



# **SRI VENKATESWARA ENGINEERING COLLEGE FOR WOMEN**

**R15 - LECTURE NOTES**

**ON**

**MOBILE APPLICATION DEVELOPMENT**

**(15A05703)**

# MOBILE APPLICATION DEVELOPMENT

## LECTURE NOTES

### UNIT: 1

#### Introduction to android:

The android 4.1 jelly bean SDK, understanding the android software stack, installing the android sdk, creating android virtual devices, creating the first android project, using the text view control, using the android emulator, the android debug bridge(ADB), Launching android applications on a handset.



#### 1. The android 4.1 jelly bean SDK:

with [Google I/O 2012](#) now kicked off and [Android 4.1 Jelly Bean](#) unveiled, the time has come to download the tools to make the magic. During this mornings keynote address, it was announced Android 4.1 would be rolling out mid-July while the SDK would be made available to developers today. The update has gone live now and is available for download to those who can make use of it. Some of the highlights to look forward to:

- ) **Faster, Smoother, More Responsive** - Android 4.1 is optimized to deliver Android's best performance and lowest touch latency, in an effortless, intuitive UI.
- ) **Enhanced Accessibility** - New APIs for accessibility services let you handle gestures and manage accessibility focus as the user moves through the on-screen elements and navigation buttons using accessibility gestures, accessories, and other input. The Talkback system and explore-by-touch are redesigned to use accessibility focus for easier use and offer a complete set of APIs for developers.
- ) **Support For International Users** - Android 4.1 helps you to reach more users through support for bi-directional text in TextView and EditText elements. Apps can display text or handle text editing in left-to-right or right-to-left scripts. Apps can make use of new Arabic and Hebrew locales and associated fonts.
- ) **Expandable Notifications** - Android 4.1 brings a major update to the Android notifications framework. Apps can now display larger, richer notifications to users that can be expanded and collapsed with a pinch. Notifications support new types of content, including photos, have configurable priority, and can even include multiple actions.

- ) **Resizable app widgets** - Android 4.1 introduces improved App Widgets that can automatically resize, based on where the user drops them on the home screen, the size to which the user expands them, and the amount of room available on the home screen. New App Widget APIs let you take advantage of this to optimize your app widget content as the size of widgets changes.

## Android Architecture

**Android architecture** or **Android software stack** is categorized into five parts:

1. linux kernel
2. native libraries (middleware),
3. Android Runtime
4. Application Framework
5. Applications

Let's see the android architecture first.

### 1) Linux kernel

It is the heart of android architecture that exists at the root of android architecture. **Linux kernel** is responsible for device drivers, power management, memory management, device management and resource access.

### 2) Native Libraries

On the top of linux kernel, there are **Native libraries** such as WebKit, OpenGL, FreeType, SQLite, Media, C runtime library (libc) etc. The WebKit library is responsible for browser support, SQLite is for database, FreeType for font support, Media for playing and recording audio and video formats.

### 3) Android Runtime

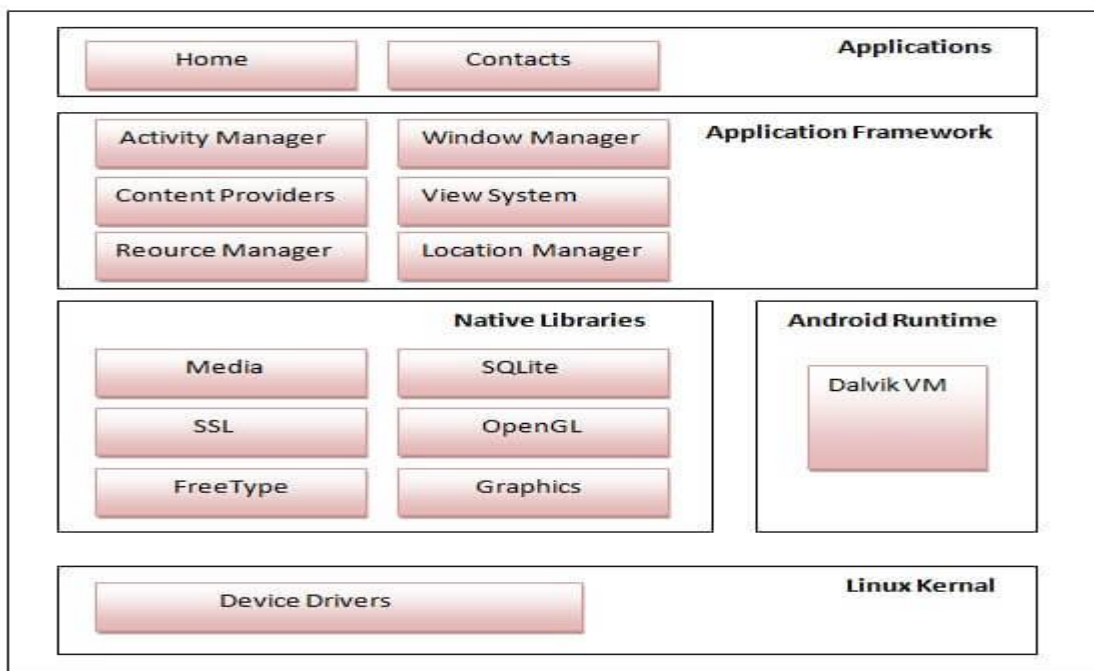
In android runtime, there are core libraries and DVM (Dalvik Virtual Machine) which is responsible to run android application. DVM is like JVM but it is optimized for mobile devices. It consumes less memory and provides fast performance.

#### 4) Android Framework

On the top of Native libraries and android runtime, there is android framework. Android framework includes **Android API's** such as UI (User Interface), telephony, resources, locations, Content Providers (data) and package managers. It provides a lot of classes and interfaces for android application development.

#### 5) Applications

On the top of android framework, there are applications. All applications such as home, contact, settings, games, browsers are using android framework that uses android runtime and libraries. Android runtime and native libraries are using linux kernel.



### Download Android Studio

Google provides Android Studio for the Windows, Mac OS X, and Linux platforms. You can [download this software](#) from the Android Studio homepage. (You'll also find the traditional SDKs, with Android Studio's command-line tools, available from the Downloads page.) Before downloading Android Studio, make sure your platform meets one of the following requirements:

#### Windows OS

- ) Microsoft Windows 7/8/10 (32-bit or 64-bit)
- ) 2 GB RAM minimum, 8 GB RAM recommended
- ) 2 GB of available disk space minimum, 4 GB Recommended (500 MB for IDE + 1.5 GB for Android SDK and emulator system image)
- ) 1280 x 800 minimum screen resolution
- ) JDK 8
- ) For accelerated emulator: 64-bit operating system and Intel processor with support for Intel VT-x, Intel EM64T (Intel 64), and Execute Disable (XD) Bit functionality

Once you've ensured your operating system is compatible with Android Studio 2.1.1, download the appropriate Android Studio distribution file. The Android Studio download page auto-detected that I'm running 64-bit Windows 8.1 and selected `android-studio-bundle-143.2821654-windows.exe` for me to download.

#### Bundled installer and Android SDK

`android-studio-bundle-143.2821654-windows.exe` includes an installer and the Android SDK. Alternatively, I could have downloaded a distribution file without the installer and without the SDK.

#### Installing Android Studio on 64-bit Windows 8.1

I launched `android-studio-bundle-143.2821654-windows.exe` to start the installation process. The installer responded by presenting the Android Studio Setup dialog box shown in Figure 1.



Figure 1. Set up Android Studio

Clicking Next took me to the following dialog box, which gives you the option to decline installing the Android SDK (included with the installer) and an Android Virtual Device (AVD).

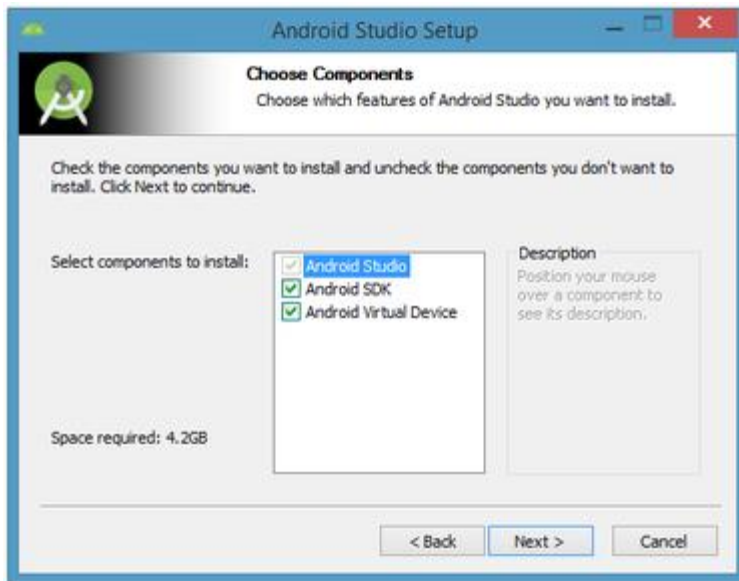


Figure 2. Do you want to install the Android SDK and AVD?

I chose to keep the default settings. After clicking Next, you'll be taken to the license agreement dialog box. Accept the license to continue the installation.

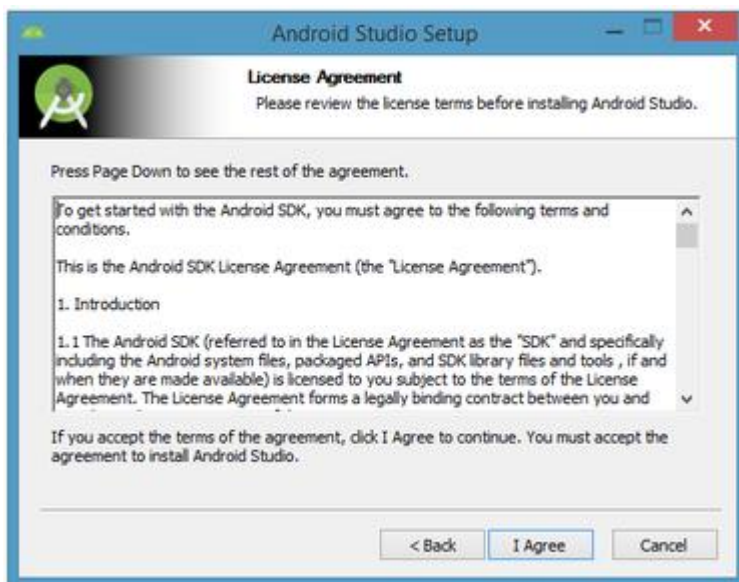


Figure 3. Accept the license agreement to continue installation

The next dialog box invites you to change the installation locations for Android Studio and the Android SDK.

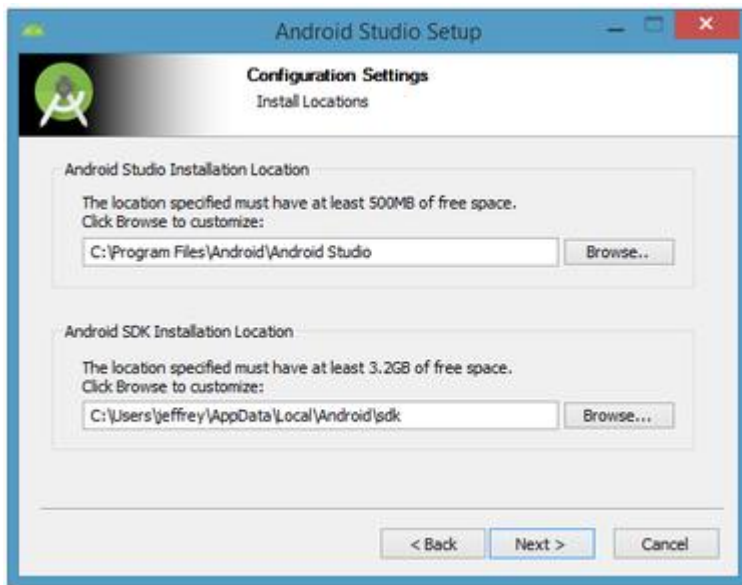


Figure 4. Set the Android Studio and Android SDK installation locations  
Change the location or accept the default locations and click Next.

The installer defaults to creating a shortcut for launching this program, or you can choose to decline. I recommend that you create the shortcut, then click the Install button to begin installation.

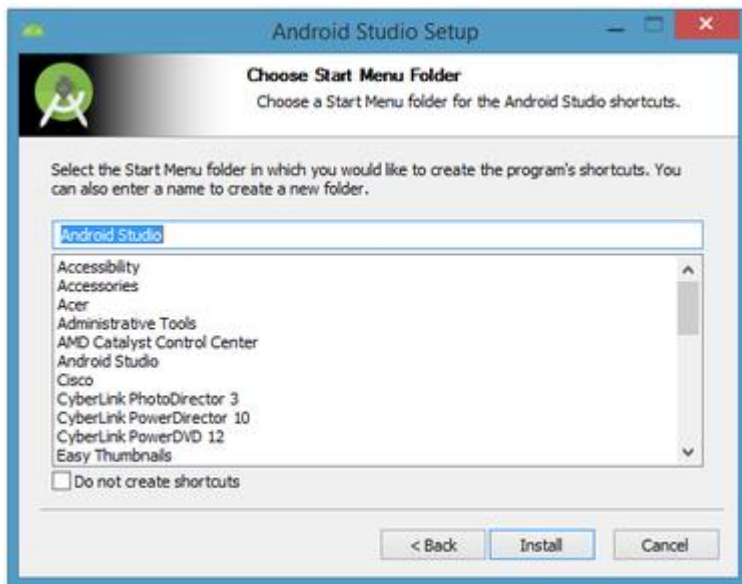


Figure 5. Create a new shortcut for Android Studio

The resulting dialog box shows the progress of installing Android Studio and the Android SDK. Clicking the Show Details button will let you view detailed information about the installation progress.

The dialog box will inform you when installation has finished. When you click Next, you should see the following:



Figure 6. Leave the Start Android Studio check box checked to run this software  
To complete your installation, leave the Start Android Studio box checked and click Finish.

## Running Android Studio

Android Studio presents a splash screen when it starts running:



Figure 7. Android Studio's start screen

On your first run, you'll be asked to respond to several configuration-oriented dialog boxes. The first dialog box focuses on importing settings from any previously installed version of Android Studio.



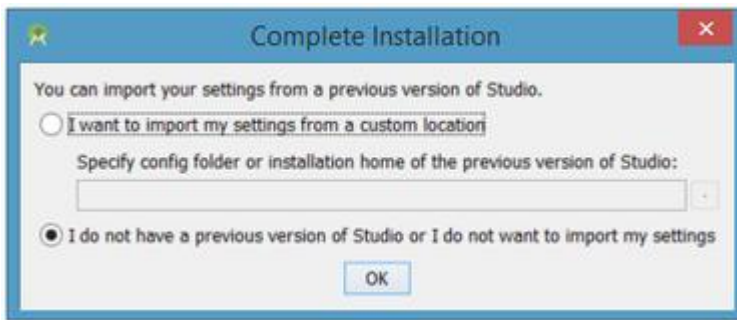


Figure 8. Import settings

If you're like me, and don't have a previously installed version, you can just keep the default setting and click OK. Android Studio will respond with a slightly enhanced version of the splash screen, followed by the Android Studio Setup Wizard dialog box:

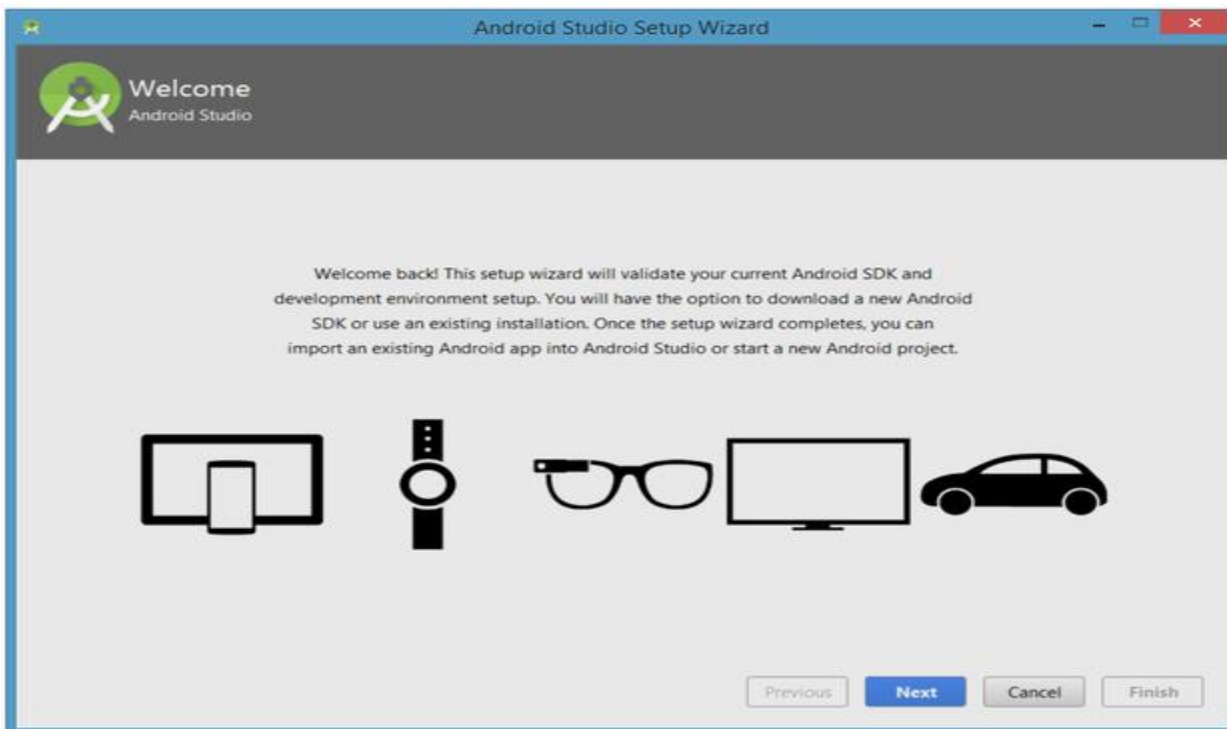


Figure 9. Validate your Android SDK and development environment setup

When you click Next, the setup wizard invites you to select an installation type for your SDK components. For now I recommend you keep the default standard setting.

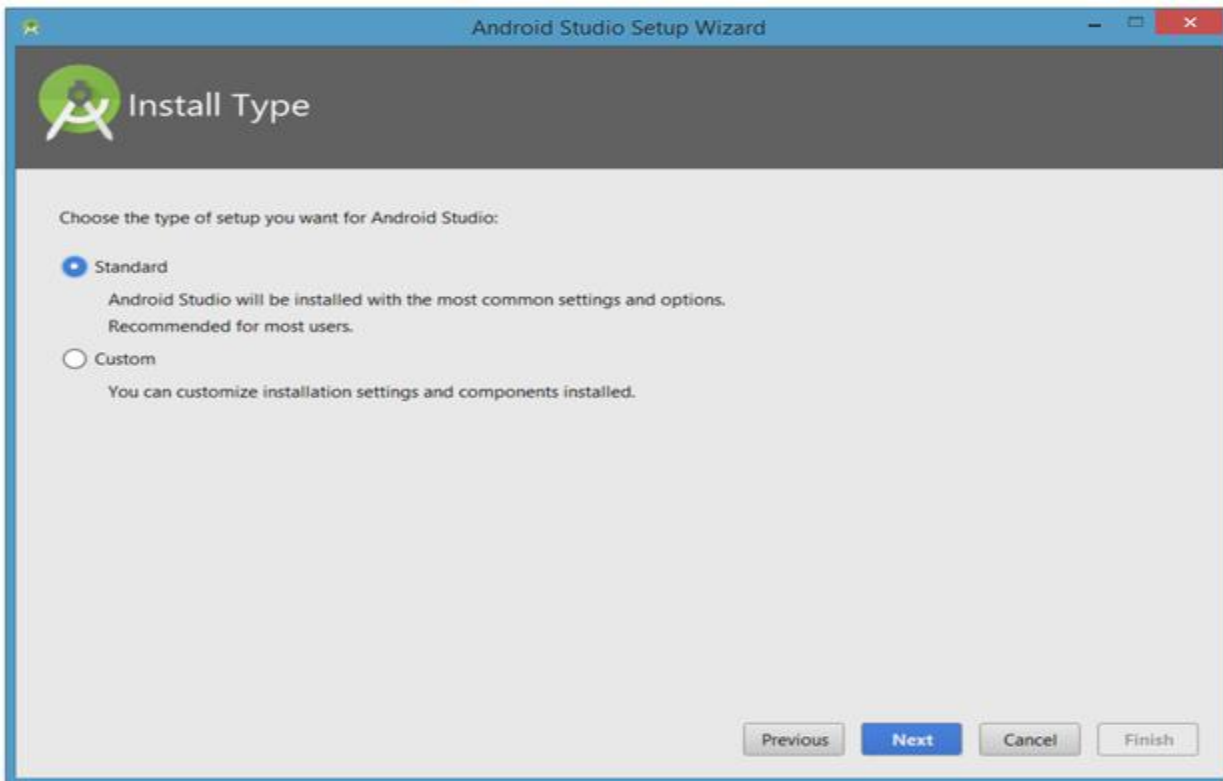


Figure 10. Choose an installation type  
Click Next and verify your settings, then click Finish to continue.

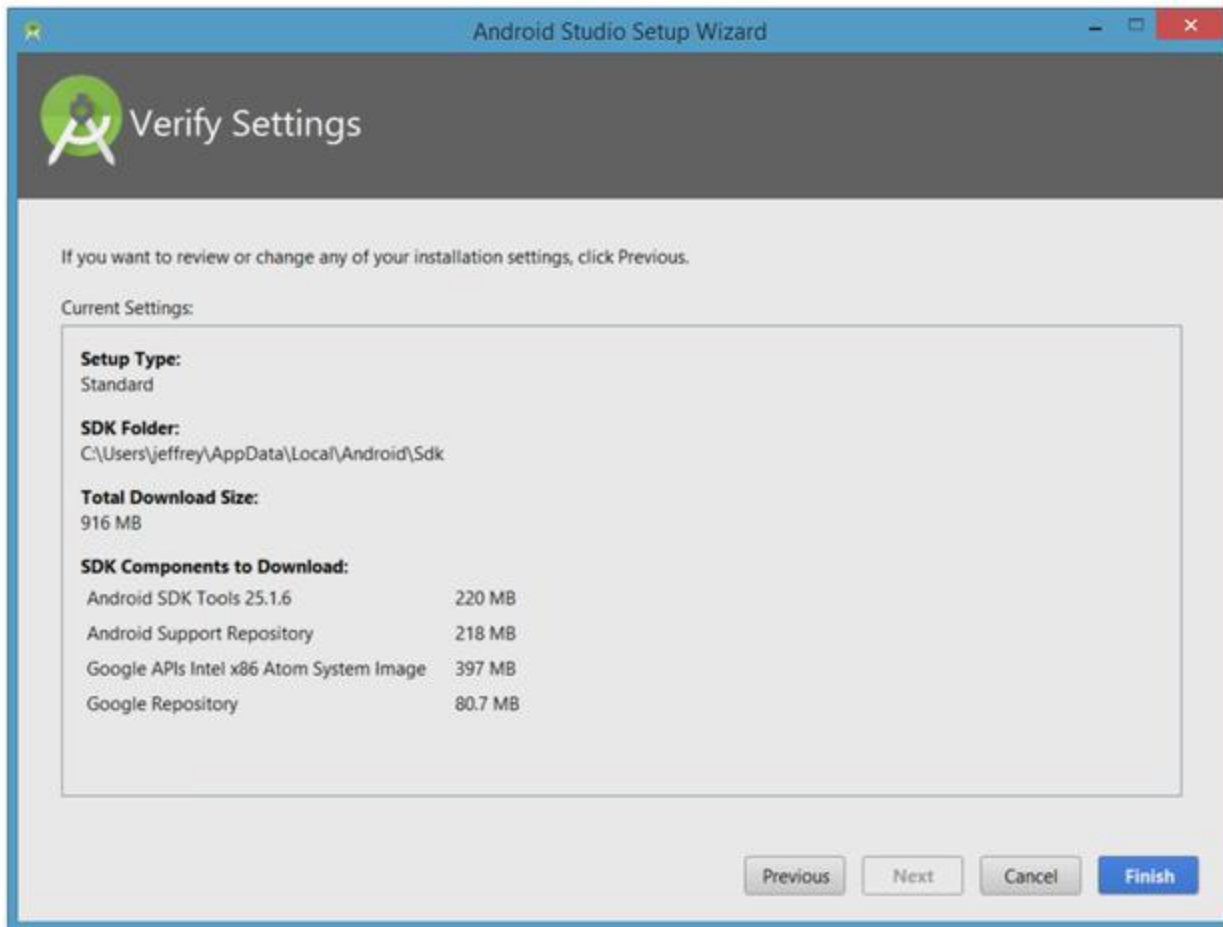


Figure 11. Review settings

The wizard will download and unzip various components. Click Show Details if you want to see more information about the archives being downloaded and their contents.

