findViewById(R.id.txtName)).getText().toString());

      values.put(StudentsProvider.GRADE,
      ((EditText)findViewById(R.id.txtGrade)).getText().toString());

      Uri uri = getContentResolver().insert(
      StudentsProvider.CONTENT_URI, values);

      Toast.makeText(getBaseContext(),
      uri.toString(), Toast.LENGTH_LONG).show();
   }

   public void onClickRetrieveStudents(View view) {
      // Retrieve student records
      String URL = "content://com.example.provider.College/students";
      Uri students = Uri.parse(URL);
      Cursor c = managedQuery(students, null, null, null, "name");
      if (c.moveToFirst()) {
         do{
            Toast.makeText(this,
            c.getString(c.getColumnIndex(StudentsProvider._ID)) +
            ", " +  c.getString(c.getColumnIndex( StudentsProvider.NAME)) +
            ", " + c.getString(c.getColumnIndex( StudentsProvider.GRADE)),
            Toast.LENGTH_SHORT).show();
         } while (c.moveToNext());
      }
   }
}
```

Create new file StudentsProvider.java under *com.example.mycontentprovider* package and following is the content of **src/com.example.mycontentprovider/StudentsProvider.java**:

```
package com.example.mycontentprovider;

import java.util.HashMap;

import android.content.ContentProvider;
import android.content.ContentUris;
import android.content.ContentValues;
import android.content.Context;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteQueryBuilder;
import android.net.Uri;
import android.text.TextUtils;

public class StudentsProvider extends ContentProvider {

   static final String PROVIDER_NAME = "com.example.provider.College";
   static final String URL = "content://" + PROVIDER_NAME + "/students";
   static final Uri CONTENT_URI = Uri.parse(URL);

   static final String _ID = "_id";
   static final String NAME = "name";
   static final String GRADE = "grade";

   private static HashMap<String, String> STUDENTS_PROJECTION_MAP;

   static final int STUDENTS = 1;
   static final int STUDENT_ID = 2;

   static final UriMatcher uriMatcher;
   static{
      uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
      uriMatcher.addURI(PROVIDER_NAME, "students", STUDENTS);
      uriMatcher.addURI(PROVIDER_NAME, "students/#", STUDENT_ID);
   }

   /**
    * Database specific constant declarations
    */
   private SQLiteDatabase db;
   static final String DATABASE_NAME = "College";
   static final String STUDENTS_TABLE_NAME = "students";
   static final int DATABASE_VERSION = 1;
   static final String CREATE_DB_TABLE =
      " CREATE TABLE " + STUDENTS_TABLE_NAME +
      " (_id INTEGER PRIMARY KEY AUTOINCREMENT, " +
      " name TEXT NOT NULL, " +
      " grade TEXT NOT NULL);";

   /**
    * Helper class that actually creates and manages
    * the provider's underlying data repository.
    */
   private static class DatabaseHelper extends SQLiteOpenHelper {
       DatabaseHelper(Context context){
```

```
            super(context, DATABASE_NAME, null, DATABASE_VERSION);
        }

        @Override
        public void onCreate(SQLiteDatabase db)
        {
            db.execSQL(CREATE_DB_TABLE);
        }

        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion,
                              int newVersion) {
            db.execSQL("DROP TABLE IF EXISTS " +  STUDENTS_TABLE_NAME);
            onCreate(db);
        }
    }

    @Override
    public boolean onCreate() {
        Context context = getContext();
        DatabaseHelper dbHelper = new DatabaseHelper(context);
        /**
         * Create a write able database which will trigger its
         * creation if it doesn't already exist.
         */
        db = dbHelper.getWritableDatabase();
        return (db == null)? false:true;
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {
        /**
         * Add a new student record
         */
        long rowID = db.insert(     STUDENTS_TABLE_NAME, "", values);
        /**
         * If record is added successfully
         */
        if (rowID > 0)
        {
            Uri _uri = ContentUris.withAppendedId(CONTENT_URI, rowID);
            getContext().getContentResolver().notifyChange(_uri, null);
            return _uri;
        }
        throw new SQLException("Failed to add a record into " + uri);
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
                        String[] selectionArgs, String sortOrder) {

        SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
        qb.setTables(STUDENTS_TABLE_NAME);

        switch (uriMatcher.match(uri)) {
        case STUDENTS:
            qb.setProjectionMap(STUDENTS_PROJECTION_MAP);
            break;
        case STUDENT_ID:
            qb.appendWhere( _ID + "=" + uri.getPathSegments().get(1));
            break;
```

```
      default:
         throw new IllegalArgumentException("Unknown URI " + uri);
      }
      if (sortOrder == null || sortOrder == ""){
         /**
          * By default sort on student names
          */
         sortOrder = NAME;
      }
      Cursor c = qb.query(db,     projection,      selection, selectionArgs,
                          null, null, sortOrder);
      /**
       * register to watch a content URI for changes
       */
      c.setNotificationUri(getContext().getContentResolver(), uri);

      return c;
   }

   @Override
   public int delete(Uri uri, String selection, String[] selectionArgs) {
      int count = 0;

      switch (uriMatcher.match(uri)){
      case STUDENTS:
         count = db.delete(STUDENTS_TABLE_NAME, selection, selectionArgs);
         break;
      case STUDENT_ID:
         String id = uri.getPathSegments().get(1);
         count = db.delete( STUDENTS_TABLE_NAME, _ID +  " = " + id +
                (!TextUtils.isEmpty(selection) ? " AND (" +
                selection + ')' : ""), selectionArgs);
         break;
      default:
         throw new IllegalArgumentException("Unknown URI " + uri);
      }

      getContext().getContentResolver().notifyChange(uri, null);
      return count;
   }

   @Override
   public int update(Uri uri, ContentValues values, String selection,
                     String[] selectionArgs) {
      int count = 0;

      switch (uriMatcher.match(uri)){
      case STUDENTS:
         count = db.update(STUDENTS_TABLE_NAME, values,
                 selection, selectionArgs);
         break;
      case STUDENT_ID:
         count = db.update(STUDENTS_TABLE_NAME, values, _ID +
                 " = " + uri.getPathSegments().get(1) +
                 (!TextUtils.isEmpty(selection) ? " AND (" +
                 selection + ')' : ""), selectionArgs);
         break;
      default:
         throw new IllegalArgumentException("Unknown URI " + uri );
      }
      getContext().getContentResolver().notifyChange(uri, null);
```

```
      return count;
   }

   @Override
   public String getType(Uri uri) {
      switch (uriMatcher.match(uri)){
      /**
       * Get all student records
       */
      case STUDENTS:
         return "vnd.android.cursor.dir/vnd.example.students";
      /**
       * Get a particular student
       */
      case STUDENT_ID:
         return "vnd.android.cursor.item/vnd.example.students";
      default:
         throw new IllegalArgumentException("Unsupported URI: " + uri);
      }
   }
}
```

Following will the modified content of *AndroidManifest.xml* file. Here we have added <provider.../> tag to include our content provider:

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

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

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.mycontentprovider.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <provider android:name="StudentsProvider"
           android:authorities="com.example.provider.College">
        </provider>
    </application>

</manifest>
```

Following will be the content of **res/layout/activity_main.xml** file to include a button to broadcast your custom intent:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
```

```
    android:orientation="vertical" >
    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Name" />
    <EditText
    android:id="@+id/txtName"
    android:layout_height="wrap_content"
    android:layout_width="fill_parent" />
    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Grade" />
    <EditText
    android:id="@+id/txtGrade"
    android:layout_height="wrap_content"
    android:layout_width="fill_parent" />
    <Button
    android:text="Add Name"
    android:id="@+id/btnAdd"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:onClick="onClickAddName" />
    <Button
    android:text="Retrieve Students"
    android:id="@+id/btnRetrieve"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:onClick="onClickRetrieveStudents" />
</LinearLayout>
```

Make sure you have following content of **res/values/strings.xml** file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="app_name">MyContentProvider</string>
    <string name="action_settings">Settings</string>
    <string name="hello_world">Hello world!</string>

</resources>;
```

Let's try to run our modified **MyContentProvider** application we just created. I assume you had created your **AVD** while doing environment setup. To run the app from Eclipse, open one of your project's activity files and click Run icon from the toolbar. Eclipse installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display following Emulator window, be patience because it may take sometime based on your computer speed:

Now let's enter student **Name** and **Grade** and finally click on **Add Name** button, this will add student record in the database and will flash a message at the bottom showing ContentProvider URI along with record number added in the database. This operation makes use of our **insert()** method. Let's repeat this process to add few more students in the database of our content provider.

Once you are done with adding records in the database, now its time to ask ContentProvider to give us those records back, so let's click **Retrieve Students** button which will fetch and display all the records one by one which is as per our the implementation of our **query()** method.

You can write activities against update and delete operations by providing callback functions in**MainActivity.java** file and then modify user interface to have buttons for update and deleted operations in the same way as we have done for add and read operations.

This way you can use existing Content Provider like Address Book or you can use Content Provider concept in developing nice database oriented applications where you can perform all sort of database operations like read, write, update and delete as explained above in the example.

CHAPTER

12

Intents and Filters

An Android **Intent** is an object carrying an *intent* ie. message from one component to another component with-in the application or outside the application. The intents can communicate messages among any of the three core components of an application - activities, services, and broadcast receivers.
The intent itself, an Intent object, is a passive data structure holding an abstract description of an operation to be performed.
For example, let's assume that you have an Activity that needs to launch an email client and sends an email using your Android device. For this purpose, your Activity would send an ACTION_SEND along with appropriate **chooser**, to the Android Intent Resolver. The specified chooser gives the proper interface for the user to pick how to send your email data.

