Android Tutorial

ANDROID TUTORIAL

Simply Easy Learning by tutorialspoint.com

tutorialspoint.com

ABOUT THE TUTORIAL

Android Tutorial

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.

This tutorial will teach you basic Android programming and will also take you through some advance concepts related to Android application development.

Audience

This tutorial has been prepared for the beginners to help them understand basic Android programming. After completing this tutorial you will find yourself at a moderate level of expertise in Android programming from where you can take yourself to next levels.

Prerequisites

Android programming is based on Java programming language so if you have basic understanding on Java programming then it will be a fun to learn Android application development.

Copyright & Disclaimer Notice

Table of Content

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.

Environment Setup

Y ou 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
```

TUTORIALS POINT
Simply Easy Learning

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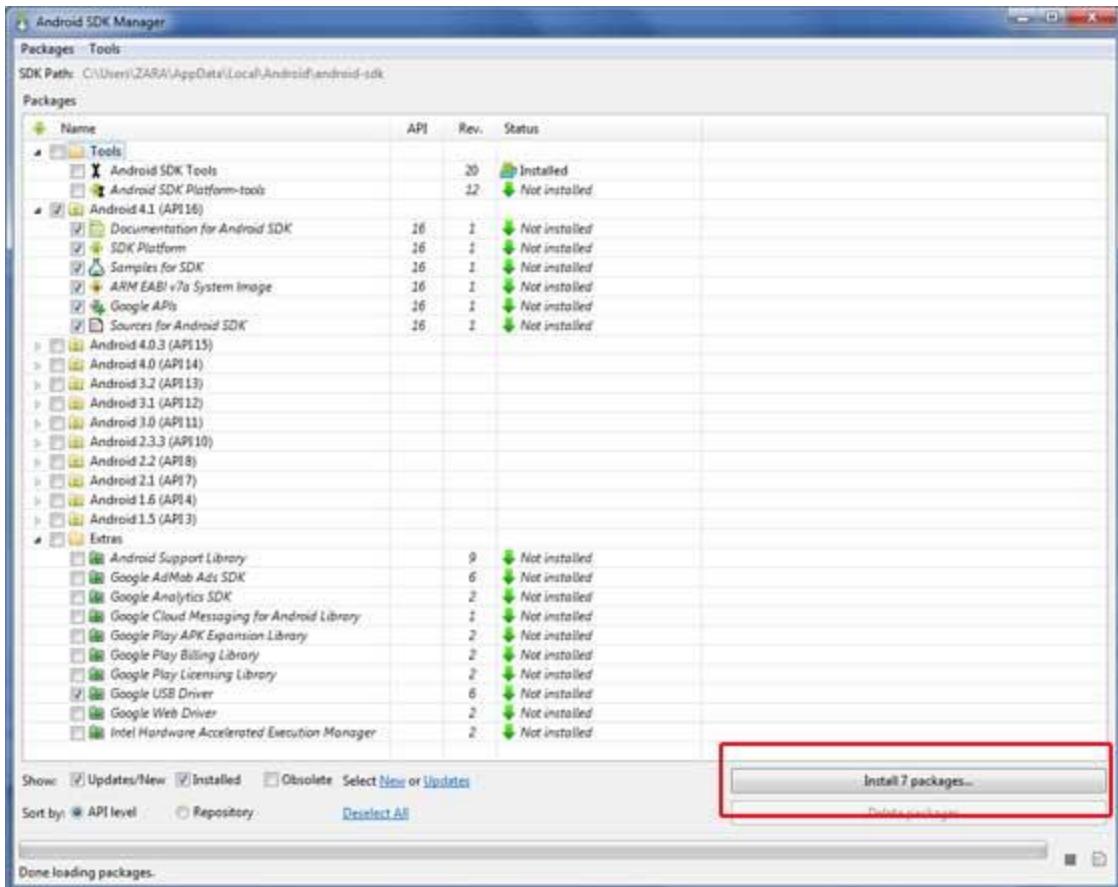
Once you launched SDK manager, its time to install other required packages. By default it will list down total 7 packages to be installed, but I will suggest to de-select *Documentation for Android SDK* and*Samples for*

SDK packages to reduce installation time. Next click **Install 7 Packages** button to proceed, which will display following dialogue box:
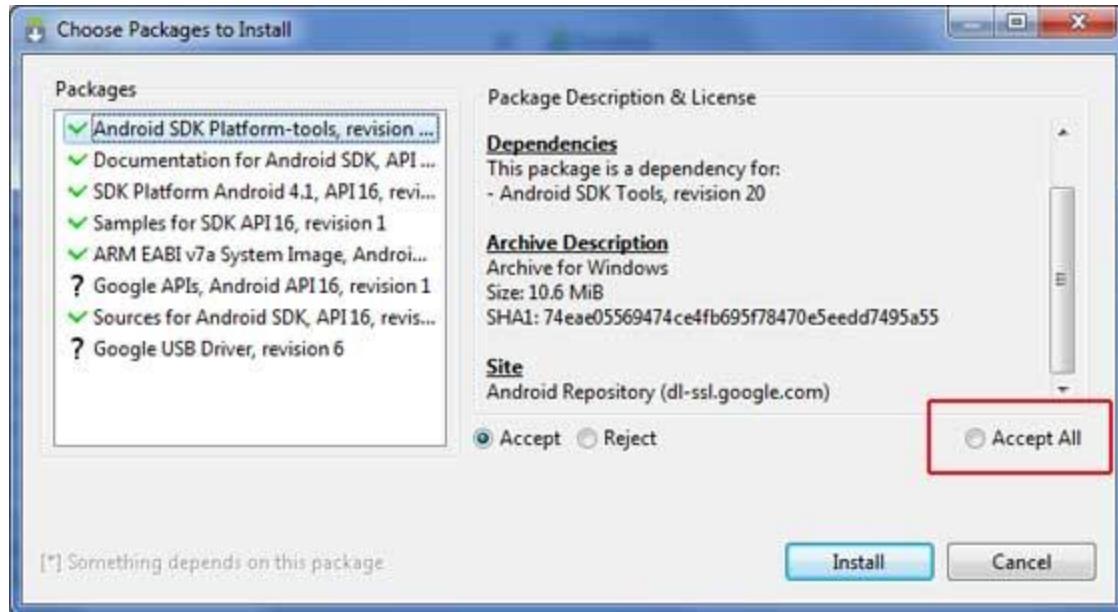
If you agree to install all the packages, select **Accept All** radio button and proceed by clicking **Install**button. Now let SDK manager do its work and you go, pick up a cup of coffee and wait until all the packages are installed. It may take some time depending on your internet connection. Once 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:
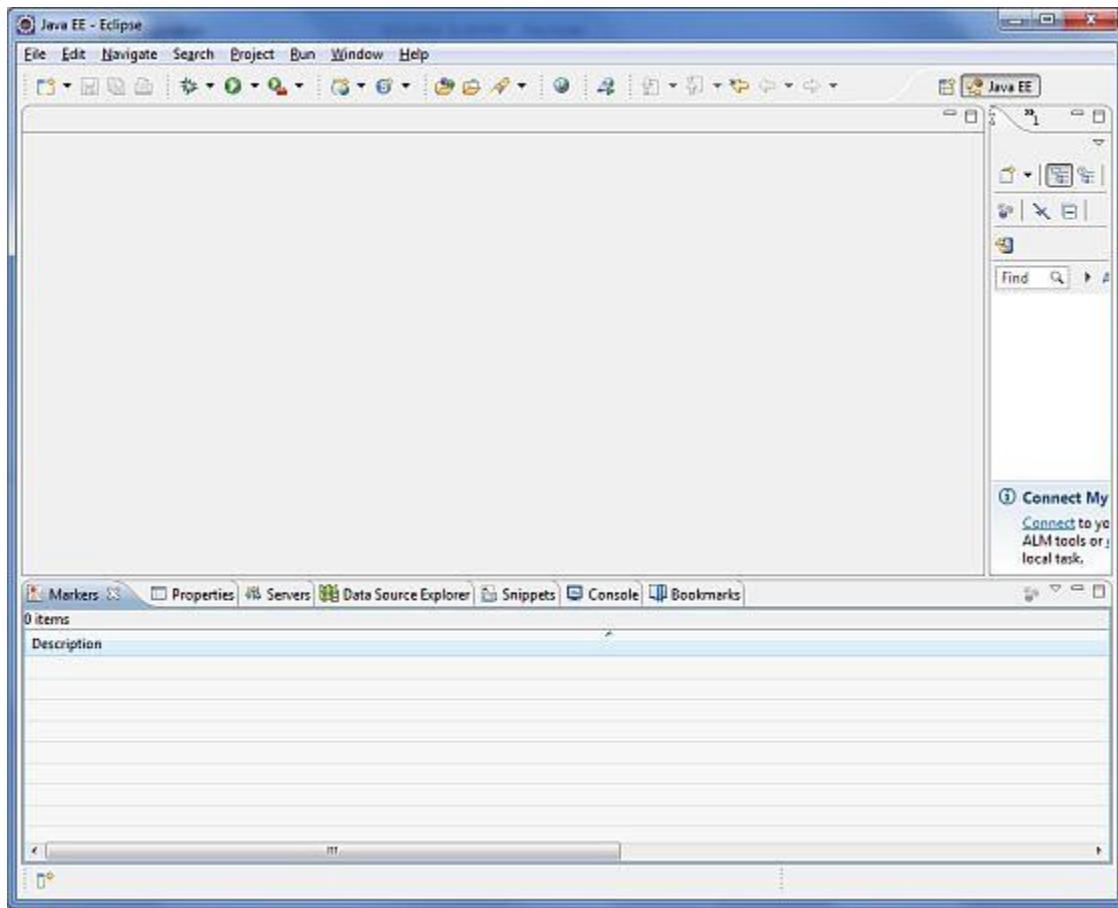
Step 4 - Setup Android Development Tools (ADT) Plugin

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.
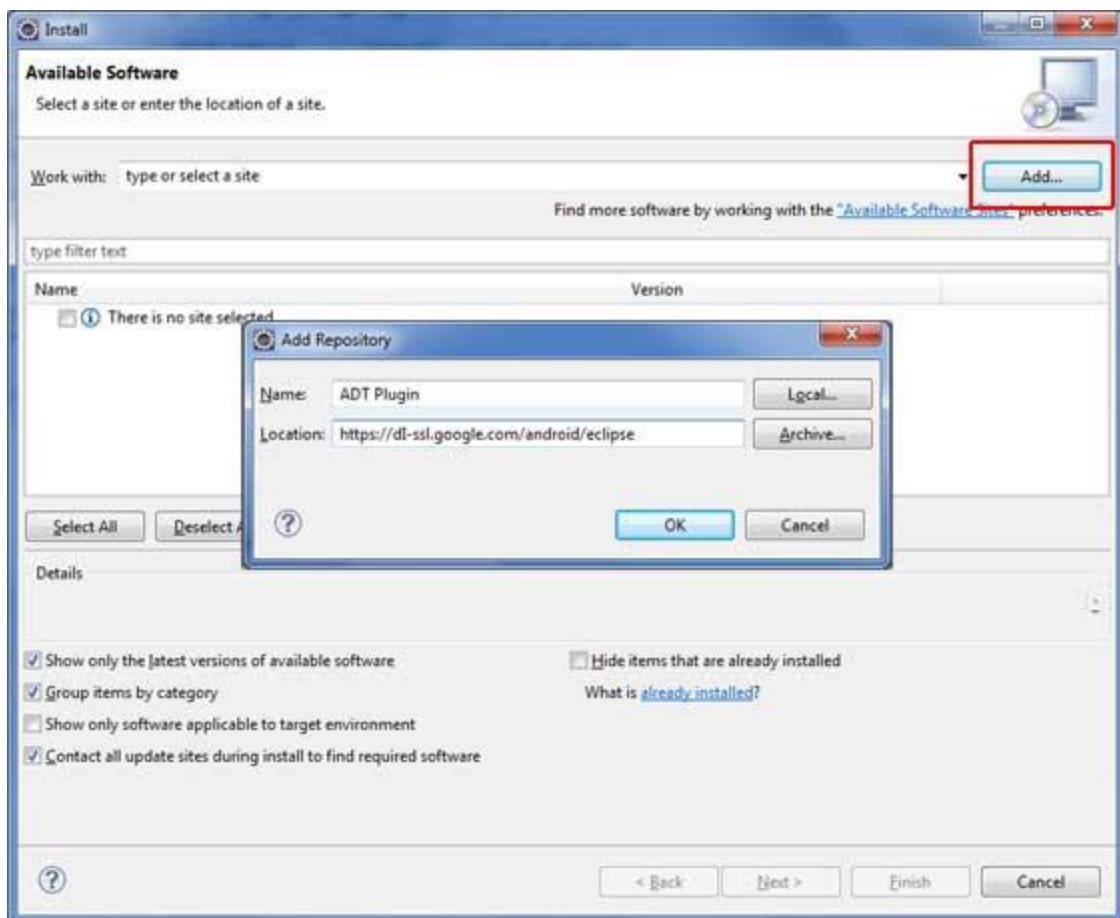
Now use **Add** button to add *ADT Plugin* as name and *https://dl-ssl.google.com/android/eclipse/* as the location. Then click OK to add this location, as soon as you will click OK button to add this location, Eclipse starts searching for the plug-in available the given location and finally lists down the found plugins.
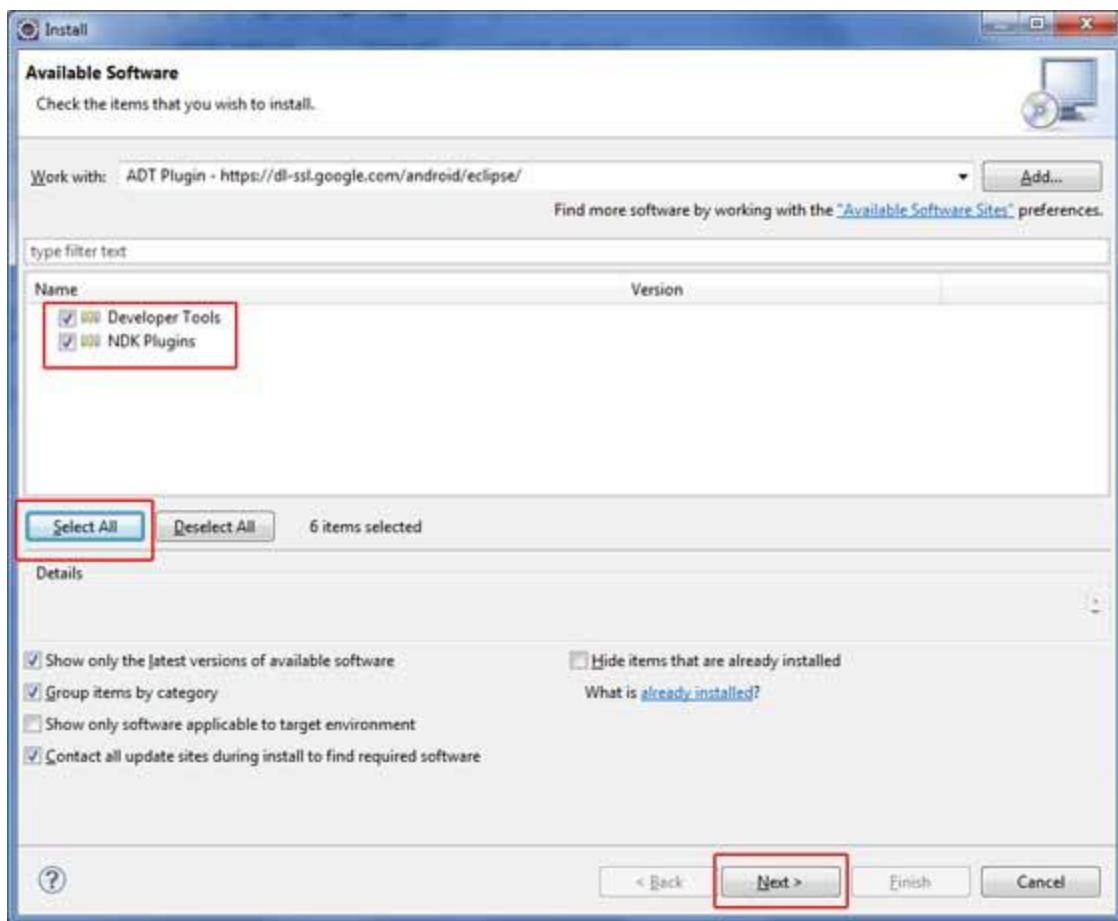
Now select all the listed plug-ins using **Select All** button and click **Next** button which will guide you ahead to install Android Development Tools and other required plugins.

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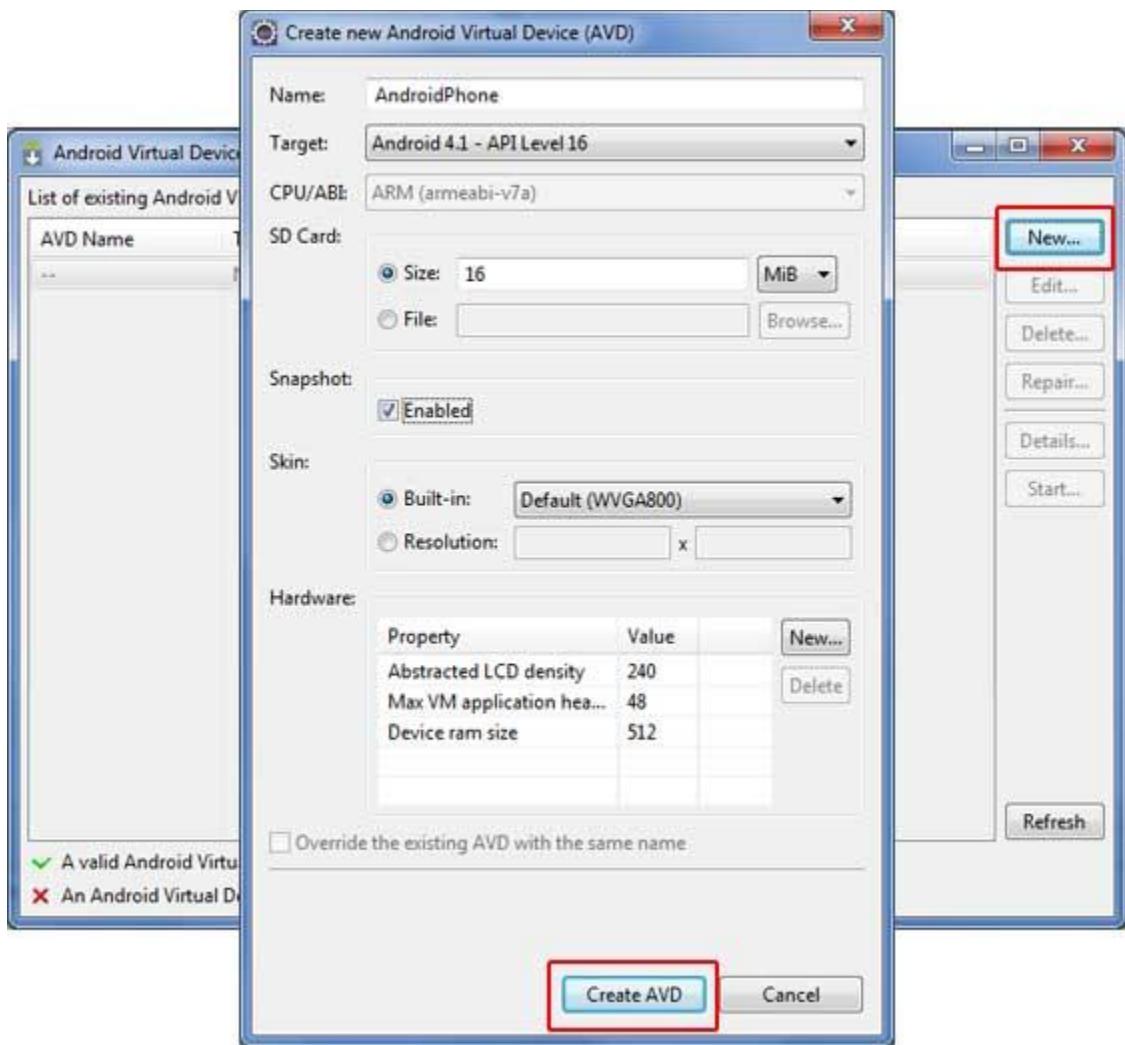
If your AVD is created successfully it means your environment is ready for Android application development. If you like, you can close this window using top-right cross button. Better you re-start your machine and once you are done with this last step, you are ready to proceed for your first Android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

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.
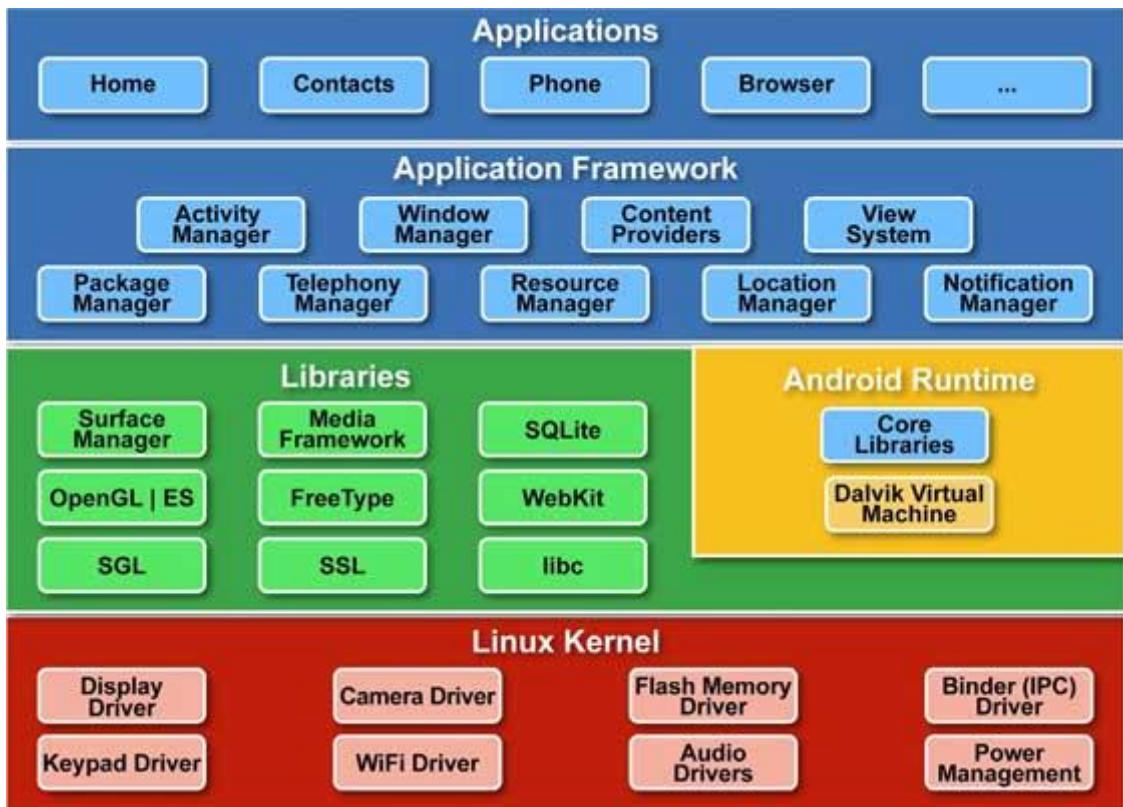
Linux kernel

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.

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:

```java
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:

```java
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.

Hello World Example

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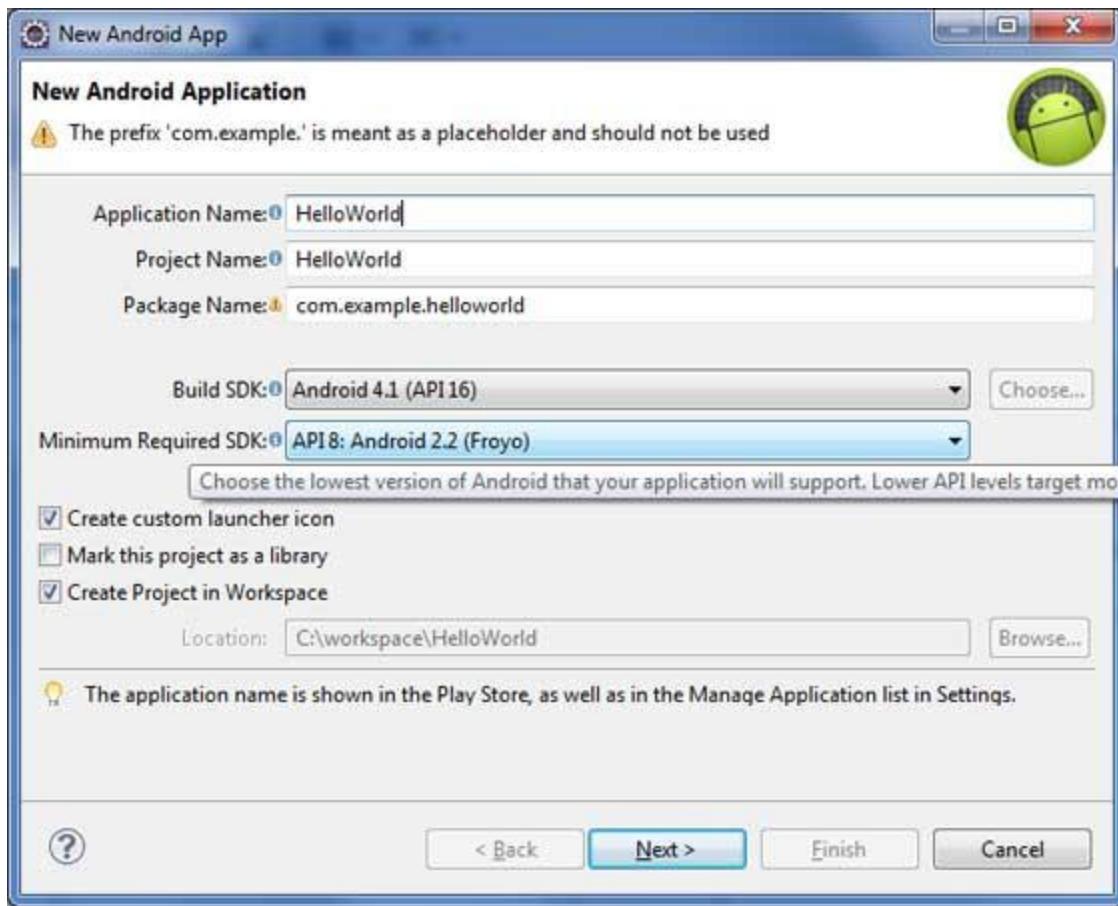
Next, follow the instructions provided and keep all other entries as default till the final step. Once your project is created successfully, you will have following project screen:
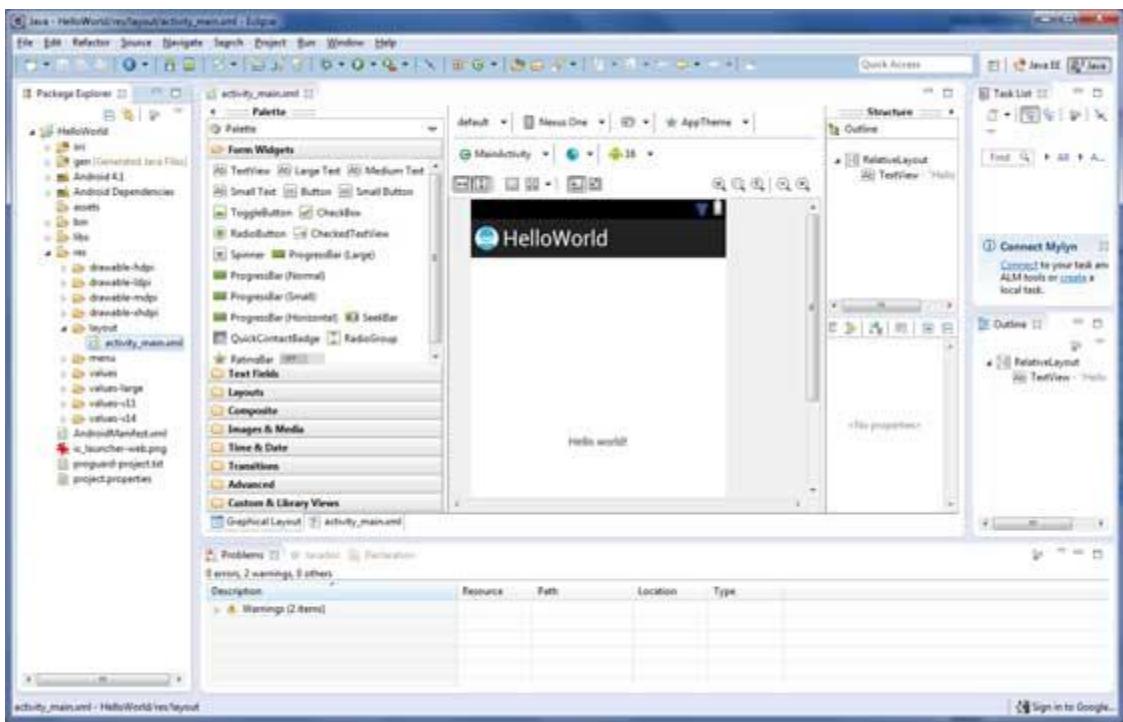
Anatomy of Android Application

Before you run your app, you should be aware of a few directories and files in the Android project:
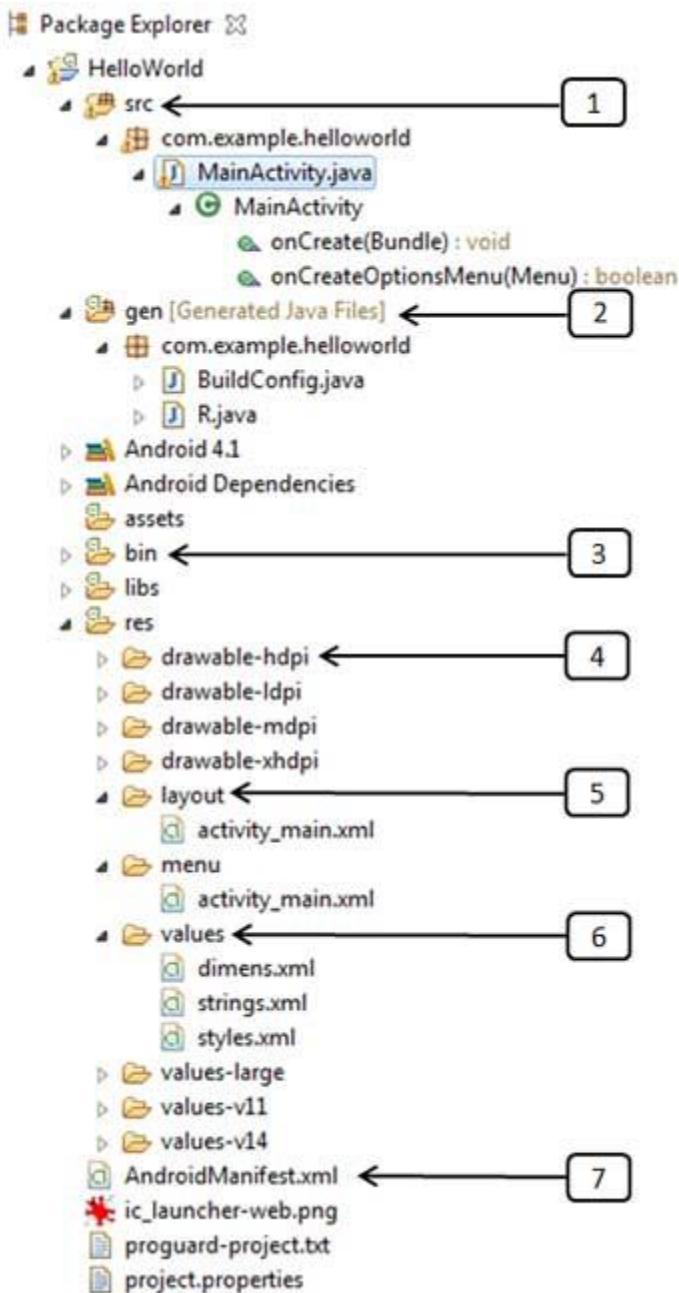
S.N.	Folder, File & Description
1	**src** This contains the **.java** source files for your project. By default, it includes an *MainActivity.java* source file having an activity class that runs when your app is launched using the app icon.
2	**gen** This contains the **.R** file, a compiler-generated 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:

```java
/* 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:

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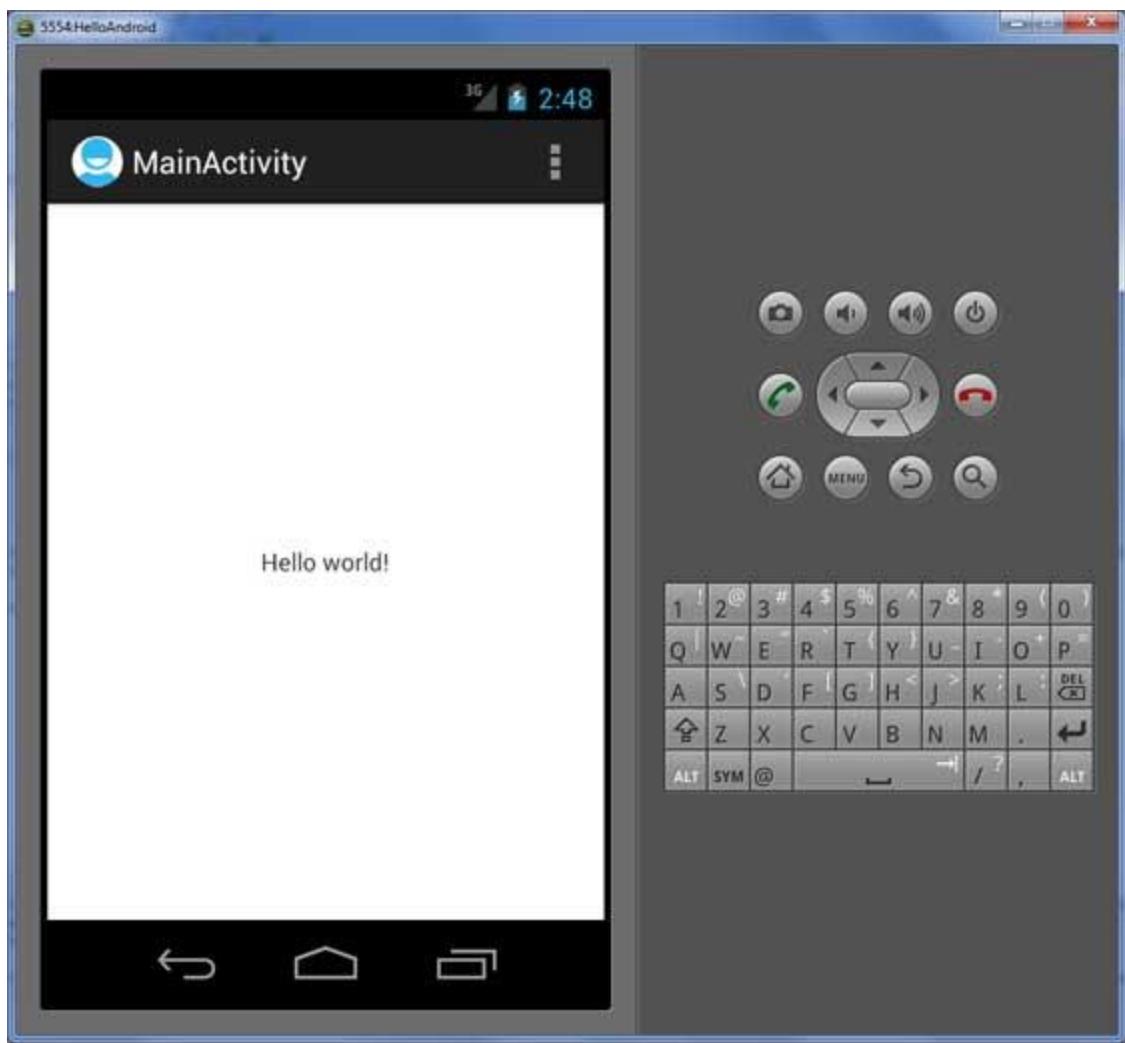
Congratulations!!! you have developed your first Android Application and now just keep following rest of the tutorial step by step to become a great Android Developer. All the very best.

Resources Organizing & Accessing

T here 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.

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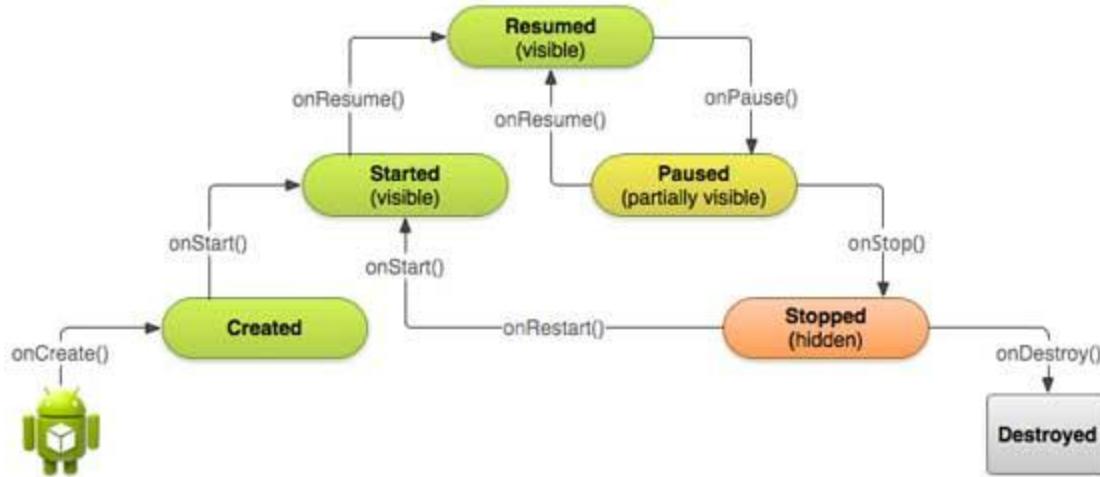
The Activity class defines the following callbacks i.e. events. You don't need to implement all the callbacks methods. However, it's important that you understand each one and implement those that ensure your app behaves the way users expect.

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:

```java
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
```
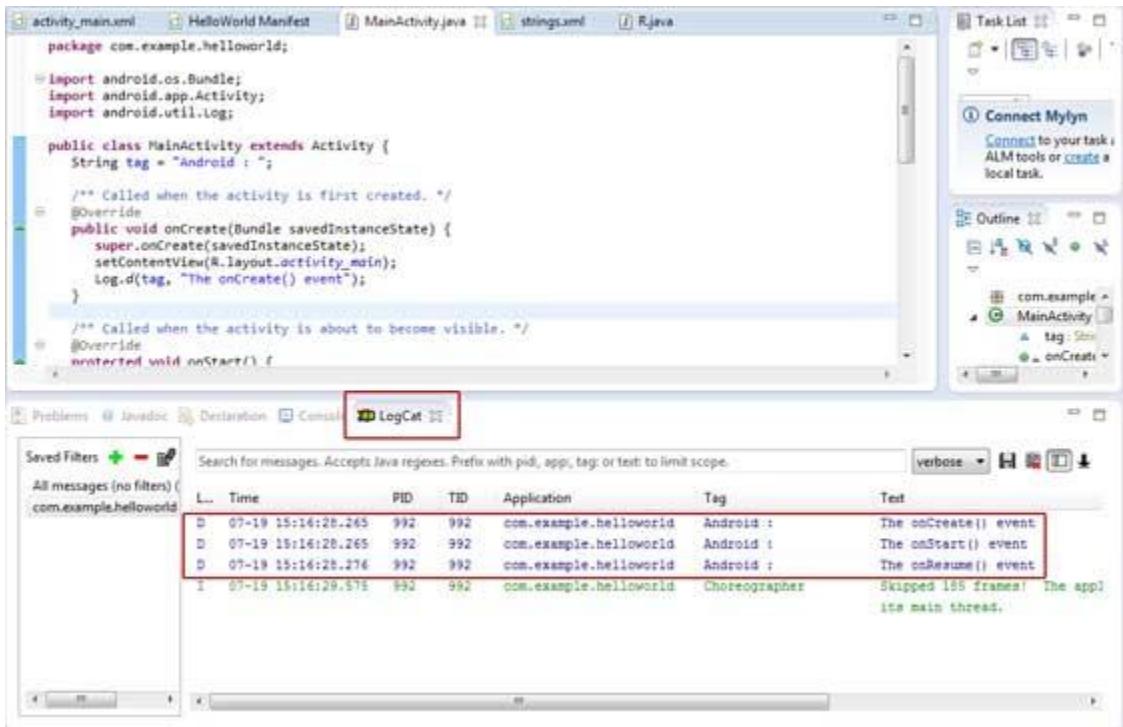

Let us try to click Red button ☎ on the Android emulator and it will generate following events messages in **LogCat** window in Eclipse IDE:

```
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 MENU 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
```

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)*
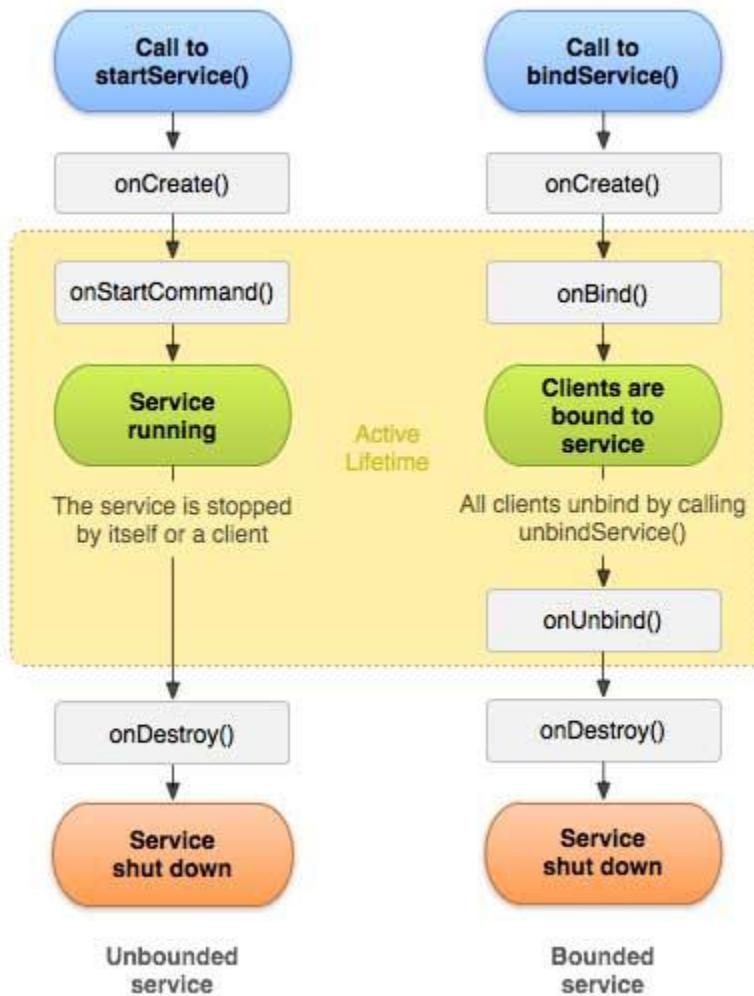
To create an service, you create a Java class that extends the Service base class or one of its existing subclasses. The **Service** base class defines various callback methods and the most important are given below. You don't need to implement all the callbacks methods. However, it's important that 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:

```java
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.

```java
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()*:

```java
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:

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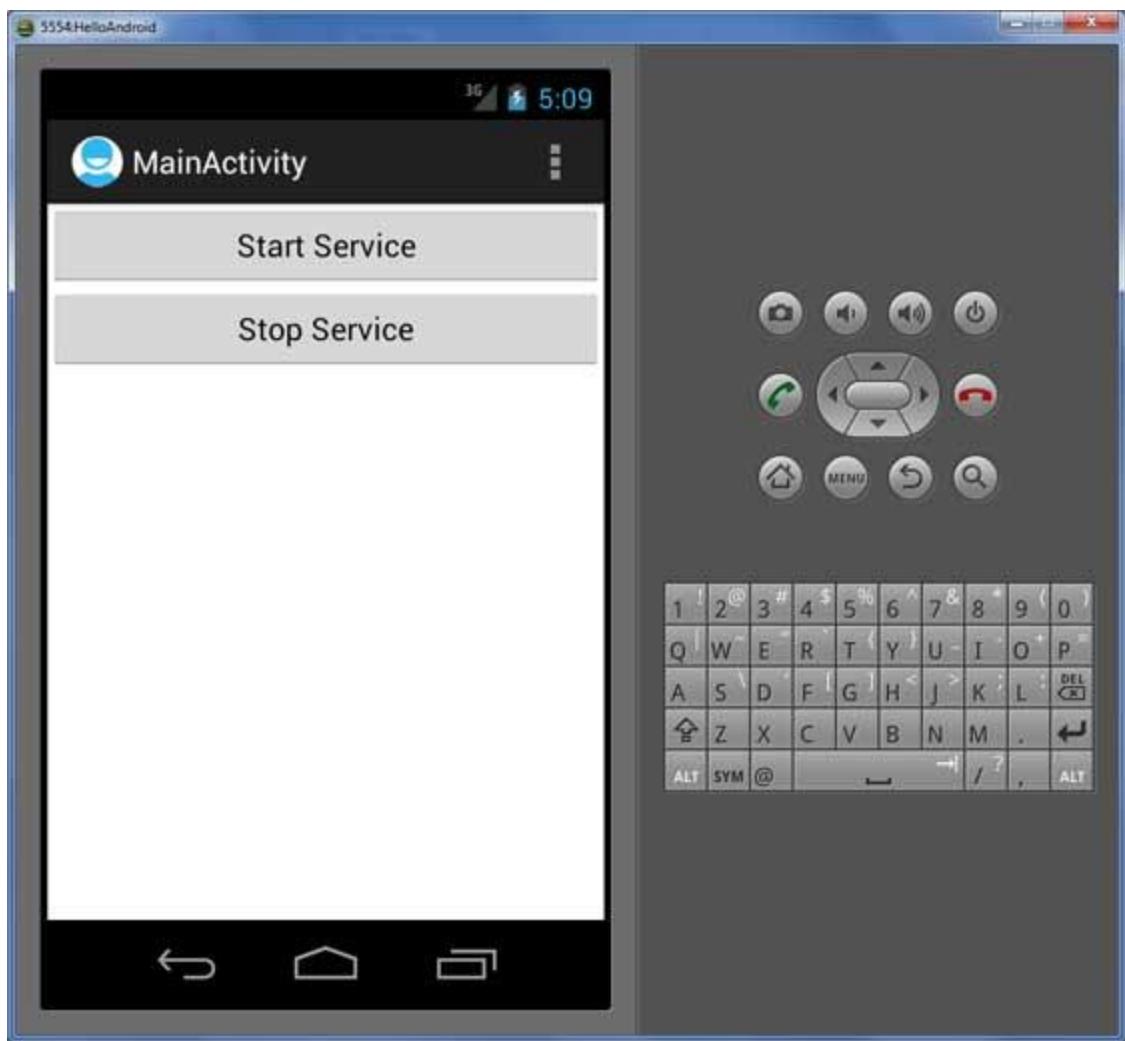
Now to start your service, let's click on **Start Service** button, this will start the service and as per our programming in *onStartCommand()* method, a message *Service Started* will appear on the bottom of the the simulator as follows:
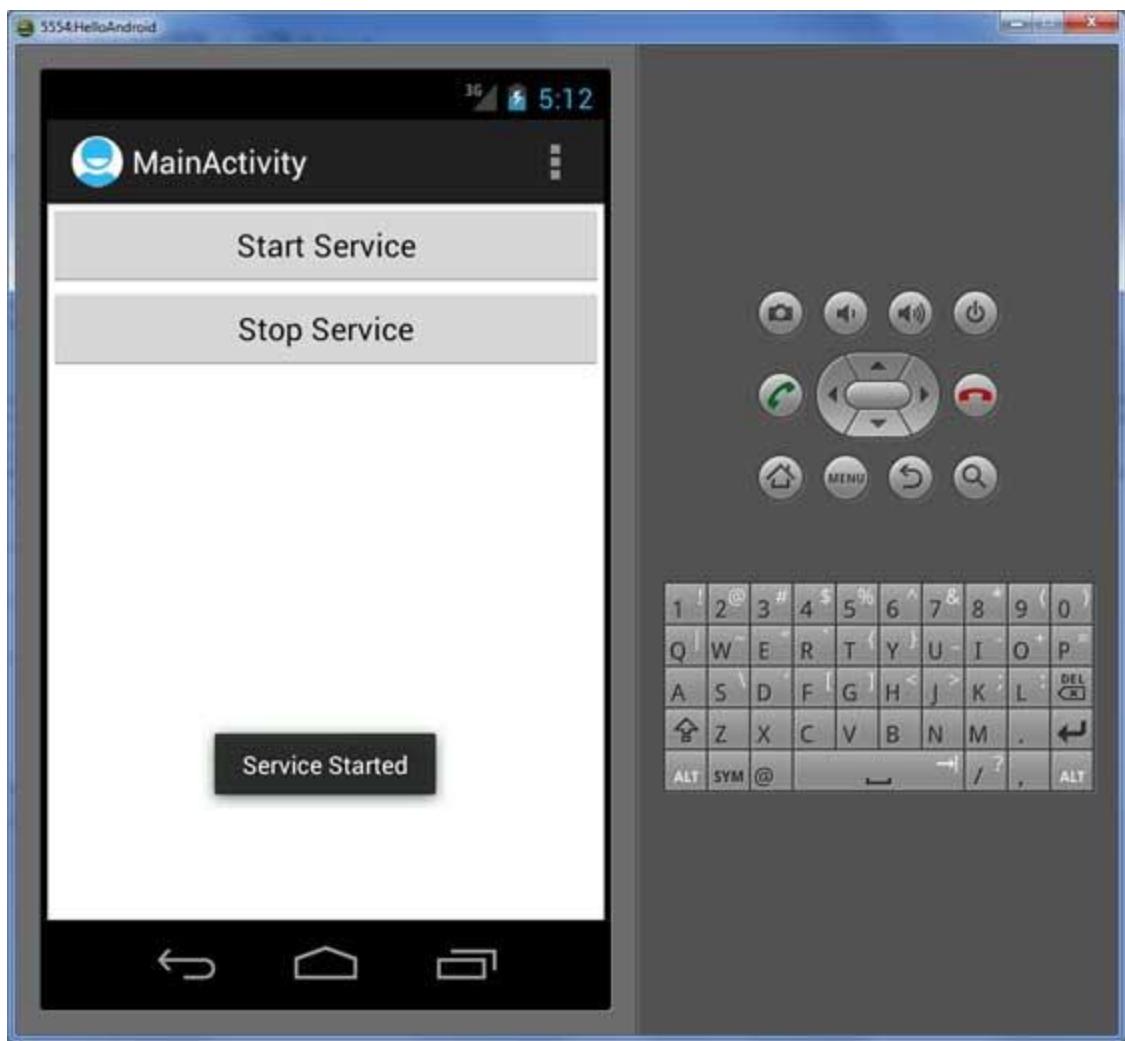
To stop the service, you can click the Stop Service button.

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 default 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"
```

TUTORIALS POINT

Simply Easy Learning

```
    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:
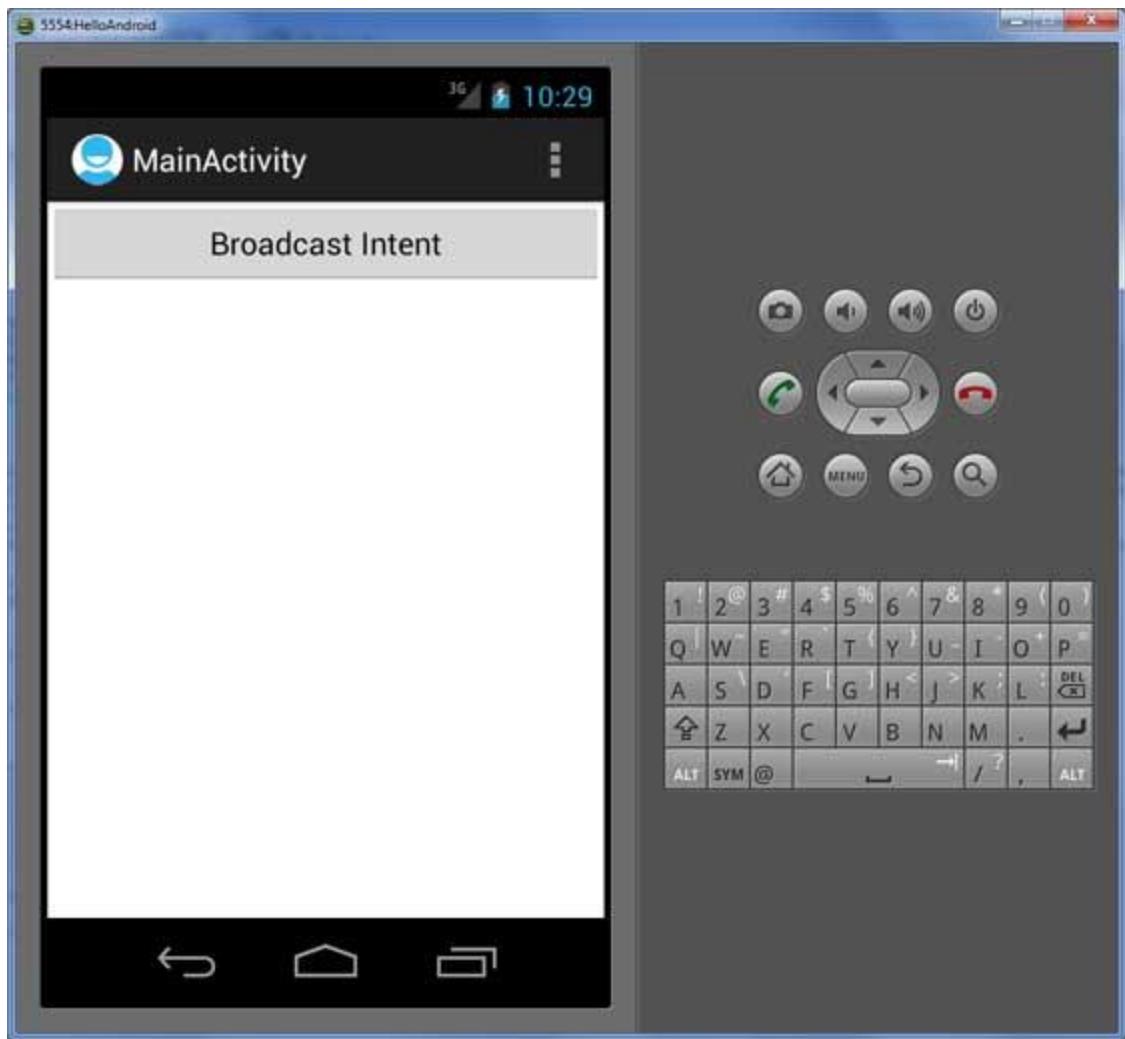
Now to broadcast our custom intent, let's click on **Broadcast Intent** button, this will broadcast our custom intent *"com.tutorialspoint.CUSTOM_INTENT"* which will be intercepted by our registered BroadcastReceiver ie. MyReceiver and as per our implemented logic a toast will appear on the bottom of the the simulator as follows:
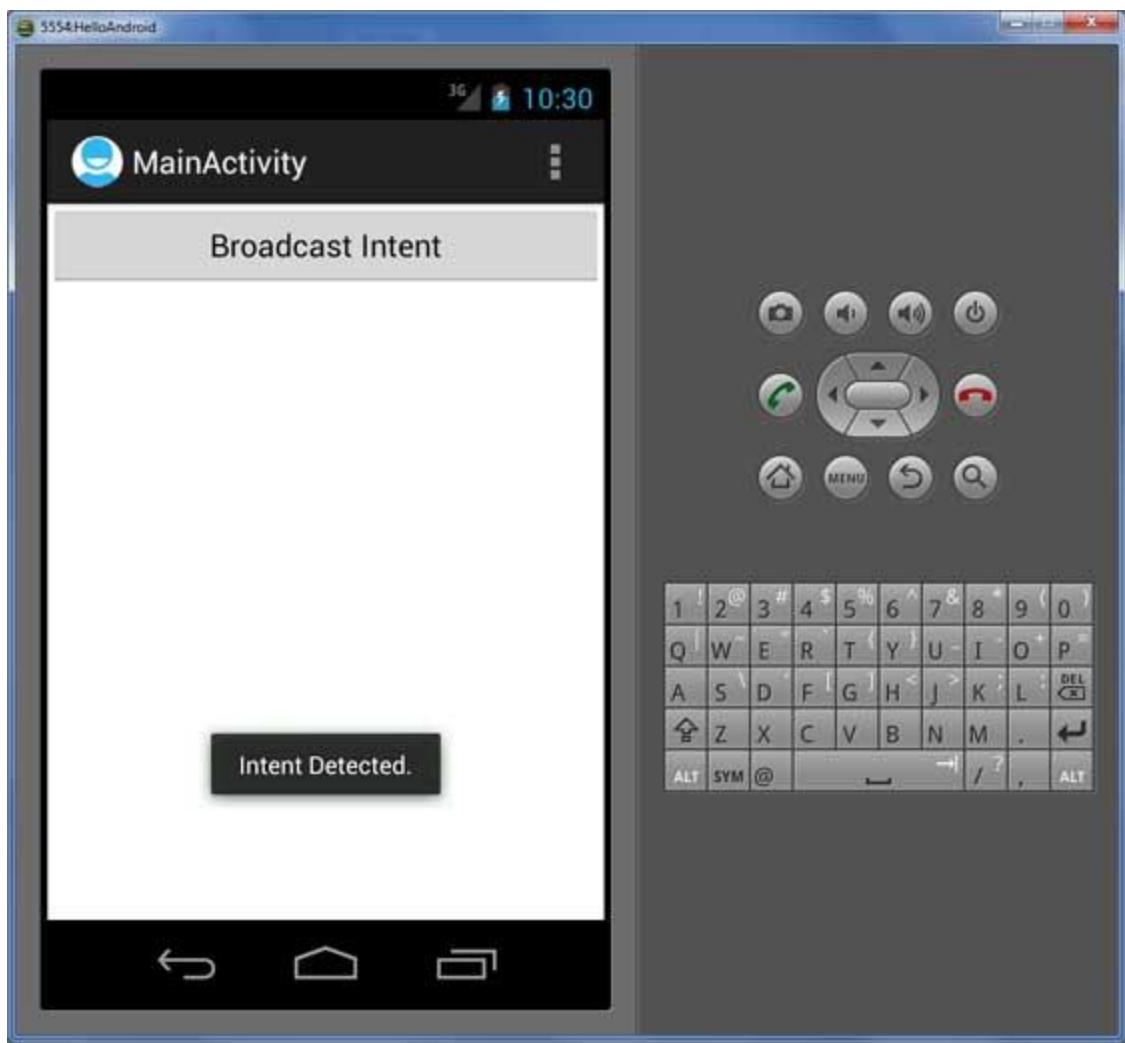
You can try implementing other BroadcastReceiver to intercept system generated intents like system bootup, date changed, low battery etc.

Content Providers

Acontent 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 detail 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 default 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**:

```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){
```

```java
            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:
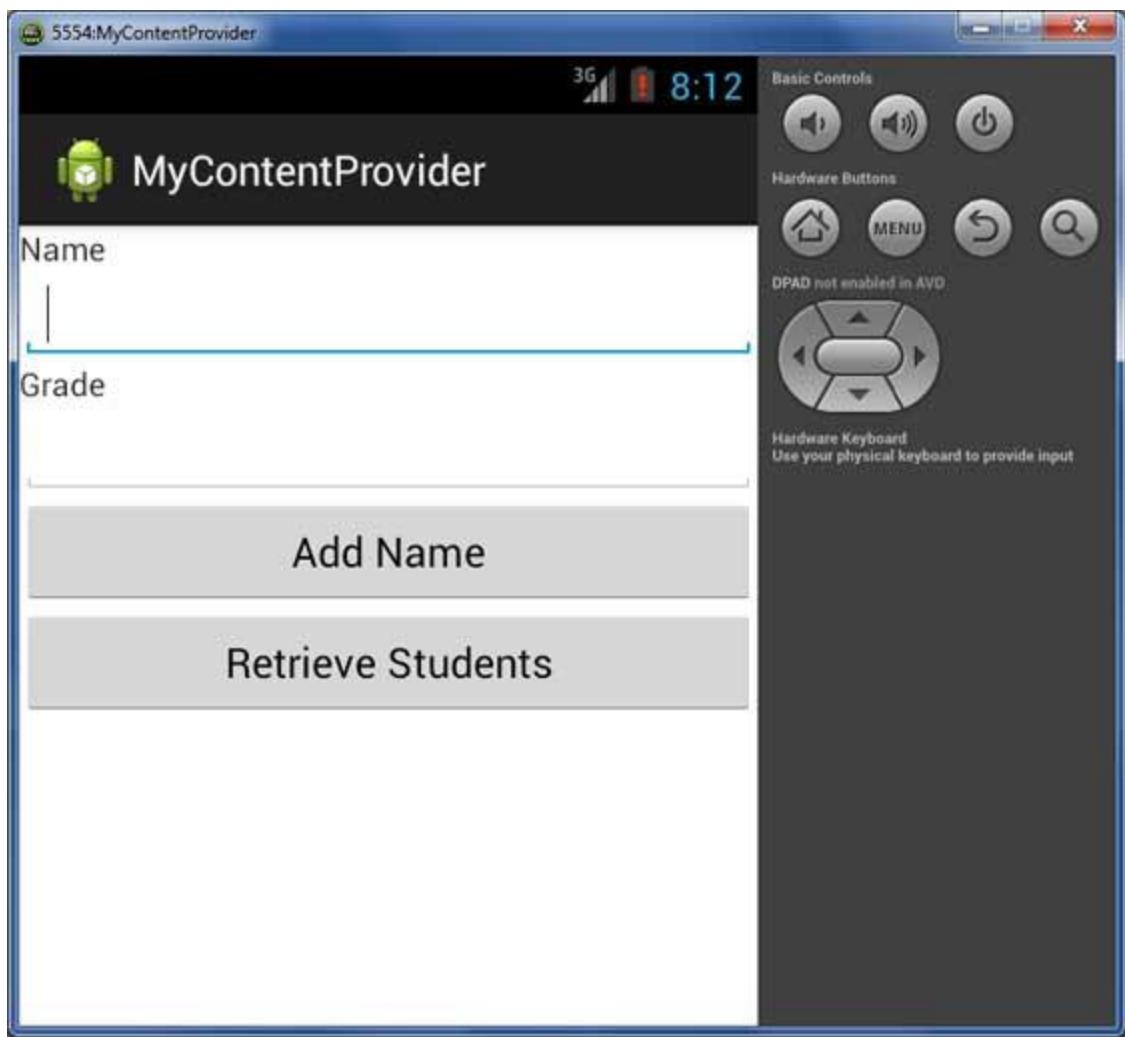
Now let's enter student **Name** and **Grade** and finally click on **Add Name** button, this will add student record in the database and will flash a message at the bottom showing ContentProvider URI along with record number added in the database. This operation makes use of our **insert()** method. Let's repeat this process to add few more students in the database of our content provider.
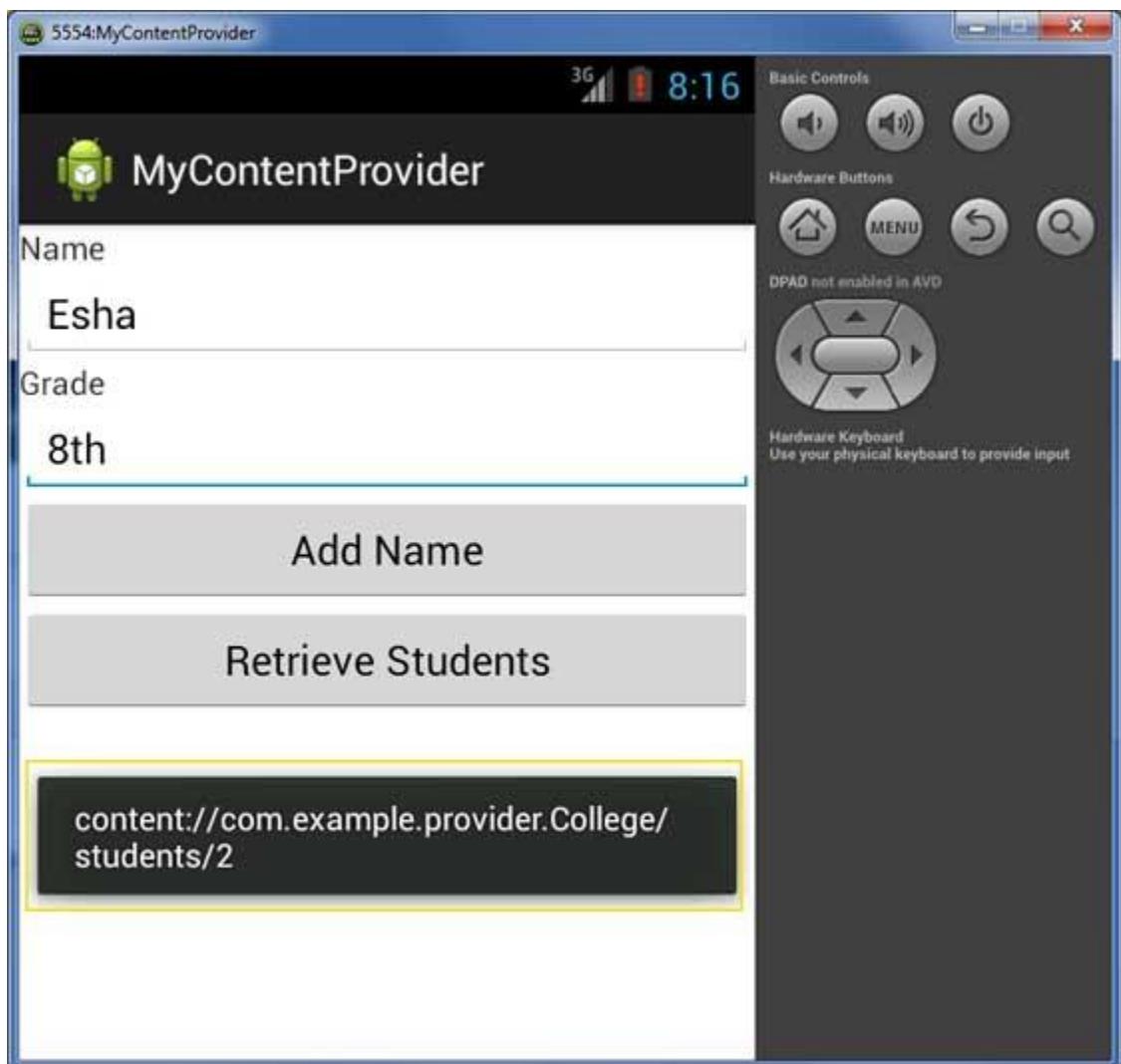
Once you are done with adding records in the database, now its time to ask ContentProvider to give us those records back, so let's click **Retrieve Students** button which will fetch and display all the records one by one which is as per our the implementation of our **query()** method.
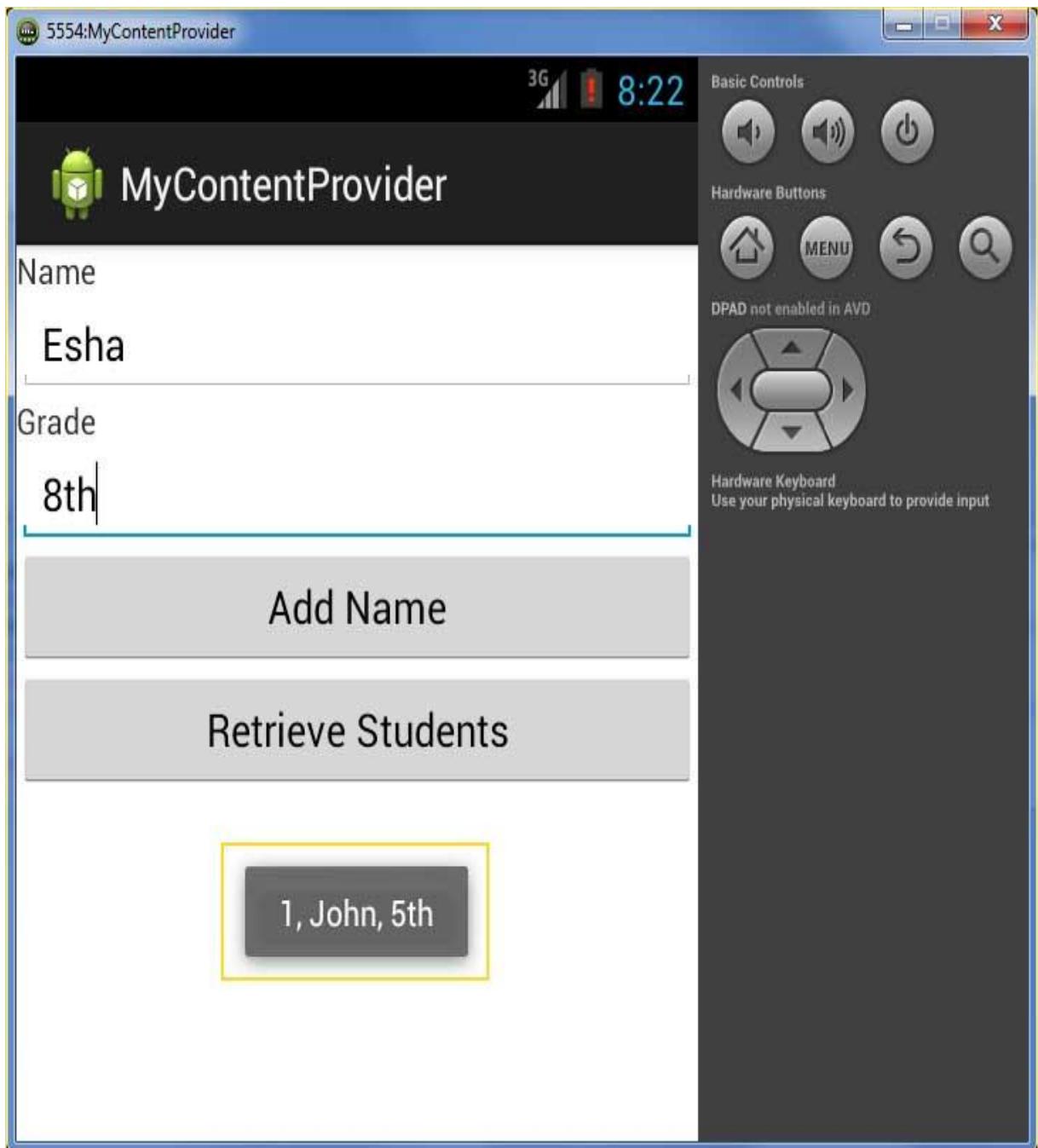
You can write activities against update and delete operations by providing callback functions in**MainActivity.java** file and then modify user interface to have buttons for update and deleted operations in the same way as we have done for add and read operations.

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.

Fragments

A Fragment is a piece of an application's user interface or behavior that can be placed in an Activity which enable more modular activity design. It will not be wrong if we say, a fragment is a kind of **sub-acitivity**. Following are important points about fragment:

- A fragment has its own layout and its own behavior with its own lifecycle callbacks.

- You can add or remove fragments in an activity while the activity is running.

- You can combine multiple fragments in a single activity to build a multi-pane UI.

- A fragment can be used in multiple activities.

- Fragment life cycle is closely related to the lifecycle of its host activity which means when the activity is paused, all the fragments available in the acivity will also be stopped.

- A fragment can implement a behavior that has no user interface component.

- Fragments were added to the Android API in Honeycomb version of Android which API version 11.

You create fragments by extending **Fragment** class and You can insert a fragment into your activity layout by declaring the fragment in the activity's layout file, as a **<fragment>** element.

Prior to fragment introduction, we had a limitation because we can show only a single activity on the screen at one given point in time. So we were not able to divide device screen and control different parts separately. But with the introduction of fragment we got more flexibility and removed the limitation of having a single activity on the screen at a time. Now we can have a single acitivity but each acitivity can comprise of multiple fragments which will have their own layout, events and complete lifecycle.

Following is a typical example of how two UI modules defined by fragments can be combined into one activity for a tablet design, but separated for a handset design.
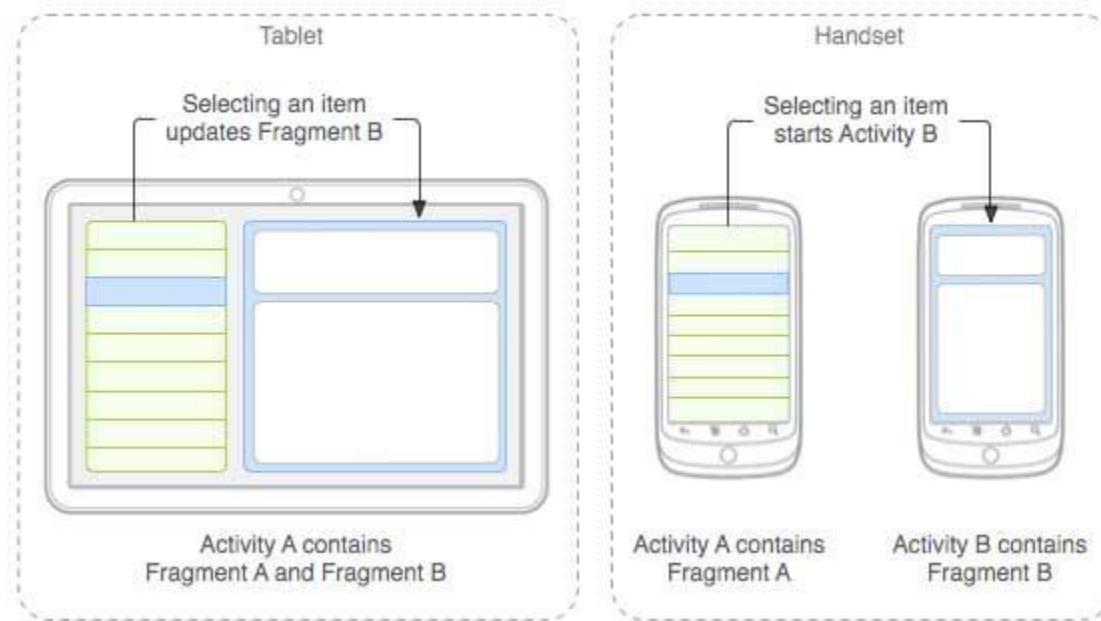
The application can embed two fragments in Activity A, when running on a tablet-sized device. However, on a handset-sized screen, there's not enough room for both fragments, so Activity A includes only the fragment for the list of articles, and when the user selects an article, it starts Activity B, which includes the second fragment to read the article.

Fragment Life Cycle

Android fragments have their own life cycle very similar to an android activity. This section briefs different stages of its life cycle.

Phase I: When a fragment gets created, it goes through the following states:

onAttach()

onCreate()

onCreateView()

onActivityCreated()

Phase II: When the fragment becomes visible, it goes through these states:

onStart()

onResume()

Phase III: When the fragment goes into the background mode, it goes through these states:

onPaused()

onStop()

Phase IV: When the fragment is destroyed, it goes through the following states:

onPaused()

onStop()

onDestroyView()

onDestroy()

onDetach()

Activity State	Fragment Callbacks
Created	onAttach()
	onCreate()
	onCreateView()
	onActivityCreated()
Started	onStart()
Resumed	onResume()
Paused	onPause()
Stopped	onStop()
Destroyed	onDestroyView()
	onDestroy()
	onDetach()
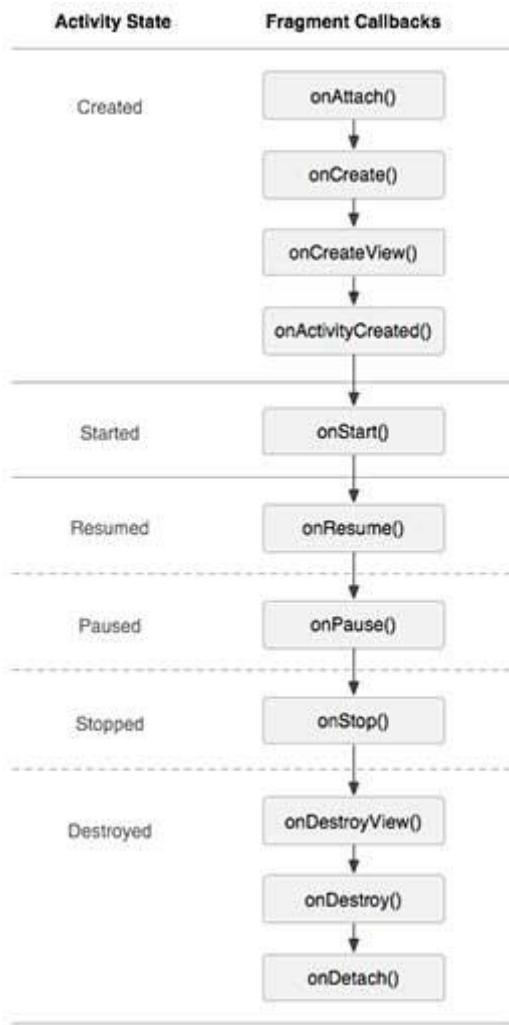
How to use Fragments?

This involves number of simple steps to create Fragments.

- First of all decide how many fragments you want to use in an activity. Fors example let's we want to use two fragments to handle landscape and portrait modes of the device.

- Next based on number of fragments, create classes which will extend the *Fragment* class. The Fragment class has above mentioned callback functions. You can override any of the functions based on your requirements.

- Corresponding to each fragment, you will need to create layout files in XML file. These files will have layout for the defined fragments.

- Finally modify activity file to define the actual logic of replacing fragments based on your requirement.

Here is the list of important methods which you can to override in your fragment class:

- **onCreate()** The system calls this when creating the fragment. You should initialize essential components of the fragment that you want to retain when the fragment is paused or stopped, then resumed.
- **onCreateView()** The system calls this callback when it's time for the fragment to draw its user interface for the first time. To draw a UI for your fragment, you must return a **View** component from this method that is the root of your fragment's layout. You can return null if the fragment does not provide a UI.
- **onPause()** The system calls this method as the first indication that the user is leaving the fragment. This is usually where you should commit any changes that should be persisted beyond the current user session.

Example

This example will explain you how to create your own *Fragments*. Here we will create two fragments and one of them will be used when device is in landscape mode and another fragment will be used in case of portrait mode. 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 *MyFragments* under a package *com.example.myfragments*, with blank Activity.
2	Modify main activity file *MainActivity.java* as shown below in the code. Here we will check orientation of the device and accordingly we will switch between different fragments.
3	Create a two java files *PM_Fragment.java* and *LM_Fragement.java* under the package*com.example.myfragments* to define your fragments and associated methods.
4	Create layouts files *res/layout/lm_fragment.xml* and *res/layout/pm_fragment.xml* and define your layouts for both the fragments.<
5	Modify the default content of *res/layout/activity_main.xml* file to include both the fragments.
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**:

```
package com.example.myfragments;

import android.os.Bundle;
import android.app.Activity;
import android.app.FragmentManager;
import android.app.FragmentTransaction;
import android.content.res.Configuration;
import android.view.WindowManager;

public class MainActivity extends Activity {

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

      Configuration config = getResources().getConfiguration();

      FragmentManager fragmentManager = getFragmentManager();
      FragmentTransaction fragmentTransaction =
      fragmentManager.beginTransaction();

      /**
      * Check the device orientation and act accordingly
```

```
        */
        if (config.orientation == Configuration.ORIENTATION_LANDSCAPE) {
           /**
            * Landscape mode of the device
            */
           LM_Fragment ls_fragment = new LM_Fragment();
           fragmentTransaction.replace(android.R.id.content, ls_fragment);
        }else{
           /**
            * Portrait mode of the device
            */
           PM_Fragment pm_fragment = new PM_Fragment();
           fragmentTransaction.replace(android.R.id.content, pm_fragment);
        }
        fragmentTransaction.commit();
    }

}
```

Create two fragment files **LM_Fragement.java** and **PM_Fragement.java** under *com.example.mycontentprovider* package.

Following is the content of **LM_Fragement.java** file:

```
package com.example.myfragments;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;


public class LM_Fragment extends Fragment{
    @Override
    public View onCreateView(LayoutInflater inflater,
        ViewGroup container, Bundle savedInstanceState) {
        /**
         * Inflate the layout for this fragment
         */
        return inflater.inflate(
        R.layout.lm_fragment, container, false);
    }
}
```

Following is the content of **PM_Fragement.java** file:

```
package com.example.myfragments;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;


public class PM_Fragment extends Fragment{
    @Override
    public View onCreateView(LayoutInflater inflater,
        ViewGroup container, Bundle savedInstanceState) {
        /**
```

```
        * Inflate the layout for this fragment
        */
      return inflater.inflate(
      R.layout.pm_fragment, container, false);
   }
}
```

Create two layout files **lm_fragement.xml** and **pm_fragment.xml** under *res/layout* directory.

Following is the content of **lm_fragement.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"
   android:background="#7bae16">

   <TextView
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:text="@string/landscape_message"
   android:textColor="#000000"
   android:textSize="20px" />

<!-- More GUI components go here  -->

</LinearLayout>
```

Following is the content of **pm_fragment.xml** file:

```
<?xml version="1.0" encoding="utf-8"?>
   <LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="horizontal"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:background="#666666">

   <TextView
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:text="@string/portrait_message"
   android:textColor="#000000"
   android:textSize="20px" />

<!-- More GUI components go here  -->

</LinearLayout>
```

Following will be the content of **res/layout/activity_main.xml** file which includes your fragments:

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

   <fragment
```

```
    android:name="com.example.fragments"
    android:id="@+id/lm_fragment"
    android:layout_weight="1"
    android:layout_width="0dp"
    android:layout_height="match_parent" />

    <fragment
    android:name="com.example.fragments"
    android:id="@+id/pm_fragment"
    android:layout_weight="2"
    android:layout_width="0dp"
    android:layout_height="match_parent" />

</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">MyFragments</string>
    <string name="action_settings">Settings</string>
    <string name="hello_world">Hello world!</string>
    <string name="landscape_message">This is Landscape mode fragment
    </string>
    <string name="portrait_message">This is Portrait mode fragment
    </string>

</resources>
```

Let's try to run our modified **MyFragments** 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 Emulator window where you will click on Menu button to see the following window. Be patience because it may take sometime based on your computer speed:
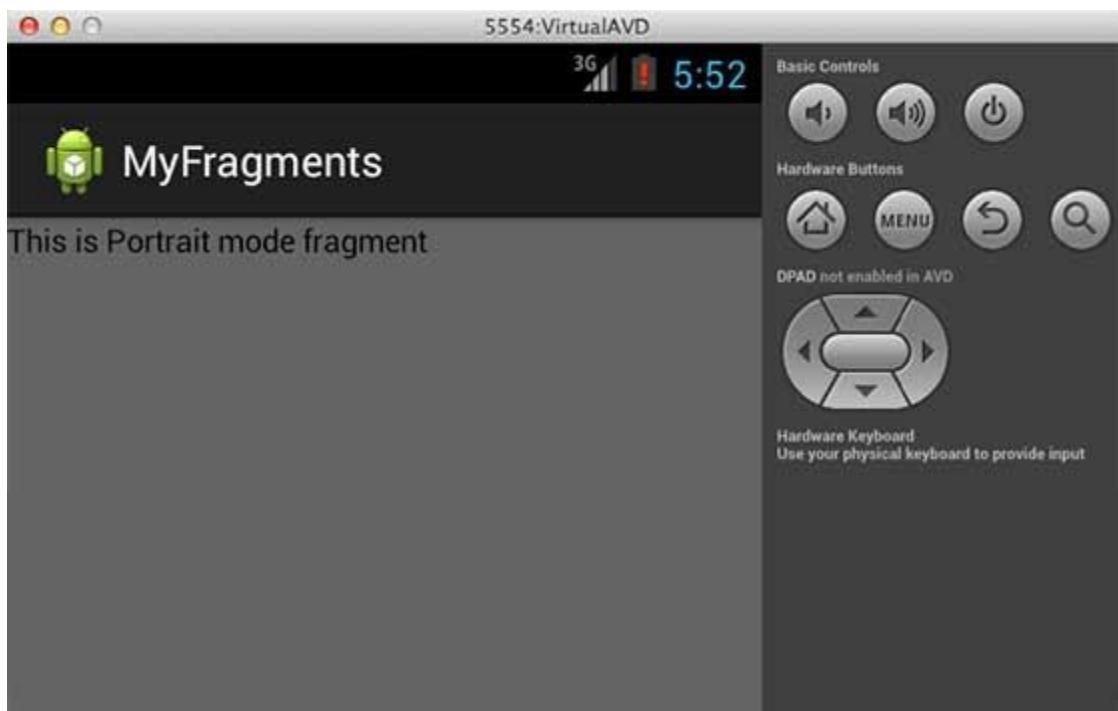
To change the mode of the emulator screen, let's do the following:

- **fn+control+F11** on Mac to change the landscape to portrait and vice versa.
- **ctrl+F11** on Windows.
- **ctrl+F11** on Linux.