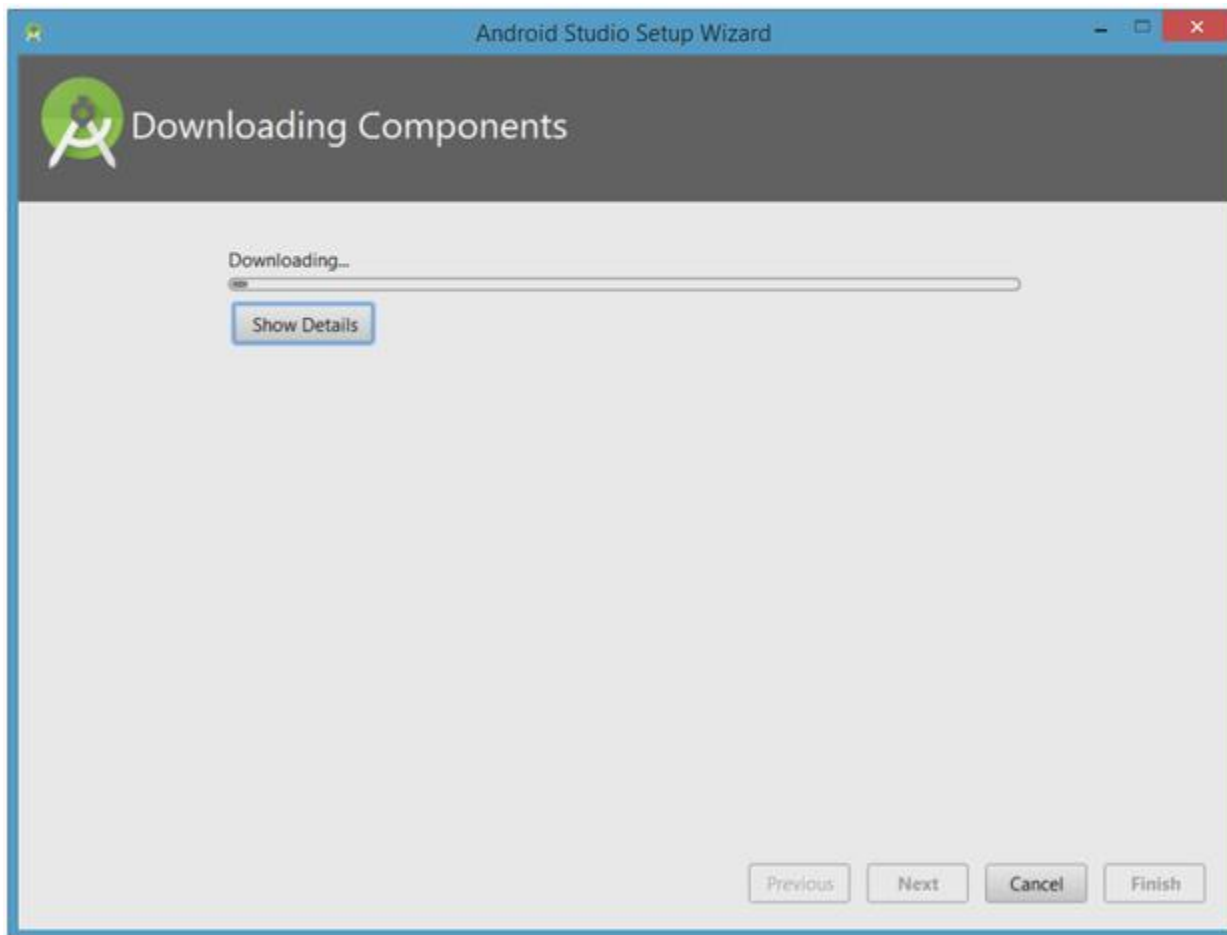


Figure 12. The wizard downloads and unzips Android Studio components

If your computer isn't Intel based, you might get an unpleasant surprise after the components have completely downloaded and unzipped:

```

C:\Program Files\Android\Android Studio\bin>
m2repository/com/google/firebase/firebase-core/9.0
.1/firebase-core-9.0.1.pom.md5
Android SDK is up to date.
Unable to install Intel HAXM
Your CPU does not support required features (VT-x or SVM).
Unfortunately, your computer does not support hardware
accelerated virtualization.
Here are some of your options:
1) Use a physical device for testing
2) Develop on a Windows/OSX computer with an Intel processor
that supports VT-x and NX
3) Develop on a Linux computer that supports VT-x or SVM
4) Use an Android Virtual Device based on an ARM system image
(This is 10x slower than hardware accelerated
virtualization)

Creating Android virtual device
Unable to create a virtual device: Unable to create Android
virtual device

```

Figure 13. Intel-based hardware acceleration is unavailable

Your options are to either put up with the slow emulator or use an Android device to speed up development. I'll discuss the latter option later in the tutorial.

Finally, click Finish to complete the wizard. You should see the Welcome to Android Studio dialog box:



Figure 14. Welcome to Android Studio

You'll use this dialog to start up a new Android Studio project, work with an existing project, and more. You can access it anytime by double-clicking the Android Studio shortcut on your desktop.

## Your first Android Project

The quickest way to get to know Android Studio is to use it to develop an app. We'll start with a variation on the "Hello, World" application: a little mobile app that displays a "Welcome to Android" message.

In the steps that follow, you'll start a new Android Studio project and get to know the project workspace, including the project editor that you'll use to code the app in Part 2.

## Starting a new project

From our setup so far, you should still have Android Studio running with the Welcome to Android Studio dialog box. From here, click Start a new Android Studio project. Android Studio will respond with the Create New Project dialog box shown in Figure 15.

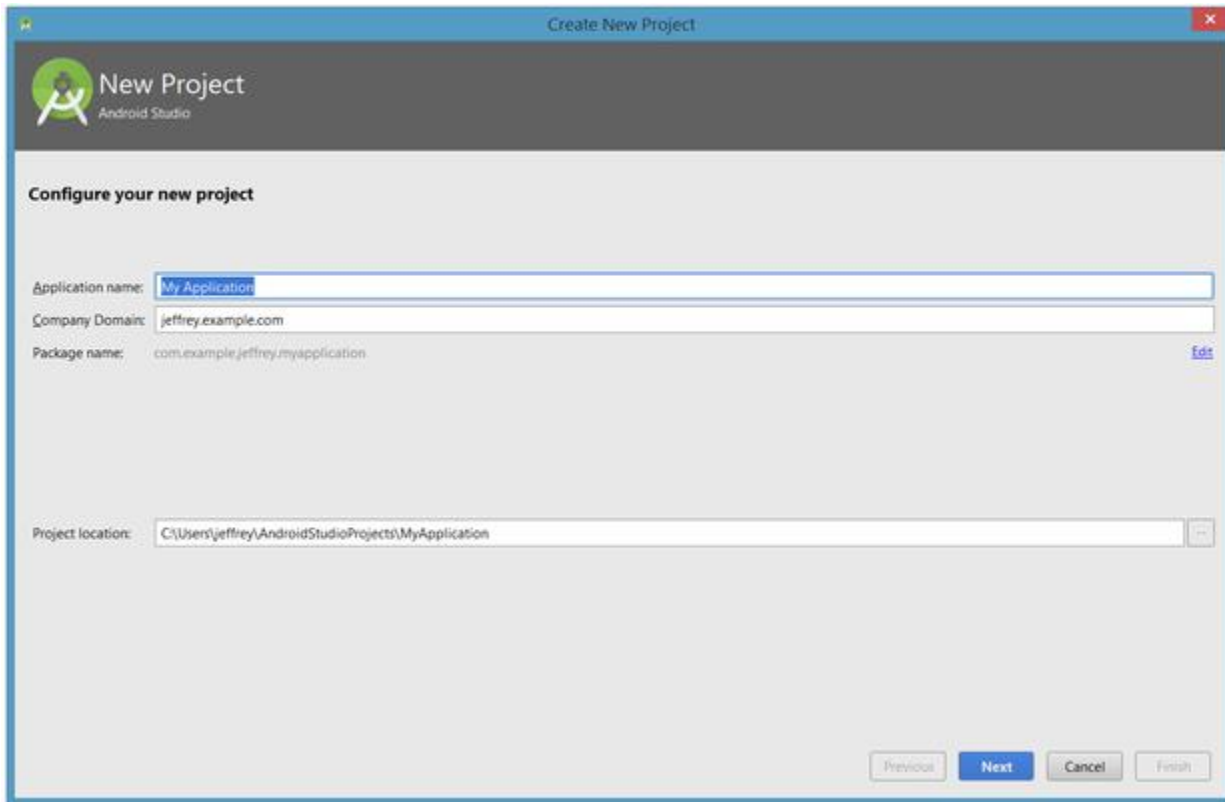


Figure 15. Create a new project

Enter W2A (Welcome to Android) as the application name and *javajeff.ca* as the company domain name. You should then see C:\Users\jeffrey\AndroidStudioProjects\W2A as the project location. Click Next to select your target devices.

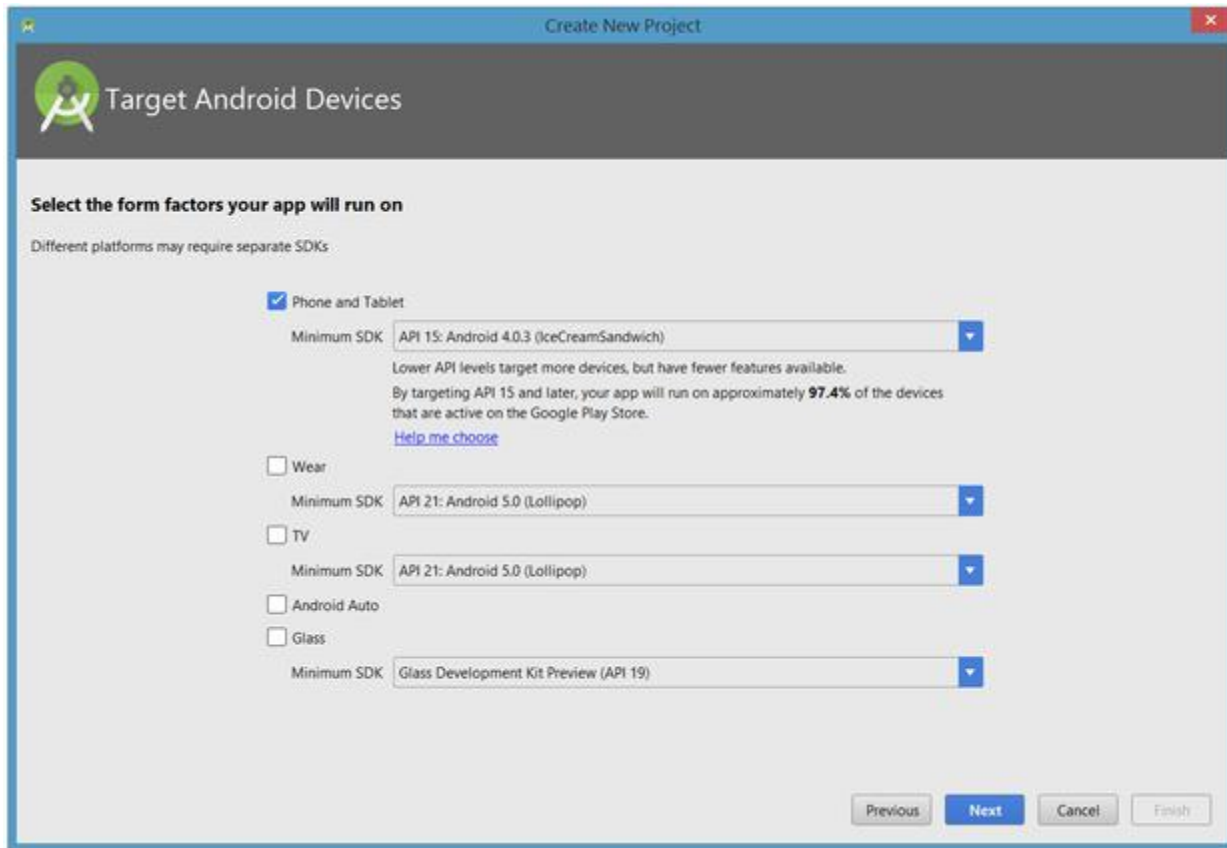


Figure 16. Select your target device categories

Android Studio lets you select *form factors*, or categories of target devices, for every app you create. I would have preferred to keep the default API 15: Android 4.0.3 (IceCreamSandwich) minimum SDK setting (under Phone and Tablet), which is supported by my Amazon Kindle Fire HD tablet. Because Android Studio doesn't currently support this API level (even when you add the 4.0.3 system image via the SDK Manager), I changed this setting to API 14: Android 4.0 (IceCreamSandwich), which is also supported by my tablet.

Click Next, and you will be given the opportunity to choose a template for your app's main activity. For now we'll stick with Empty Activity. Select this template and click Next.

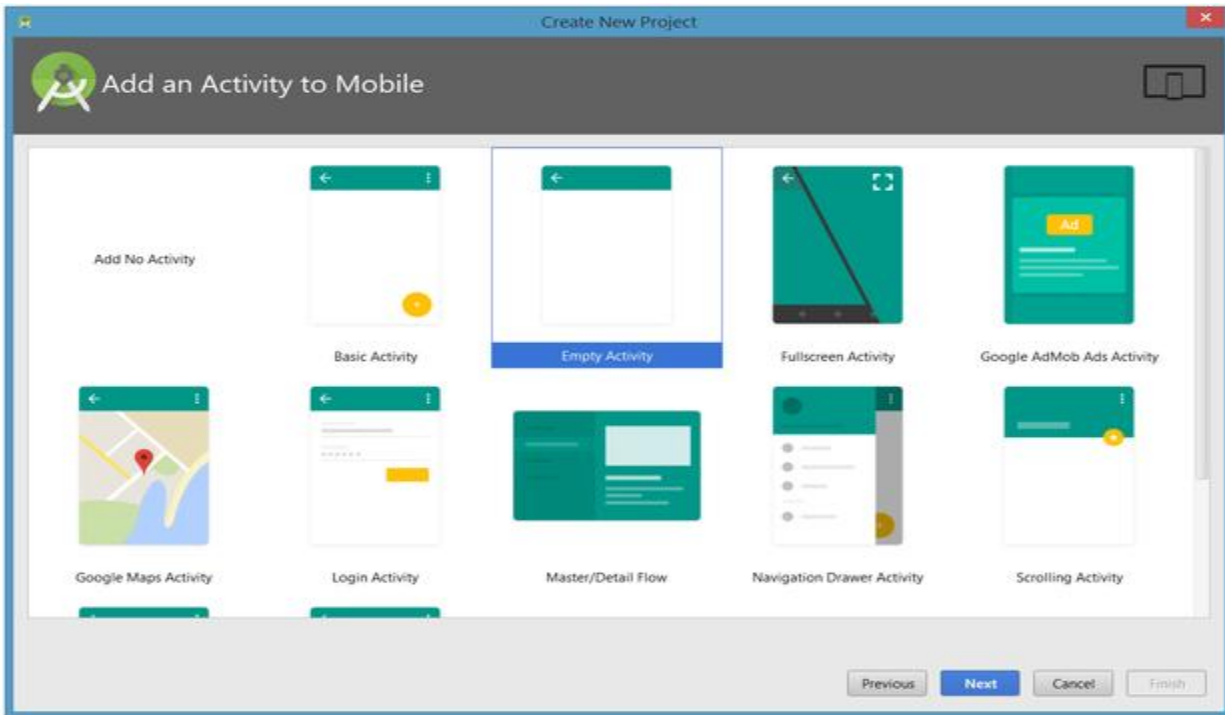


Figure 17. Specify an activity template  
Next you'll customize the activity:

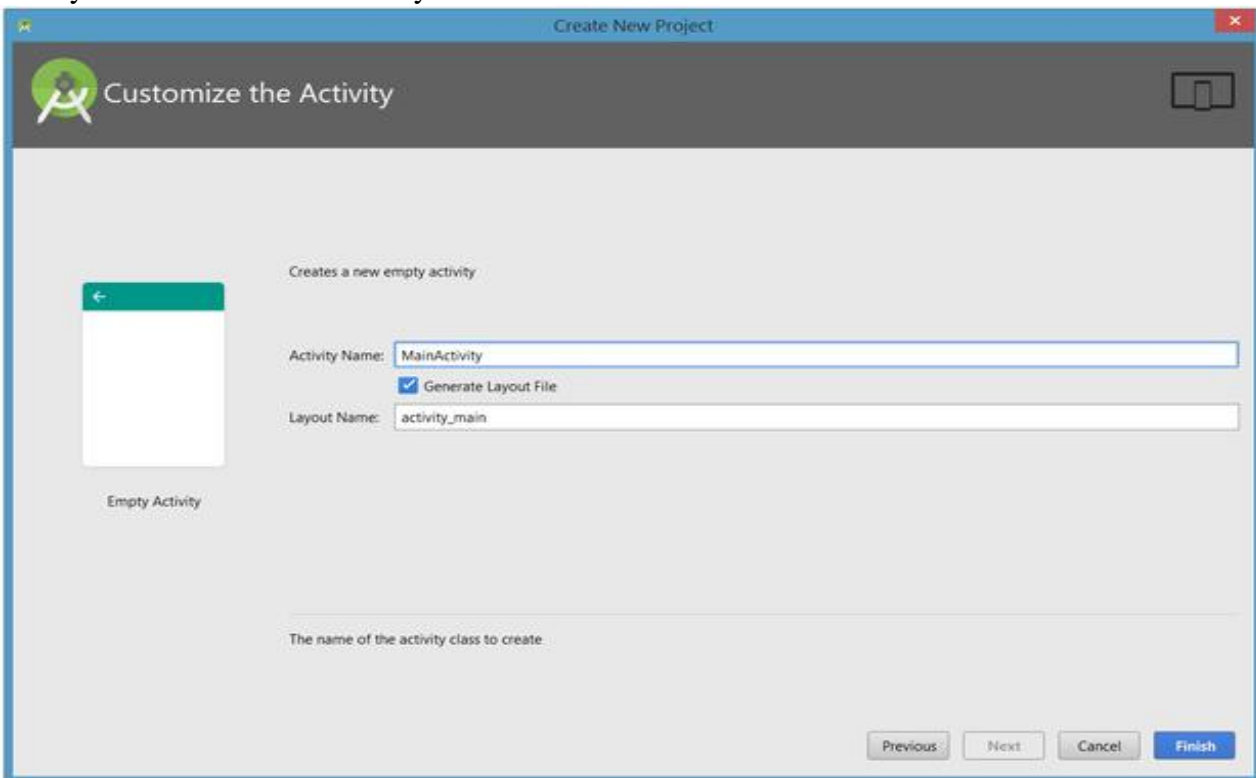


Figure 18. Customize your activity

Enter *W2A* as the activity name and *main* as the layout name, and click Finish to complete this step. Android Studio will respond that it is creating the project, then take you to the project workspace.



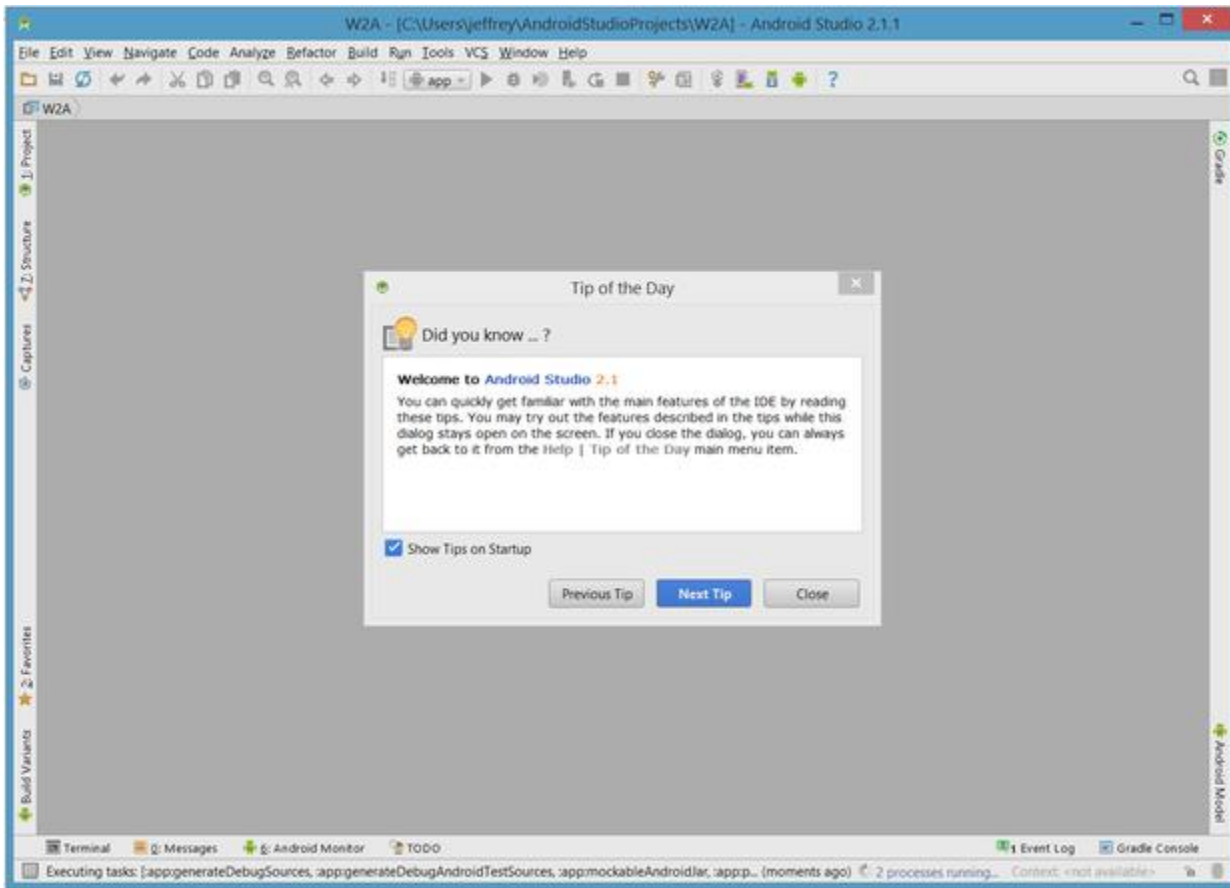


Figure 19. Android Studio workspace

The project workspace is organized around a menu bar, a tool bar, a work area, additional components that lead to more windows (such as a Gradle Console window), and a status bar. Also note the Tip of the Day dialog box, which you can disable if you like.

#### Accessing AVD Manager or SDK Manager from menu and tool bar

To access the traditional AVD Manager or SDK Manager, select Android from the Tools menu followed by AVD Manager or SDK Manager from the resulting pop-up menu (or click their tool bar icons).

#### The project and editor windows

When you enter the project workspace, W2A is identified as the current project, but you won't immediately see the project details. After a few moments, these details will appear in two new windows.