For example, assume that you have an Activity that needs to open URL in a web browser on your Android device. For this purpose, your Activity will send ACTION_WEB_SEARCH Intent to the Android Intent Resolver to open given URL in the web browser. The Intent Resolver parses through a list of Activities and chooses the one that would best match your Intent, in this case, the Web Browser Activity. The Intent Resolver then passes your web page to the web browser and starts the Web Browser Activity.

There are separate mechanisms for delivering intents to each type of component - activities, services, and broadcast receivers.

S.N.	Method & Description
1	**Context.startActivity()** The Intent object is passed to this method to launch a new activity or get an existing activity to do something new.
2	**Context.startService()** The Intent object is passed to this method to initiate a service or deliver new instructions to an ongoing service.
3	**Context.sendBroadcast()** The Intent object is passed to this method to deliver the message to all interested broadcast receivers.

Intent Objects

An Intent object is a bundle of information which is used by the component that receives the intent plus information used by the Android system.

An Intent object can contain the following components based on what it is communicating or going to perform:

ACTION

This is mandatory part of the Intent object and is a string naming the action to be performed — or, in the case of broadcast intents, the action that took place and is being reported. The action largely determines how the rest of the intent object is structured . The Intent class defines a number of action constants corresponding to different intents. Here is a list of Android Intent Standard Actions

Android Intent Standard Actions:

Following table lists down various important Android Intent Standard Actions. You can check Android Official Documentation for a complete list of Actions:

S.N.	Activity Action Intent & Description
1	**ACTION_ALL_APPS** List all the applications available on the device.
2	**ACTION_ANSWER** Handle an incoming phone call.
3	**ACTION_ATTACH_DATA** Used to indicate that some piece of data should be attached to some other place
4	**ACTION_BATTERY_CHANGED** This is a sticky broadcast containing the charging state, level, and other information about the battery.
5	**ACTION_BATTERY_LOW** This broadcast corresponds to the "Low battery warning" system dialog.
6	**ACTION_BATTERY_OKAY** This will be sent after ACTION_BATTERY_LOW once the battery has gone back up to an okay state.
7	**ACTION_BOOT_COMPLETED** This is broadcast once, after the system has finished booting.
8	**ACTION_BUG_REPORT** Show activity for reporting a bug.
9	**ACTION_CALL** Perform a call to someone specified by the data.
10	**ACTION_CALL_BUTTON** The user pressed the "call" button to go to the dialer or other appropriate UI for placing a call.
11	**ACTION_CAMERA_BUTTON** The "Camera Button" was pressed.
12	**ACTION_CHOOSER** Display an activity chooser, allowing the user to pick what they want to before proceeding.
13	**ACTION_CONFIGURATION_CHANGED** The current device Configuration (orientation, locale, etc) has changed.
14	**ACTION_DATE_CHANGED** The date has changed.
15	**ACTION_DEFAULT** A synonym for ACTION_VIEW, the "standard" action that is performed on a piece of data.

16	**ACTION_DELETE** Delete the given data from its container.
17	**ACTION_DEVICE_STORAGE_LOW** A sticky broadcast that indicates low memory condition on the device.
18	**ACTION_DEVICE_STORAGE_OK** Indicates low memory condition on the device no longer exists.
19	**ACTION_DIAL** Dial a number as specified by the data.
20	**ACTION_DOCK_EVENT** A sticky broadcast for changes in the physical docking state of the device.
21	**ACTION_DREAMING_STARTED** Sent after the system starts dreaming.
22	**ACTION_DREAMING_STOPPED** Sent after the system stops dreaming.
23	**ACTION_EDIT** Provide explicit editable access to the given data.
24	**ACTION_FACTORY_TEST** Main entry point for factory tests.
25	**ACTION_GET_CONTENT** Allow the user to select a particular kind of data and return it.
26	**ACTION_GTALK_SERVICE_CONNECTED** A GTalk connection has been established.
27	**ACTION_GTALK_SERVICE_DISCONNECTED** A GTalk connection has been disconnected.
28	**ACTION_HEADSET_PLUG** Wired Headset plugged in or unplugged.
29	**ACTION_INPUT_METHOD_CHANGED** An input method has been changed.
30	**ACTION_INSERT** Insert an empty item into the given container.
31	**ACTION_INSERT_OR_EDIT** Pick an existing item, or insert a new item, and then edit it.
32	**ACTION_INSTALL_PACKAGE** Launch application installer.
33	**ACTION_LOCALE_CHANGED** The current device's locale has changed.
34	**ACTION_MAIN** Start as a main entry point, does not expect to receive data.
35	**ACTION_MEDIA_BUTTON** The "Media Button" was pressed.

36	**ACTION_MEDIA_CHECKING** External media is present, and being disk-checked.
37	**ACTION_MEDIA_EJECT** User has expressed the desire to remove the external storage media.
38	**ACTION_MEDIA_REMOVED** External media has been removed.
39	**ACTION_NEW_OUTGOING_CALL** An outgoing call is about to be placed.
40	**ACTION_PASTE** Create a new item in the given container, initializing it from the current contents of the clipboard.
41	**ACTION_POWER_CONNECTED** External power has been connected to the device.
42	**ACTION_REBOOT** Have the device reboot. This is only for use by system code.
43	**ACTION_RUN** Run the data, whatever that means.
44	**ACTION_SCREEN_OFF** Sent after the screen turns off.
45	**ACTION_SCREEN_ON** Sent after the screen turns on.
46	**ACTION_SEARCH** Perform a search.
47	**ACTION_SEND** Deliver some data to someone else.
48	**ACTION_SENDTO** Send a message to someone specified by the data.
49	**ACTION_SEND_MULTIPLE** Deliver multiple data to someone else.
50	**ACTION_SET_WALLPAPER** Show settings for choosing wallpaper.
51	**ACTION_SHUTDOWN** Device is shutting down.
52	**ACTION_SYNC** Perform a data synchronization.
53	**ACTION_TIMEZONE_CHANGED** The timezone has changed.
54	**ACTION_TIME_CHANGED** The time was set.
55	**ACTION_VIEW** Display the data to the user.

56	**ACTION_VOICE_COMMAND** Start Voice Command.
57	**ACTION_WALLPAPER_CHANGED** The current system wallpaper has changed.
58	**ACTION_WEB_SEARCH** Perform a web search.

The action in an Intent object can be set by the setAction() method and read by getAction().

DATA

The URI of the data to be acted on and the MIME type of that data. For example, if the action field is ACTION_EDIT, the data field would contain the URI of the document to be displayed for editing.

The setData() method specifies data only as a URI, setType() specifies it only as a MIME type, and setDataAndType() specifies it as both a URI and a MIME type. The URI is read by getData() and the type by getType().

Some examples of action/data pairs are:

S.N.	Action/Data Pair & Description
1	**ACTION_VIEW content://contacts/people/1** Display information about the person whose identifier is "1".
2	**ACTION_DIAL content://contacts/people/1** Display the phone dialer with the person filled in.
3	**ACTION_VIEW tel:123** Display the phone dialer with the given number filled in.
4	**ACTION_DIAL tel:123** Display the phone dialer with the given number filled in.
5	**ACTION_EDIT content://contacts/people/1** Edit information about the person whose identifier is "1".
6	**ACTION_VIEW content://contacts/people/** Display a list of people, which the user can browse through.

CATEGORY

The category is an optional part of Intent object and it's a string containing additional information about the kind of component that should handle the intent. The addCategory() method places a category in an Intent object, removeCategory() deletes a category previously added, and getCategories() gets the set of all categories currently in the object. Here is a list of Android Intent Standard Categories.

Android Intent Standard Categories

Following table lists down various important Android Intent Standard Categories. You can check Android Official Documentation for a complete list of Categories:

S.N.	Categories & Description
1	**CATEGORY_APP_BROWSER**

	Used with ACTION_MAIN to launch the browser application.
2	**CATEGORY_APP_CALCULATOR** Used with ACTION_MAIN to launch the calculator application.
3	**CATEGORY_APP_CALENDAR** Used with ACTION_MAIN to launch the calendar application.
4	**CATEGORY_APP_CONTACTS** Used with ACTION_MAIN to launch the contacts application.
5	**CATEGORY_APP_EMAIL** Used with ACTION_MAIN to launch the email application.
6	**CATEGORY_APP_GALLERY** Used with ACTION_MAIN to launch the gallery application.
7	**CATEGORY_APP_MAPS** Used with ACTION_MAIN to launch the maps application.
8	**CATEGORY_APP_MARKET** This activity allows the user to browse and download new applications.
9	**CATEGORY_APP_MESSAGING** Used with ACTION_MAIN to launch the messaging application.
10	**CATEGORY_APP_MUSIC** Used with ACTION_MAIN to launch the music application.
11	**CATEGORY_BROWSABLE** Activities that can be safely invoked from a browser must support this category.
12	**CATEGORY_CAR_DOCK** An activity to run when device is inserted into a car dock.
13	**CATEGORY_CAR_MODE** Used to indicate that the activity can be used in a car environment.
14	**CATEGORY_DEFAULT** Set if the activity should be an option for the default action (center press) to perform on a piece of data.
15	**CATEGORY_DESK_DOCK** An activity to run when device is inserted into a car dock.
16	**CATEGORY_DEVELOPMENT_PREFERENCE** This activity is a development preference panel.
17	**CATEGORY_EMBED** Capable of running inside a parent activity container.
18	**CATEGORY_FRAMEWORK_INSTRUMENTATION_TEST** To be used as code under test for framework instrumentation tests.
19	**CATEGORY_HE_DESK_DOCK** An activity to run when device is inserted into a digital (high end) dock.
20	**CATEGORY_HOME** This is the home activity, that is the first activity that is displayed when the device boots.