Once you changed the mode, you will be able to see the GUI which you have implemented for landscape mode as below:
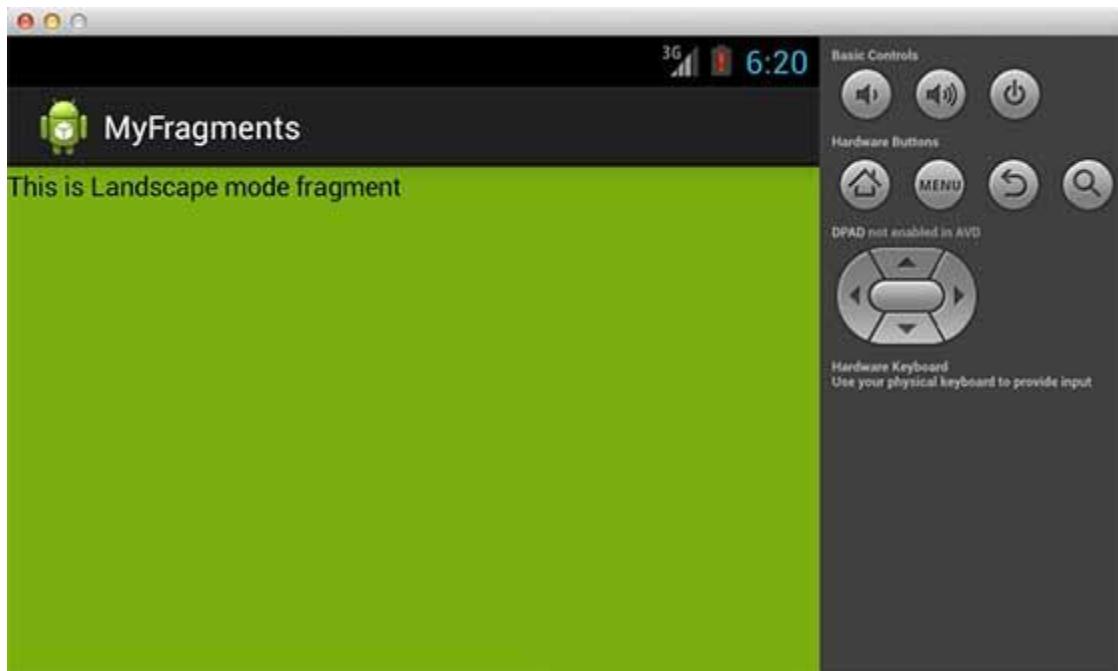
This way you can use same activity but different GUIs through different fragments. You can use different type of GUI components for different GUIs based on your requirements.

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 <u>Android Intent Standard Actions</u>

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 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:

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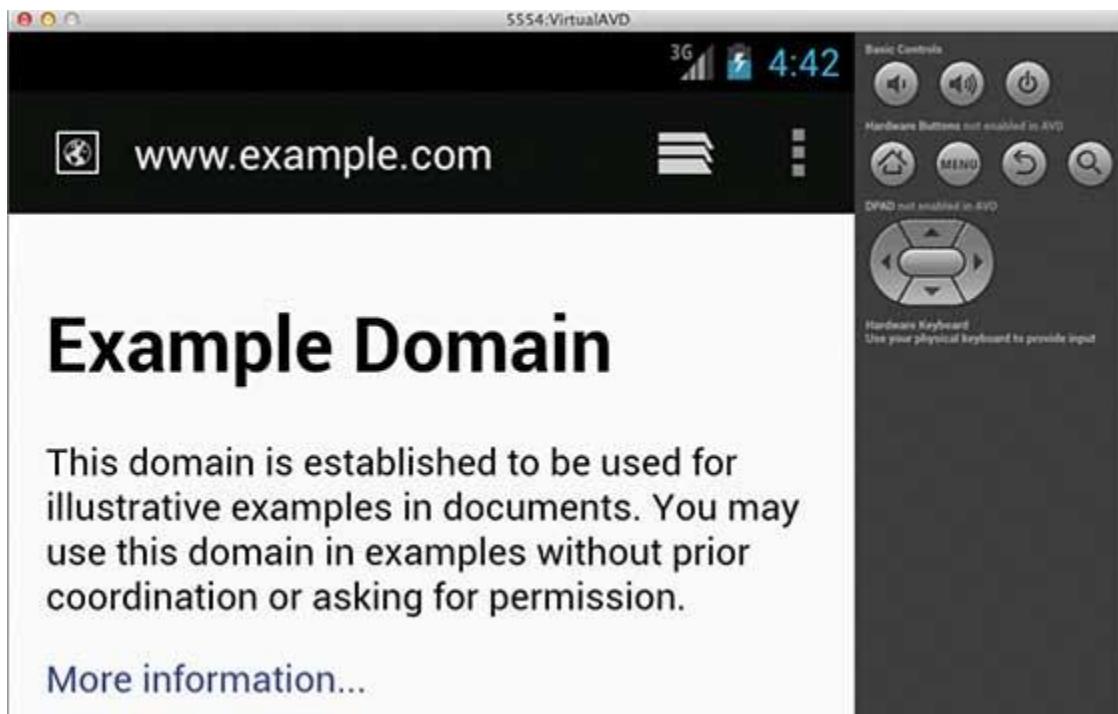
Similar way you can launch phone interface using Start Phone button, which will allow you to dial already given phone number.

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);
```

```java
        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**.

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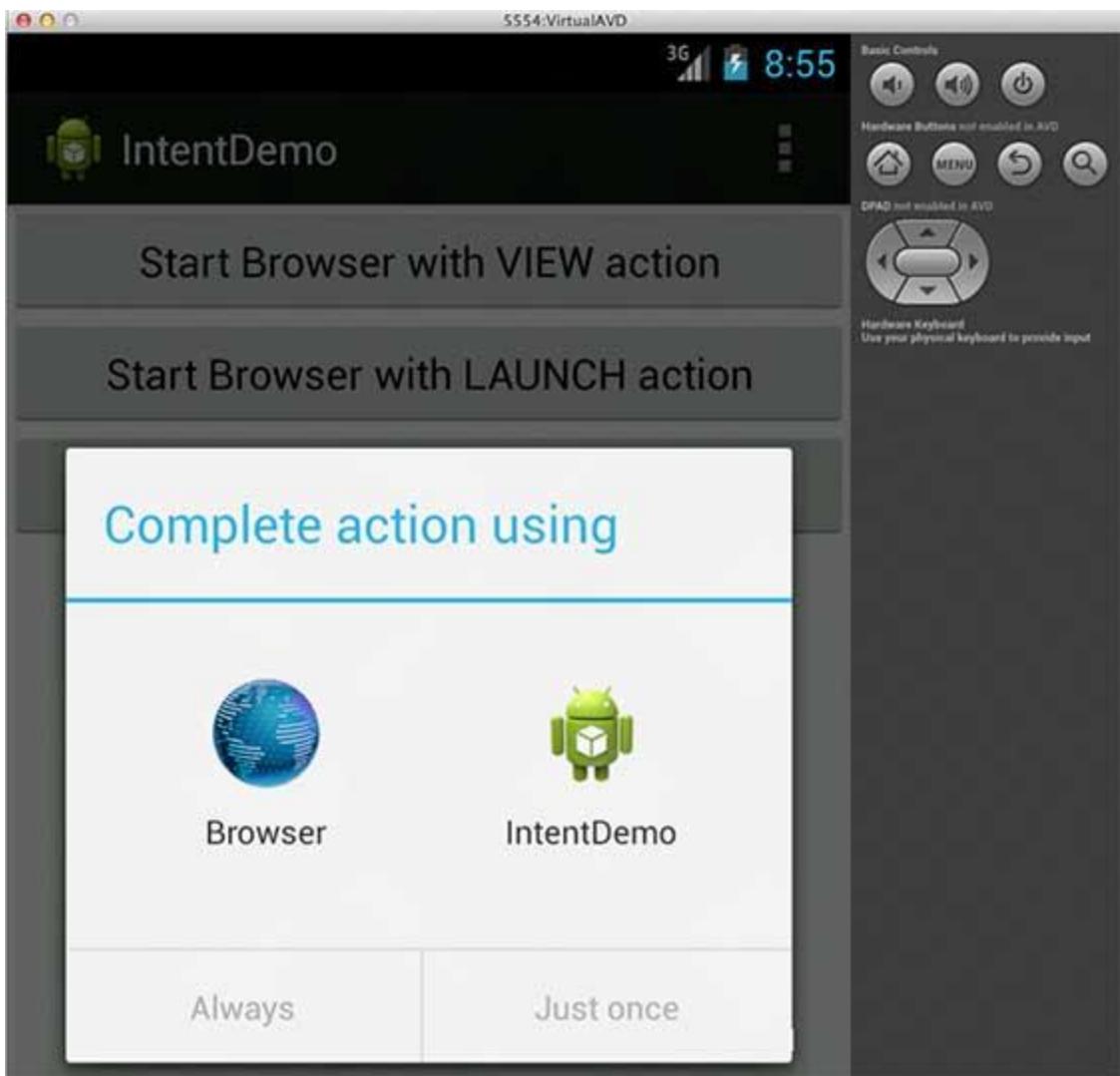
Now if you select Browser, then Android will launch web browser and open example.com website but if you select IndentDemo option then Android will launch CustomActivity which does nothing but just capture passed data and displays in a text view as follows:

http://www.example.com

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:
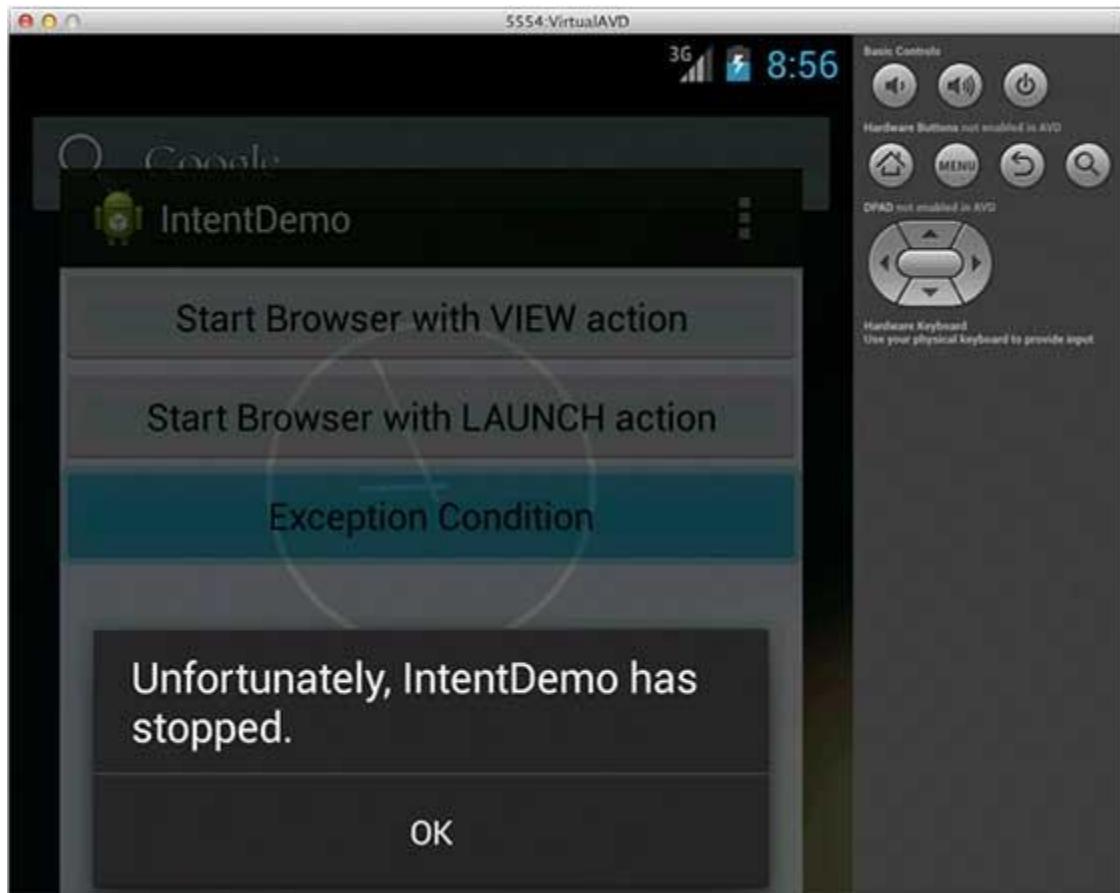
UI Layouts

The basic building block for user interface is a **View** object which is created from the View class and occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for widgets, which are used to create interactive UI components like buttons, text fields, etc.

The **ViewGroup** is a subclass of **View** and provides invisible container that hold other Views or other ViewGroups and define their layout properties.

At third level we have different layouts which are subclasses of ViewGroup class and a typical layout defines the visual structure for an Android user interface and can be created either at run time using **View/ViewGroup** objects or you can declare your layout using simple XML file **main_layout.xml** which is located in the res/layout folder of your project.

This tutorial is more about creating your GUI based on layouts defined in XML file. A layout may contain any type of widgets such as buttons, labels, textboxes, and so on. Following is a simple example of XML file having LinearLayout:

```xml
<?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="This is a TextView" />
        <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="This is a Button" />

        <!-- More GUI components go here  -->

</LinearLayout>
```

Once your layout is defined, you can load the layout resource from your application code, in your *Activity.onCreate()* callback implementation as shown below:

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

Android Layout Types

There are number of Layouts provided by Android which you will use in almost all the Android applications to provide different view, look and feel.

S.N.	Layout & Description
1	Linear Layout LinearLayout is a view group that aligns all children in a single direction, vertically or horizontally.
2	Relative Layout RelativeLayout is a view group that displays child views in relative positions.
3	Table Layout TableLayout is a view that groups views into rows and columns.
4	Absolute Layout AbsoluteLayout enables you to specify the exact location of its children.
5	Frame Layout The FrameLayout is a placeholder on screen that you can use to display a single view.
6	List View ListView is a view group that displays a list of scrollable items.
7	Grid View GridView is a ViewGroup that displays items in a two-dimensional, scrollable grid.

Linear Layout

Android LinearLayout is a view group that aligns all children in a single direction, *vertically* or *horizontally.*

LinearLayout Attributes

Following are the important attributes specific to LinearLayout:

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:baselineAligned	This must be a boolean value, either "true" or "false" and prevents the layout from aligning its children's baselines.
android:divider	This is drawable to use as a vertical divider between buttons. You use a color value, in the form of "#rgb", "#argb", "#rrggbb", or "#aarrggbb".
android:gravity	This specifies how an object should position its content, on both the X and Y axes. Possible values are top, bottom, left, right, center, center_vertical, center_horizontal etc.
android:orientation	This specifies the direction of arrangmet and you will use "horizontal" for a row, "vertical" for a column. The default is horizontal.

Example

This example will take you through simple steps to show how to create your own Android application using Linear Layout. 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 the default content of *res/layout/activity_main.xml* file to include few buttons in linear layout.
3	Define required constants *start_service, pause_service* and *stop_service* in *res/values/strings.xml* file
4	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.

```java
package com.example.helloworld;

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

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;
    }

}
```

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

```xml
<?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/btnStartService"
   android:layout_width="150px"
   android:layout_height="wrap_content"
   android:text="@string/start_service"

   <Button android:id="@+id/btnPauseService"
   android:layout_width="150px"
   android:layout_height="wrap_content"
   android:text="@string/pause_service"

   <Button android:id="@+id/btnStopService"
   android:layout_width="150px"
   android:layout_height="wrap_content"
   android:text="@string/stop_service"

</LinearLayout>
```

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

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

    <string name="app_name">HelloWorld</string>
    <string name="action_settings">Settings</string>
    <string name="start_service">Start</string>
    <string name="pause_service">Pause</string>
    <string name="stop_service">Stop</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 let's change the orientation of Layout as **android:orientation="horizontal"** and try to run the same application, it will give following screen:
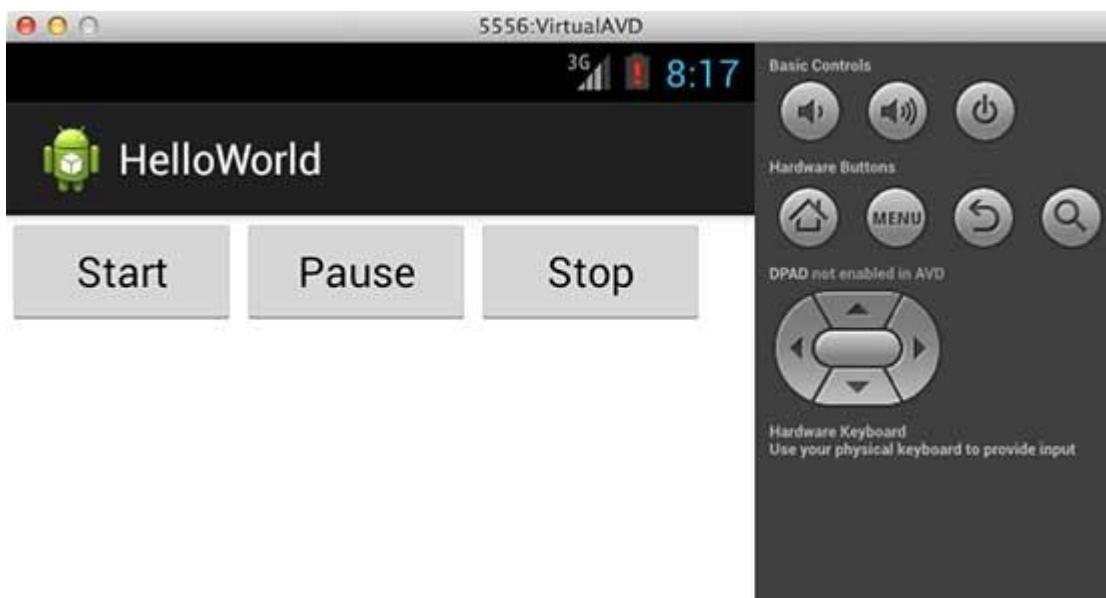
Relative Layout

Android RelativeLayout enables you to specify how child views are positioned relative to each other. The position of each view can be specified as relative to sibling elements or relative to the parent.

RelativeLayout Attributes

Following are the important attributes specific to RelativeLayout:

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:gravity	This specifies how an object should position its content, on both the X and Y axes. Possible values are top, bottom, left, right, center, center_vertical, center_horizontal etc.
android:ignoreGravity	This indicates what view should not be affected by gravity.

Using RelativeLayout, you can align two elements by right border, or make one below another, centered in the screen, centered left, and so on. By default, all child views are drawn at the top-left of the layout, so you must define the position of each view using the various layout properties available from**RelativeLayout.LayoutParams** and few of the important attributes are given below:

Attribute	Description
android:layout_above	Positions the bottom edge of this view above the given anchor view ID and must be a reference to another resource, in the form "@[+][package:]type:name"
android:layout_alignBottom	Makes the bottom edge of this view match the bottom edge of the given anchor view ID and must be a reference to another resource, in the form "@[+][package:]type:name".
android:layout_alignLeft	Makes the left edge of this view match the left edge of the given anchor view ID and must be a reference to another resource, in the form

	"@[+][package:]type:name".
android:layout_alignParentBottom	If true, makes the bottom edge of this view match the bottom edge of the parent. Must be a boolean value, either "true" or "false".
android:layout_alignParentEnd	If true, makes the end edge of this view match the end edge of the parent. Must be a boolean value, either "true" or "false".
android:layout_alignParentLeft	If true, makes the left edge of this view match the left edge of the parent. Must be a boolean value, either "true" or "false".
android:layout_alignParentRight	If true, makes the right edge of this view match the right edge of the parent. Must be a boolean value, either "true" or "false".
android:layout_alignParentStart	If true, makes the start edge of this view match the start edge of the parent. Must be a boolean value, either "true" or "false".
android:layout_alignParentTop	If true, makes the top edge of this view match the top edge of the parent. Must be a boolean value, either "true" or "false".
android:layout_alignRight	Makes the right edge of this view match the right edge of the given anchor view ID and must be a reference to another resource, in the form "@[+][package:]type:name".
android:layout_alignStart	Makes the start edge of this view match the start edge of the given anchor view ID and must be a reference to another resource, in the form "@[+][package:]type:name".
android:layout_alignTop	Makes the top edge of this view match the top edge of the given anchor view ID and must be a reference to another resource, in the form "@[+][package:]type:name".
android:layout_below	Positions the top edge of this view below the given anchor view ID and must be a reference to another resource, in the form "@[+][package:]type:name".
android:layout_centerHorizontal	If true, centers this child horizontally within its parent. Must be a boolean value, either "true" or "false".
android:layout_centerInParent	If true, centers this child horizontally and vertically within its parent. Must be a boolean value, either "true" or "false".
android:layout_centerVertical	If true, centers this child vertically within its parent. Must be a boolean value, either "true" or "false".
android:layout_toEndOf	Positions the start edge of this view to the end of the given anchor view ID and must be a reference to another resource, in the form "@[+][package:]type:name".
android:layout_toLeftOf	Positions the right edge of this view to the left of the given anchor view ID and must be a reference to another resource, in the form "@[+][package:]type:name".
android:layout_toRightOf	Positions the left edge of this view to the right of the given anchor view ID and must be a reference to another resource, in the form "@[+][package:]type:name".
android:layout_toStartOf	Positions the end edge of this view to the start of the given anchor view ID and must be a reference to another resource, in the form "@[+][package:]type:name".

Example

This example will take you through simple steps to show how to create your own Android application using Relative Layout. 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 the default content of *res/layout/activity_main.xml* file to include few widgets in Relative layout.
3	Define required constants in *res/values/strings.xml* file
4	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.

```
package com.example.helloworld;


import java.text.SimpleDateFormat;

import java.util.Calendar;

import java.util.Date;


import android.os.Bundle;

import android.app.Activity;

import android.text.format.DateFormat;

import android.view.Menu;

import android.widget.TextView;


public class MainActivity extends Activity {

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);


        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd");

        Date date = new Date();

         String nowDate = dateFormat.format(date);

         TextView dateView =  (TextView)findViewById(R.id.dates);
```

```
        dateView.setText(nowDate);


        SimpleDateFormat timeFormat = new SimpleDateFormat("HH:mm:ss");
        String nowTime = timeFormat.format(date);
        TextView timeView =  (TextView)findViewById(R.id.times);
        timeView.setText(nowTime);
    }



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

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

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp" >
    <EditText
        android:id="@+id/name"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/reminder" />
    <TextView
        android:id="@+id/dates"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/name"
        android:layout_alignParentLeft="true"
```

```
      android:layout_toLeftOf="@+id/times" />

   <TextView

      android:id="@id/times"

      android:layout_width="96dp"

      android:layout_height="wrap_content"

      android:layout_below="@id/name"

      android:layout_alignParentRight="true" />

   <Button

      android:layout_width="96dp"

      android:layout_height="wrap_content"

      android:layout_below="@id/times"

      android:layout_centerInParent="true"

      android:text="@string/done" />

</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">HelloWorld</string>

   <string name="action_settings">Settings</string>

   <string name="reminder">Enter your name</string>

   <string name="done">Done</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:
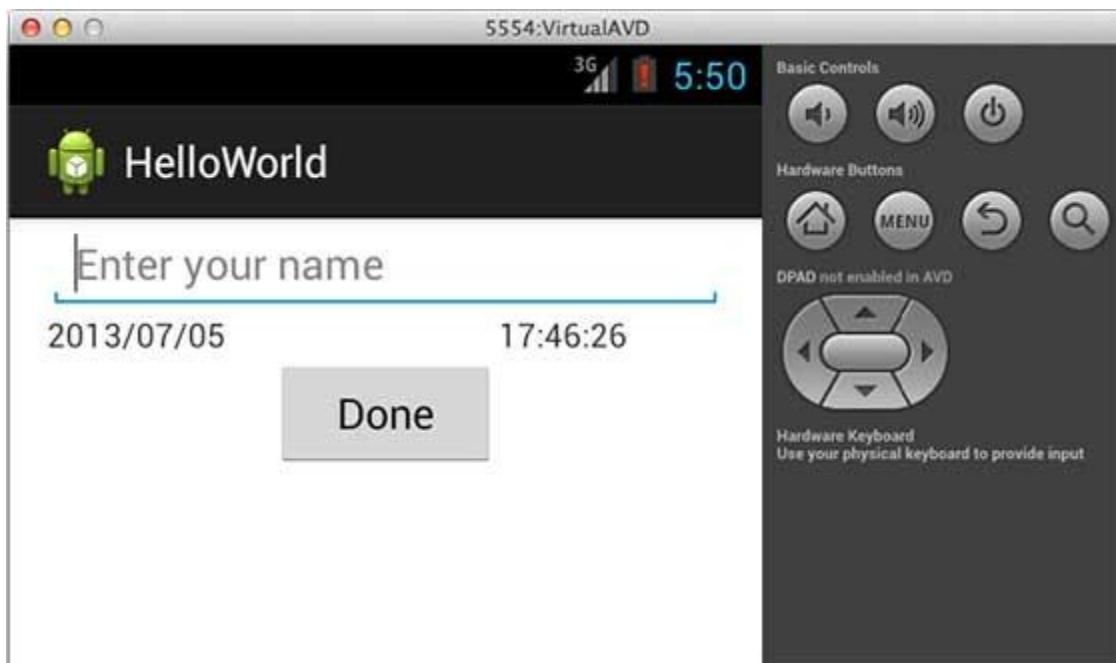
Table Layout

Android TableLayout groups views into rows and columns. You will use the <TableRow> element to build a row in the table. Each row has zero or more cells; each cell can hold one View object.

TableLayout containers do not display border lines for their rows, columns, or cells.

TableLayout Attributes

Following are the important attributes specific to TableLayout:

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:collapseColumns	This specifies the zero-based index of the columns to collapse. The column indices must be separated by a comma: 1, 2, 5.
android:collapseColumns	The zero-based index of the columns to shrink. The column indices must be separated by a comma: 1, 2, 5.
android:stretchColumns	The zero-based index of the columns to stretch. The column indices must be separated by a comma: 1, 2, 5.

Example

This example will take you through simple steps to show how to create your own Android application using Table Layout. 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 the default content of *res/layout/activity_main.xml* file to include few widgets in table layout.
3	Define required constants in *res/values/strings.xml* file
4	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.

```java
package com.example.helloworld;

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

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;
    }

}
```

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

```xml
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TableRow>
        <Button
            android:id="@+id/backbutton"
            android:text="Back"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
    </TableRow>
    <TableRow>
        <TextView
            android:text="First Name"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_column="1" />
        <EditText
            android:width="100px"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
    </TableRow>
    <TableRow>
        <TextView
            android:text="Last Name"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_column="1" />
```

```
        <EditText
        android:width="100px"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    </TableRow>
</TableLayout>
```

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">HelloWorld</string>
    <string name="action_settings">Settings</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:

Absolute Layout

An Absolute Layout lets you specify exact locations (x/y coordinates) of its children. Absolute layouts are less flexible and harder to maintain than other types of layouts without absolute positioning.

AbsoluteLayout Attributes

Following are the important attributes specific to AbsoluteLayout:

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.

android:layout_x	This specifies the x-coordinate of the view.
android:layout_y	This specifies the y-coordinate of the view.

Example

This example will take you through simple steps to show how to create your own Android application using absolute layout. 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 the default content of *res/layout/activity_main.xml* file to include few widgets in absolute layout.
3	Define required constants in *res/values/strings.xml* file
4	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.

```java
package com.example.helloworld;

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

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;
    }

}
```

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

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

    <Button
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:text="OK"
        android:layout_x="50px"
        android:layout_y="361px" />
    <Button
        android:layout_width="100dp"
```

```
        android:layout_height="wrap_content"
        android:text="Cancel"
        android:layout_x="225px"
        android:layout_y="361px" />

</AbsoluteLayout>
```

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">HelloWorld</string>
    <string name="action_settings">Settings</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:

Frame Layout

Frame Layout is designed to block out an area on the screen to display a single item. Generally, FrameLayout should be used to hold a single child view, because it can be difficult to organize child views in a way that's scalable to different screen sizes without the children overlapping each other.