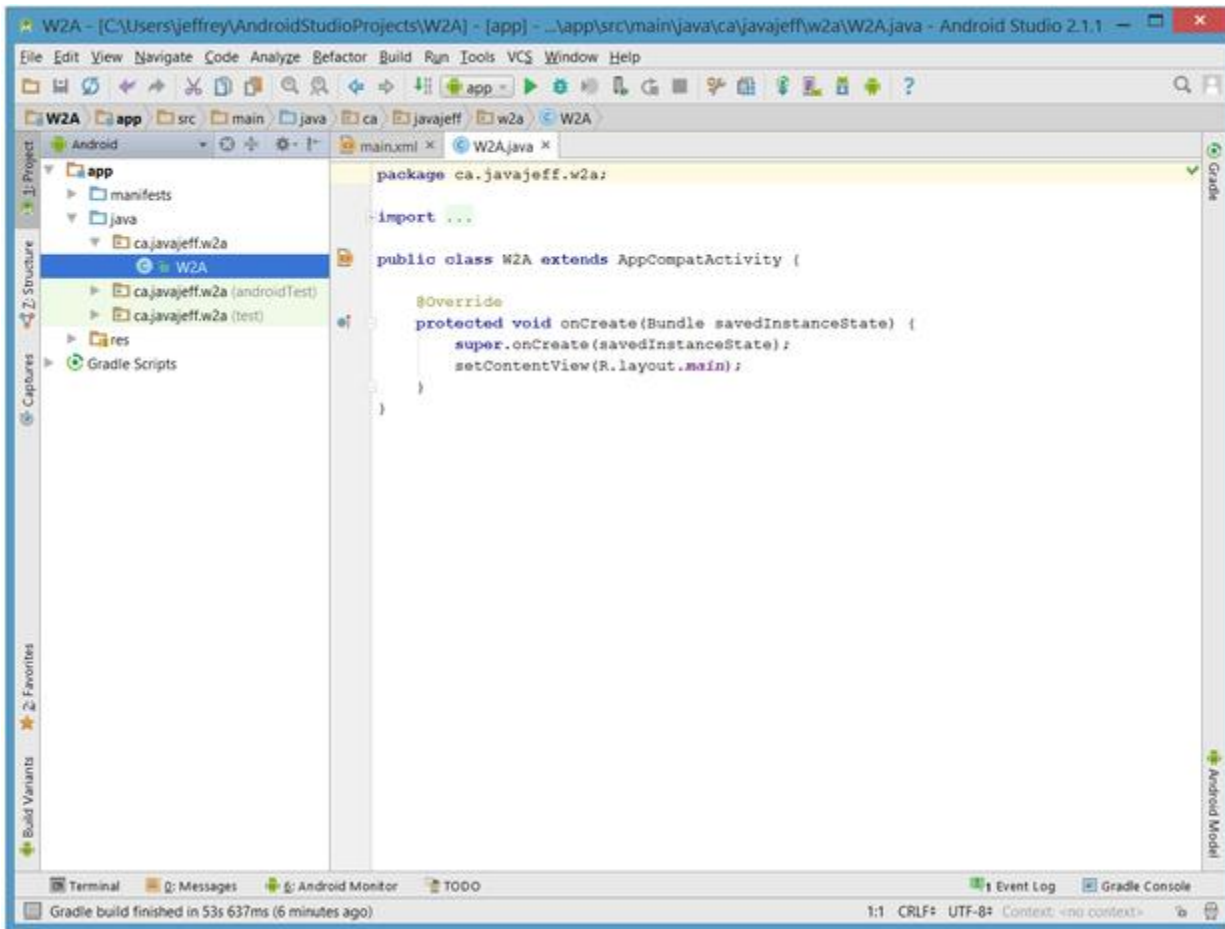


Figure 20. The project and editor windows

The project window is organized into a tree whose main branches are App and Gradle Scripts. The App branch is further organized into manifests, java, and res subbranches:

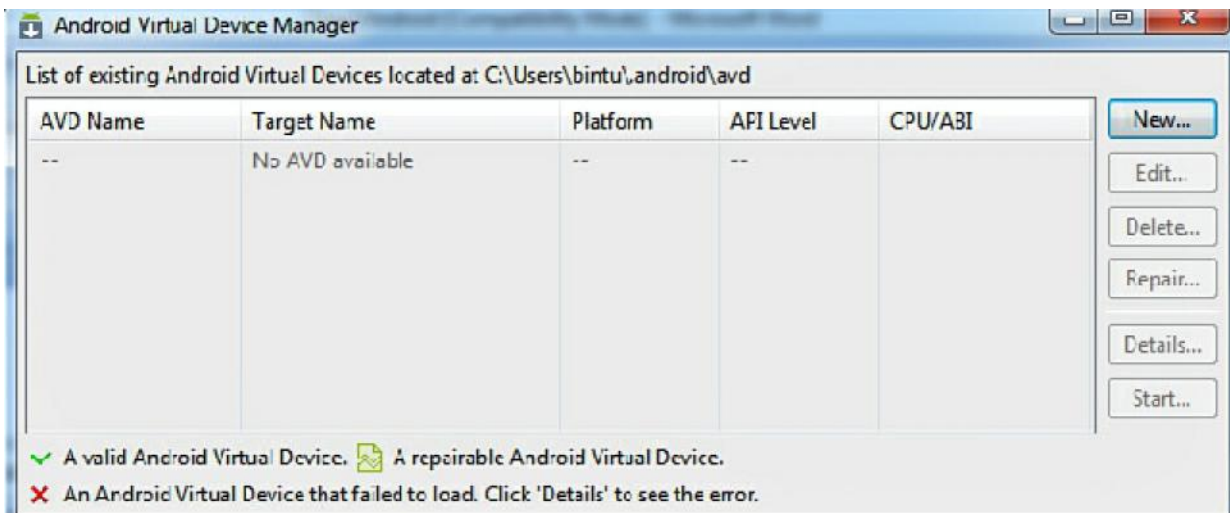
- manifests stores **AndroidManifest.xml**, which is an XML file that describes the structure of an Android app. This file also records permission settings (where applicable) and other details about the app.
    - java stores an app's Java source files according to a package hierarchy, which is **ca.javajeff.w2a** in this example.
    - res stores an app's resource files, which are organized into drawable, layout, mipmap, and values subbranches:
      - drawable: an initially empty location in which to store an app's artwork
      - layout: a location containing an app's layout files; initially, **main.xml**(the main activity's layout file) is stored here
      - mipmap: a location containing various **ic\_launcher.png** files that store launcher screen icons of different resolutions
      - values: a location containing **colors.xml**, **dimens.xml**, **strings.xml**, and **styles.xml**

The Gradle Scripts branch identifies various **.gradle** (such as **build.gradle**) and **.properties** (such as **local.properties**) files that are used by the Gradle-based build system.

### Creating Android Virtual Devices

An Android Virtual Device (AVD) represents a device configuration. There are many Android devices, each with different configuration. To test whether the Android application is compatible with a set of Android devices, you can create AVDs that represent their configuration.

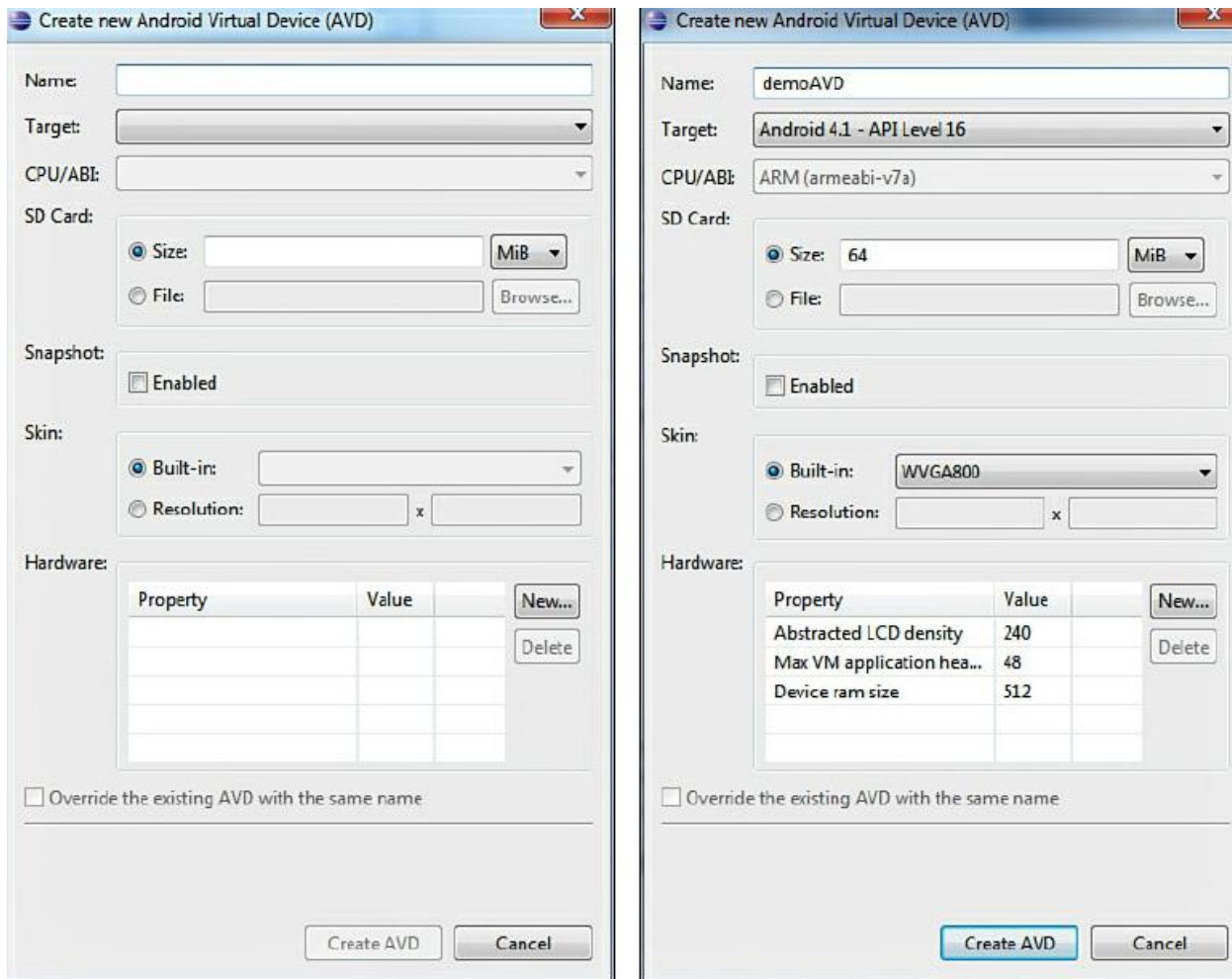
To create AVDs in Eclipse, select the Window, AVD Manager option. An Android Virtual DeviceManager dialog opens, as shown in [Figure](#). The dialog box displays a list of existing AVDs, letting you create new AVDs and manage existing AVDs. Because you haven't yet defined an AVD, an empty list is displayed.



Select the New button to define a new AVD. A Create new Android Virtual Device (AVD) dialog box, appears (see Figure —left). The fields are as follows:

- **Name**—Used to specify the name of the AVD.
- **Target**—Used to specify the target API level. Our application will be tested against the specified API level.
- **CPU/ABI**—Determines the processor that we want to emulate on our device.
- **SD Card**—Used for extending the storage capacity of the device. Large data files such as audio and video for which the built-in flash memory is insufficient are stored on the SD card.
- **Snapshot**—Enable this option to avoid booting of the emulator and start it from the last saved snapshot. Hence, this option is used to start the Android emulator quickly.
- **Skin**—Used for setting the screen size. Each built-in skin represents a specific screen size. You can try multiple skins to see if your application works across different devices.
- **Hardware**—Used to set properties representing various optional hardware that may be

present in the target device.



## Using the TextView Control

So far, our HelloWorldApp contains default code. We simply ran the application whose default structure and code was created for us by the ADT plug-in. Now we learn to use the TextView control, removing the default text and entering our own. You can assign text to the TextView in two ways:

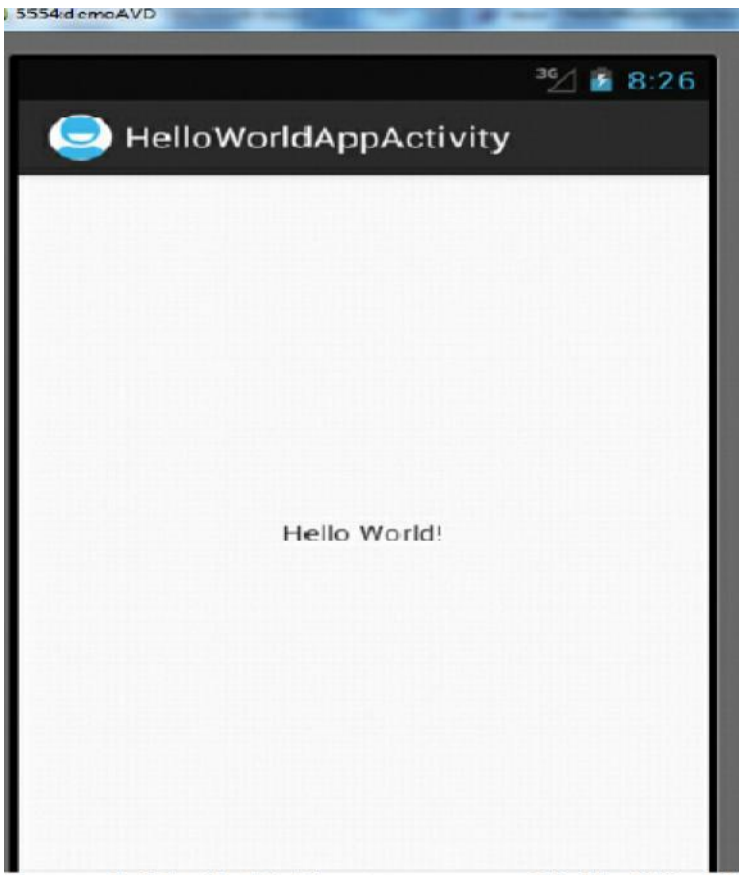
- Direct assignment to the TextView control as defined in the layout file `activity_hello_world_app.xml`
- Indirectly through the Java Activity file `HelloWorldAppActivity.java`

## Assigning the Text Directly in the Layout File

The text that you want to be displayed through the `TextView` control can be assigned to it in its XMLdefinition in the layout file `activity_hello_world_app.xml`. From the Package Explorer window,open `activity_hello_world_app.xml` by double-clicking it in the `res/layout` folder. Modify

`activity_hello_world_app.xml` to appear as shown,

```
<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:text="Hello World!"
tools:context=".HelloWorldAppActivity" />
</RelativeLayout>
```



### Assigning Text Through the Activity File

To assign text to the `TextView` control through the Java Activity file `HelloWorldAppActivity.java` you need to do the following two things:

- **Remove the text from XML definition**—The first thing that you need to do is to remove the text that was assigned to the `TextView` control in the layout file `activity_hello_world_app.xml`.

Open the `activity_hello_world_app.xml` file and remove the statement `android:text="HelloWorld!"`. The `TextView` control will appear blank and will not display anything on execution.

- **Assign an ID to the `TextView` control**—To access the `TextView` control in the Activity file, you have to uniquely identify it by assigning it an ID. To assign an ID to a control, use the `android:id` attribute. Add the following statement to the `TextView` tag in the `activity_hello_world_app.xml` file:  
`android:id="@+id/message"`

This statement declares that a constant called `message` is assigned as an ID to the `TextView`.

```
<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:id="@+id/message"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
tools:context=".HelloWorldAppActivity" />
</RelativeLayout>

```

```

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;
public class HelloWorldAppActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        TextView mesg =
        (TextView)findViewById(R.id.message); #1
        mesg.setText("Hello
        World!"); #2
    }
}

```

## Using the Android Emulator

The Android emulator is used for testing and debugging applications before they are loaded onto a real handset. The Android emulator is integrated into Eclipse through the ADT plug-in.

### Limitations of the Android Emulator

The Android emulator is useful to test Android applications for compatibility with devices of different configurations. But still, it is a piece of software and not an actual device and has several limitations:

- Emulators no doubt help in knowing how an application may operate within a given environment, but they still don't provide the actual environment to an application. For example, an actual device has memory, CPU, or other physical limitations that an emulator doesn't reveal.

- Emulators just simulate certain handset behavior. Features such as GPS, sensors, battery, power settings, and network connectivity can be easily simulated on a computer.
- SMS messages are also simulated and do not use a real network.
- Phone calls cannot be placed or received but are simulated.
- No support for device-attached headphones is available.
- Peripherals such as camera/video capture are not fully functional.
- No USB or Bluetooth support is available.

### The Android Debug Bridge (ADB)

The Android Debug Bridge (ADB) is a client-server program that is part of the Android SDK. It is used to communicate with, control, and manage the Android device and emulator.

It consists of three components:

- **Client**—Runs on a computer machine. It can be invoked from the command prompt using the ADB command.
- **Daemon**—Runs as a background process in either an emulator instance or in the device itself.
- **Server**—Runs in a computer machine in the background. It manages the communication between the client and the daemon.

When you are in the folder where ADB is found, you can issue the following commands to interact with the device or emulator:

- **adb devices**—Displays the list of devices attached to the computer. Figure 1.33 shows the currently running emulator on your computer.

**adb push**—Copies files from your computer to the device/emulator.

Syntax:

`adb push source destination`

where *source* refers to the file along with its path that you want to copy, and *destination* refers to the place in the device or emulator where you want to copy the file.

**adb pull**—Copies files from the device/emulator to your computer.

Syntax:

`adb pull source [destination]`

**adb install**—Installs an application from your computer to the device/emulator.

Syntax:

`adb install appname.apk`



## **Launching Android Applications on a Handset**

To load an application onto a real handset, you need to plug a handset into your computer, using the USB data cable. You first confirm whether the configurations for debugging your application are correct and then launch the application as described here:

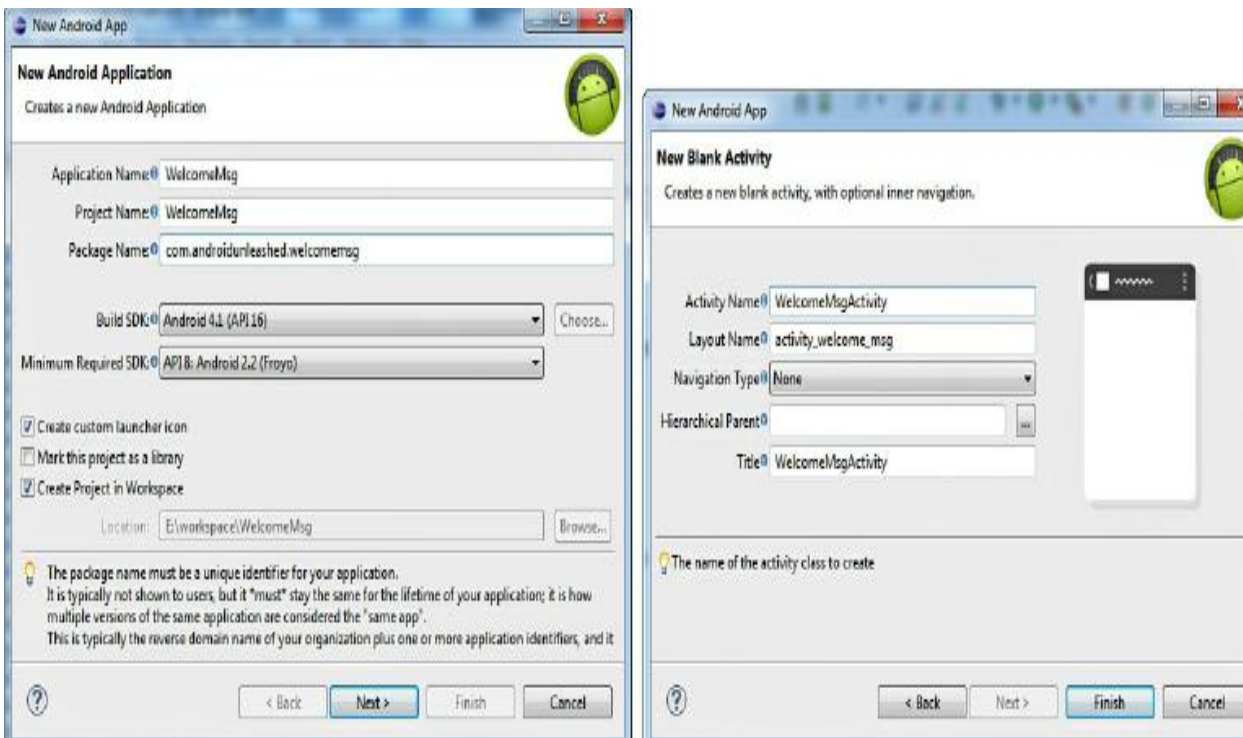
1. In Eclipse, choose the Run, Debug Configurations option.
2. Select the configuration HelloWorldApp\_configuration, which you created for the HelloWorldApp application.
3. Select the Target tab, set the Deployment Target Selection Mode to Manual. The Manual option allows us to choose the device or AVD to connect to when using this launch configuration.
4. Apply the changes to the configuration file by clicking the Apply button.
5. Plug an Android device into your computer, using a USB cable.
6. Select Run, Debug in Eclipse or press the F11 key. A dialog box appears, showing all available configurations for running and debugging your application. The physical device(s) connected to the computer are also listed. Double-click the running Android device. Eclipse now installs the Android application on the handset, attaches a debugger, and runs the application.

## **Unit 2**

### **The Role of Android Application Components**

To gain an understanding of the role and functionality of different components that make up an Android application, you create a new Android application and study each of its components.

To create an application, open Eclipse and choose, File, New, Android Application Project or click the Android Project.



In the Application Name box, enter the name of the Android project. Let's name the application WelcomeMsg. The Project Name is automatically assigned, which is the same as the application name by default. Being a unique identifier, the Package Name is com.androidunleashed.welcomemsg.

- From the Build SDK drop-down, select Android 4.1 (API 16) as the target platform as we expect it to be the version commonly used by your target audience.
- Select API 8: Android 2.2 (Froyo) from the Minimum Required SDK drop-down to indicate that the application requires at least API level 8 to run.
- Because this is a new project, select the Create Project in Workspace check box to create the project files at the Workspace location specified when opening Eclipse for the first time.

The Create custom launcher icon check box is selected by default and enables us to define the icon of our application. Click the Next button to move to the next dialog box.

- Keeping the auto-assigned layout filename and the title unchanged, click the Finish button after supplying the required information. The application is created by ADT, along with all the necessary files.

The next dialog (see [Figure 2.1](#)—right) asks us to enter information about the newly created activity. Name the activity `WelcomeMsgActivity`. The layout filename and title name automatically change to reflect the newly assigned activity name. The layout name becomes `activity_welcome_msg`, and the Title of the application also changes to `WelcomeMsgActivity`.

We can always change the auto-assigned layout filename and title of the application.

- Keeping the auto-assigned layout filename and the title unchanged, click the `Finish` button after supplying the required information. The application is created by ADT, along with all the necessary files.

### **Understanding the Utility of Android API**

The Android platform provides a framework API that applications can use to interact with the underlying Android system.

The framework API consists of a core set of packages and classes; XML elements for declaring layouts, resources, and so on; a manifest file to configure applications;

The framework API is specified through an integer called API level, and each Android platform version supports exactly one API level, although backward compatibility is there;

**Table 2.1. The Android Platform and Corresponding API Levels**

Platform Version	API Level	Code Name
Android 4.1	16	Jelly Bean
Android 4.0.3	15	Ice Cream Sandwich
Android 4.0	14	Ice Cream Sandwich
Android 3.2	13	Honeycomb
Android 3.1	12	Honeycomb
Android 3.0	11	Honeycomb
Android 2.3.3	10	Gingerbread
Android 2.3.1	9	Gingerbread
Android 2.2	8	Froyo
Android 2.1	7	Eclair
Android 2.0.1	6	Eclair
Android 2.0	5	Eclair
Android 1.6	4	Donut
Android 1.5	3	Cupcake
Android 1.1	2	
Android 1.0	1	

## Overview of the Android Project Files

The following files and directories are created for the Android application. The list below is just an overview of the files and directories.