21	**CATEGORY_INFO** Provides information about the package it is in.
22	**CATEGORY_LAUNCHER** Should be displayed in the top-level launcher.
23	**CATEGORY_LE_DESK_DOCK** An activity to run when device is inserted into a analog (low end) dock.
24	**CATEGORY_MONKEY** This activity may be exercised by the monkey or other automated test tools.
25	**CATEGORY_OPENABLE** Used to indicate that a GET_CONTENT intent only wants URIs that can be opened with ContentResolver.openInputStream.
26	**CATEGORY_PREFERENCE** This activity is a preference panel.
27	**CATEGORY_TAB** Intended to be used as a tab inside of a containing TabActivity.
28	**CATEGORY_TEST** To be used as a test (not part of the normal user experience).
29	**CATEGORY_UNIT_TEST** To be used as a unit test (run through the Test Harness).

You can check detail on Intent Filters in below section to understand how do we use categories to choose appropriate acivity coressponding to an Intent.

EXTRAS

This will be in key-value pairs for additional information that should be delivered to the component handling the intent. The extras can be set and read using the putExtras() and getExtras() methods respectively. Here is a list of Android Intent Standard Extra Data

Android Intent standard Extra Data

Following table lists down various important Android Intent Standard Extra Data. You can check Android Official Documentation for a complete list of Extra Data:

S.N.	Extra Data & Description
1	**EXTRA_ALARM_COUNT** Used as an int extra field in AlarmManager intents to tell the application being invoked how many pending alarms are being delievered with the intent.
2	**EXTRA_ALLOW_MULTIPLE** Used to indicate that a ACTION_GET_CONTENT intent can allow the user to select and return multiple

	items.
3	**EXTRA_ALLOW_REPLACE** Used as a boolean extra field with ACTION_INSTALL_PACKAGE to install a package.
4	**EXTRA_BCC** A String[] holding e-mail addresses that should be blind carbon copied.
5	**EXTRA_CC** A String[] holding e-mail addresses that should be carbon copied.
6	**EXTRA_CHANGED_COMPONENT_NAME_LIST** This field is part of ACTION_PACKAGE_CHANGED, and contains a string array of all of the components that have changed.
7	**EXTRA_DATA_REMOVED** Used as a boolean extra field in ACTION_PACKAGE_REMOVED intents to indicate whether this represents a full uninstall or a partial uninstall
8	**EXTRA_DOCK_STATE** Used as an int extra field in ACTION_DOCK_EVENT intents to request the dock state.
9	**EXTRA_DOCK_STATE_CAR** Used as an int value for EXTRA_DOCK_STATE to represent that the phone is in a car dock.
10	**EXTRA_DOCK_STATE_DESK** Used as an int value for EXTRA_DOCK_STATE to represent that the phone is in a desk dock.
11	**EXTRA_EMAIL** A String[] holding e-mail addresses that should be delivered to.
12	**EXTRA_HTML_TEXT** A constant String that is associated with the Intent, used with ACTION_SEND to supply an alternative to EXTRA_TEXT as HTML formatted text.
13	**EXTRA_INTENT** An Intent describing the choices you would like shown with ACTION_PICK_ACTIVITY.
14	**EXTRA_KEY_EVENT** A KeyEvent object containing the event that triggered the creation of the Intent it is in.
15	**EXTRA_LOCAL_ONLY** Used to indicate that a ACTION_GET_CONTENT intent should only return data that is on the local device.
16	**EXTRA_ORIGINATING_URI** Used as a URI extra field with ACTION_INSTALL_PACKAGE and ACTION_VIEW to indicate the URI from which the local APK in the Intent data field originated from.
17	**EXTRA_PHONE_NUMBER** A String holding the phone number originally entered in ACTION_NEW_OUTGOING_CALL, or the actual number to call in a ACTION_CALL.
18	**EXTRA_SHORTCUT_ICON** The name of the extra used to define the icon, as a Bitmap, of a shortcut.
19	**EXTRA_SHORTCUT_INTENT** The name of the extra used to define the Intent of a shortcut.

20	**EXTRA_SHORTCUT_NAME** The name of the extra used to define the name of a shortcut.
21	**EXTRA_STREAM** URI holding a stream of data associated with the Intent, used with ACTION_SEND to supply the data being sent.
22	**EXTRA_SUBJECT** A constant string holding the desired subject line of a message.
23	**EXTRA_TEMPLATE** The initial data to place in a newly created record. Use with ACTION_INSERT.
24	**EXTRA_TEXT** A constant CharSequence that is associated with the Intent, used with ACTION_SEND to supply the literal data to be sent.
25	**EXTRA_TITLE** A CharSequence dialog title to provide to the user when used with a ACTION_CHOOSER.
26	**EXTRA_UID** Used as an int extra field in ACTION_UID_REMOVED intents to supply the uid the package had been assigned.

FLAGS

These flags are optional part of Intent object and instruct the Android system how to launch an activity, and how to treat it after it's launched etc.

COMPONENT NAME

This optional field is an android **ComponentName** object representing either Activity, Service or BroadcastReceiver class. If it is set, the Intent object is delivered to an instance of the designated class otherwise Android uses other information in the Intent object to locate a suitable target.

The component name is set by setComponent(), setClass(), or setClassName() and read by getComponent().

Types of Intents

There are following two types of intents supported by Android till version 4.1

EXPLICIT INTENTS

These intents designate the target component by its name and they are typically used for application-internal messages - such as an activity starting a subordinate service or launching a sister activity. For example:

```
// Explicit Intent by specifying its class name
Intent i = new Intent(this, TargetActivity.class);
i.putExtra("Key1", "ABC");
i.putExtra("Key2", "123");

// Starts TargetActivity
```

```
startActivity(i);
```

IMPLICIT INTENTS

These intents do not name a target and the field for the component name is left blank. Implicit intents are often used to activate components in other applications. For example:

```
// Implicit Intent by specifying a URI
Intent i = new Intent(Intent.ACTION_VIEW,
Uri.parse("http://www.example.com"));

// Starts Implicit Activity
startActivity(i);
```

The target component which receives the intent can use the **getExtras()** method to get the extra data sent by the source component. For example:

```
// Get bundle object at appropriate place in your code
Bundle extras = getIntent().getExtras();

// Extract data using passed keys
String value1 = extras.getString("Key1");
String value2 = extras.getString("Key2");
```

Example

Following example shows the functionality of a Android Intent to launch various Android built-in applications.

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *IntentDemo* under a package *com.example.intentdemo*. While creating this project, make sure you *Target SDK*and *Compile With* at the latest version of Android SDK to use higher levels of APIs.
2	Modify *src/MainActivity.java* file and add the code to define two listeners corresponding two buttons ie. Start Browser and Start Phone.
3	Modify layout XML file *res/layout/activity_main.xml* to add three buttons in linear layout.
4	Modify *res/values/strings.xml* to define required constant values
5	Run the application to launch Android emulator and verify the result of the changes done in the aplication.

Following is the content of the modified main activity file**src/com.example.intentdemo/MainActivity.java**.

```
package com.example.intentdemo;

import android.net.Uri;
import android.os.Bundle;
import android.app.Activity;
import android.content.Intent;
import android.view.Menu;
import android.view.View;
import android.widget.Button;

public class MainActivity extends Activity {

   @Override
   protected void onCreate(Bundle savedInstanceState) {
```

```
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      Button startBrowser = (Button) findViewById(R.id.start_browser);
      startBrowser.setOnClickListener(new View.OnClickListener() {
         public void onClick(View view) {
            Intent i = new Intent(android.content.Intent.ACTION_VIEW,
            Uri.parse("http://www.example.com"));
            startActivity(i);
         }
      });
      Button startPhone = (Button) findViewById(R.id.start_phone);
      startPhone.setOnClickListener(new View.OnClickListener() {
         public void onClick(View view) {
            Intent i = new Intent(android.content.Intent.ACTION_VIEW,
            Uri.parse("tel:9510300000"));
            startActivity(i);
         }
      });
   }
   @Override
   public boolean onCreateOptionsMenu(Menu menu) {
      // Inflate the menu; this adds items to the action
      // bar if it is present.
      getMenuInflater().inflate(R.menu.main, menu);
      return true;
   }

}
```

Following will be the content of **res/layout/activity_main.xml** file:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:orientation="vertical" >

   <Button android:id="@+id/start_browser"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:text="@string/start_browser"/>

   <Button android:id="@+id/start_phone"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:text="@string/start_phone" />

</LinearLayout>
```

Following will be the content of **res/values/strings.xml** to define two new constants:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="app_name">IntentDemo</string>
    <string name="action_settings">Settings</string>
    <string name="hello_world">Hello world!</string>
    <string name="start_browser">Start Browser</string>
    <string name="start_phone">Start Phone</string>
```

```
</resources>
```

Following is the default content of **AndroidManifest.xml**:

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

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

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

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Let's try to run your **IntentDemo** application. I assume you had created your **AVD** while doing environment setup. To run the app from Eclipse, open one of your project's activity files and click Run icon from the toolbar. Eclipse installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display following Emulator window:

Now click on **Start Browser** button, which will start a browser configured and display http://www.example.com as shown below:

Similar way you can launch phone interface using Start Phone button, which will allow you to dial already given phone number.

Intent Filters

You have seen how an Intent has been used to call an another activity. Android OS uses filters to pinpoint the set of Activities, Services, and Broadcast receivers that can handle the Intent with help of specified set of action, categories, data scheme associated with an Intent. You will use **<intent-filter>**element in the manifest file to list down actions, categories and data types associated with any activity, service, or broadcast receiver.