You can, however, add multiple children to a FrameLayout and control their position within the FrameLayout by assigning gravity to each child, using the android:layout_gravity attribute.

FrameLayout Attributes

Following are the important attributes specific to FrameLayout:

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:foreground	This defines the drawable to draw over the content and possible values may be a color value, in the form of "#rgb", "#argb", "#rrggbb", or "#aarrggbb".
android:foregroundGravity	Defines the gravity to apply to the foreground drawable. The gravity defaults to fill. Possible values are top, bottom, left, right, center, center_vertical, center_horizontal etc.
android:measureAllChildren	Determines whether to measure all children or just those in the VISIBLE or INVISIBLE state when measuring. Defaults to false.

Example

This example will take you through simple steps to show how to create your own Android application using frame layout. 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 the default content of *res/layout/activity_main.xml* file to include few widgets in frame layout.
3	Define required constants in *res/values/strings.xml* file
4	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.

```
package com.example.helloworld;

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

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;
    }

}
```

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

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

```
    android:layout_height="fill_parent">

    <ImageView
    android:src="@drawable/ic_launcher"
    android:scaleType="fitCenter"
    android:layout_height="250px"
    android:layout_width="250px"/>

    <TextView
    android:text="Frame Demo"
    android:textSize="30px"
    android:textStyle="bold"
    android:layout_height="fill_parent"
    android:layout_width="fill_parent"
    android:gravity="center"/>
</FrameLayout>
```

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">HelloWorld</string>
    <string name="action_settings">Settings</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:

List View

Android **ListView** is a view which groups several items and display them in vertical scrollable list. The list items are automatically inserted to the list using an **Adapter** that pulls content from a source such as an array or database.

An adapter actually bridges between UI components and the data source that fill data into UI Component. Adapter can be used to supply the data to like spinner, list view, grid view etc.

The **ListView** and **GridView** are subclasses of **AdapterView** and they can be populated by binding them to an **Adapter**, which retrieves data from an external source and creates a View that represents each data entry. Android provides several subclasses of Adapter that are useful for retrieving different kinds of data and building views for an AdapterView (ie. ListView or GridView). The two most common adapters are*ArrayAdapter* and *SimpleCursorAdapter*. We will see separate examples for both the adapters.

ListView Attributes

Following are the important attributes specific to GridView:

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:divider	This is drawable or color to draw between list items. .
android:dividerHeight	This specifies height of the divider. This could be in px, dp, sp, in, or mm.
android:entries	Specifies the reference to an array resource that will populate the ListView.
android:footerDividersEnabled	When set to false, the ListView will not draw the divider before each footer view. The default value is true.
android:headerDividersEnabled	When set to false, the ListView will not draw the divider after each header view. The default value is true.

ArrayAdapter

You can use this adapter when your data source is an array. By default, ArrayAdapter creates a view for each array item by calling toString() on each item and placing the contents in a **TextView**. Consider you have an array of strings you want to display in a ListView, initialize a new **ArrayAdapter** using a constructor to specify the layout for each string and the string array:

```
ArrayAdapter adapter = new ArrayAdapter<String>(this,
        R.layout.ListView,
        StringArray);
```

Here are arguments for this constructor:

- First argument **this** is the application context. Most of the case, keep it **this**.

- Second argument will be layout defined in XML file and having **TextView** for each string in the array.

- Final argument is an array of strings which will be populated in the text view.

Once you have array adaptor created, then simply call **setAdapter()** on your **ListView** object as follows:

```
ListView listView = (ListView) findViewById(R.id.listview);
listView.setAdapter(adapter);
```

You will define your list view under res/layout directory in an XML file. For our example we are going to using activity_main.xml file.

EXAMPLE

Following is the example which will take you through simple steps to show how to create your own Android application using ListView. 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 the default content of *res/layout/activity_main.xml* file to include ListView content with the self explanatory attributes.
3	Define required constants in *res/values/strings.xml* file.
4	Create a Text View file *res/layout/activity_listview.xml*. This file will have setting to display all the list items. So you can customize its fonts, padding, color etc. using this file.
6	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.

```java
package com.example.helloworld;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.widget.ArrayAdapter;
import android.widget.ListView;

public class MainActivity extends Activity {

   // Array of strings...
    String[] countryArray = {"India", "Pakistan", "USA", "UK"};

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

      ArrayAdapter adapter = new ArrayAdapter<String>(this,
      R.layout.activity_listview, countryArray);

      ListView listView = (ListView) findViewById(R.id.country_list);
      listView.setAdapter(adapter);
   }
}
```

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

```xml
<LinearLayout 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:orientation="vertical"
tools:context=".ListActivity" >

   <ListView
```

TUTORIALS POINT
Simply Easy Learning

```
   android:id="@+id/country_list"
   android:layout_width="match_parent"
   android:layout_height="wrap_content" >
   </ListView>

</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">HelloWorld</string>
   <string name="action_settings">Settings</string>

</resources>
```

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

```
<?xml version="1.0" encoding="utf-8"?>
<!--  Single List Item Design -->
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/label"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:padding="10dip"
        android:textSize="16dip"
        android:textStyle="bold" >
</TextView>
```

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:
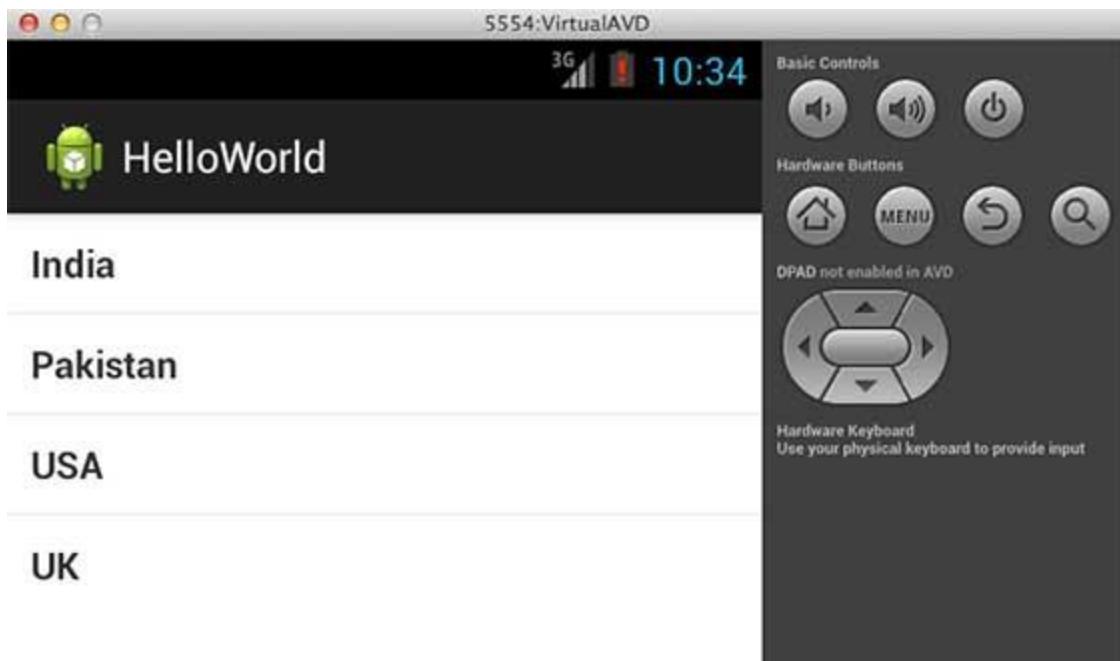
SimpleCursorAdapter

You can use this adapter when your data source is a database Cursor. When using *SimpleCursorAdapter*, you must specify a layout to use for each row in the **Cursor** and which columns in the Cursor should be inserted into which views of the layout.

For example, if you want to create a list of people's names and phone numbers, you can perform a query that returns a Cursor containing a row for each person and columns for the names and numbers. You then create a string array specifying which columns from the Cursor you want in the layout for each result and an integer array specifying the corresponding views that each column should be placed:

```
String[] fromColumns = {ContactsContract.Data.DISPLAY_NAME,
                        ContactsContract.CommonDataKinds.Phone.NUMBER};
int[] toViews = {R.id.display_name, R.id.phone_number};
```

When you instantiate the SimpleCursorAdapter, pass the layout to use for each result, the Cursor containing the results, and these two arrays:

```
SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
        R.layout.person_name_and_number, cursor, fromColumns, toViews, 0);

ListView listView = getListView();
listView.setAdapter(adapter);
```

The SimpleCursorAdapter then creates a view for each row in the Cursor using the provided layout by inserting each fromColumns item into the corresponding **toViews** view.

GridView

Android **GridView** shows items in two-dimensional scrolling grid (rows & columns) and the grid items are not necessarily predetermined but they automatically inserted to the layout using a **ListAdapter**

An adapter actually bridges between UI components and the data source that fill data into UI Component. Adapter can be used to supply the data to like spinner, list view, grid view etc.

The **ListView** and **GridView** are subclasses of **AdapterView** and they can be populated by binding them to an **Adapter**, which retrieves data from an external source and creates a View that represents each data entry.

GridView Attributes

Following are the important attributes specific to GridView:

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:columnWidth	This specifies the fixed width for each column. This could be in px, dp, sp, in, or mm.
android:gravity	Specifies the gravity within each cell. Possible values are top, bottom, left, right, center, center_vertical, center_horizontal etc.
android:horizontalSpacing	Defines the default horizontal spacing between columns. This could be in px, dp, sp, in, or mm.
android:numColumns	Defines how many columns to show. May be an integer value, such as "100" or auto_fit which means display as many columns as possible to fill the available space.

	Defines how columns should stretch to fill the available empty space, if any. This must be either of the values:
	none: Stretching is disabled.
android:stretchMode	spacingWidth: The spacing between each column is stretched.
	columnWidth: Each column is stretched equally.
	spacingWidthUniform: The spacing between each column is uniformly stretched..
android:verticalSpacing	Defines the default vertical spacing between rows. This could be in px, dp, sp, in, or mm.

Example

This example will take you through simple steps to show how to create your own Android application using GridView. 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 the default content of *res/layout/activity_main.xml* file to include GridView content with the self explanatory attributes.
3	Define required constants in *res/values/strings.xml* file.
4	Let's put few pictures in *res/drawable-hdpi* folder. I have put sample0.jpg, sample1.jpg, sample2.jpg, sample3.jpg, sample4.jpg, sample5.jpg, sample6.jpg and sample7.jpg.
5	Create a new class called **ImageAdapter** under a package com.example.helloworld that extends BaseAdapter. This class will implement functionality of an adaptor to be used to fill the view.
6	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.

```
package com.example.helloworld;


import android.os.Bundle;

import android.app.Activity;

import android.view.Menu;

import android.widget.GridView;


public class MainActivity extends Activity {


    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);


        GridView gridview = (GridView) findViewById(R.id.gridview);

        gridview.setAdapter(new ImageAdapter(this));

    }


    @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"?>

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

    android:id="@+id/gridview"

    android:layout_width="fill_parent"

    android:layout_height="fill_parent"

    android:columnWidth="90dp"

    android:numColumns="auto_fit"

    android:verticalSpacing="10dp"

    android:horizontalSpacing="10dp"

    android:stretchMode="columnWidth"

    android:gravity="center"

/>
```

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">HelloWorld</string>

    <string name="action_settings">Settings</string>


</resources>
```

Following will be the content of **src/com.example.helloworld/ImageAdapter.java** file:

```
package com.example.helloworld;


import android.content.Context;

import android.view.View;

import android.view.ViewGroup;

import android.widget.BaseAdapter;

import android.widget.GridView;

import android.widget.ImageView;


public class ImageAdapter extends BaseAdapter {

    private Context mContext;


    // Constructor

    public ImageAdapter(Context c) {

        mContext = c;

    }


    public int getCount() {

        return mThumbIds.length;

    }


    public Object getItem(int position) {

        return null;

    }


    public long getItemId(int position) {

        return 0;

    }
```

```java
// create a new ImageView for each item referenced by the Adapter
public View getView(int position, View convertView, ViewGroup parent) {
    ImageView imageView;
    if (convertView == null) {
    imageView = new ImageView(mContext);
    imageView.setLayoutParams(new GridView.LayoutParams(85, 85));
    imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
    imageView.setPadding(8, 8, 8, 8);
    } else {
    imageView = (ImageView) convertView;
    }

    imageView.setImageResource(mThumbIds[position]);
    return imageView;
}


// Keep all Images in array
public Integer[] mThumbIds = {
    R.drawable.sample_2, R.drawable.sample_3,
    R.drawable.sample_4, R.drawable.sample_5,
    R.drawable.sample_6, R.drawable.sample_7,
    R.drawable.sample_0, R.drawable.sample_1,
    R.drawable.sample_2, R.drawable.sample_3,
    R.drawable.sample_4, R.drawable.sample_5,
    R.drawable.sample_6, R.drawable.sample_7,
    R.drawable.sample_0, R.drawable.sample_1,
    R.drawable.sample_2, R.drawable.sample_3,
    R.drawable.sample_4, R.drawable.sample_5,
    R.drawable.sample_6, R.drawable.sample_7
};
}
```

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:
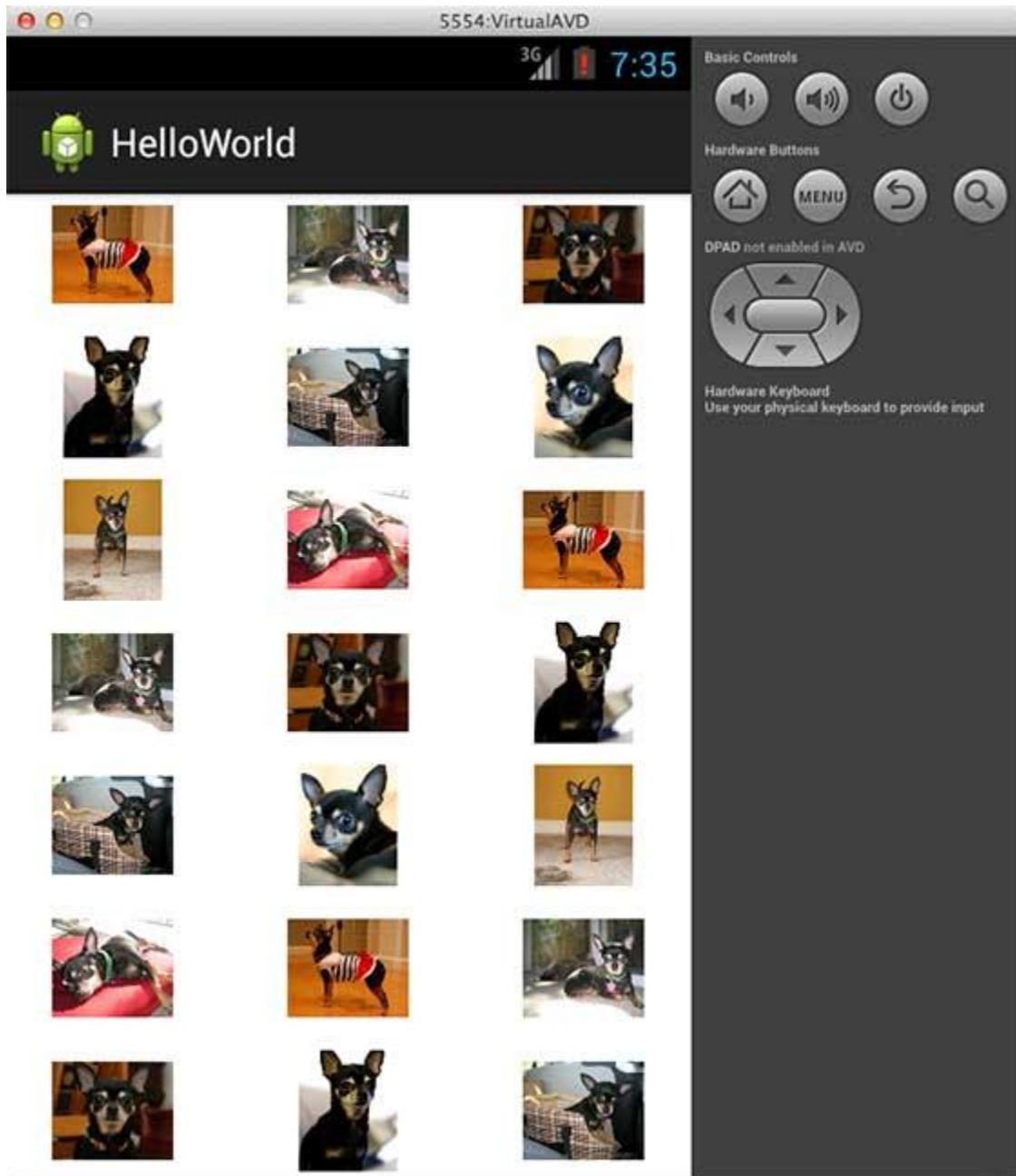
Sub-Activity Example

Let's extend the functionality of above example where we will show selected grid image in full screen. To achieve this we need to introduce a new activity. Just keep in mind for any activity we need perform all the steps like we have to implement an activity class, define that activity in AndroidManifest.xml file, define related layout and finally link that sub-activity with the main activity by it in the main activity class. So let's follow the steps to modify above example:

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	Create a new Activity class as *SingleViewActivity.java* under a package*com.example.helloworld* as shown below.
3	Create new layout file for the new activity under **res/layout/** folder. Let's name this XML file as single_view.xml.
4	Define your new activity in *AndroidManifest.xml* file using tag. An application can have one or more activities without any restrictions.
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.helloworld/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.helloworld;


import android.app.Activity;

import android.content.Intent;

import android.os.Bundle;

import android.view.Menu;

import android.view.View;

import android.widget.AdapterView;

import android.widget.AdapterView.OnItemClickListener;

import android.widget.GridView;


public class MainActivity extends Activity {

   @Override

   protected void onCreate(Bundle savedInstanceState) {

      super.onCreate(savedInstanceState);

      setContentView(R.layout.activity_main);


      GridView gridview = (GridView) findViewById(R.id.gridview);

      gridview.setAdapter(new ImageAdapter(this));


      gridview.setOnItemClickListener(new OnItemClickListener() {

         public void onItemClick(AdapterView<?> parent, View v,
```

```
                 int position, long id) {

              // Send intent to SingleViewActivity

              Intent i =

              new Intent(getApplicationContext(), SingleViewActivity.class);

              // Pass image index

              i.putExtra("id", position);

              startActivity(i);

         }

      });

   }


   @Override

   public boolean onCreateOptionsMenu(Menu menu) {

      getMenuInflater().inflate(R.menu.main, menu);

      return true;

   }

}
```

Following will be the content of new activity file **src/com.example.helloworld/SingleViewActivity.java**file:

```
package com.example.helloworld;


import android.app.Activity;

import android.content.Intent;

import android.os.Bundle;

import android.widget.ImageView;


public class SingleViewActivity extends Activity {


    @Override

    public void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

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

```
        // Get intent data

        Intent i = getIntent();


        // Selected image id

        int position = i.getExtras().getInt("id");

        ImageAdapter imageAdapter = new ImageAdapter(this);


        ImageView imageView = (ImageView) findViewById(R.id.SingleView);

        imageView.setImageResource(imageAdapter.mThumbIds[position]);

    }

}
```

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

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

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

    android:layout_width="match_parent"

    android:layout_height="match_parent"

    android:orientation="vertical" >


   <ImageView android:id="@+id/SingleView"

            android:layout_width="fill_parent"

            android:layout_height="fill_parent"/>


</LinearLayout>
```

Following will be the content of **AndroidManifest.xml** to define two new constants:

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

<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="17" />


    <application

        android:allowBackup="true"

        android:icon="@drawable/ic_launcher"

        android:label="@string/app_name"

        android:theme="@style/AppTheme" >

        <activity

            android:name="com.example.helloworld.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=".SingleViewActivity"></activity>

    </application>

</manifest>
```

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 if you click on either of the images it will be displayed as a single image, for example:
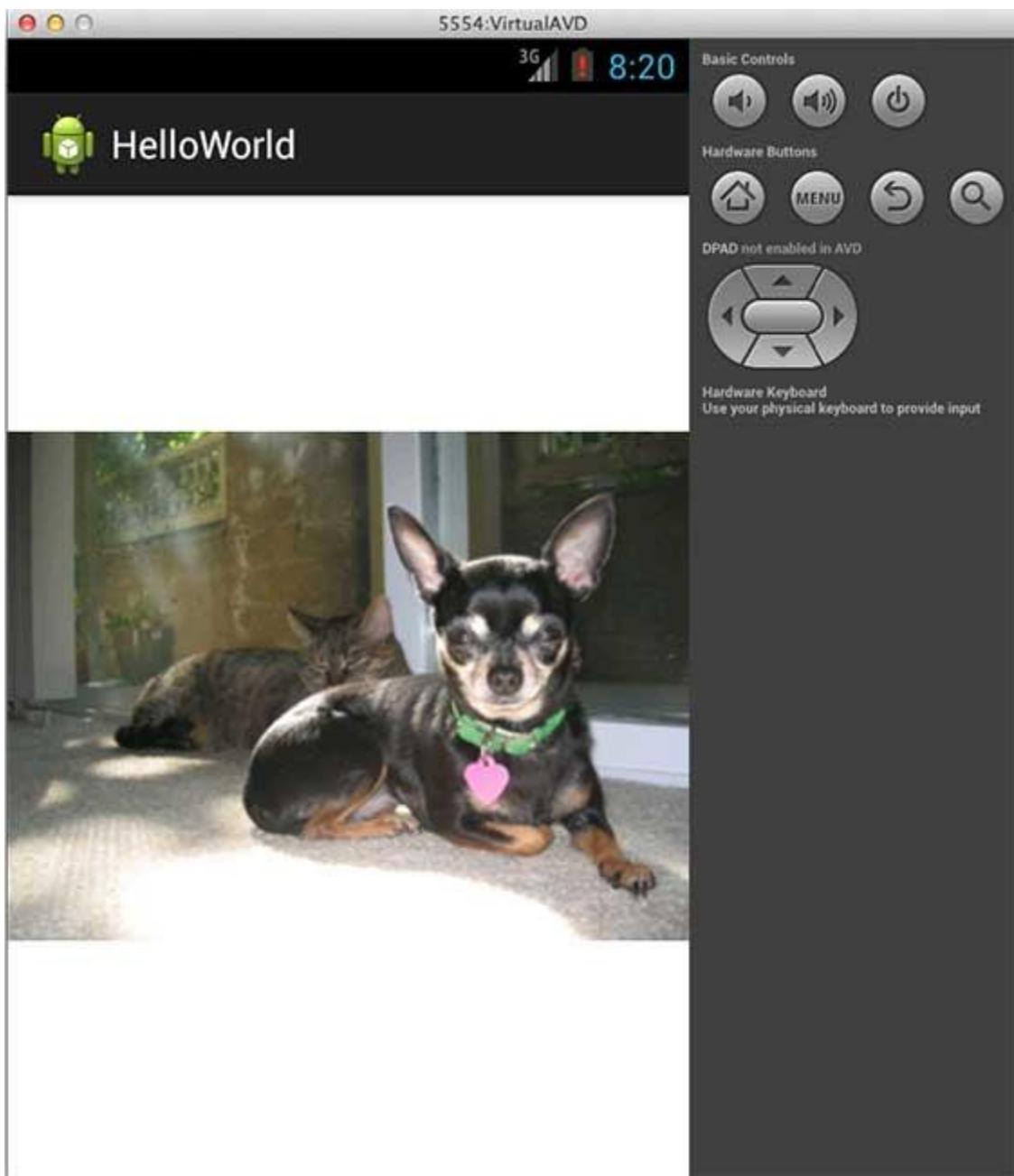
Kindly note above mentioned images have been taken from Android official website.

Layout Attributes

Each layout has a set of attributes which define the visual properties of that layout. There are few common attributes among all the layouts and their are other attributes which are specific to that layout. Following are common attributes and will be applied to all the layouts:

Attribute	Description
android:id	This is the ID which uniquely identifies the view.

android:layout_width	This is the width of the layout.
android:layout_height	This is the height of the layout
android:layout_marginTop	This is the extra space on the top side of the layout.
android:layout_marginBottom	This is the extra space on the bottom side of the layout.
android:layout_marginLeft	This is the extra space on the left side of the layout.
android:layout_marginRight	This is the extra space on the right side of the layout.
android:layout_gravity	This specifies how child Views are positioned.
android:layout_weight	This specifies how much of the extra space in the layout should be allocated to the View.
android:layout_x	This specifies the x-coordinate of the layout.
android:layout_y	This specifies the y-coordinate of the layout.
android:layout_width	This is the width of the layout.
android:layout_width	This is the width of the layout.
android:paddingLeft	This is the left padding filled for the layout.
android:paddingRight	This is the right padding filled for the layout.
android:paddingTop	This is the top padding filled for the layout.
android:paddingBottom	This is the bottom padding filled for the layout.

Here width and height are the dimension of the layout/view which can be specified in terms of dp (Density-independent Pixels), sp (Scale-independent Pixels), pt (Points which is 1/72 of an inch), px(Pixels), mm (Millimeters) and finally in (inches).

You can specify width and height with exact measurements but more often, you will use one of these constants to set the width or height:

- **android:layout_width=wrap_content** tells your view to size itself to the dimensions required by its content.
- **android:layout_width=fill_parent** tells your view to become as big as its parent view.

Gravity attribute plays important role in positioning the view object and it can take one or more (separated by '|') of the following constant values.

Constant	Value	Description
top	0x30	Push object to the top of its container, not changing its size.
bottom	0x50	Push object to the bottom of its container, not changing its size.
left	0x03	Push object to the left of its container, not changing its size.
right	0x05	Push object to the right of its container, not changing its size.
center_vertical	0x10	Place object in the vertical center of its container, not changing its size.
fill_vertical	0x70	Grow the vertical size of the object if needed so it completely fills its container.
center_horizontal	0x01	Place object in the horizontal center of its container, not changing its size.

fill_horizontal	0x07	Grow the horizontal size of the object if needed so it completely fills its container.
center	0x11	Place the object in the center of its container in both the vertical and horizontal axis, not changing its size.
fill	0x77	Grow the horizontal and vertical size of the object if needed so it completely fills its container.
clip_vertical	0x80	Additional option that can be set to have the top and/or bottom edges of the child clipped to its container's bounds. The clip will be based on the vertical gravity: a top gravity will clip the bottom edge, a bottom gravity will clip the top edge, and neither will clip both edges.
clip_horizontal	0x08	Additional option that can be set to have the left and/or right edges of the child clipped to its container's bounds. The clip will be based on the horizontal gravity: a left gravity will clip the right edge, a right gravity will clip the left edge, and neither will clip both edges.
start	0x00800003	Push object to the beginning of its container, not changing its size.
end	0x00800005	Push object to the end of its container, not changing its size.

View Identification

A view object may have a unique ID assigned to it which will identify the View uniquely within the tree. The syntax for an ID, inside an XML tag is:

```
android:id="@+id/my_button"
```

Following is a brief description of @ and + signs:

- The at-symbol (@) at the beginning of the string indicates that the XML parser should parse and expand the rest of the ID string and identify it as an ID resource.

- The plus-symbol (+) means that this is a new resource name that must be created and added to our resources. To create an instance of the view object and capture it from the layout, use the following:

```
Button myButton = (Button) findViewById(R.id.my_button);
```

UI Controls

A n Android application user interface is everything that the user can see and interact with. You have

learned about the various layouts that you can use to position your views in an activity. This chapter will give you detail on various views.

A **View** is an object that draws something on the screen that the user can interact with and a **ViewGroup** is an object that holds other View (and ViewGroup) objects in order to define the layout of the user interface.

You define your layout in an XML file which offers a human-readable structure for the layout, similar to HTML. For example, a simple vertical layout with a text view and a button looks like this:

```xml
<?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="I am a TextView" />
    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="I am a Button" />
</LinearLayout>
```

Android UI Controls

There are number of UI controls provided by Android that allow you to build the graphical user interface for your app.

S.N.	UI Control & Description
1	TextView This control is used to display text to the user.
2	EditText EditText is a predefined subclass of TextView that includes rich editing capabilities.
3	AutoCompleteTextView The AutoCompleteTextView is a view that is similar to EditText, except that it shows a list of completion

		suggestions automatically while the user is typing.
4		Button A push-button that can be pressed, or clicked, by the user to perform an action.
5		ImageButton AbsoluteLayout enables you to specify the exact location of its children.
6		CheckBox An on/off switch that can be toggled by the user. You should use checkboxes when presenting users with a group of selectable options that are not mutually exclusive.
7		ToggleButton An on/off button with a light indicator.
8		RadioButton The RadioButton has two states: either checked or unchecked.
9		RadioGroup A RadioGroup is used to group together one or more RadioButtons.
10		ProgressBar The ProgressBar view provides visual feedback about some ongoing tasks, such as when you are performing a task in the background.
11		Spinner A drop-down list that allows users to select one value from a set.
12		TimePicker The TimePicker view enables users to select a time of the day, in either 24-hour mode or AM/PM mode.
13		DatePicker The DatePicker view enables users to select a date of the day.

TextView

A TextView displays text to the user and optionally allows them to edit it. A TextView is a complete text editor, however the basic class is configured to not allow editing.

TextView Attributes

Following are the important attributes related to TextView control. You can check Android official documentation for complete list of attributes and related methods which you can use to change these attributes are run time.

Attribute	Description
android:id	This is the ID which uniquely identifies the control.
android:capitalize	If set, specifies that this TextView has a textual input method and should automatically capitalize what the user types. Don't automatically capitalize anything - 0 Capitalize the first word of each sentence - 1 Capitalize the first letter of every word - 2

	Capitalize every character - 3
android:cursorVisible	Makes the cursor visible (the default) or invisible. Default is false.
android:editable	If set to true, specifies that this TextView has an input method.
android:fontFamily	Font family (named by string) for the text.
android:gravity	Specifies how to align the text by the view's x- and/or y-axis when the text is smaller than the view.
android:hint	Hint text to display when the text is empty.
android:inputType	The type of data being placed in a text field. Phone, Date, Time, Number, Password etc.
android:maxHeight	Makes the TextView be at most this many pixels tall.
android:maxWidth	Makes the TextView be at most this many pixels wide.
android:minHeight	Makes the TextView be at least this many pixels tall.
android:minWidth	Makes the TextView be at least this many pixels wide.
android:password	Whether the characters of the field are displayed as password dots instead of themselves. Possible value either "true" or "false".
android:phoneNumber	If set, specifies that this TextView has a phone number input method. Possible value either "true" or "false".
android:text	Text to display.
android:textAllCaps	Present the text in ALL CAPS. Possible value either "true" or "false".
android:textColor	Text color. May be a color value, in the form of "#rgb", "#argb", "#rrggbb", or "#aarrggbb".
android:textColorHighlight	Color of the text selection highlight.
android:textColorHint	Color of the hint text. May be a color value, in the form of "#rgb", "#argb", "#rrggbb", or "#aarrggbb".
android:textIsSelectable	Indicates that the content of a non-editable text can be selected. Possible value either "true" or "false".
android:textSize	Size of the text. Recommended dimension type for text is "sp" for scaled-pixels (example: 15sp).
android:textStyle	Style (bold, italic, bolditalic) for the text. You can use or more of the following values separated by '|'. normal - 0 bold - 1 italic - 2
android:typeface	Typeface (normal, sans, serif, monospace) for the text. You can use or more of the following values separated by '|'.

	normal - 0
	sans - 1
	serif - 2
	monospace – 3

Example

This example will take you through simple steps to show how to create your own Android application using Linear Layout and TextView.

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *GUIDemo* under a package *com.example.guidemo* as explained in the *Hello World Example* chapter.
2	Modify *src/MainActivity.java* file to add a click event.
2	Modify the default content of *res/layout/activity_main.xml* file to include Android UI control.
3	Define required constants in *res/values/strings.xml* file
4	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.guidemo/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.guidemo;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.view.View;
import android.widget.TextView;
import android.widget.Toast;

public class MainActivity extends Activity {

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

        //--- text view---
        TextView txtView = (TextView) findViewById(R.id.text_id);
        final String Label = txtView.getText().toString();

        txtView.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                Toast.makeText(getApplicationContext(),
                "You have clicked the Label : " + Label,
                Toast.LENGTH_LONG).show();
            }

        });
```

```
    }

    @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:

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

</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">GUIDemo</string>
    <string name="action_settings">Settings</string>
    <string name="hello_world">Hello world!</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 **GUIDemo** 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 click on the Lable "Hello World", it will give following screen:
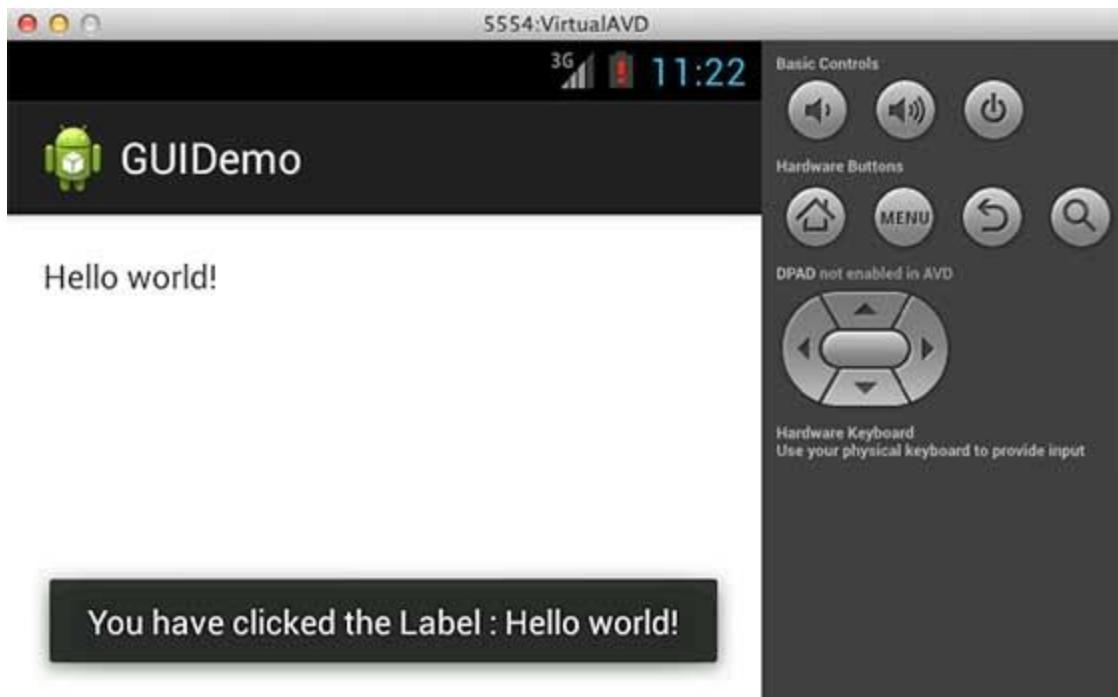
Exercise:

I will recommend to try above example with different attributes of TextView in Layout XML file as well at programming time to have different look and feel of the TextView. Try to make it editable, change to font color, font family, width, textSize etc and see the result. You can also try above example with multiple TextView controls in one acitivity.

EditText

A EditText is an overlay over TextView that configures itself to be editable. It is the predefined subclass of TextView that includes rich editing capabilities.

EditText Attributes

Following are the important attributes related to EditText control. You can check Android official documentation for complete list of attributes and related methods which you can use to change these attributes are run time.

Inherited from **android.widget.TextView** Class:

Attribute	Description
android:autoText	If set, specifies that this TextView has a textual input method and automatically corrects some common spelling errors.
android:drawableBottom	This is the drawable to be drawn below the text.
android:drawableRight	This is the drawable to be drawn to the right of the text.
android:editable	If set, specifies that this TextView has an input method.
android:text	This is the Text to display.

Inherited from **android.view.View** Class:

Attribute	Description
android:background	This is a drawable to use as the background.
android:contentDescription	This defines text that briefly describes content of the view.
android:id	This supplies an identifier name for this view,
android:onClick	This is the name of the method in this View's context to invoke when the view is clicked.
android:visibility	This controls the initial visibility of the view.

Example

This example will take you through simple steps to show how to create your own Android application using Linear Layout and EditText.

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *GUIDemo2* under a package *com.example.guidemo2* as explained in the *Hello World Example* chapter.
2	Modify *src/MainActivity.java* file to add a click event.
2	Modify the default content of *res/layout/activity_main.xml* file to include Android UI control.
3	Define required constants in *res/values/strings.xml* file
4	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.guidemo2/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.guidemo2;

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

public class MainActivity extends Activity {

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

        final EditText eText;
        final Button btn;

        eText = (EditText) findViewById(R.id.edittext);
        btn = (Button) findViewById(R.id.button);

        btn.setOnClickListener(new OnClickListener() {
```

```
        public void onClick(View v) {
           String str = eText.getText().toString();
           Toast msg = Toast.makeText(getBaseContext(),str,
           Toast.LENGTH_LONG);
           msg.show();
           msg.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:

```xml
<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:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:layout_marginLeft="14dp"
        android:layout_marginTop="18dp"
        android:text="@string/example_edittext" />

   <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/textView1"
        android:layout_below="@+id/textView1"
        android:layout_marginTop="130dp"
        android:text="@string/show_the_text" />

   <EditText
        android:id="@+id/edittext"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/button"
        android:layout_below="@+id/textView1"
        android:layout_marginTop="61dp"
        android:ems="10"
        android:text="@string/enter_text" android:inputType="text" />
```

```
</RelativeLayout>
```

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

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

    <string name="app_name">GUIDemo1</string>
    <string name="action_settings">Settings</string>
    <string name="example_edittext">Example showing EditText</string>
    <string name="show_the_text">Show the Text</string>
    <string name="enter_text">text changes</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.guidemo2"
    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.guidemo2.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 **GUIDemo2** 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:

Exercise:

I will recommend to try above example with different attributes of EditText in Layout XML file as well at programming time to have different look and feel of the EditText. Try to make it editable, change to font color, font family, width, textSize etc and see the result. You can also try above example with multiple EditText controls in one activity.

AutoCompleteTextView

A AutoCompleteTextView is a view that is similar to EditText, except that it shows a list of completion suggestions automatically while the user is typing. The list of suggestions is displayed in drop down menu. The user can choose an item from there to replace the content of edit box with.

AutoCompleteTextView Attributes

Following are the important attributes related to AutoCompleteTextView control. You can check Android official documentation for complete list of attributes and related methods which you can use to change these attributes are run time.

Attribute	Description
android:completionHint	This defines the hint displayed in the drop down menu.
android:completionHintView	This defines the hint view displayed in the drop down menu.
android:completionThreshold	This defines the number of characters that the user must type before completion suggestions are displayed in a drop down menu.
android:dropDownAnchor	This is the View to anchor the auto-complete dropdown to.
android:dropDownHeight	This specifies the basic height of the dropdown.
android:dropDownHorizontalOffset	The amount of pixels by which the drop down should be offset horizontally.
android:dropDownSelector	This is the selector in a drop down list.
android:dropDownVerticalOffset	The amount of pixels by which the drop down should be offset vertically.

android:dropDownWidth	This specifies the basic width of the dropdown.
android:popupBackground	This sets the background.

Example

This example will take you through simple steps to show how to create your own Android application using Linear Layout and AutoCompleteTextView.

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *GUIDemo3* under a package *com.example.guidemo3* as explained in the *Hello World Example* chapter.
2	Modify *src/MainActivity.java* file to add a click event.
2	Modify the default content of *res/layout/activity_main.xml* file to include Android UI control.
3	Define required constants in *res/values/strings.xml* file
4	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.guidemo3/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```java
package com.example.guidemo3;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.widget.ArrayAdapter;
import android.widget.AutoCompleteTextView;

public class MainActivity extends Activity {

    AutoCompleteTextView autocompletetextview;

    String[] arr = { "MS SQL SERVER", "MySQL", "Oracle" };

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

        autocompletetextview = (AutoCompleteTextView)
        findViewById(R.id.autoCompleteTextView1);

        ArrayAdapter adapter = new ArrayAdapter
        (this,android.R.layout.select_dialog_item, arr);

        autocompletetextview.setThreshold(1);
        autocompletetextview.setAdapter(adapter);
    }

    @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:

```xml
<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/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="25dp"
        android:text="@string/example_autocompletetextview" /
        >

    <AutoCompleteTextView
        android:id="@+id/autoCompleteTextView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/textView2"
        android:layout_below="@+id/textView2"
        android:layout_marginTop="54dp"
        android:ems="10" />

</RelativeLayout>
```

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

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

    <string name="app_name">GUIDemo3</string>
    <string name="action_settings">Settings</string>
    <string name="example_autocompletetextview">Example
    showing AutoCompleteTextView</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.guidemo3"
    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.guidemo3.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 **GUIDemo3** 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:
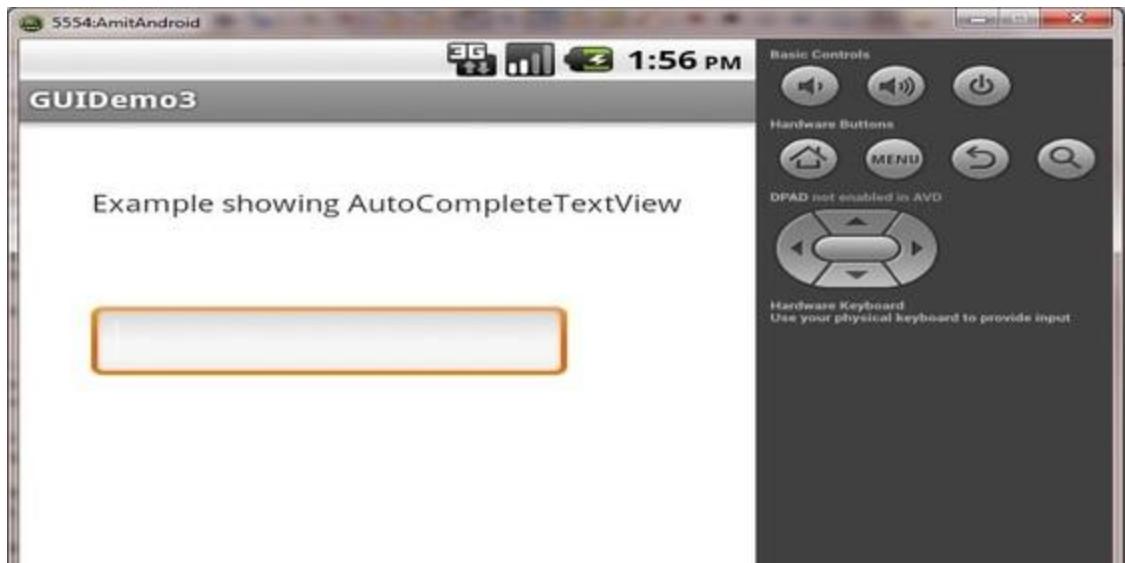
The following screen will appear after "m" will be typed in AutoCompleteTextView:

Exercise:

I will recommend to try above example with different attributes of AutoCompleteTextView in Layout XML file as well at programming time to have different look and feel of the AutoCompleteTextView. Try to make it editable, change to font color, font family, width, textSize etc and see the result. You can also try above example with multiple AutoCompleteTextView controls in one activity.

Button

A Button is a Push-button which can be pressed, or clicked, by the user to perform an action.

Button Attributes

Following are the important attributes related to Button control. You can check Android official documentation for complete list of attributes and related methods which you can use to change these attributes are run time.

Inherited from **android.widget.TextView** Class:

Attribute	Description
android:autoText	If set, specifies that this TextView has a textual input method and automatically corrects some common spelling errors.
android:drawableBottom	This is the drawable to be drawn below the text.
android:drawableRight	This is the drawable to be drawn to the right of the text.
android:editable	If set, specifies that this TextView has an input method.
android:text	This is the Text to display.

Inherited from **android.view.View** Class:

Attribute	Description
android:background	This is a drawable to use as the background.

android:contentDescription	This defines text that briefly describes content of the view.
android:id	This supplies an identifier name for this view,
android:onClick	This is the name of the method in this View's context to invoke when the view is clicked.
android:visibility	This controls the initial visibility of the view.

Example

This example will take you through simple steps to show how to create your own Android application using Linear Layout and Button.

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *GUIDemo4* under a package *com.example.guidemo4* as explained in the *Hello World Example* chapter.
2	Modify *src/MainActivity.java* file to add a click event.
2	Modify the default content of *res/layout/activity_main.xml* file to include Android UI control.
3	Define required constants in *res/values/strings.xml* file
4	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.guidemo4/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.guidemo4;

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

public class MainActivity extends Activity {

        private EditText edText1,edText2,edText3;
        private Button btnProduct;

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

        private void addListenerOnButton() {

          edText1 = (EditText)findViewById(R.id.edittext);
          edText2 = (EditText)findViewById(R.id.edittext2);
          edText3 = (EditText)findViewById(R.id.edittext3);

          btnProduct = (Button)findViewById(R.id.button1);
```

```
        btnProduct.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View view) {
                String t1 = edText1.getText().toString();
                String t2 = edText2.getText().toString();
                String t3 = edText3.getText().toString();

                int i1 = Integer.parseInt(t1);
                int i2 = Integer.parseInt(t2);
                int i3 = Integer.parseInt(t3);

                int product = i1*i2*i3;
                Toast.makeText(getApplicationContext(),
                String.valueOf(product),Toast.LENGTH_LONG).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:

```xml
<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="@string/example_button" />

    <EditText
        android:id="@+id/edittext"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/button1"
        android:layout_below="@+id/textView1"
        android:layout_marginTop="61dp"
        android:ems="10"
        android:inputType="text"
        android:text="@string/enter_text1" />

    <EditText
        android:id="@+id/edittext2"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
```

```
            android:layout_alignLeft="@+id/edittext3"
            android:layout_below="@+id/edittext"
            android:layout_marginTop="17dp"
            android:ems="10"
            android:inputType="text"
            android:text="@string/enter_text2" />

    <EditText
        android:id="@+id/edittext3"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/edittext"
        android:layout_below="@+id/edittext2"
        android:layout_marginTop="14dp"
        android:ems="10"
        android:inputType="text"
        android:text="@string/enter_text3" />

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/textView1"
        android:layout_below="@+id/edittext3"
        android:layout_marginTop="35dp"
        android:text="@string/click_button" />

</RelativeLayout>
```

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

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

    <string name="app_name">GUIDemo4</string>
    <string name="action_settings">Settings</string>
    <string name="example_button">Example showing
    Button</string>
    <string name="enter_text1"/>
    <string name="enter_text2"/>
    <string name="enter_text3"/>
    <string name="click_button">Calculate product of 3
    numbers</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.guidemo4"
    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.guidemo4.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 **GUIDemo4** 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:
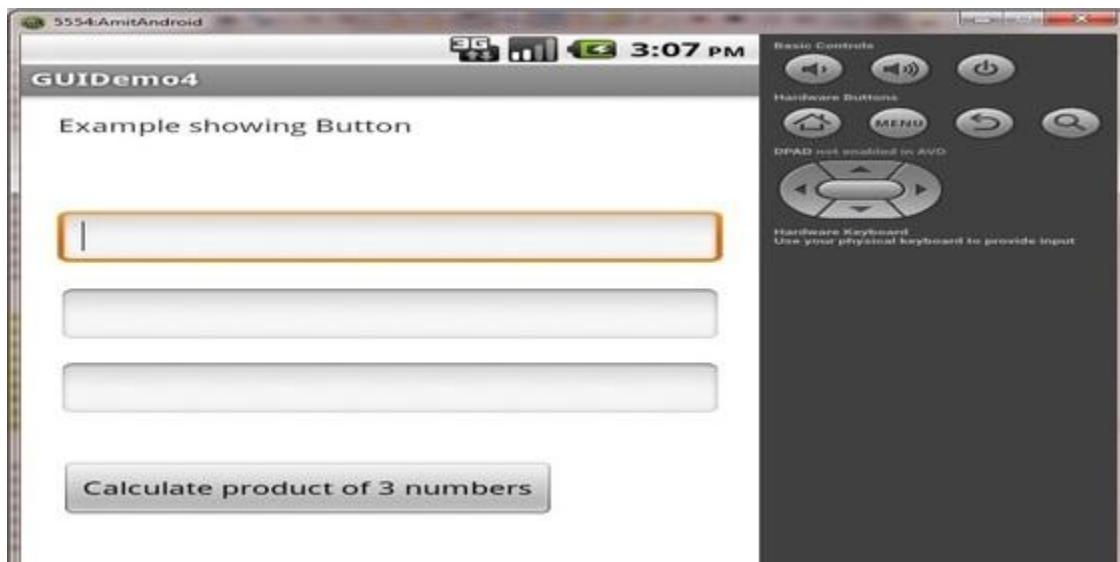
The following screen will appear after values are entered in 3 EditText and then the product is calculated by clicking on Button:

Exercise:

I will recommend to try above example with different attributes of Button in Layout XML file as well at programming time to have different look and feel of the Button. Try to make it editable, change to font color, font family, width, textSize etc and see the result. You can also try above example with multiple Button controls in one activity.

ImageButton

A ImageButton is a AbsoluteLayout which enables you to specify the exact location of its children. This shows a button with an image (instead of text) that can be pressed or clicked by the user.

ImageButton Attributes

Following are the important attributes related to ImageButton control. You can check Android official documentation for complete list of attributes and related methods which you can use to change these attributes are run time.

Inherited from **android.widget.ImageView** Class:

Attribute	Description
android:adjustViewBounds	Set this to true if you want the ImageView to adjust its bounds to preserve the aspect ratio of its drawable.
android:baseline	This is the offset of the baseline within this view.
android:baselineAlignBottom	If true, the image view will be baseline aligned with based on its bottom edge.
android:cropToPadding	If true, the image will be cropped to fit within its padding.
android:src	This sets a drawable as the content of this ImageView.

Inherited from **android.view.View** Class:

Attribute	Description

android:background	This is a drawable to use as the background.
android:contentDescription	This defines text that briefly describes content of the view.
android:id	This supplies an identifier name for this view,
android:onClick	This is the name of the method in this View's context to invoke when the view is clicked.
android:visibility	This controls the initial visibility of the view.

Example

This example will take you through simple steps to show how to create your own Android application using Linear Layout and ImageButton.

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *GUIDemo5* under a package *com.example.guidemo5* as explained in the *Hello World Example* chapter.
2	Modify *src/MainActivity.java* file to add a click event.
2	Modify the default content of *res/layout/activity_main.xml* file to include Android UI control.
3	Define required constants in *res/values/strings.xml* file
4	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.guidemo5/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.guidemo5;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.ImageButton;
import android.widget.Toast;

public class MainActivity extends Activity {

        ImageButton imgButton;

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

          addListenerOnButton();
        }

        private void addListenerOnButton() {

          imgButton = (ImageButton) findViewById
          (R.id.imageButton1);

          imgButton.setOnClickListener(new OnClickListener() {
            @Override
```

```
            public void onClick(View view) {
                Toast.makeText(MainActivity.this,"ImageButton
                Clicked : tutorialspoint.com",
                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:

```xml
<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="@string/example_imagebutton" />

    <ImageButton
        android:id="@+id/imageButton1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignRight="@+id/textView1"
        android:layout_below="@+id/textView1"
        android:layout_marginRight="35dp"
        android:layout_marginTop="32dp"
        android:contentDescription=
        "@string/android_launcher_image"
        android:src="@drawable/ic_launcher" />

</RelativeLayout>
```

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

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

    <string name="app_name">GUIDemo5</string>
    <string name="action_settings">Settings</string>
    <string name="example_imagebutton">Example showing ImageButton</string>
    <string name="android_launcher_image"></string>

</resources>
```

TUTORIALS POINT
Simply Easy Learning

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.guidemo5"
    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.guidemo5.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 **GUIDemo5** 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:

The following screen will appear after ImageButton is clicked:

Exercise:

I will recommend to try above example with different attributes of ImageButton in Layout XML file as well at programming time to have different look and feel of the ImageButton. Try to make it editable, change to font color, font family, width, textSize etc and see the result. You can also try above example with multiple ImageButton controls in one activity.

CheckBox

A CheckBox is an on/off switch that can be toggled by the user. You should use check-boxes when presenting users with a group of selectable options that are not mutually exclusive.

CheckBox Attributes

Following are the important attributes related to CheckBox control. You can check Android official documentation for complete list of attributes and related methods which you can use to change these attributes are run time.

Inherited from **android.widget.TextView** Class:

Attribute	Description
android:autoText	If set, specifies that this TextView has a textual input method and automatically corrects some common spelling errors.
android:drawableBottom	This is the drawable to be drawn below the text.
android:drawableRight	This is the drawable to be drawn to the right of the text.
android:editable	If set, specifies that this TextView has an input method.
android:text	This is the Text to display.

Inherited from **android.view.View** Class:

Attribute	Description

android:background	This is a drawable to use as the background.
android:contentDescription	This defines text that briefly describes content of the view.
android:id	This supplies an identifier name for this view,
android:onClick	This is the name of the method in this View's context to invoke when the view is clicked.
android:visibility	This controls the initial visibility of the view.

Example

This example will take you through simple steps to show how to create your own Android application using Linear Layout and CheckBox.