- **/src folder**—The folder that contains the entire Java source file of the application. The folder contains a directory structure corresponding to the package name supplied in the application.

The folder contains the project's default package: `com.androidunleashed.welcomemsg`. On expanding the package, you find the Activity of the application, the `WelcomeMsgActivity.java` file, within it.

- **/gen folder**—Contains Java files generated by ADT on compiling the application. That is, the `gen` folder will come into existence after compiling the application for the first time. The folder contains an `R.java` file that contains references for all the resources defined in the `res`.
- **/gen/com.androidunleashed.welcomemsg/R.java**—All the layout and other resource information that is coded in the XML files is converted into Java source

code and placed in the `R.java` file. It also means that the file contains the ID of all the resources of the application.

- **Android SDK jar file**—The jar file for the target platform.

drawable, layout, and values.

- **/res/drawable-xhdpi, /res/drawable-hdpi, /res/drawable-mdpi, /res/drawable-ldpi**—the application's icon and graphic resources are kept in these folders. Because devices have screens of different densities, the graphics of different resolutions are kept in these folders.

- **/res/layout**—Stores the layout file(s) in XML format.

- **/res/values**—Stores all the values resources. The values resources include many types such as string resource, dimension resource, and color resource.

- **/res/layout/activity\_welcome\_msg.xml**—The layout file used by `WelcomeMsgActivity` to draw views on the screen. The views or controls are laid in the specified layout.

- **/res/values/strings.xml**—Contains the string resources. String resources contain the text matter to be assigned to different controls of the applications. This file also defines string arrays.

- **AndroidManifest.xml**—The central configuration file for the application.

## The manifest file

---

Before the Android system can start an app component, the system must know that the component exists by reading the app's *manifest file*, `AndroidManifest.xml`. Your app must declare all its components in this file, which must be at the root of the app project directory.

The manifest does a number of things in addition to declaring the app's components, such as the following:

- ) Identifies any user permissions the app requires, such as Internet access or read-access to the user's contacts.
- ) Declares the minimum [API Level](#) required by the app, based on which APIs the app uses.
- ) Declares hardware and software features used or required by the app, such as a camera, bluetooth services, or a multitouch screen.
- ) Declares API libraries the app needs to be linked against (other than the Android framework APIs), such as the [Google Maps library](#).

## Declaring components

The primary task of the manifest is to inform the system about the app's components. For example, a manifest file can declare an activity as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
  <applicationandroid:icon="@drawable/app_icon.png" ... >
    <activityandroid:name="com.example.project.ExampleActivity"
      android:label="@string/example_label" ... >
    </activity>
    ...
  </application>
</manifest>
```

In the `<application>` element, the `android:icon` attribute points to resources for an icon that identifies the app.

In the `<activity>` element, the `android:name` attribute specifies the fully qualified class name of the `Activity` subclass and the `android:label` attribute specifies a string to use as the user-visible label for the activity.

You must declare all app components using the following elements:

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

Activities, services, and content providers that you include in your source but do not declare in the manifest are not visible to the system and, consequently, can never run. However, broadcast receivers can be either declared in the manifest or created dynamically in code as `BroadcastReceiver` objects and registered with the system by calling `registerReceiver()`.

For more about how to structure the manifest file for your app, see [The AndroidManifest.xml File](#) documentation.

## Creating the User Interface

There are three approaches to creating user interfaces in Android. You can create user interfaces entirely in Java code or entirely in XML or in both (defining the user interface in XML and then referring and modifying it through Java code). The third approach, the combined approach, is highly preferred.

```

<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="match_parent"
android:layout_height="match_parent">
<TextView
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:text="Enter your name:"/>
<EditText
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:id="@+id/user_name"/>
<Button
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:id="@+id/click_btn"
android:text="Click Me"/>
<TextView
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:id="@+id/response"/>

</LinearLayout>

```

**Code Written in the activity\_welcome\_msg.xml File.**

## Commonly Used Layouts and Controls

The views or the controls that we want to display in an application are arranged in an order or sequence by placing them in the desired layout. The layouts also known as Containers or ViewGroups are used for organizing the views or controls in the required format.

**LinearLayout**—In this layout, all elements are arranged in a descending column from top to bottom or left to right. Each element contains properties to specify how much of the screenspace it will consume. Depending on the orientation parameter, elements are either arranged in row or column format.

- **RelativeLayout**—In this layout, each child element is laid out in relation to other child elements. That is, a child element appears in relation to the previous child. Also, the elements are laid out in relation to the parent.
- **AbsoluteLayout**—In this layout, each child is given a specific location within the bounds of the parent layout object. This layout is not suitable for devices with different screen sizes and hence is deprecated.
- **FrameLayout**—This is a layout used to display a single view. Views added to this are always placed at the top left of the layout. Any other view that is added to the FrameLayout overlaps the previous view; that is, each view stacks on top of the previous one.
- **TableLayout**—In this layout, the screen is assumed to be divided in table rows, and each of the child elements is arranged in a specific row and column.
- **GridLayout**—In this layout, child views are arranged in a grid format, that is, in the rows and columns pattern. The views can be placed at the specified row and column location. Also, more than one view can be placed at the given row and column position.

## Event Handling

Three ways of event handling:

- Creating an anonymous inner class
- Implementing the OnClickListener interface
- Declaring the event handler in the XML definition of the control.

### Creating an Anonymous Inner Class

In this method of event handling, you implement a listener inline; that is, an anonymous class is defined with an OnClickListener interface, and an onClick(View v) method is implemented in it. The anonymous inner class is passed to the listener through the setOnClickListener() method. To implement the concept of anonymous inner class for event handling, the Java activity file.

```
package com.androidunleashed.welcomemsg;  
import android.app.Activity;  
import android.os.Bundle;
```



```

import android.widget.TextView;
import android.widget.EditText;
import android.widget.Button;
import android.view.View;
public class WelcomeMsgActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_welcome_msg);
        Button b = (Button)this.findViewById(R.id.click_btn);
        b.setOnClickListener(new Button.OnClickListener(){
            public void onClick(View v) {
                TextView resp = (TextView)
                findViewById(R.id.response);
                EditText name = (EditText)findViewById(R.id.user_name);
                String str = "Welcome " +name.getText().toString() + " !";
                resp.setText(str);
            }
        });
    }
}

```

### Activity Implementing the `OnClickListener` Interface

In this method of event handling, the Activity class implements the `OnClickListener` interface, which requires us to implement the `onClick()` method in this class. The `onClick()` method is declared inside the class, and the listener is set by passing a reference to the class by the following statement: `b.setOnClickListener(this);`

```

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;
import android.widget.EditText;
import android.widget.Button;
import android.view.View;
import android.view.View.OnClickListener;
public class WelcomeMsgActivity extends Activity implements
OnClickListener {

```

```

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_welcome_msg);
    Button b = (Button)this.findViewById(R.id.click_btn);
    b.setOnClickListener(this);
}
public void onClick(View v) {
    TextView resp =
    (TextView)this.findViewById(R.id.response);
    EditText name =
    (EditText)this.findViewById(R.id.user_name);
    String str = "Welcome " + name.getText().toString() + "
    !";
    resp.setText(str);
}
}

```

## Declaring the Event Handler in the XML Control Definition

Since Android SDK 1.6, there has been another way to set up a click handler for the Button controls.

In the XML definition of a Button in the layout file `activity_welcome_msg.xml`, you can add an `android:onClick` attribute. This attribute is used to represent the method you want to execute when a click event occurs via the Button control.

```

<Button
    android:id="@+id/click_btn"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Click Me"
    android:onClick="dispMessage" />

```

```

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

```

```

import android.widget.EditText;
public class WelcomeMsgActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_welcome_msg);
    }
    public void dispMessage(View v) {
        TextView resp = (TextView) findViewById(R.id.response);
        EditText name = (EditText) findViewById(R.id.user_name);
        String str = "Welcome " + name.getText().toString() + "
!";
        resp.setText(str);
    }
}

```

## Displaying Messages Through Toast

A Toast is a transient message that automatically disappears after a while without user interaction. It is usually used to inform the user about happenings that are not very important and does not create a problem if they go unnoticed. A Toast is created by calling the static method, `makeText()`, of the Toast class. The syntax of the `makeText()` method is shown here:

```
Toast.makeText(activity_context, string_to_display, duration).
```

```

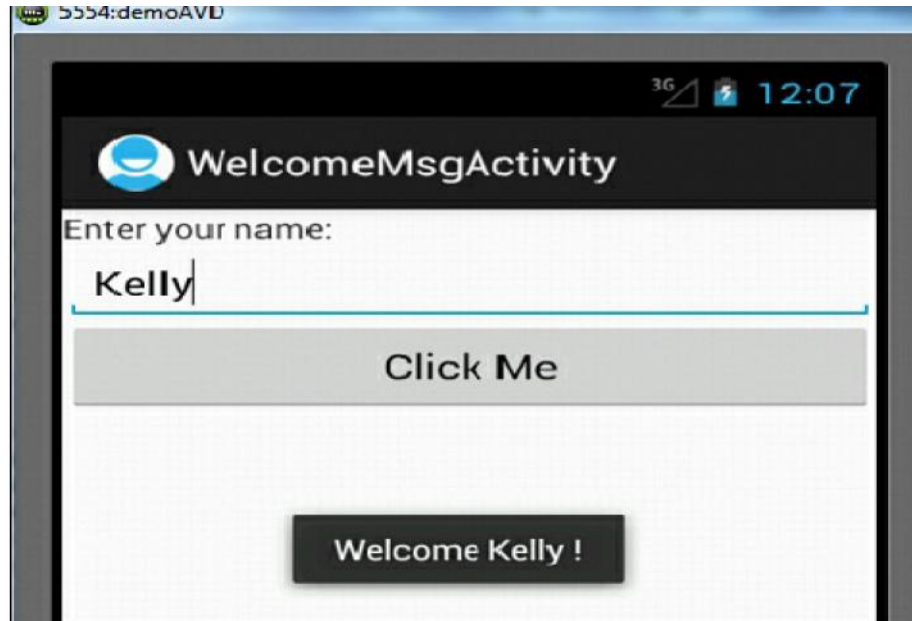
import android.app.Activity;
import android.os.Bundle;
import android.widget.EditText;
import android.view.View;
import android.widget.Toast;
public class WelcomeMsgActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_welcome_msg);
    }
}

```

```

public void dispMessage(View v) {
    EditText name = (EditText) findViewById(R.id.user_name);
    String str = "Welcome " + name.getText().toString() + "
    !";
    Toast.makeText(WelcomeMsgActivity.this, str,
    Toast.LENGTH_SHORT).show();
}
}

```



## Creating and Starting an Activity

The structure that is used to start, stop, and transition between Activities within an application is called an `Intent`.

`Intent` can be explicit or implicit as follows:

- **Explicit Intent**—In an explicit intent, you specify the Activity required to respond to the intent; that is, you explicitly designate the target component. Because the developers of other applications have no idea of the component names in your application, an explicit intent is limited to be used within an application—for example, transferring internal messages.

- **Implicit Intent**—In an implicit intent, you just declare intent and leave it to the platform to find an Activity that can respond to the intent. That is, you don't specify the target component that should respond to the intent. This type of intent is used for activating components of other applications.

The method used to start an activity **is** `startActivity()`.



## Using the `EditText` Control

The `EditText` control is a subclass of `TextView` and is used for getting input from the user. You can use several attributes to configure the `EditText` control to suit your requirements.

The default behavior of the `EditText` control is to display text as a single line and run over to the next line when the user types beyond the width of the control.

### Example:

```
<EditText
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:hint="Enter your name"
android:singleLine="false"
android:id="@+id/user_name" />
```

### Attributes:

**android:layout\_width**—Used to set the width of the `EditText` control. The two valid values are `wrap_content` and `match_parent`. The value `wrap_content` sets the width of the `EditText` control to accept only a single character.

**android:layout\_height**—Used to set the height of the `EditText` control. Valid values are `wrap_content` and `match_parent`. When the value `wrap_content` is assigned to the control, the height of the `EditText` control increases when typing text beyond the width of the control.

**android:singleLine**—When set to `true`, forces the `EditText` control to remain on a single line.

**android:hint**—Displays helpful text in the `EditText` control to guide user entry.

**android:lines**—Sets the height of the `EditText` control to accommodate the specified number of lines.

**android:textSize**—Sets the size of the text typed in the `EditText` control. You can specify the size in any of the following units of measurement: `px`, `in`, `mm`, `pts`, `dip`, and `sp`.

**android:autoText**—When set to `true` enables the `EditText` control to correct common spelling mistakes.

**android:capitalize**—Automatically converts typed text into capital letters.

**android:password**—When set to `true`, the attribute converts the typed characters into dots to hide entered text.

**android:inputType**—Specifies the type of data that will be typed in the `EditText` control.

Ex: Refer notes.

## Choosing Options with `CheckBox`

A checkbox control has two states: `checked` and `unchecked`. When the check box is selected, it toggles its state from `checked` to `unchecked` and vice versa. A check box is created via an instance of `android.widget.CheckBox`.

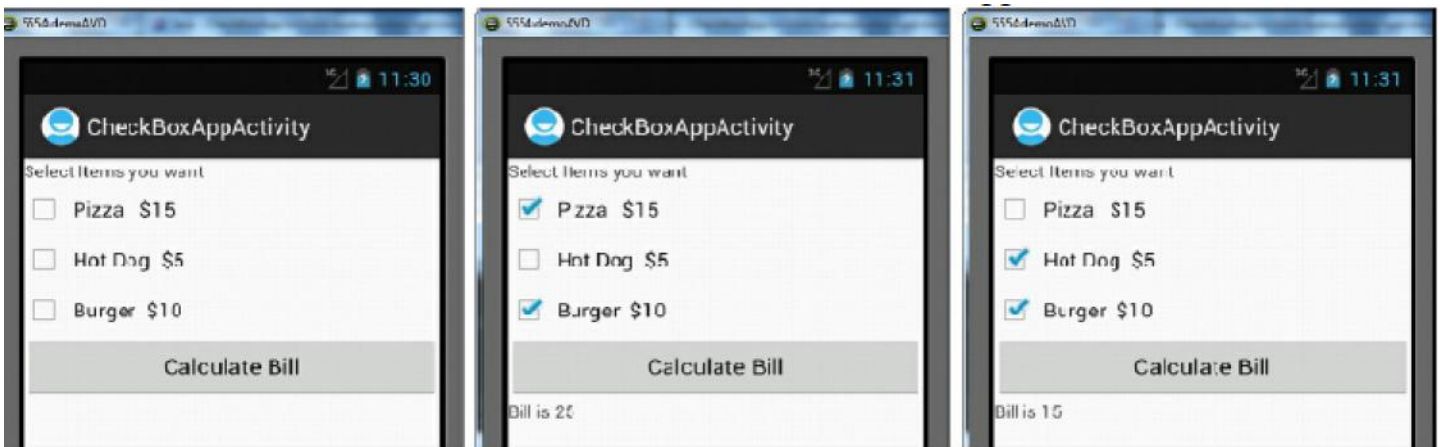
In Java, the following methods can be used with `Checkbox` controls:

- **isChecked()**—Determines whether the check box is checked
- **setChecked()**—Sets or changes the state of the check box
- **toggle()**—Toggles the state of the check box from `checked` to `unchecked` or vice versa.

## Example:

```
<CheckBox
    android:id="@+id/purchase"
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:text="Purchase" />
```





**Figure 2.9. CheckBox controls displayed on application startup (left), price displayed when the**

#### Activity\_main.XML

```
<TextView
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:text="Select Items you want"/>
<CheckBox
android:id="@+id/checkbox_pizza"
android:layout_height="wrap_content"
android:text="Pizza $15"
android:layout_width="match_parent" />
<CheckBox
android:id="@+id/checkbox_hotdog"
android:layout_height="wrap_content"
android:text="Hot Dog $5"
android:layout_width="match_parent" />
<CheckBox
android:id="@+id/checkbox_burger"
android:layout_height="wrap_content"
android:text="Burger $10"
```

```

android:layout_width="match_parent" />
<Button
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:id="@+id/bill_btn"
android:text="Calculate Bill"/>
<TextView
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:id="@+id/amount"/>
</LinearLayout>

```

### MainActivity.java

```

import android.app.Activity;
import android.os.Bundle;
import android.widget.Button;
import android.widget.TextView;
import android.widget.CheckBox;
import android.view.View;
import android.view.View.OnClickListener;
public class CheckBoxAppActivity extends Activity implements
OnClickListener {
    CheckBox c1,c2,c3;
    TextView resp;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_check_box_app);
        Button b = (Button)this.findViewById(R.id.bill_btn);
        resp = (TextView)this.findViewById(R.id.amount);
        c1 = (CheckBox)findViewById(R.id.checkbox_pizza);
        c2 = (CheckBox)findViewById(R.id.checkbox_hotdog);
        c3 = (CheckBox)findViewById(R.id.checkbox_burger);
        b.setOnClickListener(this);
    }
    public void onClick(View v) {

```

```

int amt=0;
if (c1.isChecked()) {
amt=amt+15;
}
if (c2.isChecked()) {
amt=amt+5;
}
if (c3.isChecked()) {
amt=amt+10;
}
resp.setText("Bill is " +Integer.toString(amt));
}
}

```

## Choosing Mutually Exclusive Items Using RadioButtons

`RadioButton` controls are two-state widgets that can be in either a checked or unchecked state. The difference between check boxes and radio buttons is that you can select more than one check box in a set, whereas radio buttons are mutually exclusive—only one radio button can be selected in a group. Selecting a radio button automatically deselects other radio buttons in the group.

The following methods can be applied to `RadioButtons`:

- **isChecked()**—Detects whether the `RadioButton` control is selected.
- **toggle()**—Toggles the state of the `RadioButton` from selected to unselected and vice versa.
- **check()**—Checks a specific `RadioButton` whose ID is supplied in the method.
- **getCheckedRadioButtonId()**—Gets the ID of the currently selected `RadioButton` control.

```

<RadioGroup
android:id="@+id/group_hotel"

```

```

android:layout_width="match_parent"
android:layout_height="wrap_content"
android:orientation="vertical" >
<RadioButton android:id="@+id/radio_fivestar"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Five Star " />
<RadioButton android:id="@+id/radio_threestar"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Three Star" />

```

```

public class RadioButtonAppActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_radio_button_app);
        RadioButton radioFivestar = (RadioButton)
        findViewById(R.id.radio_fivestar);
        RadioButton radioThreestar = (RadioButton)
        findViewById(R.id.radio_threestar);
        radioFivestar.setOnClickListener(listener);
        radioThreestar.setOnClickListener(listener);
    }
    private OnClickListener listener = new
    OnClickListener() {
        public void onClick(View v) {
            TextView selectedHotel = (TextView)
            findViewById(R.id.hoteltype);
            RadioButton rb = (RadioButton) v;
            selectedHotel.setText("The hotel type selected is: "
            +rb.getText());
        }
    };
}

```

## UNIT- 3

### Introduction to Layouts

Layouts are basically containers for other items known as *Views*, which are displayed on the screen. Layouts help manage and arrange views as well. Layouts are defined in the form of XML files that cannot be changed by our code during runtime.

**Table 3.1. Android Layout Managers**

Layout Manager	Description
LinearLayout	Organizes its children either horizontally or vertically
RelativeLayout	Organizes its children relative to one another or to the parent
AbsoluteLayout	Each child control is given a specific location within the bounds of the container
FrameLayout	Displays a single view; that is, the next view replaces the previous view and hence is used to dynamically change the children in the layout
TableLayout	Organizes its children in tabular form
GridLayout	Organizes its children in grid format

## LinearLayout

The LinearLayout is the most basic layout, and it arranges its elements sequentially, either horizontally or vertically. To arrange controls within a linear layout, the following attributes are used:

- **android:orientation**—Used for arranging the controls in the container in horizontal or vertical order
- **android:layout\_width**—Used for defining the width of a control
- **android:layout\_height**—Used for defining the height of a control
- **android:padding**—Used for increasing the whitespace between the boundaries of the control and its actual content
- **android:layout\_weight**—Used for shrinking or expanding the size of the control to consume the extra space relative to the other controls in the container
- **android:gravity**—Used for aligning content within a control
- **android:layout\_gravity**—Used for aligning the control within the container

### Applying the orientation Attribute

The `orientation` attribute is used to arrange its children either in horizontal or vertical order. The valid values for this attribute are `horizontal` and `vertical`. If the value of the `android:orientation` attribute is set to `vertical`, the children in the linear layout are arranged in a column layout, one below the other. Similarly, if the value of the `android:orientation` attribute is set to `horizontal`, the controls in the linear layout are arranged in a row format, side by side.

### Applying the `height` and `width` Attributes

The default height and width of a control are decided on the basis of the text or content that is displayed through it. To specify a certain height and width to the control, we use the `android:layout_width` and `android:layout_height` attributes.

### Applying the `padding` Attribute

The `padding` attribute is used to increase the whitespace between the boundaries of the control and its actual content. Through the `android:padding` attribute, we can set the same amount of padding or spacing on all four sides of the control. Similarly, by using the `android:paddingLeft`, `android:paddingRight`, `android:paddingTop`, and `android:paddingBottom` attributes, we can specify the individual spacing on the left, right, top, and bottom of the control, respectively.

Example program:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical" >
<Button
android:id="@+id/Apple"
android:text="Apple"
android:layout_width="match_parent"
android:layout_height="wrap_content" />
<Button
android:id="@+id/Mango"
android:text="Mango"
android:layout_width="match_parent"
android:layout_height="wrap_content" />
<Button
android:id="@+id/Banana"
android:text="Banana"
android:layout_width="match_parent"
android:layout_height="wrap_content" />
</LinearLayout>
```



### Applying the Gravity Attribute

The Gravity attribute is for aligning the content within a control. For example, to align the text of a control to the center, we set the value of its `android:gravity` attribute to `center`. The valid options for `android:gravity` include `left`, `center`, `right`, `top`, `bottom`, `center_horizontal`, `center_vertical`, `fill_horizontal`, and `fill_vertical`.

### Using the android:layout\_gravity Attribute

Where `android:gravity` is a setting used by the `View`, the `android:layout_gravity` is used by the container. That is, this attribute is used to align the control within the container. For example, to align the text within a `Button` control, we use the `android:gravity` attribute; to align the `Button` control itself in the `LinearLayout` (the container), we use the `android:layout_gravity` attribute.

## RelativeLayout

In `RelativeLayout`, each child element is laid out in relation to other child elements; that is, the location of a child element is specified in terms of the desired distance from the existing children.

### Layout Control Attributes

The attributes used to set the location of the control relative to a container are

- **`android:layout_alignParentTop`**—The top of the control is set to align with the top of the container.
- **`android:layout_alignParentBottom`**—The bottom of the control is set to align with the bottom of the container.
- **`android:layout_alignParentLeft`**—The left side of the control is set to align with the left side of the container.



- **android:layout\_alignParentRight**—The right side of the control is set to align with the right side of the container.
- **android:layout\_centerHorizontal**—The control is placed horizontally at the center of the container.
- **android:layout\_centerVertical**—The control is placed vertically at the center of the container.
- **android:layout\_centerInParent**—The control is placed horizontally and vertically at the center of the container.

The attributes to control the position of a control in relation to other controls are

- **android:layout\_above**—The control is placed above the referenced control.
- **android:layout\_below**—The control is placed below the referenced control.
- **android:layout\_toLeftOf**—The control is placed to the left of the referenced control.
- **android:layout\_toRightOf**—The control is placed to the right of the referenced control.