Following is an example of a part of **AndroidManifest.xml** file to specify an activity**com.example.intentdemo.CustomActivity** which can be invoked by either of the two mentioned actions, one category, and one data:

```
<activity android:name=".CustomActivity"
   android:label="@string/app_name">
   <intent-filter>
      <action android:name="android.intent.action.VIEW" />
      <action android:name="com.example.intentdemo.LAUNCH" />
      <category android:name="android.intent.category.DEFAULT" />
      <data android:scheme="http" />
   </intent-filter>
</activity>
```

Once this activity is defined along with above mentioned filters, other activities will be able to invoke this activity using either the **android.intent.action.VIEW**, or using the **com.example.intentdemo.LAUNCH**action provided their category is **android.intent.category.DEFAULT**.

The **<data>** element specifies the data type expected by the activity to be called and for above example our custom activity expects the data to start with the "http://"

There may be a situation that an intent can pass through the filters of more than one activity or service, the user may be asked which component to activate. An exception is raised if no target can be found.

There are following test Android checks before invoking an activity:

- A filter <intent-filter> may list more than one action as shown above but this list cannot be empty; a filter must contain at least one <action> element, otherwise it will block all intents. If more than one actions are mentioned then Android tries to match one of the mentioned actions before invoking the activity.

- A filter <intent-filter> may list zero, one or more than one categories. if there is no category mentioned then Android always pass this test but if more than one categories are mentioned then for an intent to pass the category test, every category in the Intent object must match a category in the filter.

- Each <data> element can specify a URI and a data type (MIME media type). There are separate attributes like **scheme, host, port**, and **path** for each part of the URI. An Intent object that contains both a URI and a data type passes the data type part of the test only if its type matches a type listed in the filter.

Example

Following example is a modification of the above example. Here we will see how Android resolves conflict if one intent is invoking two activities defined in , next how to invoke a custom activity using a filter and third one is an exception case if Android does not file appropriate activity defined for an intent.

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *IntentDemo* under a package *com.example.intentdemo*. While creating this project, make sure you *Target SDK*and *Compile With* at the latest version of Android SDK to use higher levels of APIs.
2	Modify *src/MainActivity.java* file and add the code to define three listeners corresponding to three buttons defined in layout file.
3	Add a new *src/CustomActivity.java* file to have one custom activity which will be invoked by different intents.
4	Modify layout XML file *res/layout/activity_main.xml* to add three buttons in linear layout.
5	Add one layout XML file *res/layout/custom_view.xml* to add a simple <TextView> to show the passed data through intent.
6	Modify *res/values/strings.xml* to define required constant values
7	Modify *AndroidManifest.xml* to add <intent-filter> to define rules for your intent to invoke custom activity.
8	Run the application to launch Android emulator and verify the result of the changes done in the aplication.

Following is the content of the modified main activity file**src/com.example.intentdemo/MainActivity.java**.

```
package com.example.intentdemo;

import android.net.Uri;
import android.os.Bundle;
import android.app.Activity;
import android.content.Intent;
import android.view.Menu;
import android.view.View;
import android.widget.Button;

public class MainActivity extends Activity {

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
```

```
      setContentView(R.layout.activity_main);

      // First intent to use ACTION_VIEW action with correct data
      Button startBrowser_a = (Button) findViewById(R.id.start_browser_a);
      startBrowser_a.setOnClickListener(new View.OnClickListener() {
         public void onClick(View view) {
            Intent i = new Intent(android.content.Intent.ACTION_VIEW,
            Uri.parse("http://www.example.com"));
            startActivity(i);
         }
      });

      // Second intent to use LAUNCH action with correct data
      Button startBrowser_b = (Button) findViewById(R.id.start_browser_b);
      startBrowser_b.setOnClickListener(new View.OnClickListener() {
         public void onClick(View view) {
            Intent i = new Intent("com.example.intentdemo.LAUNCH",
            Uri.parse("http://www.example.com"));
            startActivity(i);
         }
      });

      // Third intent to use LAUNCH action with incorrect data
      Button startBrowser_c = (Button) findViewById(R.id.start_browser_c);
      startBrowser_c.setOnClickListener(new View.OnClickListener() {
         public void onClick(View view) {
            Intent i = new Intent("com.example.intentdemo.LAUNCH",
            Uri.parse("https://www.example.com"));
            startActivity(i);
         }
      });

   }

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

}
```

Following is the content of the modified main activity file**src/com.example.intentdemo/CustomActivity.java**.

```
package com.example.intentdemo;

import android.app.Activity;
import android.net.Uri;
import android.os.Bundle;
import android.widget.TextView;

public class CustomActivity extends Activity {
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.custom_view);

      TextView label = (TextView) findViewById(R.id.show_data);
```

```
        Uri url = getIntent().getData();
        label.setText(url.toString());
    }

}
```

Following will be the content of **res/layout/activity_main.xml** file:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:orientation="vertical" >

   <Button android:id="@+id/start_browser_a"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:text="@string/start_browser_a"/>

   <Button android:id="@+id/start_browser_b"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:text="@string/start_browser_b"/>

   <Button android:id="@+id/start_browser_c"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:text="@string/start_browser_c"/>

</LinearLayout>
```

Following will be the content of **res/layout/custom_view.xml** file:

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

   <TextView android:id="@+id/show_data"
   android:layout_width="fill_parent"
   android:layout_height="400dp"/>

</LinearLayout>
```

Following will be the content of **res/values/strings.xml** to define two new constants:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="app_name">IntentDemo</string>
    <string name="action_settings">Settings</string>
    <string name="hello_world">Hello world!</string>
    <string name="start_browser_a">Start Browser with VIEW action</string>
    <string name="start_browser_b">Start Browser with LAUNCH action</string>
    <string name="start_browser_c">Exception Condition</string>

</resources>
```

Following is the default content of **AndroidManifest.xml**:

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

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.intentdemo.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name="com.example.intentdemo.CustomActivity"
            android:label="@string/app_name">
           <intent-filter>
              <action android:name="android.intent.action.VIEW" />
              <action android:name="com.example.intentdemo.LAUNCH" />
              <category android:name="android.intent.category.DEFAULT" />
              <data android:scheme="http" />
           </intent-filter>
        </activity>
    </application>

</manifest>
```

Let's try to run your **IntentDemo** application. I assume you had created your **AVD** while doing environment setup. To run the app from Eclipse, open one of your project's activity files and click Run icon from the toolbar. Eclipse installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display following Emulator window:

Now let's start with first button "Start Browser with VIEW Action". Here we have defined our custom activity with a filter "android.intent.action.VIEW", and there is already one default activity against VIEW action defined by Android which is launching web browser, So android displays following two options to select the activity you want to launch.

Now if you select Browser, then Android will launch web browser and open example.com website but if you select IndentDemo option then Android will launch CustomActivity which does nothing but just capture passed data and displays in a text view as follows:

Now go back using back button and click on "Start Browser with LAUNCH Action" button, here Android applies filter to choose define activity and it simply launch your custom activity and again it displays following screen:

Again, go back using back button and click on "Exception Condition" button, here Android tries to find out a valid filter for the given intent but it does not find a valid activity defined because this time we have used data as **https** instead of **http** though we are giving a correct action, so Android raises an exception and shows following screen:

CHAPTER

15

Event Handling

Events are a useful way to collect data about a user's interaction with interactive components of your app, like button presses or screen touch etc. The Android framework maintains an event queue into which events are placed as they occur and then each event is removed from the queue on a first-in, first-out (FIFO) basis. You can capture these events in your program and take appropriate action as per requirements.

There are following three concepts related to Android Event Management:

- **Event Listeners:** The **View** class is mainly involved in building up a Android GUI, same View class provides a number of Event Listeners. The Event Listener is the object that receives notification when an event happes.
- **Event Listeners Registration:** Event Registration is the process by which an Event Handler gets registered with an Event Listener so that the handler is called when the Event Listener fires the event.
- **Event Handlers:** When an event happens and we have registered and event listener fo the event, the event listener calls the Event Handlers, which is the method that actually handles the event.

Event Listeners & Event Handlers

Event Handler	Event Listener & Description
onClick()	**OnClickListener()** This is called when the user either clicks or touches or focuses upon any widget like button, text, image etc. You will use onClick() event handler to handle such event.
onLongClick()	**OnLongClickListener()** This is called when the user either clicks or touches or focuses upon any widget like button, text, image etc. for one or more seconds. You will use onLongClick() event handler to handle such event.
onFocusChange()	**OnFocusChangeListener()** This is called when the widget looses its focus ie. user goes away from the view item. You will use onFocusChange() event handler to handle such event.
onKey()	**OnFocusChangeListener()** This is called when the user is focused on the item and presses or releases a hardware key on the device. You will use onKey() event handler to handle such event.
onTouch()	**OnTouchListener()** This is called when the user presses the key, releases the key, or any movement gesture on the screen. You will use onTouch() event handler to handle such event.

onMenuItemClick()	**OnMenuItemClickListener()** This is called when the user selects a menu item. You will use onMenuItemClick() event handler to handle such event.

There are many more event listeners available as a part of **View** class like OnHoverListener, OnDragListener etc which may be needed for your application. So I recommend to refer official documentation for Android application development in case you are going to develop a sophisticated apps.

Event Listeners Registration:

Event Registration is the process by which an Event Handler gets registered with an Event Listener so that the handler is called when the Event Listener fires the event. Though there are several tricky ways to register your event listener for any event, but I'm going to list down only top 3 ways, out of which you can use any of them based on the situation.

- Using an Anonymous Inner Class
- Activity class implements the Listener interface.
- Using Layout file activity_main.xml to specify event handler directly.

Below section will provide you detailed examples on all the three scenarios:

Event Handling Examples

EVENT LISTENERS REGISTRATION USING AN ANONYMOUS INNER CLASS

Here you will create an anonymous implementation of the listener and will be useful if each class is applied to a single control only and you have advantage to pass arguments to event handler. In this approach event handler methods can access private data of Activity. No reference is needed to call to Activity.

But if you applied the handler to more than one control, you would have to cut and paste the code for the handler and if the code for the handler is long, it makes the code harder to maintain.

Following are the simple steps to show how we will make use of separate Listener class to register and capture click event. Similar way you can implement your listener for any other required event type.

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *EventDemo* under a package *com.example.eventdemo* as explained in the *Hello World Example* chapter.
2	Modify *src/MainActivity.java* file to add click event listeners and handlers for the two buttons defined.
3	Modify the detault content of *res/layout/activity_main.xml* file to include Android UI controls.
4	Define required constants in *res/values/strings.xml* file
5	Run the application to launch Android emulator and verify the result of the changes done in the aplication.

Following is the content of the modified main activity file**src/com.example.eventdemo/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.eventdemo;

import android.os.Bundle;
```

```
import android.app.Activity;
import android.view.Menu;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends Activity {

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

        //--- find both the buttons---
        Button sButton = (Button) findViewById(R.id.button_s);
        Button lButton = (Button) findViewById(R.id.button_l);

        // -- register click event with first button ---
        sButton.setOnClickListener(new View.OnClickListener() {
           public void onClick(View v) {
               // --- find the text view --
               TextView txtView = (TextView) findViewById(R.id.text_id);
               // -- change text size --
               txtView.setTextSize(14);
           }
        });

        // -- register click event with second button ---
        lButton.setOnClickListener(new View.OnClickListener() {
           public void onClick(View v) {
               // --- find the text view --
               TextView txtView = (TextView) findViewById(R.id.text_id);
               // -- change text size --
               txtView.setTextSize(24);
           }
        });
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

}
```

Following will be the content of **res/layout/activity_main.xml** file:

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


    <Button
    android:id="@+id/button_s"
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:text="@string/button_small"/>
```

```
    <Button
    android:id="@+id/button_l"
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:text="@string/button_large"/>

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

</LinearLayout>
```

Following will be the content of **res/values/strings.xml** to define two new constants:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="app_name">EventDemo</string>
    <string name="action_settings">Settings</string>
    <string name="hello_world">Hello world!</string>
   <string name="button_small">Small Font</string>
   <string name="button_large">Large Font</string>

</resources>
```

Following is the default content of **AndroidManifest.xml**:

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

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

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

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Let's try to run your **EventDemo** application. I assume you had created your **AVD** while doing environment setup. To run the app from Eclipse, open one of your project's activity files and click Run icon from the toolbar. Eclipse

installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display following Emulator window:

Now you try to click on two buttons one by one and you will see that font of the **Hello World** text will change, which happens because registered click event handler method is being called against each click event.

REGISTRATION USING THE ACTIVITY IMPLEMENTS LISTENER INTERFACE

Here your Activity class implements the Listener interface and you put the handler method in the main Activity and then you call setOnClickListener(this).

This approach is fine if your application has only a single control of that Listener type otherwise you will have to do further programming to check which control has generated event. Second you cannot pass arguments to the Listener so, again, works poorly for multiple controls.

Following are the simple steps to show how we will implement Listener class to register and capture click event. Similar way you can implement your listener for any other required event type.

Step	Description
1	We do not need to create this application from scratch, so let's make use of above created Android application *EventDemo*.
2	Modify *src/MainActivity.java* file to add click event listeners and handlers for the two buttons defined.
3	We are not making any change in *res/layout/activity_main.xml*, it will remain as shown above.
4	We are also not making any change in *res/values/strings.xml* file, it will also remain as shown above.
5	Run the application to launch Android emulator and verify the result of the changes done in the aplication.

Following is the content of the modified main activity file**src/com.example.eventdemo/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.eventdemo;

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

```
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends Activity implements OnClickListener {

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

        //--- find both the buttons---
        Button sButton = (Button) findViewById(R.id.button_s);
        Button lButton = (Button) findViewById(R.id.button_l);


        // -- register click event with first button ---
        sButton.setOnClickListener(this);
        // -- register click event with second button ---
        lButton.setOnClickListener(this);
    }

    //--- Implement the OnClickListener callback
    public void onClick(View v) {
       if(v.getId() == R.id.button_s)
       {
            // --- find the text view --
            TextView txtView = (TextView) findViewById(R.id.text_id);
            // -- change text size --
            txtView.setTextSize(14);
            return;
       }
       if(v.getId() == R.id.button_l)
       {
            // --- find the text view --
            TextView txtView = (TextView) findViewById(R.id.text_id);
            // -- change text size --
            txtView.setTextSize(24);
            return;
       }
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

}
```

Now again let's try to run your **EventDemo** application. I assume you had created your **AVD** while doing environment setup. To run the app from Eclipse, open one of your project's activity files and click Run icon from the toolbar. Eclipse installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display following Emulator window:

Now you try to click on two buttons one by one and you will see that font of the **Hello World** text will change, which happens because registered click event handler method is being called against each click event.

REGISTRATION USING LAYOUT FILE ACTIVITY_MAIN.XML

Here you put your event handlers in Activity class without implementing a Listener interface or call to any listener method. Rather you will use the layout file (activity_main.xml) to specify the handler method via the **android:onClick** attribute for click event. You can control click events differently for different control by passing different event handler methods.

The event handler method must have a void return type and take a View as an argument. However, the method name is arbitrary, and the main class need not implement any particular interface.

This approach does not allow you to pass arguments to Listener and for the Android developers it will be difficult to know which method is the handler for which control until they look into activity_main.xml file. Second, you can not handle any other event except click event using this approach.

Following are the simple steps to show how we can make use of layout file Main.xml to register and capture click event.

Step	Description
1	We do not need to create this application from scratch, so let's make use of above created Android application *EventDemo*.
2	Modify *src/MainActivity.java* file to add click event listeners and handlers for the two buttons defined.
3	Modify layout file *res/layout/activity_main.xml*, to specify event handlers for the two buttons.
4	We are also not making any change in *res/values/strings.xml* file, it will also remain as shown above.
5	Run the application to launch Android emulator and verify the result of the changes done in the aplication.

Following is the content of the modified main activity file**src/com.example.eventdemo/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.eventdemo;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
```

```
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends Activity{

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

    //--- Implement the event handler for the first button.
    public void doSmall(View v)  {
       // --- find the text view --
       TextView txtView = (TextView) findViewById(R.id.text_id);
       // -- change text size --
       txtView.setTextSize(14);
       return;
   }
   //--- Implement the event handler for the second button.
   public void doLarge(View v)  {
       // --- find the text view --
       TextView txtView = (TextView) findViewById(R.id.text_id);
       // -- change text size --
       txtView.setTextSize(24);
       return;
   }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

}
```

Following will be the content of **res/layout/activity_main.xml** file. Here we have to add**android:onClick="methodName"** for both the buttons, which will register given method names as click event handlers.

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


    <Button
    android:id="@+id/button_s"
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:text="@string/button_small"
    android:onClick="doSmall"/>

    <Button
    android:id="@+id/button_l"
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:text="@string/button_large"
    android:onClick="doLarge"/>
```

```
    <TextView
    android:id="@+id/text_id"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:capitalize="characters"
    android:text="@string/hello_world" />

</LinearLayout>
```

Again let's try to run your **EventDemo** application. I assume you had created your **AVD** while doing environment setup. To run the app from Eclipse, open one of your project's activity files and click Run icon from the toolbar. Eclipse installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display following Emulator window:

Now you try to click on two buttons one by one and you will see that font of the **Hello World** text will change, which happens because registered click event handler method is being called against each click event.

Exercise:

I will recommend to try writing different event handlers for different event types and understand exact difference in different event types and their handling. Events related to menu, spinner, pickers widgets are little different but they are also based on the same concepts as explained above.

Thread

A thread is a concurrent unit of execution. It has its own call stack. There are two methods to implement threads in applications.

One is providing a new class that extends Thread and overriding its run() method. The other is providing a new Thread instance with a Runnable object during its creation. A thread can be executed by calling its "start" method. You can set the "Priority" of a thread by calling its "setPriority(int)" method.

A thread can be used if you have no effect in the UI part. For example, you are calling some web service or download some data, and after download, you are displaying it to your screen. Then you need to use a Handler with a Thread and this will make your application complicated to handle all the responses from Threads.

A Handler allows you to send and process Message and Runnable objects associated with a thread's MessageQueue. Each thread has each message queue. (Like a to do List), and the thread will take each message and process it until the message queue is empty. So, when the Handler communicates, it just gives a message to the caller thread and it will wait to process.

AsyncTask

AsyncTask enables proper and easy use of the UI thread. This class allows performing background operations and publishing results on the UI thread without having to manipulate threads and/or handlers. An asynchronous task is defined by a computation that runs on a background thread and whose result is published on the UI thread.

AsyncTask will go through the following 4 stages:

onPreExecute()

Invoked on the UI thread before the task is executed

doInbackground(Params..)

Invoked on the background thread immediately after onPreExecute() finishes executing.

onProgressUpdate(Progress..)

Invoked on the UI thread after a call to publishProgress(Progress...).

onPostExecute(Result)

Invoked on the UI thread after the background computation finishes.

Android - Notifications

> 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.

Android **Toast** class provides a handy way to show users alerts but problem is that these alerts are not persistent which means alert flashes on the screen for a few seconds and then disappears.

To see the details of the notification, you will have to select the icon which will display notification drawer having detail about the notification. While working with emulator with virtual device, you will have to click and drag down the status bar to expand it which will give you detail as follows. This will be just **64 dp** tall and called normal view.

Above expanded form can have a **Big View** which will have additional detail about the notification. You can add upto six additional lines in the notification. The following screen shot shows such notification.

Create and Send Notifications

You have simple way to create a notification. Follow the following steps in your application to create a notification −

Step 1 - Create Notification Builder

As a first step is to create a notification builder using *NotificationCompat.Builder.build()*. You will use Notification Builder to set various Notification properties like its small and large icons, title, priority etc.

```
NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this)
```

Step 2 - Setting Notification Properties

Once you have **Builder** object, you can set its Notification properties using Builder object as per your requirement. But this is mandatory to set at least following −

- A small icon, set by **setSmallIcon()**
- A title, set by **setContentTitle()**
- Detail text, set by **setContentText()**

```
mBuilder.setSmallIcon(R.drawable.notification_icon);
mBuilder.setContentTitle("Notification Alert, Click Me!");
mBuilder.setContentText("Hi, This is Android Notification Detail!");
```

You have plenty of optional properties which you can set for your notification. To learn more about them, see the reference documentation for NotificationCompat.Builder.

Step 3 - Attach Actions

This is an optional part and required if you want to attach an action with the notification. An action allows users to go directly from the notification to an **Activity** in your application, where they can look at one or more events or do further work.

The action is defined by a **PendingIntent** containing an **Intent** that starts an Activity in your application. To associate the PendingIntent with a gesture, call the appropriate method of *NotificationCompat.Builder*. For example, if you want to start Activity when the user clicks the notification text in the notification drawer, you add the PendingIntent by calling **setContentIntent()**.

A PendingIntent object helps you to perform an action on your applications behalf, often at a later time, without caring of whether or not your application is running.

We take help of stack builder object which 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.

```
Intent resultIntent = new Intent(this, ResultActivity.class);
TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
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_CU
mBuilder.setContentIntent(resultPendingIntent);
```

Step 4 - Issue the notification

Finally, you pass the Notification object to the system by calling NotificationManager.notify() to send your notification. Make sure you call **NotificationCompat.Builder.build()** method on builder object before

notifying it. This method combines all of the options that have been set and return a new **Notification** object.

```
NotificationManager mNotificationManager = (NotificationManager) getSystemService(Context.NOTIFI

// notificationID allows you to update the notification later on.
mNotificationManager.notify(notificationID, mBuilder.build());
```

The NotificationCompat.Builder Class

The NotificationCompat.Builder class allows easier control over all the flags, as well as help constructing the typical notification layouts. Following are few important and most frequently used methods available as a part of NotificationCompat.Builder class.

Sr.No.	Constants & Description
1	**Notification build()** Combine all of the options that have been set and return a new Notification object.
2	**NotificationCompat.Builder setAutoCancel (boolean autoCancel)** Setting this flag will make it so the notification is automatically canceled when the user clicks it in the panel.
3	**NotificationCompat.Builder setContent (RemoteViews views)** Supply a custom RemoteViews to use instead of the standard one.
4	**NotificationCompat.Builder setContentInfo (CharSequence info)** Set the large text at the right-hand side of the notification.
5	**NotificationCompat.Builder setContentIntent (PendingIntent intent)** Supply a PendingIntent to send when the notification is clicked.
6	**NotificationCompat.Builder setContentText (CharSequence text)** Set the text (second row) of the notification, in a standard notification.
7	**NotificationCompat.Builder setContentTitle (CharSequence title)** Set the text (first row) of the notification, in a standard notification.
8	**NotificationCompat.Builder setDefaults (int defaults)** Set the default notification options that will be used.
9	**NotificationCompat.Builder setLargeIcon (Bitmap icon)** Set the large icon that is shown in the ticker and notification.
10	**NotificationCompat.Builder setNumber (int number)** Set the large number at the right-hand side of the notification.
11	**NotificationCompat.Builder setOngoing (boolean ongoing)** Set whether this is an ongoing notification.

12	**NotificationCompat.Builder setSmallIcon (int icon)** Set the small icon to use in the notification layouts.
13	**NotificationCompat.Builder setStyle (NotificationCompat.Style style)** Add a rich notification style to be applied at build time.
14	**NotificationCompat.Builder setTicker (CharSequence tickerText)** Set the text that is displayed in the status bar when the notification first arrives.
15	**NotificationCompat.Builder setVibrate (long[] pattern)** Set the vibration pattern to use.
16	**NotificationCompat.Builder setWhen (long when)** Set the time that the event occurred. Notifications in the panel are sorted by this time.

Example

Following example shows the functionality of a Android notification using a **NotificationCompat.Builder** Class which has been introduced in Android 4.1.

Step	Description
1	You will use Android studio IDE to create an Android application and name it as *tutorialspoint* under a package *com.example.notificationdemo*.
2	Modify *src/MainActivity.java* file and add the code to notify(""), if user click on the button,it will call android notification service.
3	Create a new Java file *src/NotificationView.java*, which will be used to display new layout as a part of new activity which will be started when user will click any of the notifications
4	Modify layout XML file *res/layout/activity_main.xml* to add Notification button in relative layout.
5	Create a new layout XML file *res/layout/notification.xml*. This will be used as layout file for new activity which will start when user will click any of the notifications.
6	No need to change default string constants. Android studio takes care of default string constants
7	Run the application to launch Android emulator and verify the result of the changes done in the application.

Android - Phone Calls

Android provides Built-in applications for phone calls, in some occasions we may need to make a phone call through our application. This could easily be done by using implicit Intent with appropriate actions. Also, we can use PhoneStateListener and TelephonyManager classes, in order to monitor the changes in some telephony states on the device.

This chapter lists down all the simple steps to create an application which can be used to make a Phone Call. You can use Android Intent to make phone call by calling built-in Phone Call functionality of the Android. Following section explains different parts of our Intent object required to make a call.

Intent Object - Action to make Phone Call

You will use **ACTION_CALL** action to trigger built-in phone call functionality available in Android device. Following is simple syntax to create an intent with ACTION_CALL action

```
Intent phoneIntent = new Intent(Intent.ACTION_CALL);
```

You can use **ACTION_DIAL** action instead of ACTION_CALL, in that case you will have option to modify hardcoded phone number before making a call instead of making a direct call.

Intent Object - Data/Type to make Phone Call

To make a phone call at a given number 91-000-000-0000, you need to specify **tel:** as URI using setData() method as follows −

```
phoneIntent.setData(Uri.parse("tel:91-000-000-0000"));
```

The interesting point is that, to make a phone call, you do not need to specify any extra data or data type.

Example

Following example shows you in practical how to use Android Intent to make phone call to the given mobile number.

> To experiment with this example, you will need actual Mobile device equipped with latest Android OS, otherwise you will have to struggle with emulator which may not work.

Step	Description
1	You will use Android studio IDE to create an Android application and name it as *My Application* under a package *com.example.saira_000.myapplication*.
2	Modify *src/MainActivity.java* file and add required code to take care of making a call.
3	Modify layout XML file *res/layout/activity_main.xml* add any GUI component if required. I'm adding a simple button to Call 91-000-000-0000 number
4	No need to define default string constants.Android studio takes care of default constants.
5	Modify *AndroidManifest.xml* as shown below
6	Run the application to launch Android emulator and verify the result of the changes done in the application.

Following is the content of the modified main activity file **src/MainActivity.java**.

```
package com.example.saira_000.myapplication;

import android.Manifest;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.net.Uri;
import android.os.Bundle;
import android.support.v4.app.ActivityCompat;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.Button;

public class MainActivity extends AppCompatActivity {
   private Button button;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      button = (Button) findViewById(R.id.buttonCall);

      button.setOnClickListener(new View.OnClickListener() {
         public void onClick(View arg0) {
            Intent callIntent = new Intent(Intent.ACTION_CALL);
            callIntent.setData(Uri.parse("tel:0377778888"));

            if (ActivityCompat.checkSelfPermission(MainActivity.this,
               Manifest.permission.CALL_PHONE) != PackageManager.PERMISSION_GRANTED) {
                  return;
               }
               startActivity(callIntent);
```

```
        }
    });

    }
}
```

Following will be the content of **res/layout/activity_main.xml** file −

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

   <Button
      android:id="@+id/buttonCall"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="call 0377778888" />

</LinearLayout>
```

Following will be the content of **res/values/strings.xml** to define two new constants −

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="app_name">My Application</string>
</resources>
```

Following is the default content of **AndroidManifest.xml** −

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.saira_000.myapplication" >

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

   <application
      android:allowBackup="true"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name"
      android:theme="@style/AppTheme" >

      <activity
         android:name="com.example.saira_000.myapplication.MainActivity"
         android:label="@string/app_name" >

         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
```

```
        </activity>

    </application>
</manifest>
```

Let's try to run your **My Application** application. I assume you have connected your actual Android Mobile device with your computer. To run the app from Android studio, open one of your project's activity files and click Run icon from the toolbar.Select your mobile device as an option and then check your mobile device which will display following screen −

Now use **Call** button to make phone call as shown below −

11:23
Dialling
0377778888

Android - Sending Email

> **Email** is messages distributed by electronic means from one system user to one or more recipients via a network.

Before starting Email Activity, You must know Email functionality with intent, Intent is carrying data from one component to another component with-in the application or outside the application.

To send an email from your application, you don't have to implement an email client from the beginning, but you can use an existing one like the default Email app provided from Android, Gmail, Outlook, K-9 Mail etc. For this purpose, we need to write an Activity that launches an email client, using an implicit Intent with the right action and data. In this example, we are going to send an email from our app by using an Intent object that launches existing email clients.

Following section explains different parts of our Intent object required to send an email.

Intent Object - Action to send Email

You will use **ACTION_SEND** action to launch an email client installed on your Android device. Following is simple syntax to create an intent with ACTION_SEND action.

```
Intent emailIntent = new Intent(Intent.ACTION_SEND);
```

Intent Object - Data/Type to send Email

To send an email you need to specify **mailto:** as URI using setData() method and data type will be to **text/plain** using setType() method as follows −

```
emailIntent.setData(Uri.parse("mailto:"));
emailIntent.setType("text/plain");
```

Intent Object - Extra to send Email

Android has built-in support to add TO, SUBJECT, CC, TEXT etc. fields which can be attached to the intent before sending the intent to a target email client. You can use following extra fields in your email −

Sr.No.	Extra Data & Description
1	**EXTRA_BCC** A String[] holding e-mail addresses that should be blind carbon copied.
2	**EXTRA_CC** A String[] holding e-mail addresses that should be carbon copied.
3	**EXTRA_EMAIL** A String[] holding e-mail addresses that should be delivered to.
4	**EXTRA_HTML_TEXT** A constant String that is associated with the Intent, used with ACTION_SEND to supply an alternative to EXTRA_TEXT as HTML formatted text.
5	**EXTRA_SUBJECT** A constant string holding the desired subject line of a message.
6	**EXTRA_TEXT** A constant CharSequence that is associated with the Intent, used with ACTION_SEND to supply the literal data to be sent.
7	**EXTRA_TITLE** A CharSequence dialog title to provide to the user when used with a ACTION_CHOOSER.

Here is an example showing you how to assign extra data to your intent −

```
emailIntent.putExtra(Intent.EXTRA_EMAIL  , new String[]{"Recipient"});
emailIntent.putExtra(Intent.EXTRA_SUBJECT, "subject");
emailIntent.putExtra(Intent.EXTRA_TEXT   , "Message Body");
```

The out-put of above code is as below shown an image

Email Example

Example

Following example shows you in practical how to use Intent object to launch Email client to send an Email to the given recipients.

> To Email experiment with this example, you will need actual Mobile device equipped with latest Android OS, otherwise you might get struggle with emulator which may not work properly. Second you will need to have an Email client like GMail(By default every android version having Gmail client App) or K9mail installed on your device.

Step	Description
1	You will use Android studio to create an Android application and name it as *Tutorialspoint* under a package *com.example.tutorialspoint*.
2	Modify *src/MainActivity.java* file and add required code to take care of sending email.
3	Modify layout XML file *res/layout/activity_main.xml* add any GUI component if required. I'm adding a simple button to launch Email Client.
4	Modify *res/values/strings.xml* to define required constant values
5	Modify *AndroidManifest.xml* as shown below
6	Run the application to launch Android emulator and verify the result of the changes done in the application.

Following is the content of the modified main activity file **src/com.example.Tutorialspoint/MainActivity.java**.

```
package com.example.tutorialspoint;

import android.net.Uri;
import android.os.Bundle;
import android.app.Activity;
import android.content.Intent;
import android.util.Log;
import android.view.Menu;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;

public class MainActivity extends Activity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      Button startBtn = (Button) findViewById(R.id.sendEmail);
      startBtn.setOnClickListener(new View.OnClickListener() {
         public void onClick(View view) {
            sendEmail();
         }
      });
   }

   protected void sendEmail() {
      Log.i("Send email", "");
      String[] TO = {""};
      String[] CC = {""};
```

```
      Intent emailIntent = new Intent(Intent.ACTION_SEND);

      emailIntent.setData(Uri.parse("mailto:"));
      emailIntent.setType("text/plain");
      emailIntent.putExtra(Intent.EXTRA_EMAIL, TO);
      emailIntent.putExtra(Intent.EXTRA_CC, CC);
      emailIntent.putExtra(Intent.EXTRA_SUBJECT, "Your subject");
      emailIntent.putExtra(Intent.EXTRA_TEXT, "Email message goes here");

      try {
         startActivity(Intent.createChooser(emailIntent, "Send mail..."));
         finish();
         Log.i("Finished sending email...", "");
      } catch (android.content.ActivityNotFoundException ex) {
         Toast.makeText(MainActivity.this, "There is no email client installed.", Toast.LENGTH_S
      }
   }
}
```

Following will be the content of **res/layout/activity_main.xml** file −

> Here abc indicates about tutorialspoint logo

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:orientation="vertical" >

   <TextView
      android:id="@+id/textView1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Sending Mail Example"
      android:layout_alignParentTop="true"
      android:layout_centerHorizontal="true"
      android:textSize="30dp" />

   <TextView
      android:id="@+id/textView2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Tutorials point "
      android:textColor="#ff87ff09"
      android:textSize="30dp"
      android:layout_above="@+id/imageButton"
      android:layout_alignRight="@+id/imageButton"
      android:layout_alignEnd="@+id/imageButton" />
```

```
   <ImageButton
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/imageButton"
      android:src="@drawable/abc"
      android:layout_centerVertical="true"
      android:layout_centerHorizontal="true" />

   <Button
      android:id="@+id/sendEmail"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="@string/compose_email"/>

</LinearLayout>
```

Following will be the content of **res/values/strings.xml** to define two new constants −

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="app_name">Tutorialspoint</string>
   <string name="compose_email">Compose Email</string>
</resources>
```

Following is the default content of **AndroidManifest.xml** −

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

   <application
      android:allowBackup="true"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name"
      android:theme="@style/AppTheme" >

      <activity
         android:name="com.example.tutorialspoint.MainActivity"
         android:label="@string/app_name" >

         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>

      </activity>

   </application>
</manifest>
```

Let's try to run your **tutorialspoint** application. I assume you have connected your actual Android Mobile device with your computer. To run the app from Android Studio, open one of your project's activity files and click Run icon from the toolbar. Before starting your application, Android studio installer will display following window to select an option where you want to run your Android application.Select your mobile device as an option and then check your mobile device which will display following screen −

Now use **Compose Email** button to list down all the installed email clients. From the list, you can choose one of email clients to send your email. I'm going to use Gmail client to send my email which will have all the provided defaults fields available as shown below. Here **From:** will be default email ID you have registered for your Android device.

You can modify either of the given default fields and finally use send email button to send your email to the mentioned recipients.

Android - Sending SMS

In Android, you can use SmsManager API or devices Built-in SMS application to send SMS's. In this tutorial, we shows you two basic examples to send SMS message −

SmsManager API

```
SmsManager smsManager = SmsManager.getDefault();
smsManager.sendTextMessage("phoneNo", null, "sms message", null, null);
```

Built-in SMS application

```
Intent sendIntent = new Intent(Intent.ACTION_VIEW);
sendIntent.putExtra("sms_body", "default content");
sendIntent.setType("vnd.android-dir/mms-sms");
startActivity(sendIntent);
```

Of course, both need **SEND_SMS permission**.

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

Apart from the above method, there are few other important functions available in SmsManager class. These methods are listed below −

Sr.No.	Method & Description
1	**ArrayList<String> divideMessage(String text)** This method divides a message text into several fragments, none bigger than the maximum SMS message size.
2	**static SmsManager getDefault()** This method is used to get the default instance of the SmsManager
3	**void sendDataMessage(String destinationAddress, String scAddress, short destinationPort, byte[] data, PendingIntent sentIntent, PendingIntent deliveryIntent)** This method is used to send a data based SMS to a specific application port.
4	**void sendMultipartTextMessage(String destinationAddress, String scAddress, ArrayList<String> parts, ArrayList<PendingIntent> sentIntents, ArrayList<PendingIntent> deliveryIntents)** Send a multi-part text based SMS.
5	**void sendTextMessage(String destinationAddress, String scAddress, String text, PendingIntent sentIntent, PendingIntent deliveryIntent)** Send a text based SMS.

Example

Following example shows you in practical how to use SmsManager object to send an SMS to the given mobile number.

> To experiment with this example, you will need actual Mobile device equipped with latest Android OS, otherwise you will have to struggle with emulator which may not work.

Step	Description
1	You will use Android Studio IDE to create an Android application and name it as *tutorialspoint* under a package *com.example.tutorialspoint*.
2	Modify *src/MainActivity.java* file and add required code to take care of sending sms.
3	Modify layout XML file *res/layout/activity_main.xml* add any GUI component if required. I'm adding a simple GUI to take mobile number and SMS text to be sent and a simple button to send SMS.
4	No need to define default string constants at res/values/strings.xml. Android studio takes care of default constants.
5	Modify *AndroidManifest.xml* as shown below
6	Run the application to launch Android emulator and verify the result of the changes done in the application.

Following is the content of the modified main activity file **src/com.example.tutorialspoint/MainActivity.java**.

```
package com.example.tutorialspoint;

import android.Manifest;
import android.content.pm.PackageManager;
import android.os.Bundle;
import android.app.Activity;

import android.support.v4.app.ActivityCompat;
import android.support.v4.content.ContextCompat;
import android.telephony.SmsManager;

import android.util.Log;
import android.view.Menu;
import android.view.View;

import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;

public class MainActivity extends Activity {
   private static final int MY_PERMISSIONS_REQUEST_SEND_SMS =0 ;
   Button sendBtn;
   EditText txtphoneNo;
   EditText txtMessage;
   String phoneNo;
   String message;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
```

```
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      sendBtn = (Button) findViewById(R.id.btnSendSMS);
      txtphoneNo = (EditText) findViewById(R.id.editText);
      txtMessage = (EditText) findViewById(R.id.editText2);

      sendBtn.setOnClickListener(new View.OnClickListener() {
         public void onClick(View view) {
            sendSMSMessage();
         }
      });
   }

   protected void sendSMSMessage() {
      phoneNo = txtphoneNo.getText().toString();
      message = txtMessage.getText().toString();

      if (ContextCompat.checkSelfPermission(this,
         Manifest.permission.SEND_SMS)
         != PackageManager.PERMISSION_GRANTED) {
            if (ActivityCompat.shouldShowRequestPermissionRationale(this,
               Manifest.permission.SEND_SMS)) {
            } else {
               ActivityCompat.requestPermissions(this,
                  new String[]{Manifest.permission.SEND_SMS},
                  MY_PERMISSIONS_REQUEST_SEND_SMS);
            }
      }
   }

   @Override
   public void onRequestPermissionsResult(int requestCode,String permissions[], int[] grantResul
      switch (requestCode) {
         case MY_PERMISSIONS_REQUEST_SEND_SMS: {
            if (grantResults.length > 0
               && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                  SmsManager smsManager = SmsManager.getDefault();
                  smsManager.sendTextMessage(phoneNo, null, message, null, null);
                  Toast.makeText(getApplicationContext(), "SMS sent.",
                     Toast.LENGTH_LONG).show();
            } else {
               Toast.makeText(getApplicationContext(),
                  "SMS faild, please try again.", Toast.LENGTH_LONG).show();
               return;
            }
         }
      }

   }
}
```

Following will be the content of **res/layout/activity_main.xml** file −

> Here abc indicates about tutorialspoint logo

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:paddingBottom="@dimen/activity_vertical_margin"
   android:paddingLeft="@dimen/activity_horizontal_margin"
   android:paddingRight="@dimen/activity_horizontal_margin"
   android:paddingTop="@dimen/activity_vertical_margin"
   tools:context="MainActivity">

   <TextView
      android:id="@+id/textView1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Sending SMS Example"
      android:layout_alignParentTop="true"
      android:layout_centerHorizontal="true"
      android:textSize="30dp" />

   <TextView
      android:id="@+id/textView2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Tutorials point "
      android:textColor="#ff87ff09"
      android:textSize="30dp"
      android:layout_below="@+id/textView1"
      android:layout_alignRight="@+id/imageButton"
      android:layout_alignEnd="@+id/imageButton" />

   <ImageButton
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/imageButton"
      android:src="@drawable/abc"
      android:layout_below="@+id/textView2"
      android:layout_centerHorizontal="true" />

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

```
      android:hint="Enter Phone Number"
      android:phoneNumber="true"
      android:textColorHint="@color/abc_primary_text_material_dark"
      android:layout_below="@+id/imageButton"
      android:layout_centerHorizontal="true" />

   <EditText
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/editText2"
      android:layout_below="@+id/editText"
      android:layout_alignLeft="@+id/editText"
      android:layout_alignStart="@+id/editText"
      android:textColorHint="@color/abc_primary_text_material_dark"
      android:layout_alignRight="@+id/imageButton"
      android:layout_alignEnd="@+id/imageButton"
      android:hint="Enter SMS" />

   <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Send Sms"
      android:id="@+id/btnSendSMS"
      android:layout_below="@+id/editText2"
      android:layout_centerHorizontal="true"
      android:layout_marginTop="48dp" />

</RelativeLayout>
```

Following will be the content of **res/values/strings.xml** to define two new constants −

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="app_name">tutorialspoint</string>
</resources>
```

Following is the default content of **AndroidManifest.xml** −

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

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

   <application
      android:allowBackup="true"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name"
      android:theme="@style/AppTheme" >

      <activity
```

```
        android:name="com.example.tutorialspoint.MainActivity"
        android:label="@string/app_name" >

        <intent-filter>
           <action android:name="android.intent.action.MAIN" />
           <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>

     </activity>

   </application>
</manifest>
```

Let's try to run your **tutorialspoint** application. I assume you have connected your actual Android Mobile device with your computer. To run the app from Android studio, open one of your project's activity files and click Run icon from the toolbar. Before starting your application, Android studio installer will display following window to select an option where you want to run your Android application.

Now you can enter a desired mobile number and a text message to be sent on that number. Finally click on **Send SMS** button to send your SMS. Make sure your GSM/CDMA connection is working fine to deliver your SMS to its recipient.

You can take a number of SMS separated by comma and then inside your program you will have to parse them into an array string and finally you can use a loop to send message to all the given numbers. That's

how you can write your own SMS client. Next section will show you how to use existing SMS client to send SMS.

Using Built-in Intent to send SMS

You can use Android Intent to send SMS by calling built-in SMS functionality of the Android. Following section explains different parts of our Intent object required to send an SMS.

Intent Object - Action to send SMS

You will use **ACTION_VIEW** action to launch an SMS client installed on your Android device. Following is simple syntax to create an intent with ACTION_VIEW action.

```
Intent smsIntent = new Intent(Intent.ACTION_VIEW);
```

Intent Object - Data/Type to send SMS

To send an SMS you need to specify **smsto:** as URI using setData() method and data type will be to **vnd.android-dir/mms-sms** using setType() method as follows −

```
smsIntent.setData(Uri.parse("smsto:"));
smsIntent.setType("vnd.android-dir/mms-sms");
```

Intent Object - Extra to send SMS

Android has built-in support to add phone number and text message to send an SMS as follows −

```
smsIntent.putExtra("address"  , new String("0123456789;3393993300"));
smsIntent.putExtra("sms_body"  , "Test SMS to Angilla");
```

> Here address and sms_body are case sensitive and should be specified in small characters only. You can specify more than one number in single string but separated by semi-colon (;).

Example

Following example shows you in practical how to use Intent object to launch SMS client to send an SMS to the given recipients.

> To experiment with this example, you will need actual Mobile device equipped with latest Android OS, otherwise you will have to struggle with emulator which may not work.

Step	Description
1	You will use Android studio IDE to create an Android application and name it as *tutorialspoint* under a package *com.example.tutorialspoint*.
2	Modify *src/MainActivity.java* file and add required code to take care of sending SMS.
3	Modify layout XML file *res/layout/activity_main.xml* add any GUI component if required. I'm adding a simple button to launch SMS Client.
4	No need to define default constants.Android studio takes care of default constants.
5	Modify *AndroidManifest.xml* as shown below
6	Run the application to launch Android emulator and verify the result of the changes done in the application.

Following is the content of the modified main activity file **src/com.example.tutorialspoint/MainActivity.java**.

```
package com.example.tutorialspoint;

import android.net.Uri;
import android.os.Bundle;
import android.app.Activity;
import android.content.Intent;
import android.util.Log;
import android.view.Menu;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;

public class MainActivity extends Activity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      Button startBtn = (Button) findViewById(R.id.button);
      startBtn.setOnClickListener(new View.OnClickListener() {
         public void onClick(View view) {
            sendSMS();
         }
      });
   }

   protected void sendSMS() {
      Log.i("Send SMS", "");
      Intent smsIntent = new Intent(Intent.ACTION_VIEW);
```

```
      smsIntent.setData(Uri.parse("smsto:"));
      smsIntent.setType("vnd.android-dir/mms-sms");
      smsIntent.putExtra("address"  , new String ("01234"));
      smsIntent.putExtra("sms_body"  , "Test ");

      try {
         startActivity(smsIntent);
         finish();
         Log.i("Finished sending SMS...", "");
      } catch (android.content.ActivityNotFoundException ex) {
         Toast.makeText(MainActivity.this,
         "SMS faild, please try again later.", Toast.LENGTH_SHORT).show();
      }
   }

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

Following will be the content of **res/layout/activity_main.xml** file −

> Here abc indicates about tutorialspoint logo

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:paddingLeft="@dimen/activity_horizontal_margin"
   android:paddingRight="@dimen/activity_horizontal_margin"
   android:paddingTop="@dimen/activity_vertical_margin"
   android:paddingBottom="@dimen/activity_vertical_margin"
   tools:context=".MainActivity">

   <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Drag and Drop Example"
      android:id="@+id/textView"
      android:layout_alignParentTop="true"
      android:layout_centerHorizontal="true"
      android:textSize="30dp" />

   <TextView
```

```
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Tutorials Point "
      android:id="@+id/textView2"
      android:layout_below="@+id/textView"
      android:layout_centerHorizontal="true"
      android:textSize="30dp"
      android:textColor="#ff14be3c" />

   <ImageView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/imageView"
      android:src="@drawable/abc"
      android:layout_marginTop="48dp"
      android:layout_below="@+id/textView2"
      android:layout_centerHorizontal="true" />

   <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Compose SMS"
      android:id="@+id/button"
      android:layout_below="@+id/imageView"
      android:layout_alignRight="@+id/textView2"
      android:layout_alignEnd="@+id/textView2"
      android:layout_marginTop="54dp"
      android:layout_alignLeft="@+id/imageView"
      android:layout_alignStart="@+id/imageView" />

</RelativeLayout>
```

Following will be the content of **res/values/strings.xml** to define two new constants −

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="app_name">tutorialspoint</string>
</resources>
```

Following is the default content of **AndroidManifest.xml** −

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

   <application
      android:allowBackup="true"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name"
      android:theme="@style/AppTheme" >
```

```
        <activity
            android:name="com.example.tutorialspoint.MainActivity"
            android:label="@string/app_name" >

            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>

        </activity>

    </application>
</manifest>
```

Let's try to run your **tutorialspoint** application. I assume you have connected your actual Android Mobile device with your computer. To run the app from Android studio, open one of your project's activity files and click Run icon from the toolbar. Before starting your application, Android studio will display following window to select an option where you want to run your Android application.

Select your mobile device as an option and then check your mobile device which will display following screen −

Now use **Compose SMS** button to launch Android built-in SMS clients which is shown below −

You can modify either of the given default fields and finally use send SMS button to send your SMS to the mentioned recipient.