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *GUIDemo6* under a package *com.example.guidemo6* as explained in the *Hello World Example* chapter.
2	Modify *src/MainActivity.java* file to add a click event.
2	Modify the default content of *res/layout/activity_main.xml* file to include Android UI control.
3	Define required constants in *res/values/strings.xml* file
4	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.guidemo5/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.guidemo6;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.CheckBox;
import android.widget.Toast;

public class MainActivity extends Activity {

    private CheckBox chk1, chk2, chk3;

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

        // 3 methods
        addListenerOnCheck1();
        addListenerOnCheck2();
        addListenerOnCheck3();
    }

    // method for CheckBox1 - Java
    private void addListenerOnCheck1() {
        chk1 = (CheckBox) findViewById(R.id.checkBox1);
        chk2 = (CheckBox) findViewById(R.id.checkBox2);
```

```java
      chk3 = (CheckBox) findViewById(R.id.checkBox3);
      chk2.setOnClickListener(new OnClickListener() {

        @Override
        public void onClick(View v) {

          StringBuffer result = new StringBuffer();
          result.append("Java Selection : ").append
          (chk1.isChecked());
          result.append("\nPerl Selection : ").append
          (chk2.isChecked());
          result.append("\nPython Selection :").append
          (chk3.isChecked());

                  Toast.makeText(MainActivity.this, result.toString(),
           Toast.LENGTH_LONG).show();
        }
      });

    }

  // method for CheckBox2 - Perl
  private void addListenerOnCheck2() {

     chk1 = (CheckBox) findViewById(R.id.checkBox1);
     chk2 = (CheckBox) findViewById(R.id.checkBox2);
     chk3 = (CheckBox) findViewById(R.id.checkBox3);
     chk3.setOnClickListener(new OnClickListener() {

                @Override
                public void onClick(View v) {

            StringBuffer result = new StringBuffer();
            result.append("Java Selection : ").append
            (chk1.isChecked());
            result.append("\nPerl Selection : ").append
            (chk2.isChecked());
            result.append("\nPython Selection :").append
            (chk3.isChecked());

            Toast.makeText(MainActivity.this, result.toString
            (),Toast.LENGTH_LONG).show();
          }
       });
     }

      /* method for CheckBox3 - Python */
      private void addListenerOnCheck3() {
              // TODO Auto-generated method stub
          chk1 = (CheckBox) findViewById(R.id.checkBox1);
          chk2 = (CheckBox) findViewById(R.id.checkBox2);
          chk3 =  (CheckBox) findViewById(R.id.checkBox3);

              chk1.setOnClickListener(new OnClickListener() {

              @Override
              public void onClick(View v) {

                      StringBuffer result = new StringBuffer();
                       result.append("Java Selection :
").append(chk1.isChecked());
```

```
                                result.append("\nPerl Selection :
").append(chk2.isChecked());
                                result.append("\nPython Selection
:").append(chk3.isChecked());

                                Toast.makeText(MainActivity.this, result.toString(),
Toast.LENGTH_LONG).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:

```xml
<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="@string/example_checkbox" />

    <CheckBox
        android:id="@+id/checkBox1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/textView1"
        android:layout_below="@+id/textView1"
        android:layout_marginTop="88dp"
        android:text="@string/check_one" />

    <CheckBox
        android:id="@+id/checkBox2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/checkBox1"
        android:layout_below="@+id/checkBox1"
        android:layout_marginTop="22dp"
        android:text="@string/check_two" />

    <CheckBox
```

```
        android:id="@+id/checkBox3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@+id/checkBox2"
        android:layout_marginTop="24dp"
        android:text="@string/check_three" />

    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/checkBox1"
        android:layout_below="@+id/textView1"
        android:layout_marginTop="39dp"
        android:text="@string/example_question" />

</RelativeLayout>
```

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

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

    <string name="app_name">GUIDemo6</string>
    <string name="action_settings">Settings</string>
    <string name="example_checkbox">Example showing CheckBox Control</string>
    <string name="check_one">JAVA</string>
    <string name="check_two">PERL</string>
    <string name="check_three">PYTHON</string>
    <string name="example_question">Worked on following Languages-</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.guidemo6"
    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.guidemo5.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 **GUIDemo6** 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:
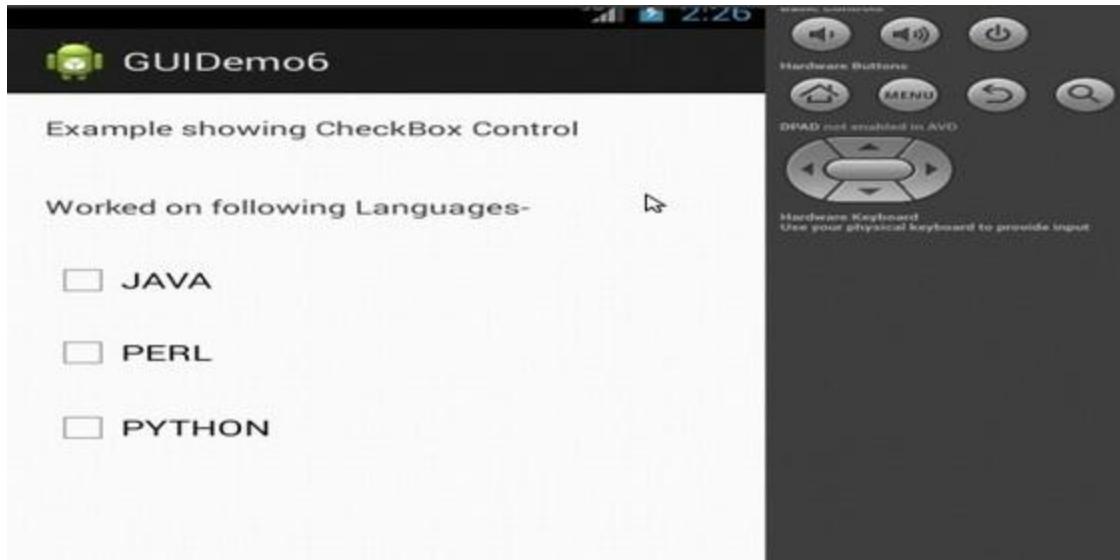
The following screen will appear after CheckBox1 i.e Java is clicked:

The following screen will appear after CheckBox3 i.e Python is clicked, now it will appear for both Java(previous selection) and Perl:
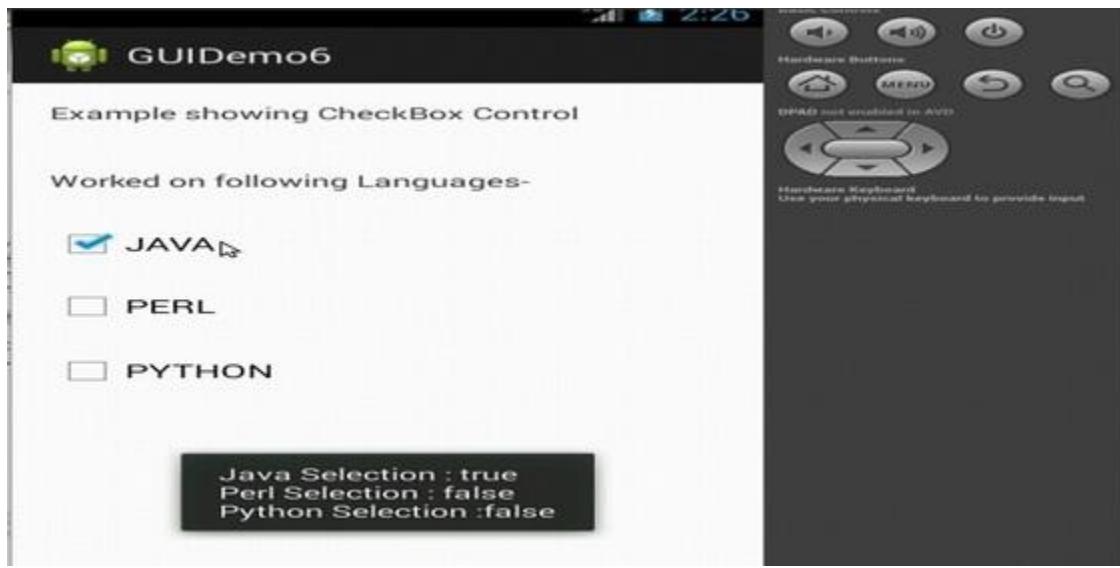
Exercise:

I will recommend to try above example with different attributes of CheckBox in Layout XML file as well at programming time to have different look and feel of the CheckBox. Try to make it editable, change to font color, font family, width, textSize etc and see the result. You can also try above example with multiple CheckBox controls in one activity.

ToggleButton

A ToggleButton displays checked/unchecked states as a button. It is basically an on/off button with a light indicator.

ToggleButton Attributes

Following are the important attributes related to ToggleButton control. You can check Android official documentation for complete list of attributes and related methods which you can use to change these attributes are run time.

Attribute	Description
android:disabledAlpha	This is the alpha to apply to the indicator when disabled.
android:textOff	This is the text for the button when it is not checked.
android:textOn	This is the text for the button when it is checked.

Inherited from **android.widget.TextView** Class:

Attribute	Description
android:autoText	If set, specifies that this TextView has a textual input method and automatically corrects some common spelling errors.
android:drawableBottom	This is the drawable to be drawn below the text.
android:drawableRight	This is the drawable to be drawn to the right of the text.

android:editable	If set, specifies that this TextView has an input method.
android:text	This is the Text to display.

Inherited from **android.view.View** Class:

Attribute	Description
android:background	This is a drawable to use as the background.
android:contentDescription	This defines text that briefly describes content of the view.
android:id	This supplies an identifier name for this view,
android:onClick	This is the name of the method in this View's context to invoke when the view is clicked.
android:visibility	This controls the initial visibility of the view.

Example

This example will take you through simple steps to show how to create your own Android application using Linear Layout and ToggleButton.

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *GUIDemo7* under a package *com.example.guidemo7* as explained in the *Hello World Example* chapter.
2	Modify *src/MainActivity.java* file to add a click event.
2	Modify the default content of *res/layout/activity_main.xml* file to include Android UI control.
3	Define required constants in *res/values/strings.xml* file
4	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.guidemo7/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.guidemo7;

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.Toast;
import android.widget.ToggleButton;

public class MainActivity extends Activity {

    private ToggleButton toggleBtn1, toggleBtn2;
    private Button btResult;

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

```
      addListenerOnToggleButton();
   }

   private void addListenerOnToggleButton() {

      toggleBtn1 = (ToggleButton) findViewById
      (R.id.toggleButton1);
      toggleBtn2 = (ToggleButton) findViewById
      (R.id.toggleButton2);
      btResult = (Button) findViewById(R.id.button1);
      btResult.setOnClickListener(new OnClickListener() {

         @Override
         public void onClick(View v) {
            StringBuffer result = new StringBuffer();
            result.append("START Condition - ").append
            (toggleBtn1.getText());
            result.append("\nSTOP Condition - ").append
            (toggleBtn2.getText());
            Toast.makeText(MainActivity.this, result.toString(),
            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:

```xml
<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="@string/example_togglebutton" />

    <ToggleButton
        android:id="@+id/toggleButton1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/textView1"
        android:layout_below="@+id/textView1"
        android:layout_marginTop="24dp" />

    <Button
```

```
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/toggleButton1"
        android:layout_marginLeft="19dp"
        android:layout_marginTop="30dp"
        android:layout_toRightOf="@+id/toggleButton1"
        android:text="@string/example_result" />

    <ToggleButton
        android:id="@+id/toggleButton2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignTop="@+id/toggleButton1"
        android:layout_toRightOf="@+id/textView1"
        android:textOff="@string/stop_togglebutton"
        android:textOn="@string/start_togglebutton"
        android:checked="true"/>

</RelativeLayout>
```

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

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

    <string name="app_name">GUIDemo7</string>
    <string name="action_settings">Settings</string>
    <string name="example_togglebutton">Example showing
    ToggleButton</string>
    <string name="start_togglebutton">START</string>
    <string name="stop_togglebutton">STOP</string>
    <string name="example_result">Click Me</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.guidemo7"
    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.guidemo7.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 **GUIDemo7** 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:

The following screen will appear:

The following screen will appear, conditions are shown when state of both the toggle buttons are changed:

The following screen will appear, conditions are shown when state of 2nd toggle button is changed to START:

Exercise:

I will recommend to try above example with different attributes of ToggleButton in Layout XML file as well at programming time to have different look and feel of the ToggleButton. Try to make it editable, change to font color, font family, width, textSize etc and see the result. You can also try above example with multiple ToggleButton controls in one activity.

RadioButton

A RadioButton has two states: either checked or unchecked.This allows the user to select one option from a set.

RadioButton Attributes

Following are the important attributes related to RadioButton control. You can check Android official documentation for complete list of attributes and related methods which you can use to change these attributes are run time.

Inherited from **android.widget.TextView** Class:

Attribute	Description
android:autoText	If set, specifies that this TextView has a textual input method and automatically corrects some common spelling errors.
android:drawableBottom	This is the drawable to be drawn below the text.
android:drawableRight	This is the drawable to be drawn to the right of the text.
android:editable	If set, specifies that this TextView has an input method.
android:text	This is the Text to display.

Inherited from **android.view.View** Class:

Attribute	Description
android:background	This is a drawable to use as the background.

android:contentDescription	This defines text that briefly describes content of the view.
android:id	This supplies an identifier name for this view,
android:onClick	This is the name of the method in this View's context to invoke when the view is clicked.
android:visibility	This controls the initial visibility of the view.

Example

This example will take you through simple steps to show how to create your own Android application using Linear Layout and RadioButton.

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *GUIDemo8* under a package *com.example.guidemo8* as explained in the *Hello World Example* chapter.
2	Modify *src/MainActivity.java* file to add a click event.
2	Modify the default content of *res/layout/activity_main.xml* file to include Android UI control.
3	Define required constants in *res/values/strings.xml* file
4	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.guidemo8/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.guidemo8;

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.RadioButton;
import android.widget.RadioGroup;
import android.widget.Toast;

public class MainActivity extends Activity {

    private RadioGroup radioGroupWebsite;
    private RadioButton radioBtn1;
    private Button btnWebsiteName;

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

      addListenerRadioButton();
    }


    private void addListenerRadioButton() {

        radioGroupWebsite = (RadioGroup) findViewById
```

```
        (R.id.radioGroup1);
        btnWebsiteName = (Button) findViewById(R.id.button1);

        btnWebsiteName.setOnClickListener(new OnClickListener() {

          @Override
          public void onClick(View v) {

            // get selected radio button from radioGroupWebsite
            int selected =
            radioGroupWebsite.getCheckedRadioButtonId();
            radioBtn1 = (RadioButton) findViewById(selected);
            Toast.makeText(MainActivity.this,
            radioBtn1.getText(), 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:

```xml
<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="@string/example_radiobutton" />

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/textView1"
        android:layout_centerVertical="true"
        android:text="@string/website_name" />

    <RadioGroup
        android:id="@+id/radioGroup1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/textView1"
        android:layout_below="@+id/textView1"
        android:layout_marginTop="30dp" >
```

```
        <RadioButton
            android:id="@+id/radio0"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:checked="true"
            android:text="@string/website_radio0" />

        <RadioButton
            android:id="@+id/radio1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/website_radio1" />

        <RadioButton
            android:id="@+id/radio2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/website_radio2" />
    </RadioGroup>

</RelativeLayout>
```

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

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

    <string name="app_name">GUIDemo8</string>
    <string name="action_settings">Settings</string>
    <string name="example_radiobutton">Example showing RadioButton</string>
    <string name="website_name">Website URL</string>
    <string
    name="website_radio0">www.tutorialspoint.com</string>
    <string name="website_radio1">www.compileonline.com</string>
    <string name="website_radio2">www.photofuntoos.com</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.guidemo8"
    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.guidemo8.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 **GUIDemo8** 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:

It checks the 2nd radiobutton and text "www.compileonline.com" is shown when Button is clicked:

The following screen will appear when 3rd radiobutton is checked and text "www.photofuntoos.com" is shown when Button is clicked::

Exercise:

I will recommend to try above example with different attributes of RadioButton in Layout XML file as well at programming time to have different look and feel of the RadioButton. Try to make it editable, change to font color, font family, width, textSize etc and see the result. You can also try above example with multiple RadioButton controls in one activity.

RadioGroup

A RadioGroup class is used for set of radio buttons. If we check one radio button that belongs to a radio group, it automatically unchecks any previously checked radio button within the same group.

RadioGroup Attributes

Following are the important attributes related to RadioGroup control. You can check Android official documentation for complete list of attributes and related methods which you can use to change these attributes are run time.

Attribute	Description
android:checkedButton	This is the id of child radio button that should be checked by default within this radio group.

Inherited from **android.view.View** Class:

Attribute	Description
android:background	This is a drawable to use as the background.
android:contentDescription	This defines text that briefly describes content of the view.
android:id	This supplies an identifier name for this view,
android:onClick	This is the name of the method in this View's context to invoke when the view is clicked.
android:visibility	This controls the initial visibility of the view.

Example

This example will take you through simple steps to show how to create your own Android application using Linear Layout and RadioGroup.

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *GUIDemo9* under a package *com.example.guidemo9* as explained in the *Hello World Example* chapter.
2	Modify *src/MainActivity.java* file to add a click event.
2	Modify the default content of *res/layout/activity_main.xml* file to include Android UI control.
3	Define required constants in *res/values/strings.xml* file
4	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.guidemo9/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.guidemo9;

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.RadioButton;
import android.widget.RadioGroup;
import android.widget.Toast;

public class MainActivity extends Activity {

  private RadioGroup radioGroupCricket;
  private RadioGroup radioGroupTutorials;
  private RadioButton radioBtn1;
  private RadioButton radioBtn2;
  private Button btnCricketer;
  private Button btnTutorialsPoint;

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

    // group1
    addListenerRadioGroup1();
    // group2
    addListenerRadioGroup2();
  }

  private void addListenerRadioGroup2() {
    radioGroupTutorials = (RadioGroup) findViewById
    (R.id.radioGroup2);
    btnTutorialsPoint = (Button) findViewById(R.id.button2);
    btnTutorialsPoint.setOnClickListener(new OnClickListener() {

      @Override
      public void onClick(View v) {
        // get selected radio button from radioGroupTutorials
        int selected =
        radioGroupTutorials.getCheckedRadioButtonId();
        radioBtn2 = (RadioButton) findViewById(selected);
        Toast.makeText(MainActivity.this,
        radioBtn2.getText(), Toast.LENGTH_SHORT).show();
  }
    });
  }


  private void addListenerRadioGroup1() {

    radioGroupCricket = (RadioGroup) findViewById
    (R.id.radioGroup1);
    btnCricketer = (Button) findViewById(R.id.button1);
    btnCricketer.setOnClickListener(new OnClickListener() {

      @Override
      public void onClick(View v) {
        // get selected radio button from radioGroupCricket
```

```
        int selected = radioGroupCricket.getCheckedRadioButtonId();
        radioBtn1 = (RadioButton) findViewById(selected);
        Toast.makeText(MainActivity.this,
        radioBtn1.getText(), 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:

```
<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="@string/example_radiogroup" />

    <RadioGroup
        android:id="@+id/radioGroup1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@+id/textView1"
        android:layout_marginTop="24dp" >

        <RadioButton
            android:id="@+id/radioGroupButton0"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:checked="true"
            android:text="@string/example_radiogroup1button0" />

        <RadioButton
            android:id="@+id/radioGroupButton1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/example_radiogroup1button1" />

        <RadioButton
            android:id="@+id/radioGroupButton3"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/example_radiogroup1button2" />
    </RadioGroup>
```

```
    <RadioGroup
        android:id="@+id/radioGroup2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_alignTop="@+id/radioGroup1"
        android:layout_marginRight="15dp" >

        <RadioButton
            android:id="@+id/radioGroup2Button0"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:checked="true"
            android:text="@string/example_radiogroup2button0" />

        <RadioButton
            android:id="@+id/radioGroup2Button1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/example_radiogroup2button1" />

        <RadioButton
            android:id="@+id/radioGroup2Button3"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/example_radiogroup2button2" />
    </RadioGroup>

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/radioGroup1"
        android:layout_marginTop="25dp"
        android:text="@string/group_button1" />

    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBaseline="@+id/button1"
        android:layout_alignBottom="@+id/button1"
        android:layout_alignParentRight="true"
        android:layout_marginRight="14dp"
        android:text="@string/group_button2" />

</RelativeLayout>
```

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

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

    <string name="app_name">GUIDemo9</string>
    <string name="action_settings">Settings</string>
    <string name="example_radiogroup">Example showing RadioGroup</string>
    <string name="example_radiogroup1button0">Sachin</string>
    <string name="example_radiogroup1button1">Saurav</string>
    <string name="example_radiogroup1button2">Rahul</string>
    <string name="example_radiogroup2button0">MySQL Tutorial</string>
    <string name="example_radiogroup2button1">SQL Tutorial</string>
```

TUTORIALS POINT
Simply Easy Learning

```
    <string name="example_radiogroup2button2">SQLite Tutorial</string>
    <string name="group_button1">cricketer</string>
    <string name="group_button2">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.guidemo9"
    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.guidemo9.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 **GUIDemo9** 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:

The following screen will appear, here we have 2 RadioGroups i.e. radioGroupCricket and radioGroupTutorials. The cricketer Button is clicked after checking "Rahul" Radio Button:
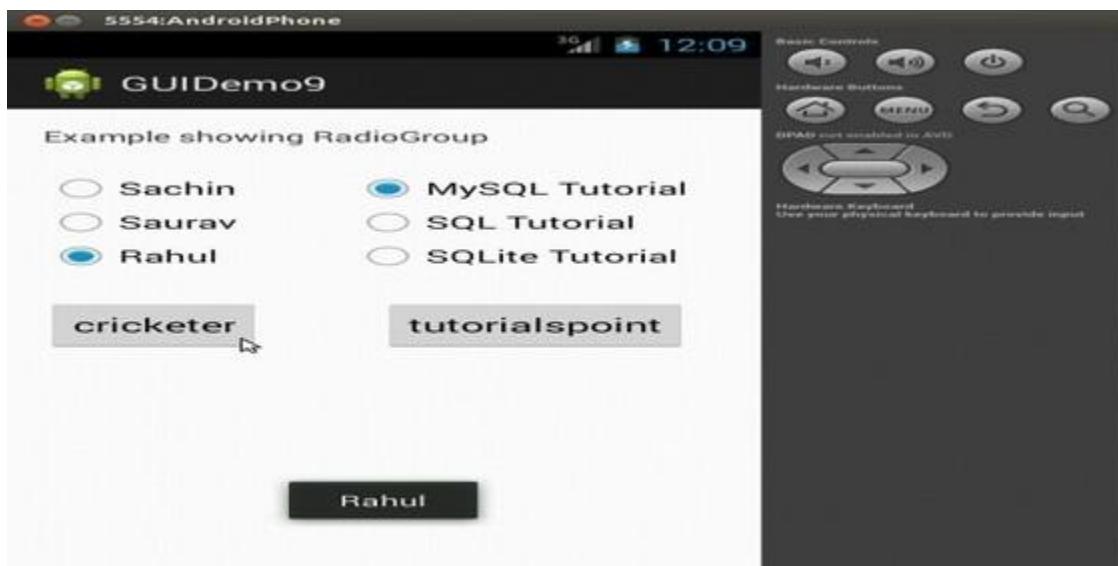
The following screen will appear if we check one radio button that belongs to radioGroupTutorials radio group, it automatically unchecks any previously checked radio button within the same radioGroupTutorials group. The tutorialspoint Button is clicked after checking "SQL Tutorial" Radio Button:

Exercise:

I will recommend to try above example with different attributes of RadioButton in Layout XML file as well at programming time to have different look and feel of the RadioButton. Try to make it editable, change to font color, font family, width, textSize etc and see the result. You can also try above example with multiple RadioButton controls in one activity.

Create UI Controls

As explained in previous chapter, a view object may have a unique ID assigned to it which will identify the View uniquely within the tree. The syntax for an ID, inside an XML tag is:

```
android:id="@+id/text_id"
```

To create a UI Control/View/Widget you will have to define a view/widget in the layout file and assign it a unique ID as follows:

```xml
<?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_id"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="I am a TextView" />
</LinearLayout>
```

Then finally create an instance of the Control object and capture it from the layout, use the following:

```java
TextView myText = (TextView) findViewById(R.id.text_id);
```

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 default 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;
```

```java
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
<?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;
```

TUTORIALS POINT
Simply Easy Learning

```
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.

Styles and Themes

I f you already know about Cascading Style Sheet (CSS) in web design then to understand Android Style also

works very similar way. There are number of attributes associated with each Android widget which you can set to change your application look and feel. A style can specify properties such as height, padding, font color, font size, background color, and much more.

You can specify these attributes in Layout file as follows:

```xml
<?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_id"

   android:layout_width="wrap_content"

   android:layout_height="wrap_content"

   android:capitalize="characters"

   android:textColor="#00FF00"

   android:typeface="monospace"

   android:text="@string/hello_world" />


</LinearLayout>
```

But this way we need to define style attributes for every attribute separately which is not good for source code maintenance point of view. So we work with styles by defining them in separate file as explained below.

Defining Styles

A style is defined in an XML resource that is separate from the XML that specifies the layout. This XML file resides under **res/values/** directory of your project and will have **<resources>** as the root node which is mandatory for the style file. The name of the XML file is arbitrary, but it must use the .xml extension.

You can define multiple styles per file using **<style>** tag but each style will have its name that uniquely identifies the style. Android style attributes are set using **<item>** tag as shown below:

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

<resources>

    <style name="CustomFontStyle">

        <item name="android:layout_width">fill_parent</item>

        <item name="android:layout_height">wrap_content</item>

        <item name="android:capitalize">characters</item>

        <item name="android:typeface">monospace</item>

        <item name="android:textSize">12pt</item>

        <item name="android:textColor">#00FF00</item>/>

    </style>

</resources>
```

The value for the <item> can be a keyword string, a hex color, a reference to another resource type, or other value depending on the style property.

Using Styles

Once your style is defined, you can use it in your XML Layout file using **style** attribute as follows:

```
<?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_id"

    style="@style/CustomFontStyle"

    android:text="@string/hello_world" />


</LinearLayout>
```

To understand the concept related to Android Style, you can check Style Demo Example given below:

Following example demonstrates how you can use a Style for individual elements. Let's start with creating a simple Android application as per the following steps:

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *StyleDemo* under a package *com.example.styledemo* 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	Define your style in global style file **res/values/style.xml** to define custom attributes for a button.
4	Modify the default content of *res/layout/activity_main.xml* file to include a set of Android UI controls and make use of the defined style.
5	Define required constants in *res/values/strings.xml* file
6	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.styledemo/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.styledemo;

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(20);
           }
        });

        // -- 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/values/style.xml** file which will have addition style**CustomButtonStyle** defined:

```
<resources>

    <!--
      Base application theme, dependent on API level. This theme is replaced
      by AppBaseTheme from res/values-vXX/styles.xml on newer devices.
    -->
    <style name="AppBaseTheme" parent="android:Theme.Light">
      <!--
          Theme customizations available in newer API levels can go in
          res/values-vXX/styles.xml, while customizations related to
          backward-compatibility can go here.
      -->
    </style>

    <!-- Application theme. -->
    <style name="AppTheme" parent="AppBaseTheme">
      <!-- All customizations that are NOT specific to a particular API-level can go
here. -->
    </style>

    <!-- Custom Style defined for the buttons. -->
    <style name="CustomButtonStyle">
        <item name="android:layout_width">100dp</item>
        <item name="android:layout_height">38dp</item>
        <item name="android:capitalize">characters</item>
        <item name="android:typeface">monospace</item>
        <item name="android:shadowDx">1.2</item>
        <item name="android:shadowDy">1.2</item>
        <item name="android:shadowRadius">2</item>
        <item name="android:textColor">#494948</item>/>
        <item name="android:gravity" >center</item>
        <item name="android:layout_margin" >3dp</item>
        <item name="android:textSize" >5pt</item>
        <item name="android:shadowColor" >#000000</item>
    </style>

</resources>
```

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" >
```

```xml
    <Button
    android:id="@+id/button_s"
    style="@style/CustomButtonStyle"
    android:text="@string/button_small"
    android:onClick="doSmall"/>

    <Button
    android:id="@+id/button_l"
    style="@style/CustomButtonStyle"
    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>
```

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

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

    <string name="app_name">StyleDemo</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
<?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 **StyleDemo** 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:

Style Inheritance

Android supports style Inheritance in very much similar way as cascading style sheet in web design. You can use this to inherit properties from an existing style and then define only the properties that you want to change or add.

Its simple, to create a new style **LargeFont** that inherits the **CustomFontStyle** style defined above, but make the font size big, you can author the new style like this:

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

<resources>

    <style name="CustomFontStyle.LargeFont">

        <item name="android:textSize">20ps</item>

    </style>

</resources>
```

You can reference this new style as **@style/CustomFontStyle.LargeFont** in your XML Layout file. You can continue inheriting like this as many times as you'd like, by chaining names with periods. For example, you can extend FontStyle.LargeFont to be Red, with:

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

<resources>

    <style name="CustomFontStyle.LargeFont.Red">

        <item name="android:textColor">#FF0000</item>/>
```

```
        </style>

</resources>
```

This technique for inheritance by chaining together names only works for styles defined by your own resources. You can't inherit Android built-in styles this way. To reference an Android built-in style, such as **TextAppearance**, you must use the **parent** attribute as shown below:

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

<resources>

    <style name="CustomFontStyle" parent="@android:style/TextAppearance">

        <item name="android:layout_width">fill_parent</item>

        <item name="android:layout_height">wrap_content</item>

        <item name="android:capitalize">characters</item>

        <item name="android:typeface">monospace</item>

        <item name="android:textSize">12pt</item>

        <item name="android:textColor">#00FF00</item>/>

    </style>

</resources>
```

Android Themes

Hope you understood the concept of Style, so now let's try to understand what is a **Theme**. A theme is nothing but an Android style applied to an entire Activity or application, rather than an individual View.

Thus, when a style is applied as a theme, every **View** in the Activity or application will apply each style property that it supports. For example, you can apply the same **CustomFontStyle** style as a theme for an Activity and then all text inside that **Activity** will have green monospace font.