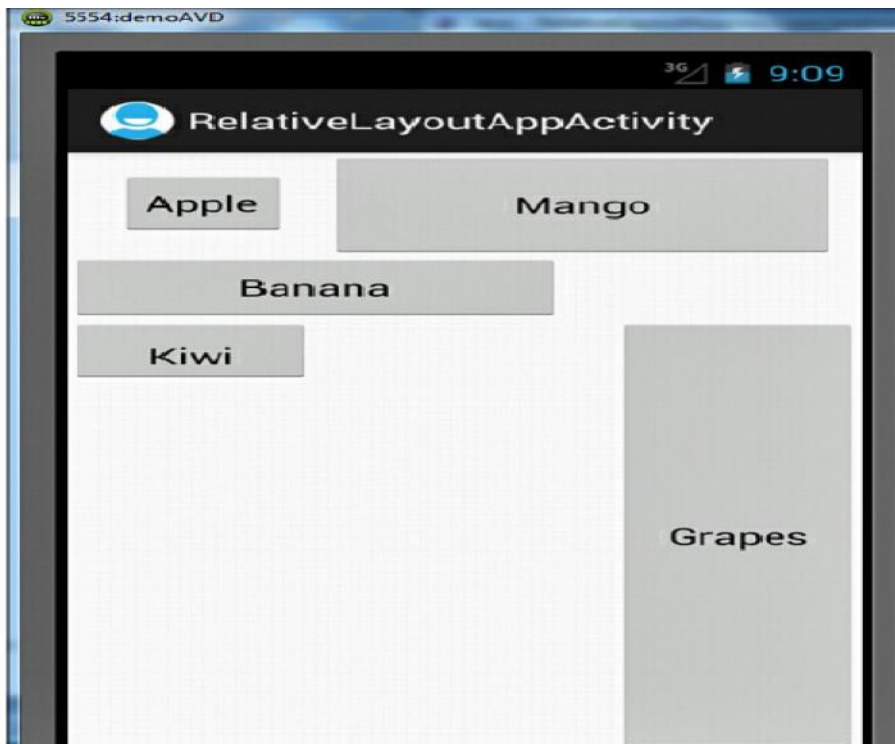
Ex:

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="match_parent"
android:layout_height="match_parent">
<Button
android:id="@+id/Apple"
android:text="Apple"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_marginTop="15dip"
android:layout_marginLeft="20dip" />
<Button
android:id="@+id/Mango"
android:text="Mango"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:padding="28dip"
android:layout_toRightOf="@id/Apple"
android:layout_marginLeft="15dip"
android:layout_marginRight="10dip"
android:layout_alignParentTop="true" />
<Button
android:id="@+id/Banana"
android:text="Banana"
DEPARTMENT OF CSE, SVEU, TIRUPATI
```

```

android:layout_width="200dip"
android:layout_height="50dip"
android:layout_marginTop="15dip"
android:layout_below="@id/Apple"
android:layout_alignParentLeft="true" />
<Button
android:id="@+id/Grapes"
android:text="Grapes"
android:layout_width="wrap_content"
android:layout_height="match_parent"
android:minWidth="100dp"
android:layout_alignParentRight="true"
android:layout_below="@id/Banana" />
<Button
android:id="@+id/Kiwi"
android:text="Kiwi"
android:layout_width="100dip"
android:layout_height="wrap_content"
android:layout_below="@id/Banana"
android:paddingTop="15dip"
android:paddingLeft="25dip"
android:paddingRight="25dip" />
</RelativeLayout>

```



## AbsoluteLayout

Each child in an `AbsoluteLayout` is given a specific location within the bounds of the container. Such fixed locations make `AbsoluteLayout` incompatible with devices of different screen size and resolution. The controls in `AbsoluteLayout` are laid out by specifying their exact X and Y positions. The coordinate 0,0 is the origin and is located at the top-left corner of the screen.

Ex:

```
<AbsoluteLayout>

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="New Product Form"
    android:textSize="20sp"
    android:textStyle="bold"
    android:layout_x="90dip"
    android:layout_y="2dip"/>
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Product Code:"
    android:layout_x="5dip"
    android:layout_y="40dip" />
<EditText
    android:id="@+id/product_code"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:minWidth="100dip"
    android:layout_x="110dip"
    android:layout_y="30dip" />

</AbsoluteLayout>
```

## Using `ImageView`

An `ImageView` control is used to display images in Android applications. An image can be displayed by assigning it to the `ImageView` control and including the `android:src` attribute in the XML definition of the control.

```
<ImageView
    android:id="@+id/first_image"
    android:src="@drawable/bintupic"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:scaleType="fitXY"
```

```

android:adjustViewBounds="true"
android:maxHeight="100dip"
android:maxLength="250dip"
android:minHeight="100dip"
android:minWidth="250dip"
android:resizeMode="horizontal|vertical" />

```

**android:src**—Used to assign the image from drawable resources.

**android:scaleType**—Used to scale an image to fit its container.

**android:resizeMode**—The `resizeMode` attribute is used to make a control resizable so we can resize it horizontally, vertically, or around both axes.

## FrameLayout

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.

FrameLayout is used to display a single `View`.

The `View` added to a FrameLayout is placed at the top left edge of the layout. Any other `View` added to the FrameLayout overlaps the previous `View`; that is, each `View` stacks on top of the previous one. Let's create an application to see how controls can be laid out using FrameLayout.

```

<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>

```

```

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

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

```



## TableLayout

## MOBIEI APPLICATION DEVELOPMENT

Android `TableLayout` going to be arranged groups of 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.

Following are the important attributes specific to `TableLayout` –

Sr.No.	Attribute & Description
1	<code>android:id</code>  This is the ID which uniquely identifies the layout.
2	<code>android:collapseColumns</code>  This specifies the zero-based index of the columns to collapse. The column indices must be separated by a comma: 1, 2, 5.
3	<code>android:shrinkColumns</code>  The zero-based index of the columns to shrink. The column indices must be separated by a comma: 1, 2, 5.
4	<code>android:stretchColumns</code>  The zero-based index of the columns to stretch. The column indices must be separated by a comma: 1, 2, 5.

```
<TableRow  
  android:layout_width="fill_parent"  
  android:layout_height="fill_parent">
```

```
<TextView  
  android:text="Time"  
  android:layout_width="wrap_content"  
  android:layout_height="wrap_content"  
  android:layout_column="1" />
```

```
<TextClock  
  android:layout_width="wrap_content"  
  android:layout_height="wrap_content"  
  android:id="@+id/textClock"  
  android:layout_column="2" />
```

```

</TableRow>

<TableRow>

<TextView
android:text="First Name"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_column="1" />

<EditText
android:width="200px"
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>

<TableRow
android:layout_width="fill_parent"
android:layout_height="fill_parent">

<RatingBar
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:id="@+id/ratingBar"
android:layout_column="2" />
</TableRow>

<TableRow
android:layout_width="fill_parent"
android:layout_height="fill_parent"/>

<TableRow
android:layout_width="fill_parent"
android:layout_height="fill_parent">

<Button
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Submit"
android:id="@+id/button"
android:layout_column="2" />
</TableRow>

</TableLayout>

```

## Adapting to Screen Orientation

As with almost all smartphones, Android supports two screen orientations: portrait and landscape.

When the screen orientation of an Android device is changed, the current activity being displayed is destroyed and re-created automatically to redraw its content in the new orientation.

There are two ways to handle changes in screen orientation:

- **Anchoring controls**—Set the controls to appear at the places relative to the four edges of the screen. When the screen orientation changes, the controls do not disappear but are rearranged relative to the four edges.
- **Defining layout for each mode**—A new layout file is defined for each of the two screen orientations. One has the controls arranged to suit the `Portrait` mode, and the other has the controls arranged to suit the `Landscape` mode.

## Anchoring Controls

For anchoring controls relative to the four edges of the screen, we use a `RelativeLayout` container.

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="match_parent"
android:layout_height="match_parent">
<Button
android:id="@+id/Apple"
android:text="Apple"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_marginTop="15dip"
android:layout_marginLeft="20dip" />
<Button
android:id="@+id/Mango"
android:text="Mango"
android:layout_width="match_parent"
```



```
android:layout_height="wrap_content"
android:padding="28dip"
android:layout_toRightOf="@id/Apple"
android:layout_marginLeft="15dip"
android:layout_marginRight="10dip"
android:layout_alignParentTop="true" />
<Button
android:id="@+id/Banana"
android:text="Banana"
android:layout_width="200dip"
android:layout_height="50dip"
android:layout_marginTop="15dip"
android:layout_below="@id/Apple"
android:layout_alignParentLeft="true" />
<Button
android:id="@+id/Grapes"
android:text="Grapes"
android:layout_width="wrap_content"
android:layout_height="match_parent"
android:minWidth="100dp"
android:layout_alignParentRight="true"
android:layout_below="@id/Banana" />
<Button
android:id="@+id/Kiwi"
android:text="Kiwi"
android:layout_width="100dip"
android:layout_height="wrap_content"
android:layout_below="@id/Banana"
android:paddingTop="15dip"
android:paddingLeft="25dip"
android:paddingRight="25dip" />
</RelativeLayout>
```

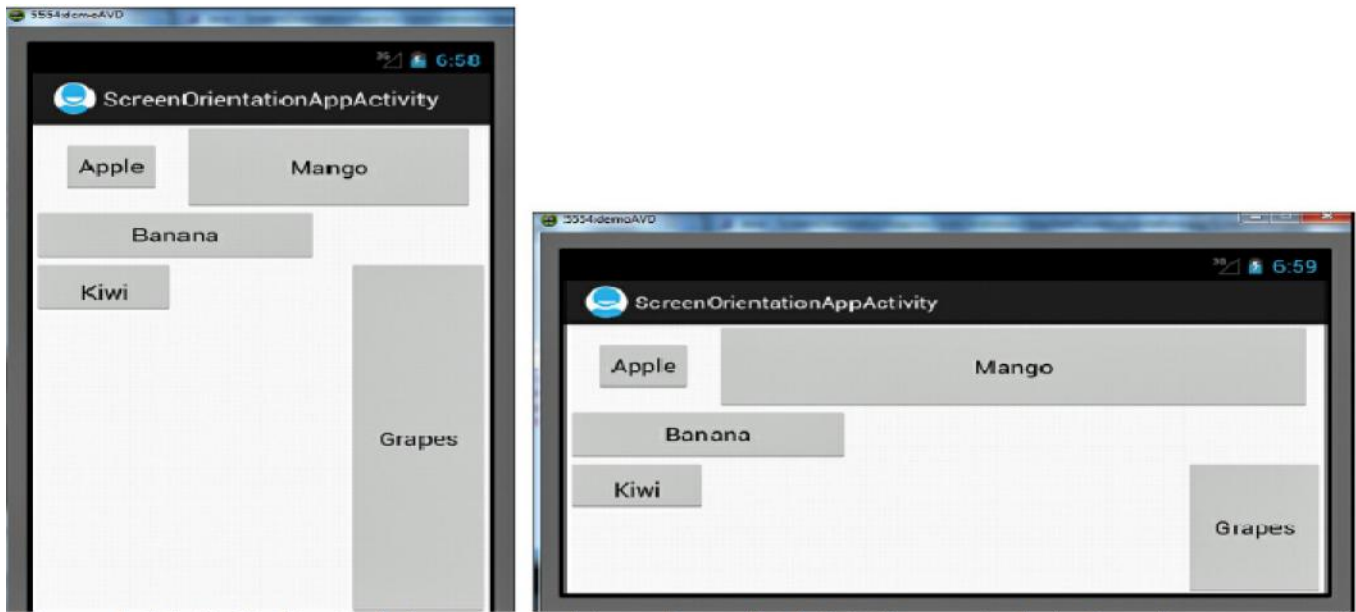


Figure 3.16. (left) Controls in portrait mode, and (right) the controls in landscape mode

## Defining Layout for Each Mode

In this method, we define two layouts. One arranges the controls in the default `portrait` mode, and the other arranges the controls in `landscape` mode.

For portrait mode we use `Linear Layout`, for landscape use `Relative Layout`.

## App resources overview

Resources are the additional files and static content that your code uses, such as bitmaps, layout definitions, user interface strings, animation instructions, and more.

You should always externalize app resources such as images and strings from your code, so that you can maintain them independently. You should also provide alternative resources for specific device configurations, by grouping them in specially-named resource directories. At runtime, Android uses the appropriate resource based on the current configuration. For example, you might want to provide a different UI layout depending on the screen size or different strings depending on the language setting.

Once you externalize your app resources, you can access them using resource IDs that are generated in your project's `R` class. This document shows you how to group your resources in your Android project and provide alternative resources for specific device configurations, and then access them from your app code or other XML files.

## Grouping resource types

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/

graphic.png

layout/

main.xml

info.xml

mipmap/

icon.png

values/

strings.xml

**As you can see in this example, the `res/` directory contains all the resources (in subdirectories): an image resource, two layout resources, `mipmap/` directories for launcher icons, and a string resource file. The resource directory names are important and are described in table 1.**

**Table 1.** Resource directories supported inside project `res/` directory.

Directory	Resource Type
animator/	XML files that define <a href="#">property animations</a> .

anim/	XML files that define <a href="#">tween animations</a> . (Property animations can also be saved in this directory, but the animator/ directory is preferred for property animations to distinguish between the two types.)
color/	XML files that define a state list of colors. See <a href="#">Color State List Resource</a>
drawable/	<p>Bitmap files (.png, .9.png, .jpg, .gif) or XML files that are compiled into the following drawable resource subtypes:</p> <ul style="list-style-type: none"> <li>) Bitmap files</li> <li>) Nine-Patches (re-sizable bitmaps)</li> <li>) State lists</li> <li>) Shapes</li> <li>) Animation drawables</li> <li>) Other drawables</li> </ul> <p>See <a href="#">Drawable Resources</a>.</p>
mipmap/	Drawable files for different launcher icon densities. For more information on managing launcher icons with mipmap/ folders, see <a href="#">Managing Projects Overview</a> .
layout/	XML files that define a user interface layout. See <a href="#">Layout Resource</a> .
menu/	XML files that define app menus, such as an Options Menu, Context Menu, or Sub Menu. See <a href="#">Menu Resource</a> .
raw/	<p>Arbitrary files to save in their raw form. To open these resources with a raw <a href="#">InputStream</a>, call <a href="#">Resources.openRawResource()</a> with the resource ID, which is <code>R.raw.filename</code>.</p> <p>However, if you need access to original file names and file hierarchy, you might consider saving some resources in the assets/ directory (instead of res/raw/). Files in assets/ aren't given a resource ID, so you can read them only using <a href="#">AssetManager</a>.</p>
values/	XML files that contain simple values, such as strings, integers, and colors.

Whereas XML resource files in other res/ subdirectories define a single resource based on the XML filename, files in the values/ directory describe multiple resources. For a file in this directory, each child of the <resources> element defines a single resource. For example, a <string> element creates an R.stringresource and a <color> element creates an R.color resource.

Because each resource is defined with its own XML element, you can name the file whatever you want and place different resource types in one file. However, for clarity, you might want to place unique resource types in different files. For example, here are some filename conventions for resources you can create in this directory:

- ) arrays.xml for resource arrays ([typed arrays](#)).
- ) colors.xml for [color values](#)
- ) dims.xml for [dimension values](#).
- ) strings.xml for [string values](#).
- ) styles.xml for [styles](#).

See [String Resources](#), [Style Resource](#), and [More Resource Types](#).

xml/	Arbitrary XML files that can be read at runtime by calling <a href="#">Resources.getXML()</a> . Various XML configuration files must be saved here, such as a <a href="#">searchable configuration</a> .
font/	Font files with extensions such as .ttf, .otf, or .ttc, or XML files that include a <font-family> element. For more information about fonts as resources, go to <a href="#">Fonts in XML</a> .

## String resources

A string resource provides text strings for your application with optional text styling and formatting. There are three types of resources that can provide your application with strings:

### [String](#)

XML resource that provides a single string.

## String Array

XML resource that provides an array of strings.

## Quantity Strings (Plurals)

XML resource that carries different strings for pluralization.

All strings are capable of applying some styling markup and formatting arguments. For information about styling and formatting strings, see the section about [Formatting and Styling](#).

## String

A single string that can be referenced from the application or from other resource files (such as an XML layout).

**Note:** A string is a simple resource that is referenced using the value provided in the **name** attribute (not the name of the XML file). So, you can combine string resources with other simple resources in the one XML file, under one **<resources>** element.

### file location:

res/values/*filename.xml*

The filename is arbitrary. The **<string>** element's **name** is used as the resource ID.

### compiled resource datatype:

Resource pointer to a [String](#).

### resource reference:

In Java: `R.string.string_name`

In XML: `@string/string_name`

### syntax:

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

```
>text_string</string>
</resources>
```

**elements:**

```
<resources>
```

**Required.** This must be the root node.

No attributes.

```
<string>
```

A string, which can include styling tags. Beware that you must escape apostrophes and quotation marks. For more information about how to properly style and format your strings see [Formatting and Styling](#), below.

attributes:

```
name
```

*String.* A name for the string. This name is used as the resource ID.

**example:**

XML file saved at res/values/strings.xml:

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

This layout XML applies a string to a View:

```
<TextView
  android:layout_width="fill_parent"
```

```
android:layout_height="wrap_content"
android:text="@string/hello"/>
```

This application code retrieves a string:

```
Stringstring=getString(R.string.hello);
```

You can use either [getString\(int\)](#) or [getText\(int\)](#) to retrieve a string. [getText\(int\)](#) retains any rich text styling applied to the string.

## String array

An array of strings that can be referenced from the application.

**Note:** A string array is a simple resource that is referenced using the value provided in the `name` attribute (not the name of the XML file). As such, you can combine string array resources with other simple resources in the one XML file, under one `<resources>` element.

### file location:

```
res/values/filename.xml
```

The filename is arbitrary. The `<string-array>` element's `name` is used as the resource ID.

### compiled resource datatype:

Resource pointer to an array of [Strings](#).

### resource reference:

In Java: `R.array.string_array_name`

### syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array
    name="string_array_name">
```



```

<item
    >text_string</item>
</string-array>
</resources>

```

**elements:**

<resources>

**Required.** This must be the root node.

No attributes.

<string-array>

Defines an array of strings. Contains one or more <item> elements.

attributes:

name

*String.* A name for the array. This name is used as the resource ID to reference the array.

<item>

A string, which can include styling tags. The value can be a reference to another string resource. Must be a child of a <string-array> element. Beware that you must escape apostrophes and quotation marks. See [Formatting and Styling](#), below, for information about to properly style and format your strings.

No attributes.

**example:**

XML file saved at res/values/strings.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="planets_array">
        <item>Mercury</item>
        <item>Venus</item>
        <item>Earth</item>
        <item>Mars</item>
    </string-array>
</resources>
```

This application code retrieves a string array:

```
Resources res = getResources();
String[] planets = res.getStringArray(R.array.planets_array);
```

## Quantity strings (plurals)

Different languages have different rules for grammatical agreement with quantity. In English, for example, the quantity 1 is a special case. We write "1 book", but for any other quantity we'd write "*n* books". This distinction between singular and plural is very common, but other languages make finer distinctions. The full set supported by Android is `zero`, `one`, `two`, `few`, `many`, and `other`.

The rules for deciding which case to use for a given language and quantity can be very complex, so Android provides you with methods such as `getQuantityString()` to select the appropriate resource for you.

Although historically called "quantity strings" (and still called that in API), quantity strings should *only* be used for plurals. It would be a mistake to use quantity strings to implement something like Gmail's "Inbox" versus "Inbox (12)" when there are unread messages, for example. It might seem convenient to use quantity strings instead of an `if` statement, but it's important to note that some languages (such as Chinese) don't make these grammatical distinctions at all, so you'll always get the `other` string.

## Drawable resources

A drawable resource is a general concept for a graphic that can be drawn to the screen and which you can retrieve with APIs such as [getDrawable\(int\)](#) or apply to another XML resource with attributes such as `android:drawable` and `android:icon`. There are several different types of drawables:

### Bitmap File

A bitmap graphic file (.png, .jpg, or .gif). Creates a [BitmapDrawable](#).

### Nine-Patch File

A PNG file with stretchable regions to allow image resizing based on content (.9.png). Creates a [NinePatchDrawable](#).

### Layer List

A Drawable that manages an array of other Drawables. These are drawn in array order, so the element with the largest index is drawn on top. Creates a [LayerDrawable](#).

### State List

An XML file that references different bitmap graphics for different states (for example, to use a different image when a button is pressed). Creates a [StateListDrawable](#).

### Level List

An XML file that defines a drawable that manages a number of alternate Drawables, each assigned a maximum numerical value. Creates a [LevelListDrawable](#).

### Transition Drawable

An XML file that defines a drawable that can cross-fade between two drawable resources. Creates a [TransitionDrawable](#).

### Inset Drawable

An XML file that defines a drawable that insets another drawable by a specified distance. This is useful when a View needs a background drawable that is smaller than the View's actual bounds.

### Clip Drawable

An XML file that defines a drawable that clips another Drawable based on this Drawable's current level value. Creates a [ClipDrawable](#).

### Scale Drawable

An XML file that defines a drawable that changes the size of another Drawable based on its current level value. Creates a [ScaleDrawable](#)

### Shape Drawable

An XML file that defines a geometric shape, including colors and gradients. Creates a [GradientDrawable](#).

## Toggle Buttons

A toggle button allows the user to change a setting between two states.

You can add a basic toggle button to your layout with the [ToggleButton](#) object. Android 4.0 (API level 14) introduces another kind of toggle button called a switch that provides a slider control, which you can add with a [Switch](#) object. [SwitchCompat](#) is a version of the Switch widget which runs on devices back to API 7.

If you need to change a button's state yourself, you can use the [CompoundButton.setChecked\(\)](#) or [CompoundButton.toggle\(\)](#) method.



*Toggle buttons*



Key classes are the following:

- ) [ToggleButton](#)
- ) [Switch](#)
- ) [SwitchCompat](#)
- ) [CompoundButton](#)

## Responding to Button Presses

To detect when the user activates the button or switch, create

an [CompoundButton.OnCheckedChangeListener](#) object and assign it to the button by

calling [setOnCheckedChangeListener\(\)](#).

### For example:

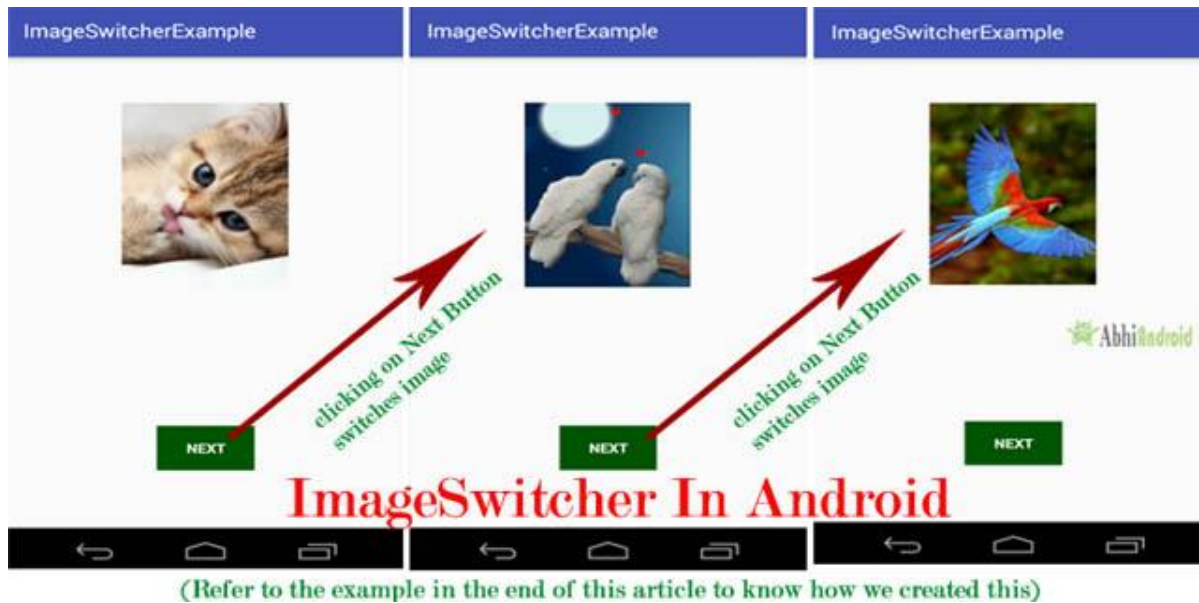
---

```
ToggleButton toggle =(ToggleButton) findViewById(R.id.togglebutton);
toggle.setOnCheckedChangeListener(newCompoundButton.OnCheckedChangeListener(){
    publicvoid onCheckedChanged(CompoundButton buttonView,boolean isChecked){
        if(isChecked){
            // The toggle is enabled
        }else{
            // The toggle is disabled
        }
    }
});
```

## ImageSwitcher Tutorial with Example in Android Studio

In Android, [ImageSwitcher](#) is a specialized [ViewSwitcher](#) that contain [ImageView](#) type children. ImageSwitcher is available in Android from v1.6+.

It is an element of transition widget which helps us to add transitions on the images. It is mainly useful to animate an image on screen. ImageSwitcher switches smoothly between two images and thus provides a way of transitioning from one Image to another through appropriate animations.



*Basic ImageSwitcher code in [XML](#)*

```
<ImageSwitcher
    android:id="@+id/simpleImageSwitcher"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
```

### *Steps for Implementation of ImageSwitcher:*

1. Get the reference of ImageSwitcher in class using findViewById() method, or you can also create an object dynamically.
2. Set a factory using switcherid.setFactory()
3. Set an in-animation using switcherid.setInAnimation()
4. Set an out-animation using switcherid.setOutAnimation()

```

ImageSwitcher simpleImageSwitcher=(ImageSwitcher)findViewById(R.id. simpleImageSwitcher);// get
reference of ImageSwitcher

// Set the ViewFactory of the ImageSwitcher that will create ImageView object when asked
imageSwitcher.setFactory(new ViewSwitcher.ViewFactory(){

    public View makeView(){

        // TODO Auto-generated method stub

        // Create a new ImageView and set it's properties
        ImageView imageView =new ImageView(getApplicationContext());
        imageView.setScaleType(ImageView.ScaleType.FIT_CENTER);
        imageView.setLayoutParams(new ImageSwitcher.LayoutParams(LinearLayout.LayoutParams.FILL_PARENT,
        LinearLayout.LayoutParams.FILL_PARENT));

        return imageView;

    }

});

```

## Working with the ScrollView

When an app has layout content that might be longer than the height of the device and that content should be vertically scrollable, then we need to use a [ScrollView](#).

The **android.widget.ScrollView** class provides the functionality of scroll view. ScrollView is used to scroll the child elements of palette inside ScrollView. Android supports vertical scroll view as default scroll view. Vertical ScrollView scrolls elements vertically.

Android uses *HorizontalScrollView* for horizontal ScrollView.

1. **<TextView**

```

2.     android:layout_width="wrap_content"
3.     android:layout_height="wrap_content"
4.     android:textAppearance="?android:attr/textAppearanceMedium"
5.     android:text="Vertical ScrollView example"
6.     android:id="@+id/textView"
7.     android:layout_gravity="center_horizontal"
8.     android:layout_centerHorizontal="true"
9.     android:layout_alignParentTop="true" />

```

10.

11.

12. **<ScrollView** android:layout\_marginTop="30dp"

```

13.     android:layout_width="fill_parent"
14.     android:layout_height="wrap_content"
15.     android:id="@+id/scrollView">

```

16.

17.

18. **<LinearLayout**

```

19.     android:layout_width="fill_parent"
20.     android:layout_height="fill_parent"
21.     android:orientation="vertical" >

```

22.

23. **<Button**

```

24.     android:layout_width="fill_parent"
25.     android:layout_height="wrap_content"
26.     android:text="Button 1" />

```



```
27. <Button
28.     android:layout_width="fill_parent"
29.     android:layout_height="wrap_content"
30.     android:text="Button 2" />
31. <Button
32.     android:layout_width="fill_parent"
33.     android:layout_height="wrap_content"
34.     android:text="Button 3" />
35. <Button
36.     android:layout_width="fill_parent"
37.     android:layout_height="wrap_content"
38.     android:text="Button 4" />
39. <Button
40.     android:layout_width="fill_parent"
41.     android:layout_height="wrap_content"
42.     android:text="Button 5" />
43. <Button
44.     android:layout_width="fill_parent"
45.     android:layout_height="wrap_content"
46.     android:text="Button 6" />
47. <Button
48.     android:layout_width="fill_parent"
49.     android:layout_height="wrap_content"
50.     android:text="Button 7" />
51. <Button
52.     android:layout_width="fill_parent"
53.     android:layout_height="wrap_content"
```

```

54.         android:text="Button 8" />
55.     <Button
56.         android:layout_width="fill_parent"
57.         android:layout_height="wrap_content"
58.         android:text="Button 9" />
59.     <Button
60.         android:layout_width="fill_parent"
61.         android:layout_height="wrap_content"
62.         android:text="Button 10" />
63. </TextView>

```

## Playing Audio

**Android Media Player Example** We can play and control the audio files in android by the help of **MediaPlayer class**. Here, we are going to see a simple example to play the audio file. In the next page, we will see the example to control the audio playback like start, stop, pause etc.

### MediaPlayer class

The **android.media.MediaPlayer** class is used to control the audio or video files.

```
package com.example.audioplay;
```

```
import android.media.MediaPlayer;
```

```
import android.os.Bundle;
```

```
import android.os.Environment;
```

```
import android.app.Activity;
```

```
import android.view.Menu;
```

```
import android.view.View;
```

```

import android.view.View.OnClickListener;

import android.widget.Button;

public class MainActivity extends Activity {

    Button start,pause,stop;

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);


        start=(Button)findViewById(R.id.button1);
        pause=(Button)findViewById(R.id.button2);
        stop=(Button)findViewById(R.id.button3);


        final MediaPlayer mp=new MediaPlayer();
        try{
            mp.setDataSource(Environment.getExternalStorageDirectory().getPath()+"/Music/main.mp3");


            mp.prepare();
        }catch(Exception e){e.printStackTrace();}


        start.setOnClickListener(new OnClickListener() {

            @Override

            public void onClick(View v) {

                mp.start();

            }

        });
    }
}

```

```

        pause.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                mp.pause();
            }
        });
        stop.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                mp.stop();
            }
        });
    }
}

```

## Playing Video

By the help of **MediaController** and **VideoView** classes, we can play the video files in android.

### *MediaController class*

The **android.widget.MediaController** is a view that contains media controls like play/pause, previous, next, fast-forward, rewind etc.

### *VideoView class*

The **android.widget.VideoView** class provides methods to play and control the video player.

```
package com.example.video1;
```

```
import android.net.Uri;
```

```
import android.os.Bundle;
```

```

import android.app.Activity;

import android.view.Menu;

import android.widget.MediaController;

import android.widget.VideoView;


public class MainActivity extends Activity {

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);


        VideoView videoView =(VideoView)findViewById(R.id.videoView1);


        MediaController mediaController= new MediaController(this);

        mediaController.setAnchorView(videoView);


        Uri uri=Uri.parse("/sdcard/media/1.mp4");

        videoView.setMediaController(mediaController);

        videoView.setVideoURI(uri);

        videoView.requestFocus();

        videoView.start();

    }


    @Override

    public boolean onCreateOptionsMenu(Menu menu) {

```

```

        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.activity_main, menu);
        return true;
    }

}

```

## Android ProgressBar

We can display the **android progress bar** dialog box to display the status of work being done e.g. downloading file, analyzing status of work etc.

In this example, we are displaying the progress dialog for dummy file download operation.

Here we are using **android.app.ProgressDialog** class to show the progress bar. Android ProgressDialog is the subclass of AlertDialog class.

The **ProgressDialog** class provides methods to work on progress bar like `setProgress()`, `setMessage()`, `setProgressStyle()`, `setMax()`, `show()` etc. The progress range of Progress Dialog is 0 to 10000.

Let's see a simple example to display progress bar in android.

1. `ProgressDialog progressBar = new ProgressDialog(this);`
2. `progressBar.setCancelable(true);` //you can cancel it by pressing back button
3. `progressBar.setMessage("File downloading ...");`
4. `progressBar.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);`
5. `progressBar.setProgress(0);` //initially progress is 0
6. `progressBar.setMax(100);` //sets the maximum value 100
7. `progressBar.show();` //displays the progress bar

<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" >
<TextView
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:gravity="center"
android:id="@+id/response"/>
<ToggleButton android:id="@+id/playstop_btn"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:textOn=""
android:textOff=""
android:layout_gravity="center"
android:background="@drawable/play" />
<ProgressBar android:id="@+id/progressbar"
android:layout_width="match_parent"
android:layout_height="wrap_content"
style="@android:style/Widget.ProgressBar.Horizontal"
android:layout_marginTop="20dip" />
</LinearLayout>

```

### MainActivity.java

```

public class ProgressBarAppActivity extends Activity {
MediaPlayer mp;

```

```

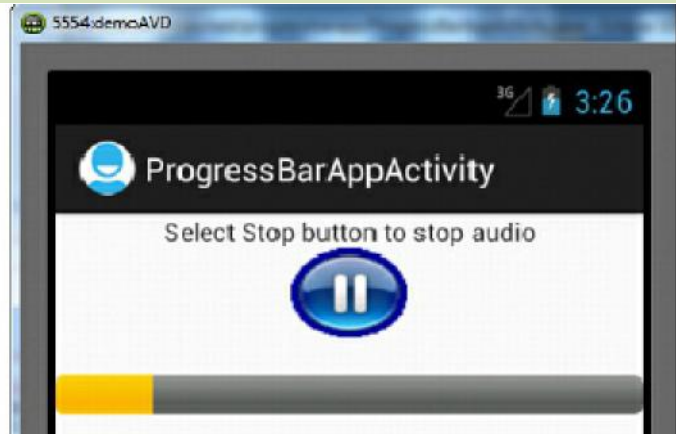
ProgressBar progressBar;
private final Handler handler = new Handler();
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_progress_bar_app);
    final TextView response =
        (TextView)findViewById(R.id.response);
    response.setText("Select Play button to play audio");
    progressBar=(ProgressBar)findViewById(R.id.progressbar);
    mp =
        MediaPlayer.create(ProgressBarAppActivity.this,R.raw.song1);
    final ToggleButton playStopButton = (ToggleButton)
        findViewById(R.id.playstop_btn);
    progressBar.setProgress(0);
    progressBar.setMax(mp.getDuration());
    playStopButton.setOnClickListener(new OnClickListener()
    {
        public void onClick(View v) {
            if (playStopButton.isChecked()) {
                response.setText("Select Stop button to stop
                audio");
                playStopButton.setBackgroundDrawable(
                    getResources().getDrawable(R.drawable.stop));
                mp.start();
                updateProgressBar();
            }
            else {
                response.setText("Select Play button to play
                audio");
                playStopButton.setBackgroundDrawable(
                    getResources().getDrawable(R.drawable.play));
                mp.pause();
            }
        }
    });
}

private void updateProgressBar() {
    progressBar.setProgress(mp.getCurrentPosition());
    if (mp.isPlaying()) {
        Runnable notification = new Runnable() {
            public void run() {

```



```
updateProgressBar();  
}  
};  
handler.postDelayed(notification,1000);  
}  
}  
}
```



## UNIT-4

### Using 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 holds the data and send the data to adapter view, the view can takes the data from adapter view and shows the data on different views like as spinner, list view,gridviewetc.



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 ( i.e. ListView or GridView). The common adapters are **ArrayAdapter**,**BaseAdapter**,**CursorAdapter**, **SimpleCursorAdapter**,**SpinnerAdapter** and **WrapperListAdapter**. We will see separate examples for both the adapters.

## ListView Attributes

Following are the important attributes specific to GridView –

Sr.No	Attribute & Description
1	<b>android:id</b> This is the ID which uniquely identifies the layout.
2	<b>android:divider</b> This is drawable or color to draw between list items.
3	<b>android:dividerHeight</b> This specifies height of the divider. This could be in px, dp, sp, in, or mm.
4	<b>android:entries</b> Specifies the reference to an array resource that will populate the ListView.
5	<b>android:footerDividersEnabled</b> When set to false, the ListView will not draw the divider before each footer view. The default value is true.
6	<b>android:headerDividersEnabled</b> 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 =newArrayAdapter<String>(this,R.layout.ListView,StringArray);
```

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

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

```

import android.os.Bundle;

import android.app.Activity;

import android.view.Menu;

import android.widget.AdapterView;

import android.widget.ListView;


public class ListDisplay extends Activity {

    // Array of strings...

    String[] mobileArray = { "Android", "iPhone", "Windows Mobile", "Blackberry",
        "WebOS", "Ubuntu", "Windows7", "Max OS X" };


    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);


        ArrayAdapter adapter = new ArrayAdapter<String>(this,

            R.layout.activity_listview, mobileArray);


        ListView listView = (ListView) findViewById(R.id.mobile_list);

        listView.setAdapter(adapter);

    }

}

```

## Using spinner control

Spinner allows you to select an item from a drop down menu. For example, when you are using the Gmail application you would get a drop down menu as shown below, you need to select an item from a drop down menu.



This example demonstrates the category of computers, you need to select a category from the category.

```
import java.util.ArrayList;
import java.util.List;
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ArrayAdapter;
import android.widget.Spinner;
import android.widget.Toast;

class AndroidSpinnerExampleActivity extends Activity implements AdapterView.OnItemClickListener {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // Spinner element
        Spinner spinner = (Spinner) findViewById(R.id.spinner);
```

```
// Spinner click listener
```

```
spinner.setOnItemSelectedListener(this);
```

```
// Spinner Drop down elements
```

```
List<String> categories =newArrayList<String>();
```

```
categories.add("Automobile");
```

```
categories.add("Business Services");
```

```
categories.add("Computers");
```

```
categories.add("Education");
```

```
categories.add("Personal");
```

```
categories.add("Travel");
```

```
// Creating adapter for spinner
```

```
ArrayAdapter<String> dataAdapter =newArrayAdapter<String>(this, android.R.layout.simple_spinner_item, categories);
```

```
// Drop down layout style - list view with radio button
```

```
dataAdapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
```

```
// attaching data adapter to spinner
```

```
spinner.setAdapter(dataAdapter);
```

```
}
```

```
publicvoid onItemSelected(AdapterView<?> parent,View view,int position,long id){
```

```
// On selecting a spinner item
```

```
String item = parent.getItemAtPosition(position).toString();
```

```
// Showing selected spinner item
```

```
Toast.makeText(parent.getContext(),"Selected: "+ item,Toast.LENGTH_LONG).show();
```

```
<Spinner
```

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

```
android:layout_width="fill_parent"
```

```
android:layout_height="wrap_content"
```

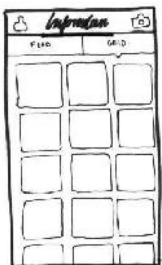
```
android:prompt="@string/spinner_title"/>
```

```
<Spinner
    android:id="@+id/spinner"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:prompt="@string/spinner_title"/>
```

```
<Spinner
    android:id="@+id/spinner"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:prompt="@string/spinner_title"/>
```

## Using Grid control view:

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 –

Sr.No	Attribute & Description

1	<b>android:id</b>  This is the ID which uniquely identifies the layout.
2	<b>android:columnWidth</b>  This specifies the fixed width for each column. This could be in px, dp, sp, in, or mm.
3	<b>android:gravity</b>  Specifies the gravity within each cell. Possible values are top, bottom, left, right, center, center_vertical, center_horizontal etc.
4	<b>android:horizontalSpacing</b>  Defines the default horizontal spacing between columns. This could be in px, dp, sp, in, or mm.
5	<b>android:numColumns</b>  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.
6	<b>android:stretchMode</b>  Defines how columns should stretch to fill the available empty space, if any. This must be either of the values – <ul style="list-style-type: none"> <li>) none – Stretching is disabled.</li> <li>) spacingWidth – The spacing between each column is stretched.</li> <li>) columnWidth – Each column is stretched equally.</li> <li>) spacingWidthUniform – The spacing between each column is uniformly stretched..</li> </ul>
7	<b>android:verticalSpacing</b>  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 Android studio IDE to create an Android application and name it as <i>HelloWorld</i> under a package <i>com.example.helloworld</i> as explained in the <i>Hello World Example</i> chapter.
2	Modify the default content of <i>res/layout/activity_main.xml</i> file to include GridView content with the self explanatory attributes.
3	No need to change string.xml, Android studio takes care of defaults strings which are placed at string.xml
4	Let's put few pictures in <i>res/drawable-hdpi</i> 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 <b>ImageAdapter</b> under a package <i>com.example.helloworld</i> that extends <i>BaseAdapter</i> . This class will implement functionality of an adapter 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 application.

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));
    }
}

```

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 **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;

    }

    // 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);

```


```

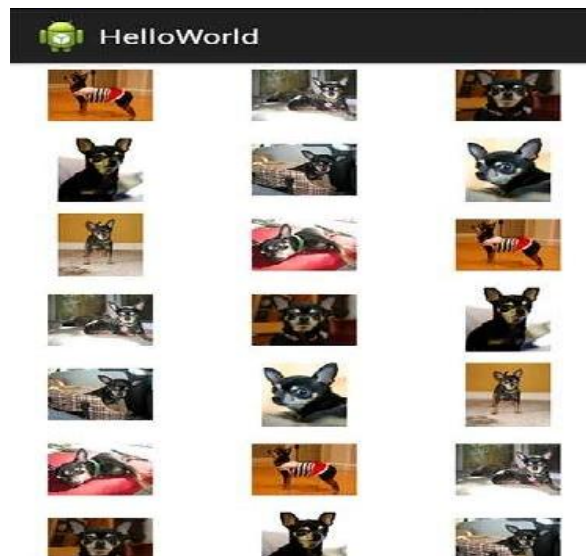
        imageView.setPadding(8,8,8,8);
    }
else
{
    imageView =(ImageView) convertView;

    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 Android Studio, open one of your project's activity files and click Run  icon from the toolbar. Android studio 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 –



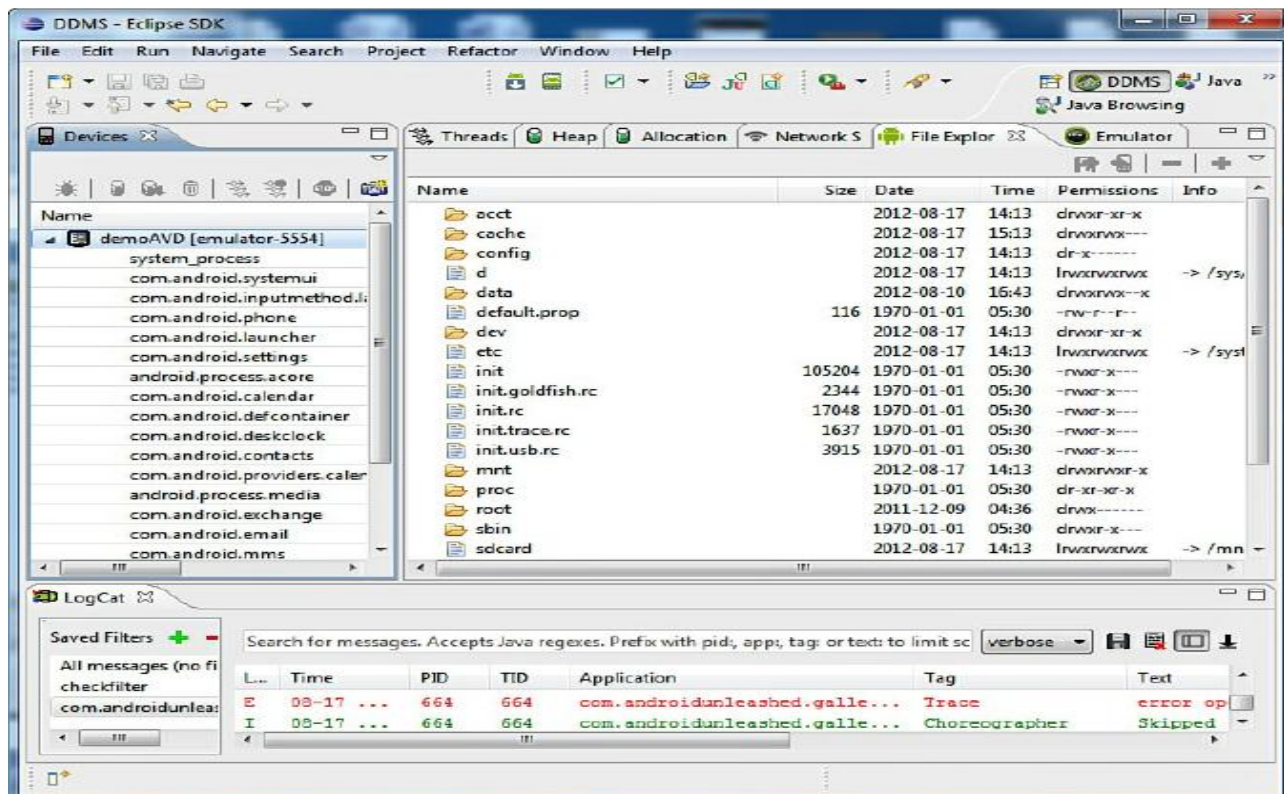
## Using the Debugging Tool: Dalvik Debug Monitor Service (DDMS)

The DDMS is a powerful debugging tool that is downloaded as part of the Android SDK. DDMS can be run either by selecting the DDMS icon on the top-right corner of the Eclipse IDE or by selecting the Window, Open Perspective, DDMS option.

When we run DDMS, it automatically connects to the attached Android device or any running emulator.

DDMS helps with a variety of tasks, including

- Finding bugs in applications running either on an emulator or on the physical device.
- Providing several services such as port forwarding, on-device screen capture, incoming call, SMS, and location data spoofing.
- Showing the status of active processes, viewing the stack and heap, viewing the status of active threads, and exploring the file system of any active emulator.
- Providing the logs generated by LogCat, so we can see log messages about the state of the application and the device. LogCat displays the line number on which the error(s) occurred.
- Simulating different types of networks, such as GPRS and EDGE.



### The DDMS tool window

In the upper-left pane of the DDMS window, we see a **Devices** tab that displays the list of Android devices connected to your PC, along with the running AVDs (if any).

**Debug**—Used to debug the selected process.

- **Update Heap**—Enables heap information of the process. After clicking this icon, use the **Heap** icon on the right pane to get heap information.
- **Dump HPROF file**—Shows the HPROF file that can be used for detecting memory leaks.
- **Cause GC**—Invokes Garbage Collection.
- **Update Threads**—Enables fetching the thread information of the selected process. After clicking this icon, we need to click the **Threads** icon in the right pane to display information about the threads that are created and destroyed in the selected process.
- **Start Method Profiling**—Used to find the number of times different methods are called in an application and the time consumed in each of them. Click the **Start Method Profiling** icon, interact with the application, and click the **Stop Method Profiling** icon to obtain information related to the different methods called in the application.
- **Stop Process**—Stops the selected process.

- **Screen Capture**—Captures our device/emulator screen.

Back to DDMS, on the right pane (refer to Above [Figure](#) ), we find the following tabs:

- **Threads**—Displays information about the threads within each process,
- **Thread ID**—Displays the unique ID assigned to each thread
- **Status**—Displays the current status of the thread—whether it is in running, sleeping, starting, waiting, native, monitor, or zombie state
- **utime**—Indicates the cumulative time spent executing user code
- **stime**—Indicates the cumulative time spent executing system code
- **Name**—Displays the name of the thread
- **Heap**—Displays the heap information of the process (provided the `Update Heap` button from the `Devices` tab has been clicked).

**Allocation Tracker**—Tracks the objects allocated to an application.

**File Explorer**—Displays the file system on the device.

## Debugging Applications

The two most common ways of debugging an application and finding out what went wrong are placing breakpoints and displaying log messages.

### Placing Breakpoints in an Application

Breakpoints are used to temporarily pause the execution of the application, allowing us to examine the content of variables and objects. To place a breakpoint in an application, select the line of code where you want to place a breakpoint and either press `Ctrl+Shift+B`, select `Run, Toggle Breakpoint`, or double-click in the marker bar to the left of the line in the Eclipse code editor.

```
import android.util.Log;
public class HelloWorldAppActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello_world_app);
        TextView msg = (TextView)findViewById(R.id.message);
        msg.setText("Hello World!");
        int a,b,c;
        a=10;
```



```

b=5;
c=a*b;
Log.v("CheckValue1", "a = " + a);
Log.v("CheckValue2", "b = " + b);
Log.v("CheckValue3", "c = " + c);
Log.i("InfoTag", "Program is working correctly up till
here");
Log.e("ErrorTag", "Error--Some error has occurred
here");
}
}

```

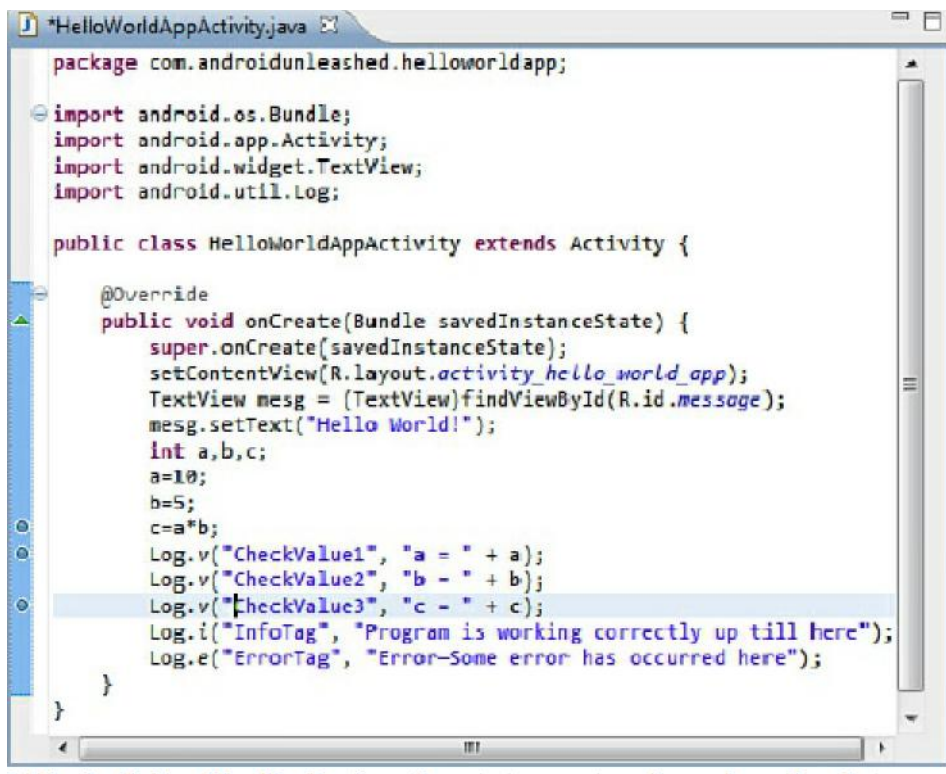
Let's place breakpoints at the following three statements in the activity file:

```

c=a*b;
Log.v("CheckValue1", "a = " + a);
Log.v("CheckValue3", "c = " + c);

```

When we place these breakpoints, a blue dot appears on the left, indicating that the breakpoints were successfully inserted



## What Are Dialogs?

We usually create a new activity or screen for interacting with users, but when we want only a little information, or want to display an essential message, dialogs are preferred. Dialogs are also used to



guide users in providing requested information, confirming certain actions, and displaying warnings or error messages.

The following is an outline of different dialog window types provided by the Android SDK:

- **Dialog**—The basic class for all dialog types.
- **AlertDialog**—A dialog with one, two, or three `Button` controls.
- **CharacterPickerDialog**—A dialog that enables you to select an accented character associated with a regular character source.
- **DatePickerDialog**—A dialog that enables you to set and select a date with a `DatePickerControl`.
- **ProgressDialog**—A dialog that displays a `ProgressBar` control showing the progress of a designated operation.
- **TimePickerDialog**—A dialog that enables you to set and select a time with a `TimePickerControl`.

A dialog is created by creating an instance of the `Dialog` class. The `Dialog` class creates a dialog in the form of a floating window containing messages and controls for user interaction.

The following is a list of the Activity class dialog methods:

- **showDialog()**—Displays a dialog and creates a dialog if one does not exist. Each dialog has a special dialog identifier that is passed to this method as a parameter.
- **onCreateDialog()**—The callback method that executes when the dialog is created for the first time. It returns the dialog of the specified type.
- **onPrepareDialog()**—The callback method used for updating a dialog.
- **dismissDialog()**—Closes the dialog whose dialog identifier is supplied to this method.

The dialog can be displayed again through the `showDialog()` method.

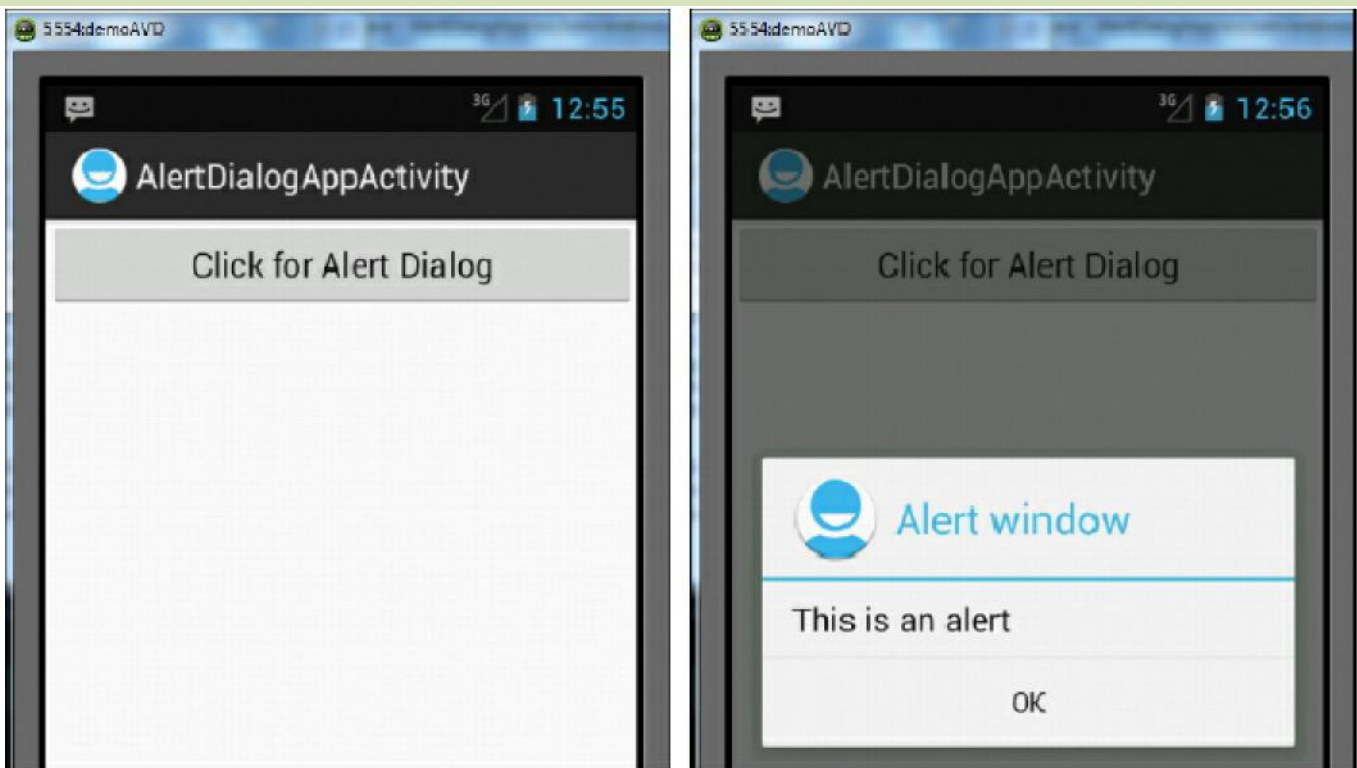
- **removeDialog()**—The `dismissDialog()` method doesn't destroy a dialog. The dismissed dialog can be redisplayed from the cache.

```
import android.app.AlertDialog;
import android.content.DialogInterface;
public class AlertDialogAppActivity extends Activity implements
OnClickListener {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_alert_dialog_app);
        Button b = (Button)this.findViewById(R.id.click_btn);
        b.setOnClickListener(this);
    }
}
```

```

}
@Override
public void onClick(View v) {
    AlertDialog.Builder alertDialog = new
    AlertDialog.Builder(this);
    alertDialog.setTitle("Alert window");
    alertDialog.setIcon(R.drawable.ic_launcher);
    alertDialog.setMessage("This is an alert");
    alertDialog.setPositiveButton("OK", new
    DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int
        buttonId) {
            return;
        }
    });
    alertDialog.show();
}
}

```



## Selecting the Date and Time in One Application

To see how the system date and time can be set in an application, let's create a new Android application and name it DateTimePickerApp. In this application, we use a

TextView and two Button controls. The TextView control displays the current system date and time, and the two Button controls, Set Date and Set Time, are used to invoke the respective dialogs. When the Set Date button is selected, the DatePickerDialog is invoked, and when the Set Time button is selected, the TimePickerDialog is invoked.

```
<LinearLayout>
<TextView android:id="@+id/datetimevw"
android:layout_width="wrap_content"
android:layout_height="wrap_content" />

<Button android:id="@+id/date_button"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:text="Set Date" />

<Button android:id="@+id/time_button"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:text="Set Time" />
</LinearLayout>
```

After defining the controls in the layout file, we write Java code into the

DateTimePickerAppActivity.java activity file to perform the following tasks:

- Display the current system date and time in the TextView control.
- Invoke DatePickerDialog and TimePickerDialog when the Set Date and Set Time Button controls are clicked.
- Initialize DatePickerDialog and TimePickerDialog to display the current system date and time via the Calendar instance.
- Display the modified date and time set by the user via the DatePickerDialog and TimePickerDialog through the TextView control.

Ex:

```
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_date_time_picker_app);
dateTimeView = (TextView) findViewById(R.id.datetimevw);
Button timeButton = (Button)
findViewById(R.id.time_button);
Button dateButton = (Button)
```

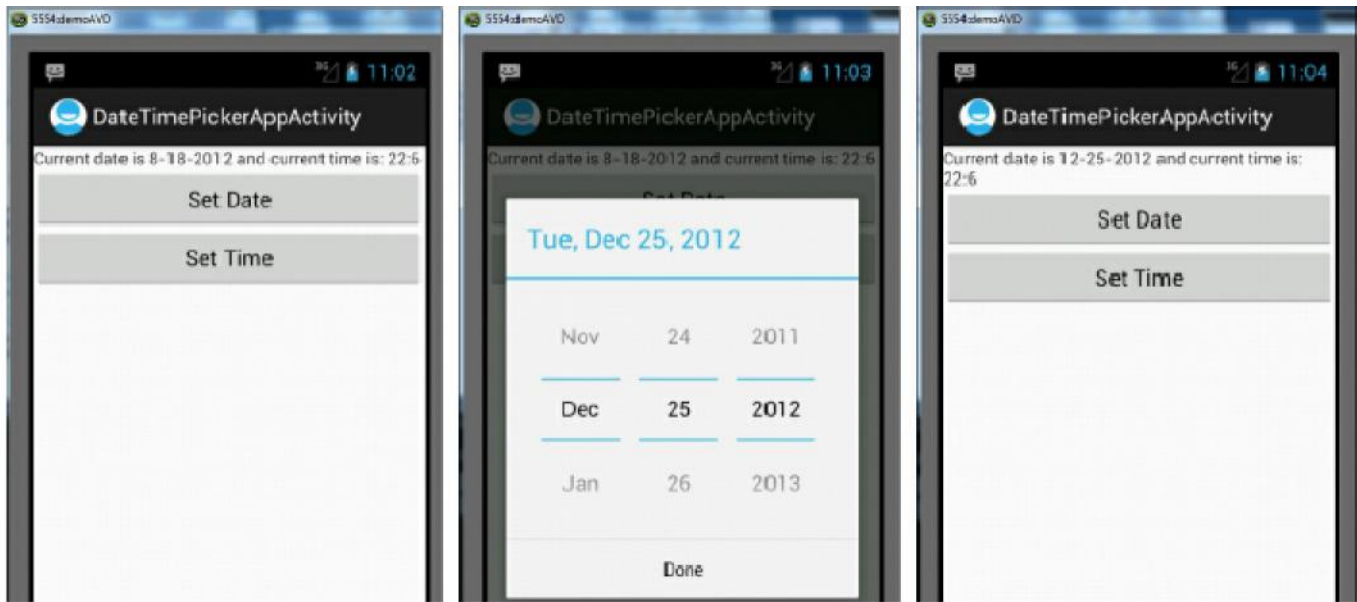
```

findViewById(R.id.date_button);
c = Calendar.getInstance();
h = c.get(Calendar.HOUR_OF_DAY);
m = c.get(Calendar.MINUTE);
yr = c.get(Calendar.YEAR);
mon = c.get(Calendar.MONTH);
dy = c.get(Calendar.DAY_OF_MONTH);
dateTimeView.setText("Current date is "+ (mon+1)+"-"+dy+"-"+yr+" and currenttime is: "+h+": "+m);
dateButton.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        new DatePickerDialog(DateTimePickerAppActivity.this, dateListener, yr, mon, dy).show();
    }
});
timeButton.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        new TimePickerDialog(DateTimePickerAppActivity.this, timeListener, h, m, true).show();
    }
});
}

private DatePickerDialog.OnDateSetListener dateListener =new DatePickerDialog.
OnDateSetListener() {
    public void onDateSet(DatePicker view, int year, intmonthOfYear, int dayOf-Month)
    {
        yr = year;
        mon = monthOfYear;
        dy = dayOfMonth;
        dateTimeView.setText("Current date is "+ (mon+1)+"-"+dy+"-"+yr+" andcurrent time is: "+h+": "+m);
    }
};

private TimePickerDialog.OnTimeSetListener timeListener =new TimePickerDialog.
OnTimeSetListener() {
    public void onTimeSet(TimePicker view, int hour, intminute) {
        h = hour;
        m = minute;
        dateTimeView.setText("Current date is "+ (mon+1)+"-"+dy+"-"+yr+" andcurrent time is: "+h+": "+m);
    }
};
}

```



***The TextView displaying the current date and time and two Button controls (left), the DatePicker dialog appears when the Set Date button is clicked (middle), and the date selected from the DatePicker dialog displayed in the TextView (right)***

## Fragments

The size of the screen changes when a device is oriented from portrait to landscape mode. In landscape mode, the screen becomes wider and shows empty space on the right. The height becomes smaller and hides the controls on the bottom of the display. There is a difference in screen sizes between the Android phone and Android tablet, as well. Android tablets have a 7–10 inch display, whereas Android phones are in the range of 3–5 inches.

### The Structure of a Fragment

A fragment is a combination of an activity and a layout and contains a set of views that make up an independent and atomic user interface. For example, one or more fragments can be embedded in the activity to fill up the blank space that appears on the right when switching from portrait to landscape. Similarly, the fragments can be dynamically removed if the screen size is unable to accommodate the Views. That is, the fragments make it possible for us to manage the Views depending on the target device.

### The Life Cycle of a Fragment

The life cycle of a fragment includes several callback methods, as listed here:

- **onAttach()**—Called when the fragment is attached to the activity.
- **onCreate()**—Called when creating the fragment.
- **onCreateView()**—Called to create the view for the fragment.
- **onActivityCreated()**—Called when the activity's `onCreate()` method is returned.
- **onStart()**—Called when the fragment is visible to the user. This method is associated with the activity's `onStart()`.
- **onResume()**—Called when the fragment is visible and is running. The method is associated with the activity's `onResume()`.
- **onPause()**—Called when the fragment is visible but does not have focus. The method is associated with the activity's `onPause()`.
- **onStop()**—Called when fragment is not visible. The method is associated with the activity's `onStop()`.
- **onDestroyView()**—Called when the fragment is supposed to be saved or destroyed. The view hierarchy is removed from the fragment.
- **onDestroy()**—Called when the fragment is no longer in use. No view hierarchy is associated with the fragment, but the fragment is still attached to the activity.
- **onDetach()**—Called when the fragment is detached from the activity and resources allocated to the fragment are released.

To understand the concept of fragments, let's create an Android project called `FragmentsApp`. In this application, we are going to create two fragments: `Fragment1` and `Fragment2`. `Fragment1` contains a selection widget, `ListView`, that displays a couple of fruits to choose from. `Fragment2` contains a `TextView` control to display the fruit selected from the `ListView` of `Fragment1`. The fragments use individual XML layout files to define their Views, so for the two fragments, let's add two XML files called `fragment1.xml` and `fragment2.xml` to the `res/layout` folder.

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical"
android:background="#0000FF" >
<ListView
```

```

android:id="@+id/fruits_list"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:drawSelectorOnTop="false"/>
</LinearLayout>

```

### Code Written into the XML File fragment1.xml

```

<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical" >
<TextView
android:id="@+id/selectedopt"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:text="Please select a fruit" />
</LinearLayout>

```

### Code Written into the XML File fragment2.xml

```

public class Fragment1Activity extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
    container, Bundle
    savedInstanceState) {
        Context c = getActivity().getApplicationContext();
        View vw = inflater.inflate(R.layout.fragment1,container, false);
        final String[] fruits={"Apple", "Mango", "Orange","Grapes", "Banana"};
        ListView fruitsList = (ListView)
        vw.findViewById(R.id.fruits_list);
        ArrayAdapter<String> arrayAdpt= new ArrayAdapter<String>(c,android.R.layout.simple_list_item_1,
        fruits);
        fruitsList.setAdapter(arrayAdpt);

        fruitsList.setOnItemClickListener(new OnItemClickListener(){
            @Override
            public void onItemClick(AdapterView<?> parent, Viewv, int position,long id)
            {
                TextView selectedOpt = (TextView)
                getActivity().findViewById(R.id.selectedopt);

                selectedOpt.setText("You have selected "+((TextView) v).getText().toString());
            }
        });
        return vw;
    }
}

```

**Code Written into the Java Class File Fragment1Activity.java**

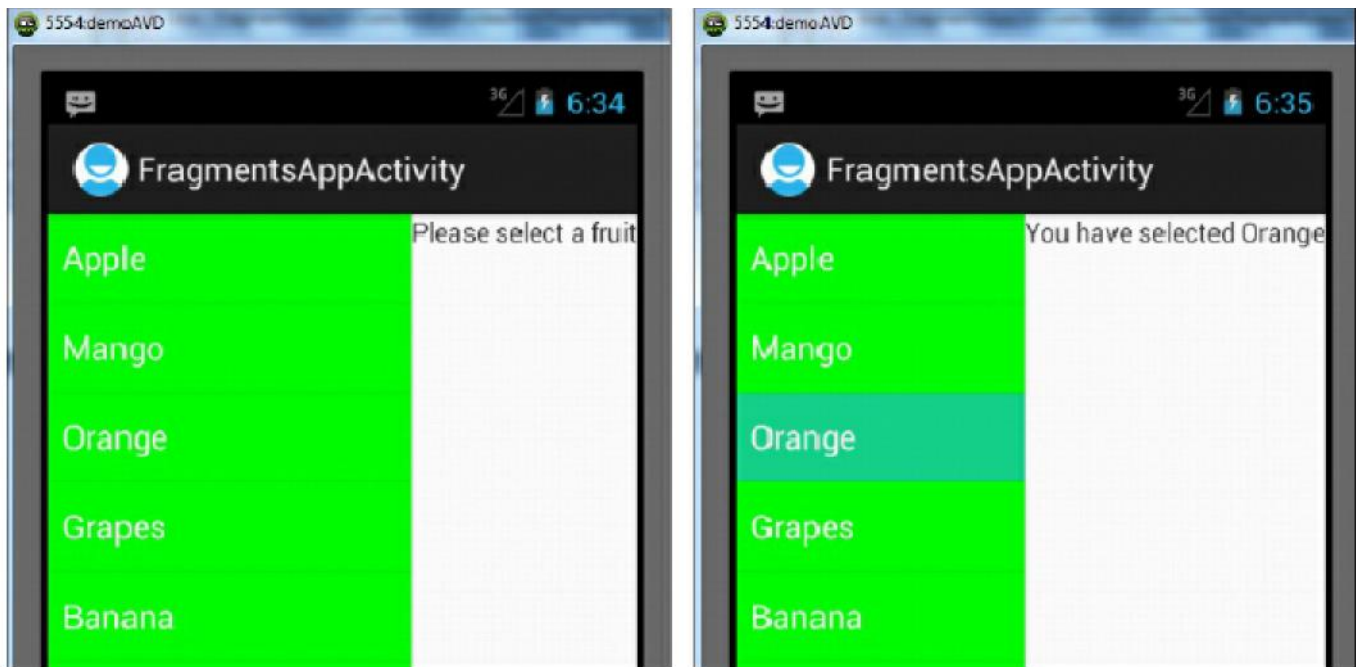
```
import android.os.Bundle;
import android.view.ViewGroup;
import android.view.View;
import android.view.LayoutInflater;
public class Fragment2Activity extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
    container, Bundle
    savedInstanceState) {
    return inflater.inflate(R.layout.fragment2, container,false);
    }
}
```

**Code Written into the Java Class File Fragment2Activity.java**

```
<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="horizontal" >
<fragment
android:name="com.androidunleashed.fragmentsapp.Fragment1 Activity"
android:id="@+id/fragment1"
android:layout_weight="1"
android:layout_width="wrap_content"
android:layout_height="match_parent" />
<fragment
android:name="com.androidunleashed.fragmentsapp.Fragment2Activity"
android:id="@+id/fragment2"
android:layout_weight="0"
android:layout_width="wrap_content"
android:layout_height="match_parent" />
</LinearLayout>
```

**The Layout File activity\_fragments\_app.xml After Adding the Two Fragments****OUTPUT:**





*ListView and TextView controls displayed via two fragments (left), and the TextView of the second fragment, showing the item selected from the ListView of the first fragment (right)*

## Creating Fragments with Java Code

Until now, we have been defining fragments statically by using `<fragment>` elements in the layout file of the application. Let's now learn how to add fragments to the activity during runtime. For creating, adding, and replacing fragments to an activity dynamically, we use the `FragmentManager`.

The `FragmentManager` is used to manage fragments in an activity. It provides the methods to access the fragments that are available in the activity. It also enables us to perform the `FragmentManagerTransaction` required to add, remove, and replace fragments. To access the `FragmentManager`, the method used is `getFragmentManager()`, as shown here:

```
FragmentManager fragmentManager = getFragmentManager();
```

1. Create Android New project (Click here for setup new Android project).
2. Add Following xml code into activity\_main.xml

File : res/layout/activity\_main.xml

```
<LinearLayout >
<LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:weightSum="4">
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Fragment One"
    android:onClick="FragmentOneClick"
    android:layout_weight="2"/>

<Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Fragment Two"
    android:onClick="FragmentTwoClick"
    android:layout_weight="2"/>
</LinearLayout>

<fragment
    android:id="@+id/fragment_switch"
    android:name="androidinterview.com.androidfragment.FragmentOne"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginLeft="5dp"
    android:layout_marginRight="5dp"
    android:layout_marginBottom="5dp"/>

</LinearLayout>
```

3. Create "**fragment\_one.xml**" file and add following code.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

<TextView
    android:id="@+id/textView1"
```

```

        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:text="Fragment One"
        android:textStyle="bold"
        android:textSize="20dp"
        android:textColor="#ffffff"
        android:gravity="center"
        android:background="#369636"/>

```

```
</LinearLayout>
```

4. Create “**fragment\_two.xml**” file and add following code.

```

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

```

```

    <TextView
        android:id="@+id/textView2"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Fragment Two"
        android:textStyle="bold"
        android:textSize="20dp"
        android:textColor="#ffffff"
        android:gravity="center"
        android:background="#563256"/>

```

```
</LinearLayout>
```

5. Open “**FragmentOne.java**” file and add following JAVA code.

```

public class FragmentOne extends Fragment {

    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {

        return inflater.inflate(R.layout.fragment_one, container, false);

    }
}

```

6. Open “**FragmentTwo.java**” file and add following JAVA code.

```
public class FragmentTwo extends Fragment{

    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState) {

        return inflater.inflate(R.layout.fragment_two, container,false);

    }
}
```

7. Open “**MainActivity.java**” file and add following JAVA code.

```
public class MainActivity extends Activity {

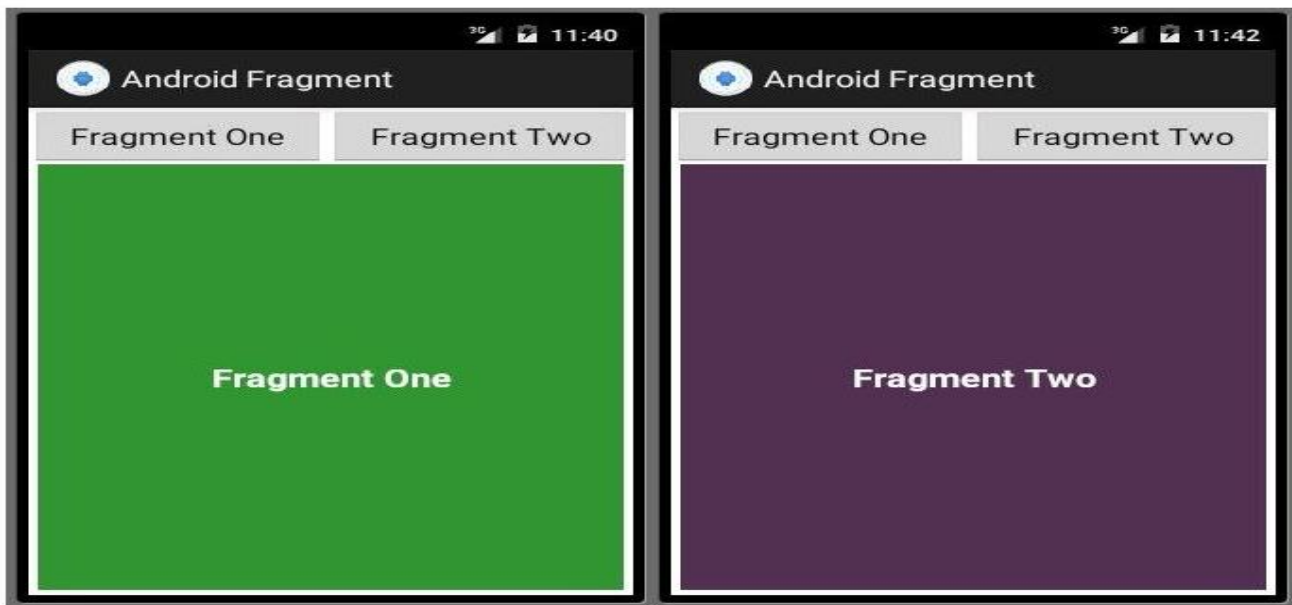
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void FragmentOneClick(View view) {
        Fragment myfragment;
        myfragment = new FragmentOne();

        FragmentManager fm = getFragmentManager();
        FragmentTransaction fragmentTransaction = fm.beginTransaction();
        fragmentTransaction.replace(R.id.fragment_switch, myfragment);
        fragmentTransaction.commit();

    }
    public void FragmentTwoClick(View view) {
        Fragment myfragment;
        myfragment = new FragmentTwo();

        FragmentManager fm = getFragmentManager();
        FragmentTransaction fragmentTransaction = fm.beginTransaction();
        fragmentTransaction.replace(R.id.fragment_switch, myfragment);
        fragmentTransaction.commit();

    }
}
```



## Unit-5

### Fragments

A **Fragment** is a piece of an activity which enable more modular activity design. It will not be wrong if we say, a fragment is a kind of **sub-activity**.

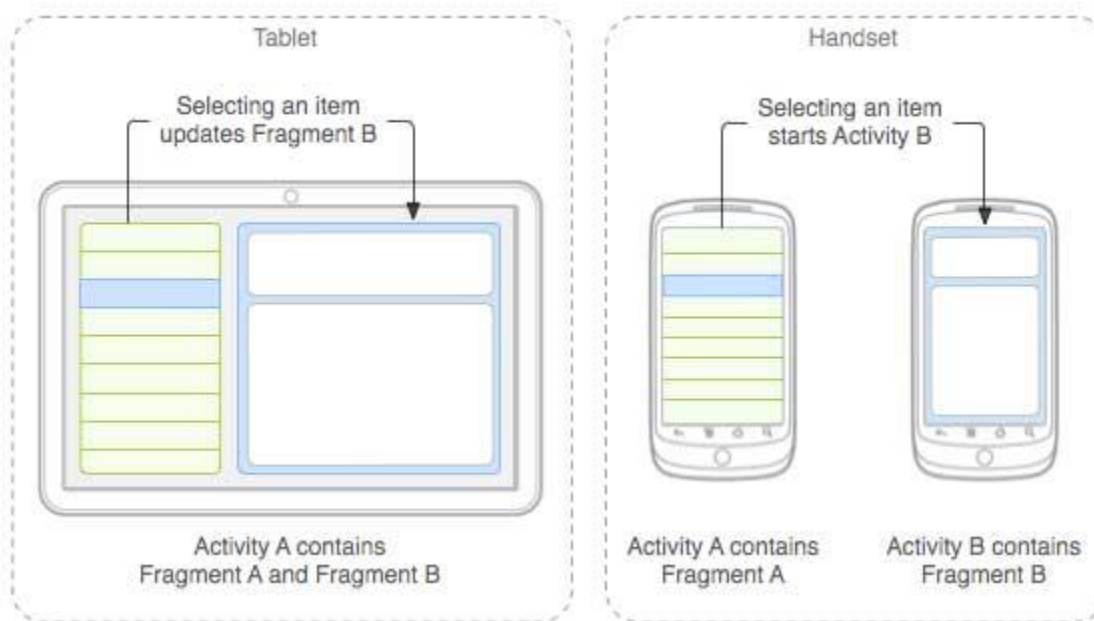
Following are important points about fragment –

- ) A fragment has its own layout and its own behaviour with its own life cycle 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 life cycle of its host activity which means when the activity is paused, all the fragments available in the activity will also be stopped.
- ) A fragment can implement a behaviour 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 activity but each activity can comprise of multiple fragments which will have their own layout, events and complete life cycle.

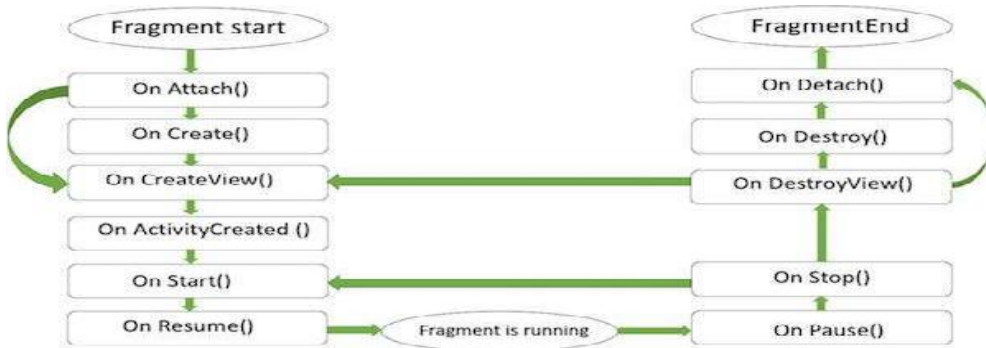
Following is a typical example of how two UI modules defined by fragments can be combined into one activity for a tablet design, but separated for a handset design.



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.



### FRAGMENT LIFECYCLE

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

- ) **onAttach()** The fragment instance is associated with an activity instance. The fragment and the activity is not fully initialized. Typically you get in this method a reference to the activity which uses the fragment for further initialization work.
- ) **onCreate()** The system calls this method 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.
- ) **onActivityCreated()** The `onActivityCreated()` is called after the `onCreateView()` method when the host activity is created. Activity and fragment instance have been created as well as the view hierarchy of the activity. At this point, view can be accessed with the `findViewById()` method. example. In this method you can instantiate objects which require a Context object
- ) **onStart()** The `onStart()` method is called once the fragment gets visible.
- ) **onResume()** Fragment becomes active.
- ) **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.
- ) **onStop()** Fragment going to be stopped by calling `onStop()`
- ) **onDestroyView()** Fragment view will destroy after call this method
- ) **onDestroy()** `onDestroy()` called to do final clean up of the fragment's state but Not guaranteed to be called by the Android platform.

## 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. For 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.

## Types of Fragments

Basically, fragments are divided as three stages as shown below.

- ) Single frame fragments – Single frame fragments are using for hand hold devices like mobiles, here we can show only one fragment as a view.
- ) List fragments – fragments having special list view is called as list fragment
- ) Fragments transaction – Using with fragment transaction. we can move one fragment to another fragment.

## Android - Intents and Filters

An Android **Intent** is an abstract description of an operation to be performed. It can be used with **startActivity** to launch an Activity, **broadcastIntent** to send it to any interested BroadcastReceiver components, and **startService(Intent)** or **bindService(Intent, ServiceConnection, int)** to communicate with a background Service.

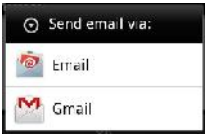
**The intent itself, an Intent object, is a passive data structure holding an abstract description of an operation to be performed.**

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.

```
Intent email = new Intent(Intent.ACTION_SEND, Uri.parse("mailto:"));
email.putExtra(Intent.EXTRA_EMAIL, recipients);
email.putExtra(Intent.EXTRA_SUBJECT, subject.getText().toString());
email.putExtra(Intent.EXTRA_TEXT, body.getText().toString());
startActivity(Intent.createChooser(email, "Choose an email client from..."));
```



Above syntax is calling startActivity method to start an email activity and result should be as shown below –



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.

```
String q ="tutorialspoint";

Intent intent =newIntent(Intent.ACTION_WEB_SEARCH );

intent.putExtra(SearchManager.QUERY, q);

startActivity(intent);
```

Above example will search as **tutorialspoint** on android search engine and it gives the result of tutorialspoint in your an activity

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

Sr.No	Method & Description
1	<b>Context.startActivity()</b> The Intent object is passed to this method to launch a new activity or get an existing activity to do something new.
2	<b>Context.startService()</b> The Intent object is passed to this method to initiate a service or deliver new instructions to an ongoing service.
3	<b>Context.sendBroadcast()</b> 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 as well as information used by the Android system.

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

### Action

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

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

### Data

Adds a data specification to an intent filter. The specification can be just a data type (the `mimeType` attribute), just a URI, or both a data type and a URI. A URI is specified by separate attributes for each of its parts –

These attributes that specify the URL format are optional, but also mutually dependent –

- ) If a scheme is not specified for the intent filter, all the other URI attributes are ignored.
- ) If a host is not specified for the filter, the port attribute and all the path attributes are ignored.

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 –

Sr.No.	Action/Data Pair & Description
1	<b>ACTION_VIEW content://contacts/people/1</b> Display information about the person whose identifier is "1".
2	<b>ACTION_DIAL content://contacts/people/1</b> Display the phone dialer with the person filled in.
3	<b>ACTION_VIEW tel:123</b>

	Display the phone dialer with the given number filled in.
4	<b>ACTION_DIAL tel:123</b> Display the phone dialer with the given number filled in.
5	<b>ACTION_EDIT content://contacts/people/1</b> Edit information about the person whose identifier is "1".
6	<b>ACTION_VIEW content://contacts/people/</b> Display a list of people, which the user can browse through.
7	<b>ACTION_SET_WALLPAPER</b> Show settings for choosing wallpaper
8	<b>ACTION_SYNC</b> It going to be synchronous the data,Constant Value is <b>android.intent.action.SYNC</b>
9	<b>ACTION_SYSTEM_TUTORIAL</b> It will start the platform-defined tutorial(Default tutorial or start up tutorial)
10	<b>ACTION_TIMEZONE_CHANGED</b> It intimates when time zone has changed
11	<b>ACTION_UNINSTALL_PACKAGE</b> It is used to run default uninstaller

## 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](#).

You can check detail on Intent Filters in below section to understand how do we use categories to choose appropriate activity corresponding 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](#)

## 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.

Sr.No	Flags & Description
1	<b>FLAG_ACTIVITY_CLEAR_TASK</b> If set in an Intent passed to <code>Context.startActivity()</code> , this flag will cause any existing task that would be associated with the activity to be cleared before the activity is started. That is, the activity becomes the new root of an otherwise empty task, and any old activities are finished. This can only be used in conjunction with <code>FLAG_ACTIVITY_NEW_TASK</code> .
2	<b>FLAG_ACTIVITY_CLEAR_TOP</b> If set, and the activity being launched is already running in the current task, then instead of launching a new instance of that activity, all of the other activities on top of it will be closed and this Intent will be delivered to the (now on top) old activity as a new Intent.
3	<b>FLAG_ACTIVITY_NEW_TASK</b> This flag is generally used by activities that want to present a "launcher" style behavior: they give the user a list of separate things that can be done, which otherwise run completely independently of the activity launching them.

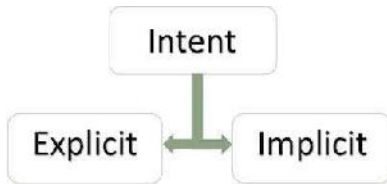
## Component Name

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



### Explicit Intents

Explicit intent going to be connected internal world of application, suppose if you wants to connect one activity to another activity, we can do this quote by explicit intent, below image is connecting first activity to second activity by clicking button.



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 =newIntent(FirstActivity.this,SecondActivity.class);

// 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 –

```
Intent read1=new Intent();
read1.setAction(android.content.Intent.ACTION_VIEW);
read1.setData(ContactsContract.Contacts.CONTENT_URI);
startActivity(read1);
```

Above code will give result as shown below



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 Android studio IDE to create an Android application and name it as <i>My Application</i> under a package <i>com.example.saira_000.myapplication</i> .
2	Modify <i>src/main/java/MainActivity.java</i> file and add the code to define two listeners corresponding two buttons ie. Start Browser and Start Phone.
3	Modify layout XML file <i>res/layout/activity_main.xml</i> to add three buttons in linear layout.
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.My Application/MainActivity.java**.

```

package com.example.saira_000.myapplication;

import android.content.Intent;
import android.net.Uri;
import android.support.v7.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;

public class MainActivity extends AppCompatActivity {

    Button b1, b2;

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        b1 = (Button) findViewById(R.id.button);

        b1.setOnClickListener(new View.OnClickListener() {

            @Override

            public void onClick(View v) {

                Intent i = new Intent(android.content.Intent.ACTION_VIEW,

                Uri.parse("http://www.example.com"));

                startActivity(i);

            }

        });

        b2 = (Button) findViewById(R.id.button2);

```

```

        b2.setOnClickListener(new View.OnClickListener(){

@Override

public void onClick(View v){

Intent i = new Intent(android.content.Intent.ACTION_VIEW,

Uri.parse("tel:9510300000"));

        startActivity(i);

}

});

}

}

```

Following will be the content of **res/layout/activity\_main.xml** file –

```

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

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

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

android:layout_width="match_parent"

android:layout_height="match_parent"

android:paddingLeft="@dimen/activity_horizontal_margin"

android:paddingRight="@dimen/activity_horizontal_margin"

android:paddingTop="@dimen/activity_vertical_margin"

android:paddingBottom="@dimen/activity_vertical_margin"

tools:context=".MainActivity">

<TextView

android:id="@+id/textView1"

android:layout_width="wrap_content"

android:layout_height="wrap_content"

android:text="Intent Example"

android:layout_alignParentTop="true"

```



```

android:layout_centerHorizontal="true"

android:textSize="30dp"/>

```

```

<TextView

```

```

    android:id="@+id/textView2"

```

```

    android:layout_width="wrap_content"

```

```

    android:layout_height="wrap_content"

```

```

    android:text="Tutorials point"

```

```

    android:textColor="#ff87ff09"

```

```

    android:textSize="30dp"

```

```

    android:layout_below="@+id/textView1"

```

```

    android:layout_centerHorizontal="true"/>

```

```

<ImageButton

```

```

    android:layout_width="wrap_content"

```

```

    android:layout_height="wrap_content"

```

```

    android:id="@+id/imageButton"

```

```

    android:src="@drawable/abc"

```

```

    android:layout_below="@+id/textView2"

```

```

    android:layout_centerHorizontal="true"/>

```

```

<EditText

```

```

    android:layout_width="wrap_content"

```

```

    android:layout_height="wrap_content"

```

```

    android:id="@+id/editText"

```

```

    android:layout_below="@+id/imageButton"

```

```

    android:layout_alignRight="@+id/imageButton"

```

```

    android:layout_alignEnd="@+id/imageButton"/>

```

```

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Start Browser"
    android:id="@+id/button"

    android:layout_alignTop="@+id/editText"
    android:layout_alignRight="@+id/textView1"
    android:layout_alignEnd="@+id/textView1"
    android:layout_alignLeft="@+id/imageButton"

    android:layout_alignStart="@+id/imageButton"/>

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Start Phone"
    android:id="@+id/button2"
    android:layout_below="@+id/button"

    android:layout_alignLeft="@+id/button"
    android:layout_alignStart="@+id/button"
    android:layout_alignRight="@+id/textView2"
    android:layout_alignEnd="@+id/textView2"/>
</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">My Applicaiton</string>

</resources>

```

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


```

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.saira_000.myapplication">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>

```

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

- J 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.
- J 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.
- J 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 android studio to create an Android application and name it as <i>My Application</i> under a package <i>com.example.tutorialspoint7.myapplication</i> ;
2	Modify <i>src/Main/Java/MainActivity.java</i> file and add the code to define three listeners corresponding to three buttons defined in layout file.
3	Add a new <i>src/Main/Java/CustomActivity.java</i> file to have one custom activity which will be invoked by different intents.
4	Modify layout XML file <i>res/layout/activity_main.xml</i> to add three buttons in linear layout.
5	Add one layout XML file <i>res/layout/custom_view.xml</i> to add a simple <b>&lt;TextView&gt;</b> to show the

	passed data through intent.
6	Modify <i>AndroidManifest.xml</i> to add <intent-filter> to define rules for your intent to invoke custom activity.
7	Run the application to launch Android emulator and verify the result of the changes done in the application.

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

```
package com.example.tutorialspoint7.myapplication;

import android.content.Intent;
import android.net.Uri;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;

public class MainActivity extends AppCompatActivity {

    Button b1, b2, b3;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        b1 = (Button) findViewById(R.id.button);
        b1.setOnClickListener(new View.OnClickListener() {

            @Override
            public void onClick(View v) {
                Intent i = new Intent(android.content.Intent.ACTION_VIEW,
```

```

Uri.parse("http://www.example.com"));

        startActivity(i);
    }
});

```

```

b2 =(Button)findViewById(R.id.button2);

b2.setOnClickListener(new View.OnClickListener(){

@Override

public void onClick(View v){

```

```

Intent i =new Intent("com.example.

        tutorialspoint7.myapplication.

        LAUNCH",Uri.parse("http://www.example.com"));

        startActivity(i);

```

```

}

});

```

```

b3 =(Button)findViewById(R.id.button3);

```

```

b3.setOnClickListener(new View.OnClickListener(){

@Override

public void onClick(View v){

Intent i =new Intent("com.example.

```

```

        My Application.LAUNCH",

Uri.parse("https://www.example.com"));

        startActivity(i);
    }

```

```

});

}

}

```

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

```
package com.example.tutorialspoint7.myapplication;

import android.app.Activity;
import android.net.Uri;

import android.os.Bundle;
import android.widget.TextView;

/**
 * Created by Tutorialspoint7 on 8/23/2016.
 */
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 –

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

<RelativeLayout

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

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

android:layout_width="match_parent"
```



```

android:layout_height="match_parent"

android:paddingBottom="@dimen/activity_vertical_margin"

android:paddingLeft="@dimen/activity_horizontal_margin"

android:paddingRight="@dimen/activity_horizontal_margin"

android:paddingTop="@dimen/activity_vertical_margin"

tools:context="com.example.tutorialspoint7.myapplication.MainActivity">

```

```
<TextView
```

```
android:id="@+id/textView1"
```

```
android:layout_width="wrap_content"
```

```
android:layout_height="wrap_content"
```

```
android:text="Intent Example"
```

```
android:layout_alignParentTop="true"
```

```
android:layout_centerHorizontal="true"
```

```
android:textSize="30dp"/>
```

```
<TextView
```

```
android:id="@+id/textView2"
```

```
android:layout_width="wrap_content"
```

```
android:layout_height="wrap_content"
```

```
android:text="Tutorials point"
```

```
android:textColor="#ff87ff09"
```

```
android:textSize="30dp"
```

```
android:layout_below="@+id/textView1"
```

```
android:layout_centerHorizontal="true"/>
```

```
<ImageButton
```

```
android:layout_width="wrap_content"
```

```
android:layout_height="wrap_content"
```

```

android:id="@+id/imageButton"

android:src="@drawable/abc"

android:layout_below="@+id/textView2"

android:layout_centerHorizontal="true"/>

```

```

<EditText

android:layout_width="wrap_content"

android:layout_height="wrap_content"

android:id="@+id/editText"

android:layout_below="@+id/imageButton"

android:layout_alignRight="@+id/imageButton"

android:layout_alignEnd="@+id/imageButton"/>

```

```

<Button

android:layout_width="wrap_content"

android:layout_height="wrap_content"

android:text="Start Browser"

android:id="@+id/button"

android:layout_alignTop="@+id/editText"

android:layout_alignLeft="@+id/imageButton"

android:layout_alignStart="@+id/imageButton"

android:layout_alignEnd="@+id/imageButton"/>

```

```

<Button

android:layout_width="wrap_content"

android:layout_height="wrap_content"

android:text="Start browsing with launch action"

android:id="@+id/button2"

android:layout_below="@+id/button"

```

```

android:layout_alignLeft="@+id/button"

android:layout_alignStart="@+id/button"

android:layout_alignEnd="@+id/button"/>

<Button

android:layout_width="wrap_content"

android:layout_height="wrap_content"

android:text="Exceptional condition"

android:id="@+id/button3"

android:layout_below="@+id/button2"

android:layout_alignLeft="@+id/button2"

android:layout_alignStart="@+id/button2"

android:layout_toStartOf="@+id/editText"

android:layout_alignParentEnd="true"/>

</RelativeLayout>

```

Following will be the content of **res/layout/custom\_view.xml** file –

```

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

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

android:orientation="vertical"android:layout_width="match_parent"

android:layout_height="match_parent">

<TextViewandroid: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>

<stringname="app_name">My Application</string>

</resources>

```

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

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.tutorialspoint7.myapplication">

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

        <activity android:name=".MainActivity">
            <intent-filter>

                <action android:name="android.intent.action.MAIN"/>


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

        <activity android:name="com.example.tutorialspoint7.myapplication.CustomActivity">
            <intent-filter>

                <action android:name="android.intent.action.VIEW"/>
                <action android:name="com.example.tutorialspoint7.myapplication.LAUNCH"/>
                <category android:name="android.intent.category.DEFAULT"/>
                <data android:scheme="http"/>
            </intent-filter>
        </activity>
```

&lt;/application&gt;

&lt;/manifest&gt;

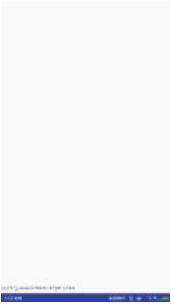
Let's try to run your **My Application** application. I assume you had created your **AVD** while doing environment setup. To run the app from Android Studio, open one of your project's activity files and click Run  icon from the toolbar. Android Studio installs the app on your AVD and starts it and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



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



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



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