To set a theme for all the activities of your application, open the **AndroidManifest.xml** file and edit the**<application>** tag to include the **android:theme** attribute with the style name. For example:

```
<application android:theme="@style/CustomFontStyle">
```

But if you want a theme applied to just one Activity in your application, then add the android:theme attribute to the <activity> tag only. For example:

```
<activity android:theme="@style/CustomFontStyle">
```

There are number of default themes defined by Android which you can use directly or inherit them using **parent** attribute as follows:

```
<style name="CustomTheme" parent="android:Theme.Light">

    ...

</style>
```

To understand the concept related to Android Theme, you can check Theme Demo Example given below:

Following example demonstrates how you can use a theme for an application. For demo purpose we will modify our default **AppTheme** wehere default text, its size, family, shadow etc will be changed. Let's start with creating a simple Android application as per the following steps:

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *ThemeDemo* under a package *com.example.themedemo* 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	Define your style in global style file **res/values/style.xml** to define custom attributes for a button and change default theme of the application to play with the text.
4	Modify the detault content of *res/layout/activity_main.xml* file to include a set of Android UI controls and make use of the defined style.
5	Define required constants in *res/values/strings.xml* file
6	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.themedemo/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.themedemo;

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(20);
           }
        });

        // -- 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/values/style.xml** file which will have addition style**CustomButtonStyle** defined:

```
<resources>

    <!--
      Base application theme, dependent on API level. This theme is replaced
      by AppBaseTheme from res/values-vXX/styles.xml on newer devices.
    -->
    <style name="AppBaseTheme" parent="android:Theme.Light">
      <!--
          Theme customizations available in newer API levels can go in
          res/values-vXX/styles.xml, while customizations related to
          backward-compatibility can go here.
      -->
    </style>

    <!-- Application theme. -->
    <style name="AppTheme" parent="AppBaseTheme">
      <!-- All customizations that are NOT specific to a particular API-level can go
here. -->
        <item name="android:capitalize">characters</item>
        <item name="android:typeface">monospace</item>
        <item name="android:shadowDx">1.2</item>
        <item name="android:shadowDy">1.2</item>
        <item name="android:shadowRadius">2</item>
        <item name="android:textColor">#494948</item>/>
        <item name="android:gravity" >center</item>
        <item name="android:layout_margin" >3dp</item>
        <item name="android:textSize" >5pt</item>
        <item name="android:shadowColor" >#000000</item>
    </style>

    <!-- Custom Style defined for the buttons. -->
    <style name="CustomButtonStyle">
        <item name="android:layout_width">100dp</item>
        <item name="android:layout_height">38dp</item>
    </style>

</resources>
```

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"
   style="@style/CustomButtonStyle"
   android:text="@string/button_small"
   android:onClick="doSmall"/>

   <Button
   android:id="@+id/button_l"
   style="@style/CustomButtonStyle"
   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>
```

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">ThemeDemo</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**. Here we do not need to change anything because we kept out theme name unchanged. But if you define fresh new theme or inherit a default them with different name then you will have to replace **AppTheme** name with the new them name.

```
<?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 **ThemeDemo** 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:

Default Styles & Themes

The Android platform provides a large collection of styles and themes that you can use in your applications. You can find a reference of all available styles in the **R.style** class. To use the styles listed here, replace all underscores in the style name with a period. For example, you can apply the Theme_NoTitleBar theme with "@android:style/Theme.NoTitleBar". You can see the following source code for Android styles and themes:

- Android Styles (styles.xml)
- Android Themes (themes.xml)
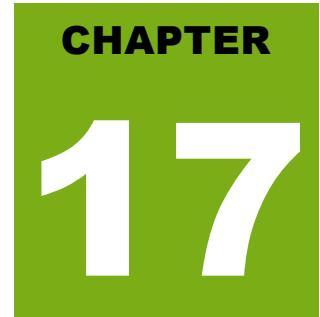
Custom Components

A ndroid offers a great list of pre-built widgets like Button, TextView, EditText, ListView, CheckBox,

RadioButton, Gallery, Spinner, AutoCompleteTextView etc. which you can use directly in your Android application development, but there may be a situation when you are not satisfied with existing functionality of any of the available widgets. Android provides you with means of creating your own custom components which you can customized to suit your needs.

If you only need to make small adjustments to an existing widget or layout, you can simply subclass the widget or layout and override its methods which will give you precise control over the appearance and function of a screen element.

This tutorial explains you how to create custom Views and use them in your application using simple and easy steps.

Creating a Simple Custom Component

The simplest way to create your custom component is to extend an existing widget class or subclass with your own class if you want to extend the functionality of existing widget like Button, TextView, EditText, ListView, CheckBox etc. otherwise you can do everything yourself by starting with the *android.view.View* class.

At its simplest form you will have to write your constructors corresponding to all the constructors of the base class. For example if you are going to extend **TextView** to create a **DateView** then following three constructors will be created for DateView class:

```
public class DateView extends TextView {

   public DateView(Context context) {

      super(context);

      //--- Additional custom code --

   }


   public DateView(Context context, AttributeSet attrs) {

      super(context, attrs);

      //--- Additional custom code --
```

```
    }


    public DateView(Context context, AttributeSet attrs, int defStyle) {

        super(context, attrs, defStyle);

        //--- Additional custom code --

    }

}
```

Because you have created DateView as child of TextView so it will have access on all the attributes, methods and events related to TextView and you will be able to use them without any further implementation. You will implement additional custom functionality inside your own code as explained in the given examples below.

If you have requirement for implementing custom drawing/sizing for your custom widgets then you need to override **onMeasure(int widthMeasureSpec, int heightMeasureSpec)** and **onDraw(Canvas canvas)** methods. If you are not going to resize or change the shape of your built-in component then you do not need either of these methods in your custom component.
The *onMeasure()* method coordinate with the layout manager to report the widget's width and height, and you need to call *setMeasuredDimension(int width, int height)* from inside this method to report the dimensions.
You can then execute your custom drawing inside the *onDraw(Canvas canvas)* method, where android.graphis.Canvas is pretty similar to its counterpart in Swing, and has methods such as drawRect(), drawLine(), drawString(), drawBitmap() etc. which you can use to draw your component.

Once you are done with the implementation of a custom component by extending existing widget, you will be able to instantiate these custom components in two ways in your application development:

INSTANTIATE USING CODE INSIDE ACTIVITY CLASS

It is very similar way of instantiating custom component the way you instantiate built-in widget in your activity class. For example you can use following code to instantiate above defined custom component:

```
@Override

 protected void onCreate(Bundle savedInstanceState) {

     super.onCreate(savedInstanceState);

     setContentView(R.layout.activity_main);


     DateView dateView = new DateView(this);

     setContentView(dateView);

 }
```

Check this below given example to understand how to Instantiate a Basic Android Custom Component using code inside an activity:

Instantiate a Basic Android Custom Component(using code)

Following example shows you how to define a simple Android custom component and then how to instantiate it inside activity code without using layout file.

Step	Description

1	You will use Eclipse IDE to create an Android application and name it as *DateViewDemo*under a package *com.example.dateviewdemo* as explained in the *Hello World Example*chapter.
2	Create *src/DateView.java* file and add the code to define your custom component. It will extend TextView and will have additional functionality to show current date.
3	Modify *src/MainActivity.java* file and add the code to create DateView instance and use*setContentView()* method to set it in the layout.
4	Run the application to launch Android emulator and verify the result of the changes done in the aplication.

Following will be the content of new file **src/com.example.dateviewdemo/DateView.java**, which will have additional functionality to show current date:

```java
package com.example.dateviewdemo;

import java.text.SimpleDateFormat;
import java.util.Calendar;

import android.content.Context;
import android.util.AttributeSet;
import android.widget.TextView;

public class DateView extends TextView {
    public DateView(Context context) {
        super(context);
        setDate();
    }

    public DateView(Context context, AttributeSet attrs) {
        super(context, attrs);
        setDate();
    }

    public DateView(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
        setDate();
    }

    private void setDate() {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd");
        String today = dateFormat.format(Calendar.getInstance().getTime());
        setText(today);  // self = DateView is a subclass of TextView
    }

}
```

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

```java
package com.example.dateviewdemo;

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

public class MainActivity extends Activity {

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

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

        //-- Create DateView instance and set it in layout.
        DateView dateView = new DateView(this);
        setContentView(dateView);
    }

    @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:

```
<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:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello_world" />

</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">DateViewDemo</string>
    <string name="action_settings">Settings</string>
    <string name="hello_world">Hello world!</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 **DateViewDemo** 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:

INSTANTIATE USING LAYOUT XML FILE

Traditionally you use Layout XML file to instantiate your built-in widgets, same concept will apply on your custom widgets as well so you will be able to instantiate your custom component using Layout XML file as explained below. Here **com.example.dateviewdemo** is the package where you have put all the code related to **DateView** class and **DateView** is Java class name where you have put complete logic of your custom component.

```
<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" >


    <com.example.dateviewdemo.DateView

     android:layout_width="match_parent"

     android:layout_height="wrap_content"

     android:textColor="#fff"

     android:textSize="40sp"

     android:background="#000"

     />

</RelativeLayout>
```

It is important to note here that we are using all TextView attributes along with custom component without any change. Similar way you will be able to use all the events, and methods along with DateView component.

Check this example to understand how to [Instantiate a Basic Android Custom Component](#) using Layout XML file.

Instantiate a Basic Android Custom Component (using Layout XML file)

Following example shows you how to define a simple Android custom component and then how to instantiate it inside activity code without using layout file.

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *DateViewDemo*under a package *com.example.dateviewdemo* as explained in the *Hello World Example*chapter.
2	Create *src/DateView.java* file and add the code to define your custom component. It will extend TextView and will have additional functionality to show current date.
3	Modify *res/layout/activity_main.xml* file and add the code to create DateView instance along with few default attributes.
4	Run the application to launch Android emulator and verify the result of the changes done in the aplication.

Following will be the content of new file **src/com.example.dateviewdemo/DateView.java**, which will have additional functionality to show current date:

```
package com.example.dateviewdemo;


import java.text.SimpleDateFormat;

import java.util.Calendar;
```

```
import android.content.Context;

import android.util.AttributeSet;

import android.widget.TextView;


public class DateView extends TextView {

   public DateView(Context context) {

      super(context);

      setDate();

   }


   public DateView(Context context, AttributeSet attrs) {

      super(context, attrs);

      setDate();

   }


   public DateView(Context context, AttributeSet attrs, int defStyle) {

      super(context, attrs, defStyle);

      setDate();

   }


   private void setDate() {

      SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd");

      String today = dateFormat.format(Calendar.getInstance().getTime());

      setText(today);  // self = DateView is a subclass of TextView

   }


}
```

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

```
package com.example.dateviewdemo;


import android.os.Bundle;

import android.app.Activity;
```

```
import android.view.Menu;


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) {

        // 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:

```
<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" >


    <com.example.dateviewdemo.DateView

     android:layout_width="match_parent"

     android:layout_height="wrap_content"

     android:textColor="#fff"
```

```
        android:textSize="40sp"

        android:background="#000"

        />



</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">DateViewDemo</string>

    <string name="action_settings">Settings</string>

    <string name="hello_world">Hello world!</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 **DateViewDemo** 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:

Custom Component with Custom Attributes

We have seen how we can extend functionality of built-in widgets but in both the examples given above we saw that extended component can make use of all the default attributes of its parent class. But consider a situation when you want to create your own attribute from scratch. Below is a simple procedure to create and use new attributes for Android Custom components. Consider we want to introduce three attributes and will use them as shown below:

```
<com.example.dateviewdemo.DateView

    android:layout_width="match_parent"

    android:layout_height="wrap_content"

    android:textColor="#fff"

    android:textSize="40sp"
```

```
   custom:delimiter="-"

   custom:fancyText="true"

/>
```

STEP 1

The first step to enable us to use our custom attributes is to define them in a new xml file under *res/values/* and call it **attrs.xml**. Let's have a look on an example attrs.xml file:

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

<resources>

   <declare-styleable name="DateView">

   <attr name="delimiter" format="string"/>

   <attr name="fancyText" format="boolean"/>

   </declare-styleable>

</resources>
```

Here the **name=value** is what we want to use in our Layout XML file as attribute, and the **format=type** is the type of attribute.

STEP 2

Your second step will be to read these attributes from Layout XML file and set them for the component. This logic will go in the constructors that get passed an *AttributeSet*, since that is what contains the XML attributes. To read the values in the XML, you need to first create a *TypedArray* from the *AttributeSet*, then use that to read and set the values as shown in the below example code:

```
TypedArray a = context.obtainStyledAttributes(attrs, R.styleable.DateView);


final int N = a.getIndexCount();

for (int i = 0; i < N; ++i)

{

   int attr = a.getIndex(i);

   switch (attr)

   {

      case R.styleable.DateView_delimiter:

         String delimiter = a.getString(attr);

         //...do something with delimiter...

         break;

      case R.styleable.DateView_fancyText:

         boolean fancyText = a.getBoolean(attr, false);
```

```
            //...do something with fancyText...

            break;

    }

}

a.recycle();
```

STEP 3

Finally you can use your defined attributes in your Layout XML file as follows:

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

    xmlns:tools="http://schemas.android.com/tools"

    xmlns:custom="http://schemas.android.com/apk/res/com.example.dateviewdemo"

    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" >


    <com.example.dateviewdemo.DateView

    android:layout_width="match_parent"

    android:layout_height="wrap_content"

    android:textColor="#fff"

    android:textSize="40sp"

    custom:delimiter="-"

    custom:fancyText="true"

    />


</RelativeLayout>
```

The important part is *xmlns:custom="http://schemas.android.com/apk/res/com.example.dateviewdemo"*. Note that *http://schemas.android.com/apk/res/* will remain as is, but last part will be set to your package name and also that you can use anything after xmlns:, in this example I used **custom**, but you could use any name you like. Check this below given example to understand how to Create Custom Attributes for Android Custom Component with simple steps.

Create Custom Attributes for Android Custom Component

Following example shows you how to define a simple Android custom component with custom attributes.

Step	Description
1	You will use Eclipse IDE to create an Android application and name it as *DateViewDemo*under a package *com.example.dateviewdemo* as explained in the *Hello World Example*chapter.
2	Create an XML *res/values/attrs.xml* file to define new attributes alongwith their data type.
3	Create *src/DateView.java* file and add the code to define your custom component. It will extend TextView and will have additional functionality to show current date. You will provide attributes parsing logic in of the constructors having AttributeSet as a parameter.
4	Modify *res/layout/activity_main.xml* file and add the code to create DateView instance along with few default attributes and new attributes.
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.dateviewdemo/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.dateviewdemo;

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

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) {
        // 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/values/attrs.xml** file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <declare-styleable name="DateView">
   <attr name="delimiter" format="string"/>
   <attr name="fancyText" format="boolean"/>
   </declare-styleable>
</resources>
```

Following will be the content of new file **src/com.example.dateviewdemo/DateView.java**, which will have additional functionality to show current date:

```
package com.example.dateviewdemo;
```

```java
import java.text.SimpleDateFormat;
import java.util.Calendar;

import android.content.Context;
import android.content.res.TypedArray;
import android.graphics.Color;
import android.util.AttributeSet;
import android.util.Log;
import android.widget.TextView;

public class DateView extends TextView {
   public String delimiter;
   public boolean fancyText;

   public DateView(Context context) {
      super(context);
      setDate();
   }

   public DateView(Context context, AttributeSet attrs) {
      super(context, attrs);
      TypedArray a = context.obtainStyledAttributes(attrs,
      R.styleable.DateView );
      final int N = a.getIndexCount();
      for (int i = 0; i < N; ++i)
      {
         int attr = a.getIndex(i);

         switch (attr)
         {
            case R.styleable.DateView_delimiter:
            delimiter = a.getString(attr);
            setDate();
            break;
            case R.styleable.DateView_fancyText:
            fancyText = a.getBoolean(attr, false);
            fancyText();
            break;
         }
      }
      a.recycle();
   }

   public DateView(Context context, AttributeSet attrs, int defStyle) {
      super(context, attrs, defStyle);
      setDate();
   }

   private void setDate() {
      SimpleDateFormat dateFormat =
      new SimpleDateFormat("yyyy" + delimiter + "MM" + delimiter + "dd");
      String today = dateFormat.format(Calendar.getInstance().getTime());
      setText(today);   // self = DateView = subclass of TextView
   }

   private void fancyText() {
      if( this.fancyText){
         setShadowLayer(9, 1, 1, Color.rgb(44, 44, 40));
      }
   }
```

```
    }
```

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

```xml
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:custom="http://schemas.android.com/apk/res/com.example.dateviewdemo"
    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" >

    <com.example.dateviewdemo.DateView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textColor="#fff"
    android:textSize="40sp"
    custom:delimiter="-"
    custom:fancyText="true"
    />

</RelativeLayout>
```

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

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

    <string name="app_name">DateViewDemo</string>
    <string name="action_settings">Settings</string>
    <string name="hello_world">Hello world!</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.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 **DateViewDemo** 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 ![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:

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 Eclipse IDE to create an Android application and name it as *NotificationDemo*under a package *com.example.notificationdemo*. 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 methods startNotification(), cancelNotification() and updateNotification() to cover maximum functionality related to Android notifications.
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	Copy image woman.png in *res/drawable-** folders and this image will be used as Notification icons. You can use images with different resolution in case you want to provide them for different devices.
5	Modify layout XML file *res/layout/activity_main.xml* to add three buttons in linear layout.
6	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.
7	Modify *res/values/strings.xml* to define required constant values
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.notificationdemo/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.notificationdemo;

import android.os.Bundle;
import android.app.Activity;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.app.TaskStackBuilder;
import android.content.Context;
import android.content.Intent;
import android.support.v4.app.NotificationCompat;
import android.util.Log;
import android.view.View;
import android.widget.Button;

public class MainActivity extends Activity {
   private NotificationManager mNotificationManager;
   private int notificationID = 100;
   private int numMessages = 0;

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

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

```java
      Button cancelBtn = (Button) findViewById(R.id.cancel);
      cancelBtn.setOnClickListener(new View.OnClickListener() {
         public void onClick(View view) {
            cancelNotification();
         }
      });

      Button updateBtn = (Button) findViewById(R.id.update);
      updateBtn.setOnClickListener(new View.OnClickListener() {
         public void onClick(View view) {
            updateNotification();
         }
      });
   }
   protected void displayNotification() {
      Log.i("Start", "notification");

      /* Invoking the default notification service */
      NotificationCompat.Builder  mBuilder =
      new NotificationCompat.Builder(this);

      mBuilder.setContentTitle("New Message");
      mBuilder.setContentText("You've received new message.");
      mBuilder.setTicker("New Message Alert!");
      mBuilder.setSmallIcon(R.drawable.woman);

      /* Increase notification number every time a new notification arrives */
      mBuilder.setNumber(++numMessages);

      /* Creates an explicit intent for an Activity in your app */
      Intent resultIntent = new Intent(this, NotificationView.class);

      TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
      stackBuilder.addParentStack(NotificationView.class);

      /* Adds the Intent that starts the Activity to the top of the stack */
      stackBuilder.addNextIntent(resultIntent);
      PendingIntent resultPendingIntent =
         stackBuilder.getPendingIntent(
            0,
            PendingIntent.FLAG_UPDATE_CURRENT
         );

      mBuilder.setContentIntent(resultPendingIntent);

      mNotificationManager =
      (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);

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

   protected void cancelNotification() {
      Log.i("Cancel", "notification");
      mNotificationManager.cancel(notificationID);
   }

   protected void updateNotification() {
      Log.i("Update", "notification");
```

```
      /* Invoking the default notification service */
      NotificationCompat.Builder  mBuilder =
      new NotificationCompat.Builder(this);

      mBuilder.setContentTitle("Updated Message");
      mBuilder.setContentText("You've got updated message.");
      mBuilder.setTicker("Updated Message Alert!");
      mBuilder.setSmallIcon(R.drawable.woman);

    /* Increase notification number every time a new notification arrives */
      mBuilder.setNumber(++numMessages);

      /* Creates an explicit intent for an Activity in your app */
      Intent resultIntent = new Intent(this, NotificationView.class);

      TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
      stackBuilder.addParentStack(NotificationView.class);

      /* Adds the Intent that starts the Activity to the top of the stack */
      stackBuilder.addNextIntent(resultIntent);
      PendingIntent resultPendingIntent =
         stackBuilder.getPendingIntent(
            0,
            PendingIntent.FLAG_UPDATE_CURRENT
         );

      mBuilder.setContentIntent(resultPendingIntent);

      mNotificationManager =
      (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);

      /* Update the existing notification using same notification ID */
      mNotificationManager.notify(notificationID, mBuilder.build());
   }
}
```

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

```
package com.example.notificationdemo;

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

public class NotificationView extends Activity{
   @Override
   public void onCreate(Bundle savedInstanceState)
   {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.notification);
   }

}
```

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"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/start_note"/>


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

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

</LinearLayout>
```

Following will be the content of **res/layout/notification.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:layout_width="fill_parent"
    android:layout_height="400dp"
    android:text="Hi, Your Detailed notification view goes here...." />
</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">NotificationDemo</string>
    <string name="action_settings">Settings</string>
    <string name="hello_world">Hello world!</string>
    <string name="start_note">Start Notification</string>
    <string name="cancel_note">Cancel Notification</string>
    <string name="update_note">Update Notification</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.notificationdemo"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="17"
        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.notificationdemo.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=".NotificationView"
                android:label="Details of notification"
                android:parentActivityName=".MainActivity">
        <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity"/>
            </activity>
    </application>

</manifest>
```

Let's try to run your **NotificationDemo** 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 **Start Notification** button, you will see at the top a message "New Message Alert!" will display momentarily and after that you will have following screen having a small icon at the top left corner.
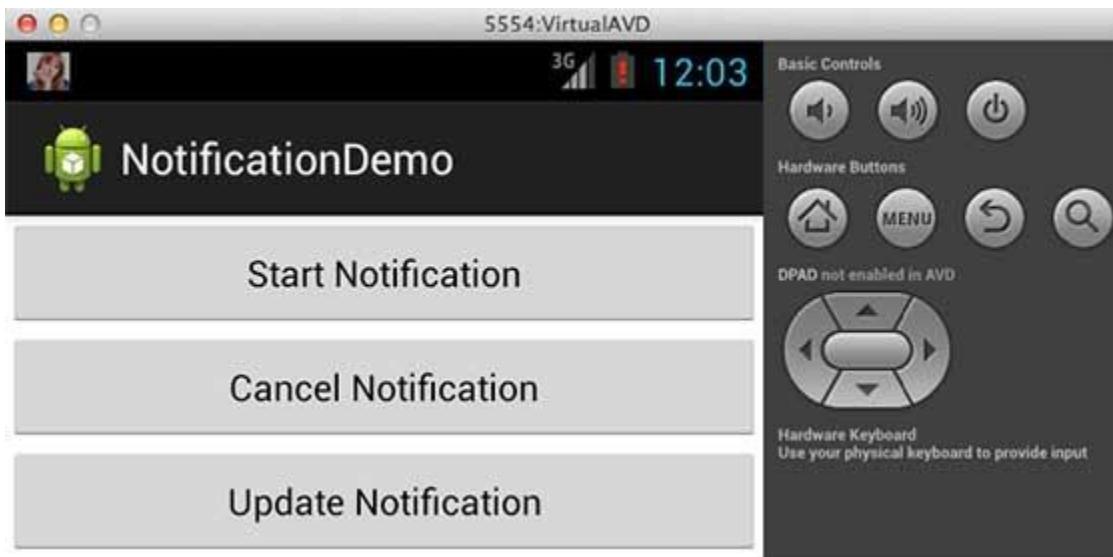
Now lets expand the view, long click on the small icon, after a second it will display date information and this is the time when you should drag status bar down without releasing mouse. You will see status bar will expand and you will get following screen:
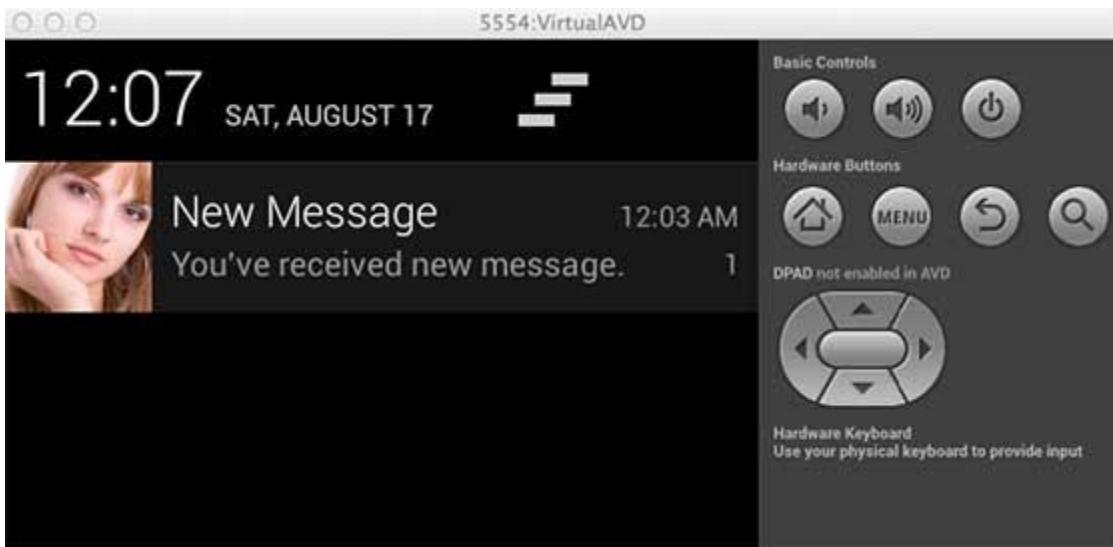
Now let's try to click on the image icon, this will launch your new activity which you have set using intent and you will have following screen:
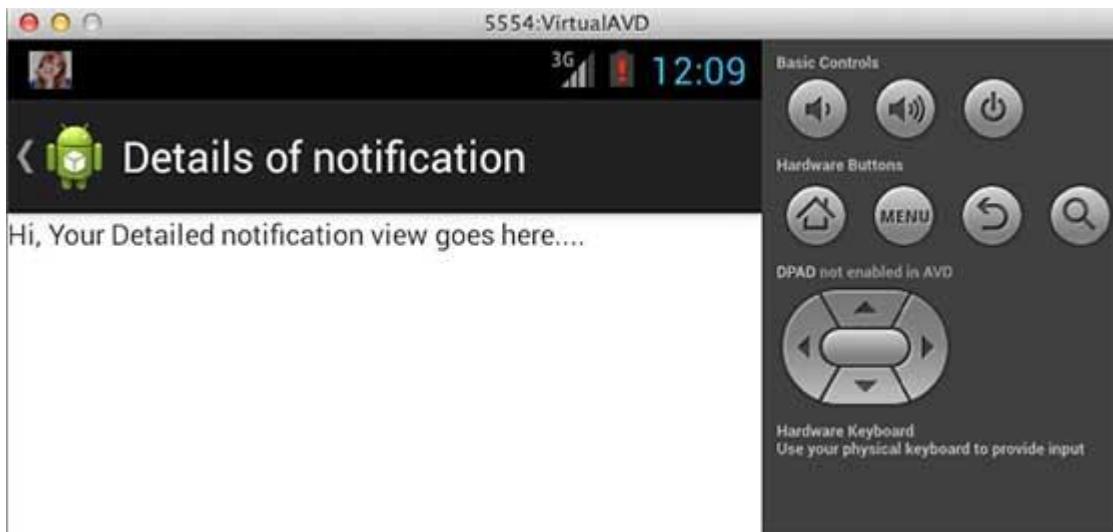
Next, you can click on "Detail of notification" and it will take you back to the main screen where you can try using **Update Notification** button which will update existing notification and number will increase by 1 but if you will send notification with new notification ID then it will keep adding in the stack and you see them separately listed on the screen.

Big View Notification

The following code snippet demonstrates how to alter the notification created in the previous snippet to use the Inbox big view style. I'm going to update displayNotification() modification method to show this functionality:

```java
protected void displayNotification() {
    Log.i("Start", "notification");

    /* Invoking the default notification service */
    NotificationCompat.Builder  mBuilder =
    new NotificationCompat.Builder(this);

    mBuilder.setContentTitle("New Message");
    mBuilder.setContentText("You've received new message.");
    mBuilder.setTicker("New Message Alert!");
    mBuilder.setSmallIcon(R.drawable.woman);

    /* Increase notification number every time a new notification arrives */
    mBuilder.setNumber(++numMessages);


    /* Add Big View Specific Configuration */
    NotificationCompat.InboxStyle inboxStyle =
            new NotificationCompat.InboxStyle();

    String[] events = new String[6];
    events[0] = new String("This is first line....");
    events[1] = new String("This is second line...");
    events[2] = new String("This is third line...");
    events[3] = new String("This is 4th line...");
    events[4] = new String("This is 5th line...");
    events[5] = new String("This is 6th line...");

    // Sets a title for the Inbox style big view
    inboxStyle.setBigContentTitle("Big Title Details:");
    // Moves events into the big view
```

```
        for (int i=0; i < events.length; i++) {

            inboxStyle.addLine(events[i]);
        }
        mBuilder.setStyle(inboxStyle);


        /* Creates an explicit intent for an Activity in your app */
        Intent resultIntent = new Intent(this, NotificationView.class);

        TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
        stackBuilder.addParentStack(NotificationView.class);

        /* Adds the Intent that starts the Activity to the top of the stack */
        stackBuilder.addNextIntent(resultIntent);
        PendingIntent resultPendingIntent =
            stackBuilder.getPendingIntent(
                0,
                PendingIntent.FLAG_UPDATE_CURRENT
            );

        mBuilder.setContentIntent(resultPendingIntent);

        mNotificationManager =
        (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);

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

Now if you will try to run your application then you will find following result in expanded form of the view: