

B.TECH IV-1SEM(CSE)
JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY ANANTAPUR
15A05703 MOBILE APPLICATION DEVELOPMENT

Course Objectives: To understand fundamentals of android operating systems. • Illustrate the various components, layouts and views in creating android applications To understand fundamentals of android programming. • Course Outcomes: Create data sharing with different applications and sending and intercepting • SMS. Develop applications using services and publishing android applications. • To demonstrate their skills of using Android software development tools •

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.

Unit 2: Basic Widgets: Understanding the Role of Android Application Components, Understanding the Utility of Android API, Overview of the Android Project Files, Understanding Activities, Role of the Android Manifest File, Creating the User Interface, Commonly Used Layouts and Controls, Event Handling, Displaying Messages Through Toast, Creating and Starting an Activity, Using the Edit Text Control, Choosing Options with Checkbox, Choosing Mutually Exclusive Items Using Radio Buttons

Unit 3: Building Blocks for Android Application Design: Introduction to Layouts, Linear Layout, Relative Layout, Absolute Layout, Using Image View, Frame Layout, Table Layout, Grid Layout, Adapting to Screen orientation. Utilizing Resources and Media Resources, Creating Values Resources, Using Drawable Resources, Switching States with Toggle Buttons, Creating an Images Switcher Application, Scrolling Through Scroll View, playing Audio, Playing Video, All Displaying Progress with Progress Bar, Using Assets.

Unit 4: Using Selection widgets and Debugging: Using List View, Using the Spinner control, Using the GridView Control, Creating an Image Gallery Using the ViewPager Control, Using the Debugging Tool: Dalvik Debug Monitor Service(DDMS), Debugging Application, Using the Debug Perspective. Displaying And Fetching Information Using Dialogs and Fragments: What Are Dialogs?, Selecting the Date and Time in One Application, Fragments, Creating Fragments with java Code, Creating Special Fragments

Unit 5: Building Menus and Storing Data: Creating Interface Menus and Action Bars, Menus and Their Types, Creating Menus Through XML, Creating Menus Through Coding, Applying a Context Menu to a List View, Using the Action Bar, Replacing a Menu with the Action Bar, Creating a Tabbed Action Bar, Creating a Drop-Down List Action Bar Using Databases: Using the SQLiteOpenHelper class, Accessing Databases with the ADB, Creating a Data Entry Form, Communicating with SMS and Emails: Understanding Broadcast Receivers, Using the Notification System, Sending SMS Messages with Java Code, Receiving SMS Messages, Sending Email, Working With Telephony Manager.

Text Books 1. Android Programming by B.M Harwani, Pearson Education, 2013.

References Text Books: 1. Android application Development for Java Programmers, James C Sheusi, Cengage Learning 2. Android In Action by w.Frank Ableson, Robi Sen, Chris King, C. Enrique Ortiz., Dreamtech. 3. Professional Android 4 applications development, Reto Meier, Wiley India, 2012. 4. Beginning Android 4 applications development, Wei- Meng Lee, Wiley India, 2013

Q1. EXPLAIN THE FEATURES AND BENEFITS OF ANDROID OS?

Android is a complete set of software for mobile devices such as tablet computers, notebooks, smartphones, electronic book readers, set-top boxes etc.

- It contains a **linux-based Operating System, middleware and key mobile applications**.
- It can be thought of as a mobile operating system. But it is not limited to mobile only. It is currently used in various devices such as mobiles, tablets, televisions etc.
- **Android** is a software package and linux based operating system for mobile devices such as tablet computers and smartphones.
- It is developed by Google and later the OHA (Open Handset Alliance). Java language is mainly used to write the android code even though other languages can be used.
- The goal of android project is to create a successful real-world product that improves the mobile experience for end users.
- There are many code names of android such as Lollipop, Kitkat, Jelly Bean, Ice cream Sandwich, Froyo, Ecliar, Donut etc which is covered in next page.

What is Open Handset Alliance (OHA)

It's a consortium of 84 companies such as google, samsung, AKM, synaptics, KDDI, Garmin, Teleca, Ebay, Intel etc. It was established on 5th November, 2007, led by Google. It is committed to advance open standards, provide services and deploy handsets using the Android Platform.

Features of Android

After learning what is android, let's see the features of android. The important features of android are given below:

- 1) It is open-source.
- 2) Anyone can customize the Android Platform.
- 3) There are a lot of mobile applications that can be chosen by the consumer.
- 4) It provides many interesting features like weather details, opening screen, live RSS (Really Simple Syndication) feeds etc.

It provides support for messaging services(SMS and MMS), web browser, storage (SQLite), connectivity (GSM, CDMA, Blue Tooth, Wi-Fi etc.), media, handset layout etc.

Advantages:

- **Android is a Linux based open-source operating system, it can be developed by anyone**
- **Easy access to android apps**
- **You can replace the battery and mass storage, disk drive and UDB option**
- **Its supports all Google services**
- **The operating system is able to inform you of a new SMS and Emails or latest updates.**
- **It supports Multitasking**
- **Android phone can also function as a router to share internet**
- **It's free to customize**
- **Can install a modified ROM**
- **Its supports 2D and 3D graphics**

Categories of Android applications

There are many android applications in the market. The top categories are:

- | | |
|-------------------|-------------------------|
| ○ Entertainment | ○ Music and Audio |
| ○ Tools | ○ Social |
| ○ Communication | ○ Media and Video |
| ○ Productivity | ○ Travel and Local etc. |
| ○ Personalization | |

Q2. WRITE SHORT NOTES ON HISTORICAL DEVELOPMENTS AND VERSIONS OF ANDROID?

History of Android

The history and versions of android are interesting to know. The code names of android ranges from A to J currently, such as **Aestro, Blender, Cupcake, Donut, Eclair, Froyo, Gingerbread, Honeycomb, Ice Cream Sandwich, Jelly Bean, KitKat** and **Lollipop**. Let's understand the android history in a sequence.

- 1) Initially, **Andy Rubin** founded Android Incorporation in Palo Alto, California, United States in October, 2003.
- 2) In 17th August 2005, Google acquired android Incorporation. Since then, it is in the subsidiary of Google Incorporation.
- 3) The key employees of Android Incorporation are **Andy Rubin, Rich Miner, Chris White** and **Nick Sears**.
- 4) Originally intended for camera but shifted to smart phones later because of low market for camera only.
- 5) Android is the nick name of Andy Rubin given by coworkers because of his love to robots.
- 6) In 2007, Google announces the development of android OS.
- 7) In 2008, HTC launched the first android mobile.

Android Versions, Codename and API

Let's see the android versions, codenames and API Level provided by Google.

Version	Code name	API Level
1.5	Cupcake	3
1.6	Donut	4
2.1	Eclair	7
2.2	Froyo	8
2.3	Gingerbread	9 and 10
3.1 and 3.3	Honeycomb	12 and 13
4.0	Ice Cream Sandwich	15
4.1, 4.2 and 4.3	Jelly Bean	16, 17 and 18
4.4	KitKat	19
5.0	Lollipop	21
6.0	Marshmallow	23
7.0	Nougat	24-25
8.0	Oreo	26-27

- **Android 9.0, Pie:** August 6, 2018
- **Android 10.0:** September 3, 2019
- **Android 11.0:** September 8, 2020

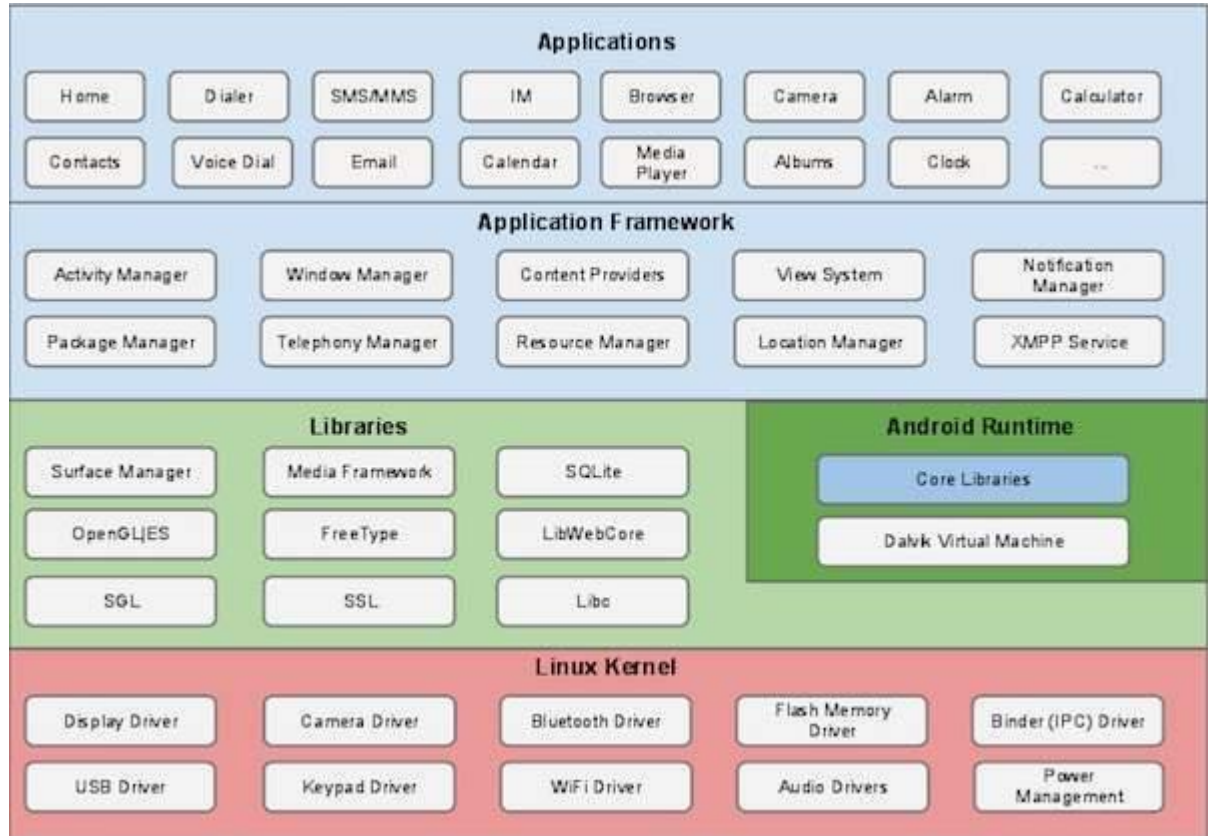
Q3. EXPLAIN ANDROID ARCHITECTURE? OR BASIC BUILDING BLOCK OF OR KERNAL OF ANDROID?

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.

Q5. EXPLAIN BASIC BUILDING BLOCKS OF ANDROID?

Android Core Building Blocks



An android **component** is simply a piece of code that has a well defined life cycle e.g. Activity, Receiver, Service etc.

The **core building blocks** or **fundamental components** of android are activities, views, intents, services, content providers, fragments and AndroidManifest.xml.

Activity

An activity is a class that represents a single screen. It is like a Frame in AWT.

View

A view is the UI element such as button, label, text field etc. Anything that you see is a view.

Intent

Intent is used to invoke components. It is mainly used to:

- Start the service
- Launch an activity
- Display a web page
- Display a list of contacts
- Broadcast a message
- Dial a phone call etc.

For example, you may write the following code to view the webpage.

1. Intent intent=**new** Intent(Intent.ACTION_VIEW);
2. intent.setData(Uri.parse("<http://www.javatpoint.com>"));
3. startActivity(intent);

Service

Service is a background process that can run for a long time.

There are two types of services local and remote. Local service is accessed from within the application whereas remote service is accessed remotely from other applications running on the same device.

Content Provider

Content Providers are used to share data between the applications.

Fragment

Fragments are like parts of activity. An activity can display one or more fragments on the screen at the same time.

AndroidManifest.xml

It contains informations about activities, content providers, permissions etc. It is like the web.xml file in Java EE.

Android Virtual Device (AVD)

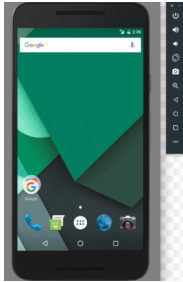
It is used to test the android application without the need for mobile or tablet etc. It can be created in different configurations to emulate different types of real devices.

Q6. WRITE SHORT NOTES ON ANDROID EMULATOR. HOW TO INSTALL AND WORK WITH ANDROID EMULATOR?

Android Emulator

The **Android emulator** is an **Android Virtual Device (AVD)**, which represents a specific Android device. We can use the Android emulator as a target device to execute and test our Android application on our PC. The Android emulator provides almost all the functionality of a real device. We can get the incoming phone calls and text messages. It also gives the location of the device and simulates different network speeds. Android emulator simulates rotation and other hardware sensors. It accesses the Google Play store, and much more.

The android emulator has all of the hardware and software features like mobile device except phone calls. It provides a variety of navigation and control keys. It also provides a screen to display your application. The emulators utilize the android virtual device configurations. Once your application is running on it, it can use services of the android platform to help other applications, access the network, play audio, video, store and retrieve the data.



Testing Android applications on emulator are sometimes faster and easier than doing on a real device. For example, we can transfer data faster to the emulator than to a real device connected through USB. The Android emulator comes with predefined configurations for several Android phones, Wear OS, tablet, Android TV devices.

Requirement and recommendations

The Android emulator takes additional requirements beyond the basic system requirement for Android Studio. These requirements are given below:

- SDK Tools 26.1.1 or higher
- 64-bit processor
- Windows: CPU with UG (unrestricted guest) support
- HAXM 6.2.1 or later (recommended HAXM 7.2.0 or later)

Install the emulator

The Android emulator is installed while installing the Android Studio. However some components of emulator may or may not be installed while installing Android Studio. To install the emulator component, select the **Android Emulator** component in the **SDK Tools** tab of the **SDK Manager**.

Run an Android app on the Emulator

We can run an Android app from the Android Studio project, or we can run an app which is installed on the Android Emulator as we run any app on a device.

Q7. WRITE SHORT NOTES ON ANDROID DEBUG BRIDGE(ADB)?

Android Debug Bridge (adb)

Android Debug Bridge (adb) is a versatile command-line tool that lets you communicate with a device. The adb command facilitates a variety of device actions, such as installing and debugging apps, and it provides access to a Unix shell that you can use to run a variety of commands on a device. It is a client-server program that includes three components:

- **A client**, which sends commands. The client runs on your development machine. You can invoke a client from a command-line terminal by issuing an adb command.
- **A daemon (adbd)**, which runs commands on a device. The daemon runs as a background process on each device.
- **A server**, which manages communication between the client and the daemon. The server runs as a background process on your development machine.

adb is included in the Android SDK Platform-Tools package. You can download this package with the [SDK Manager](#), which installs it at `android_sdk/platform-tools/`. Or if you want the standalone Android SDK Platform-Tools package, you can [download it here](#).

For information on connecting a device for use over ADB, including how to use the Connection Assistant to troubleshoot common problems, see [Run apps on a hardware device](#).

How adb works

When you start an adb client, the client first checks whether there is an adb server process already running. If there isn't, it starts the server process. When the server starts, it binds to local TCP port 5037 and listens for commands sent from adb clients—all adb clients use port 5037 to communicate with the adb server.

The server then sets up connections to all running devices. It locates emulators by scanning odd-numbered ports in the range 5555 to 5585, the range used by the first 16 emulators. Where the server finds an adb daemon (adb), it sets up a connection to that port. Note that each emulator uses a pair of sequential ports — an even-numbered port for console connections and an odd-numbered port for adb connections. For example:

Emulator 1, console: 5554
Emulator 1, adb: 5555
Emulator 2, console: 5556
Emulator 2, adb: 5557
and so on...

As shown, the emulator connected to adb on port 5555 is the same as the emulator whose console listens on port 5554.

Once the server has set up connections to all devices, you can use adb commands to access those devices. Because the server manages connections to devices and handles commands from multiple adb clients, you can control any device from any client (or from a script).

To use adb with a device connected over USB, you must enable **USB debugging** in the device system settings, under **Developer options**.

On Android 4.2 and higher, the Developer options screen is hidden by default. To make it visible, go to **Settings > About phone** and tap **Build number** seven times. Return to the previous screen to find **Developer options** at the bottom.

On some devices, the Developer options screen might be located or named differently.

Q8. WRITE ABOUT ANDROID STUDIO INSTALLATION PROCESS ?

Install Android

Android supports java, c++, c# etc. language to develop android applications. Java is the officially supported language for android. All the android examples of this site is developed using Java language and Eclipse IDE. Here, we are going to tell you, the required softwares to develop android applications using Eclipse IDE. There are two ways to install android.

1. By ADT Bundle
2. By Setup Eclipse Manually

1) By Android Studio

It is the simplest technique to install required software for android application. It includes:

- Eclipse IDE
- Android SDK
- Eclipse Plugin

If you download the Android Studio from android site, you don't need to have eclipse IDE, android SDK and eclipse Plugin because it is already included in Android Studio.

If you have downloaded the Android Studio, unjar it, go to eclipse IDE and start the eclipse by clicking on the eclipse icon. You don't need to do any extra steps here.

2) By set up eclipse manually

setting up android in eclipse manually.

Let's see the list of software required to **setup android for eclipse** IDE manually.

1. Install the JDK
2. Download and install the Eclipse for developing android application
3. Download and Install the android SDK
4. Intall the ADT plugin for eclipse
5. Configure the ADT plugin
6. Create the AVD
7. Create the hello android application

1) Install the Java Development Kit (JDK)

For creating android application, JDK must be installed if you are developing the android application with Java language. [download the JDK](#)

2) Download and install the Eclipse IDE

For developing the android application using eclipse IDE, you need to install the Eclipse. you can download it from this location [download the Eclipse](#). Eclipse classic version is recommended but we are using the Eclipse IDE for JavaEE Developers.

3) Download and install the android SDK

First of all, [download the android SDK](#). In this example we have installed the android SDK for windows (.exe version).

Now double click on the exe file, it will be installed. I am using the android 2.2 version here.

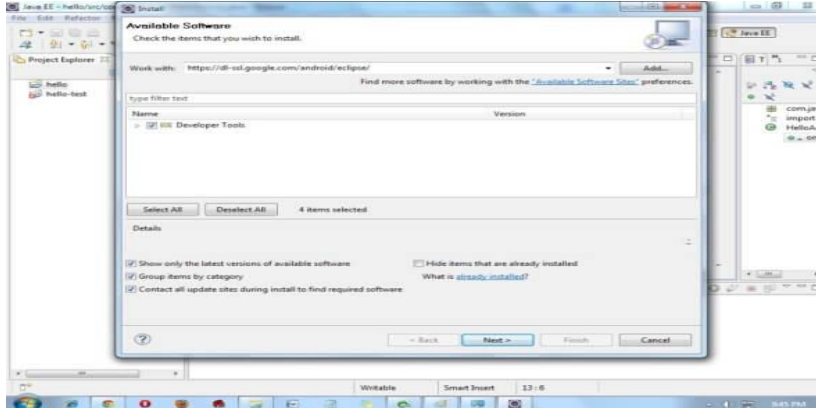
4) Download the ADT plugin for eclipse

ADT (Android Development Tools) is required for developing the android application in the eclipse IDE. It is the plugin for Eclipse IDE that is designed to provide the integrated environment.

For downloading the ADT, you need to follow these steps:

1) Start the eclipse IDE, then select **Help > Install new software...**

2) In the **work with** combo box, write <https://dl-ssl.google.com/android/eclipse/>



3) **select the checkbox** next to Developer Tools and **click next**

4) You will see, a list of tools to be downloaded here, **click next**

5) **click finish**

6) After completing the installation, restart the eclipse IDE

5) Configuring the ADT plugin

After the installing ADT plugin, now tell the eclipse IDE for your android SDK location. To do so:

1. Select the **Window menu > preferences**
2. Now select the android from the left panel. Here you may see a dialog box asking if you want to send the statistics to the google. Click **proceed**.
3. Click on the browse button and locate your SDK directory e.g. my SDK location is C:\Program Files\Android\android-sdk .
4. Click the apply button then OK.

6) Create an Android Virtual Device (AVD)

For running the android application in the Android Emulator, you need to create and AVD. For creating the AVD:

1. Select the **Window menu > AVD Manager**
2. Click on the **new** button, to create the AVD
3. Now a dialog appears, write the AVD name e.g. myavd. Now choose the target android version e.g. android2.2.
4. click the **create AVD**

7) create and run the simple android example

Q9. EXPLAIN HOW A SIMPLE ANDROID APPLICATIONS ARE DEVELOPED USING ANDRIOD STUDIO?

How to make android apps

In this page, you will know how to create the simple hello android application. We are creating the simple example of android using the Eclipse IDE. For creating the simple example:

1. Create the new android project
2. Write the message (optional)
3. Run the android application

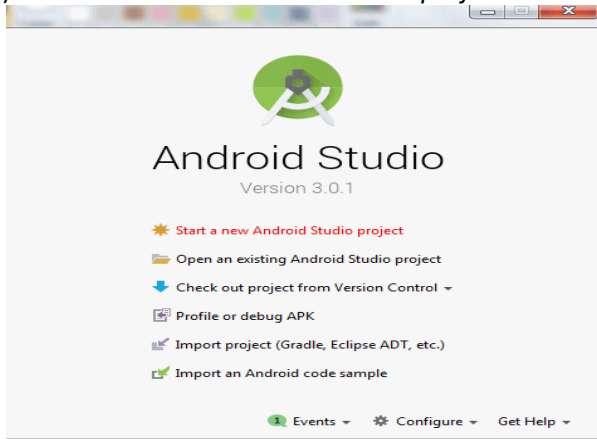
Hello Android Example

You need to follow the 3 steps mentioned above for creating the Hello android application.

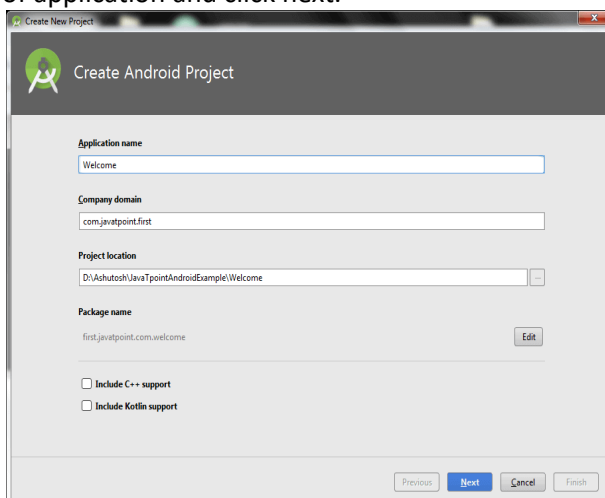
1) Create the New Android project

For creating the new android studio project:

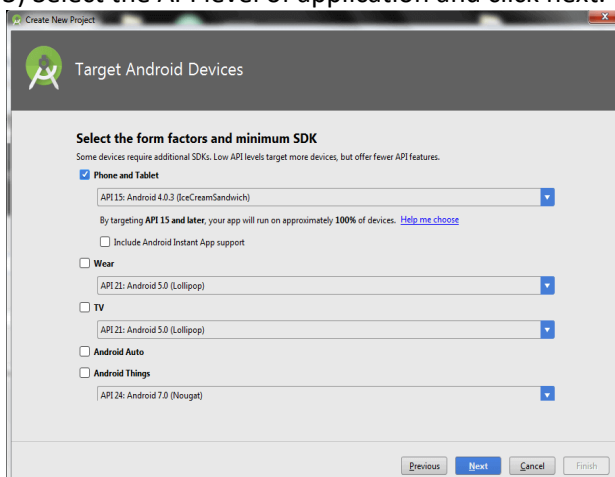
) Select *Start a new Android Studio project*



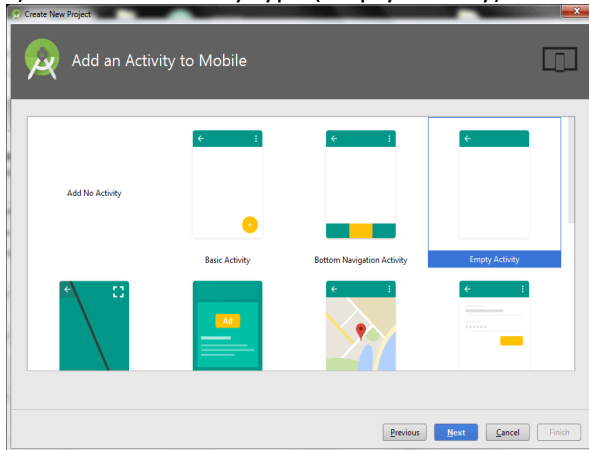
2) Provide the following information: Application name, Company domain, Project location and Package name of application and click next.



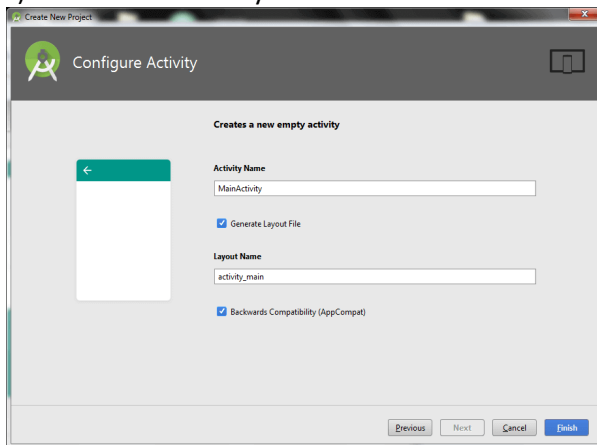
3) Select the API level of application and click next.



4) Select the Activity type (Empty Activity).

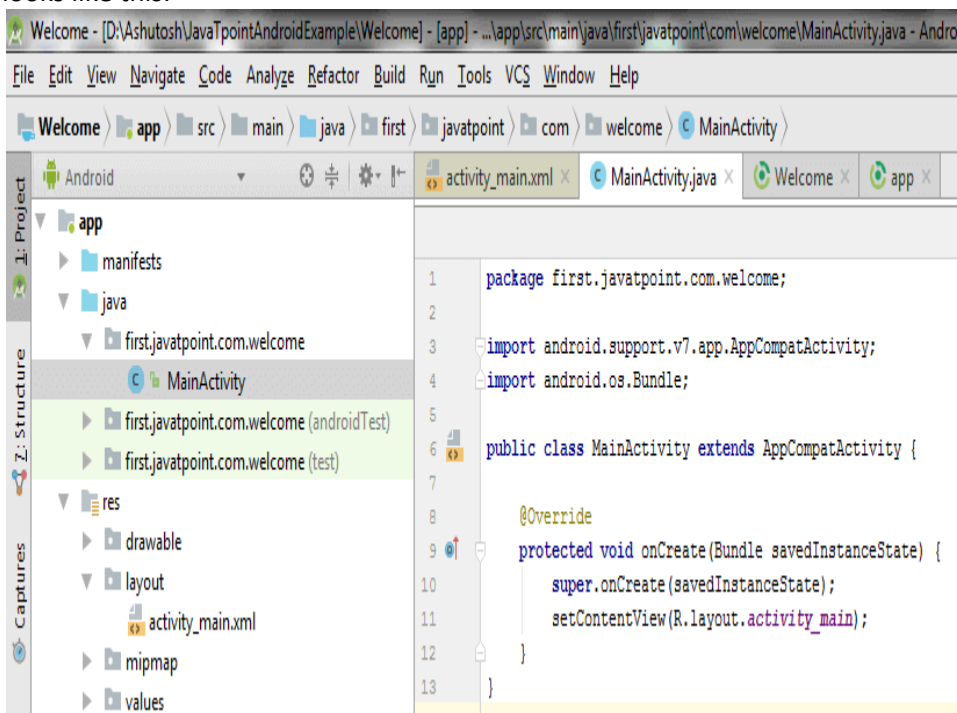


5) Provide the Activity Name and click finish.



After finishing the Activity configuration, Android Studio auto generates the activity class and other required configuration files.

Now an android project has been created. You can explore the android project and see the simple program, it looks like this:



2) Write the message

File: activity_main.xml

Android studio auto generates code for activity_main.xml file. You may edit this file according to your requirement.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="first.javatpoint.com.welcome.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello Android!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
}
```

File: MainActivity.java

```
package first.javatpoint.com.welcome;
```

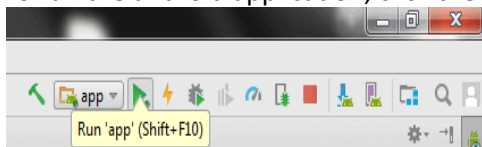
```
import android.support.v7.app.AppCompatActivity;
```

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

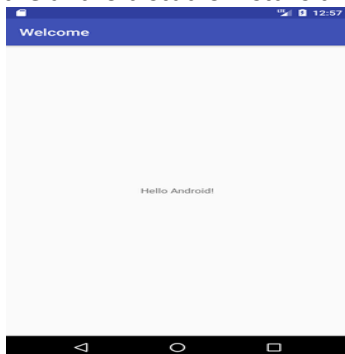
```
public class MainActivity extends AppCompatActivity {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

3) Run the android application

To run the android application, click the run icon on the toolbar or simply press Shift + F10.



The android emulator might take 2 or 3 minutes to boot. So please have patience. After booting the emulator, the android studio installs the application and launches the activity. You will see something like this:



UNIT-II

Q1. EXPLAIN ANDROID APPLICATION COMPONENTS IN DETIAL?

ANDROID APPLICATION COMPONENTS:

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

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

Sr.No	Components & Description
1	Activities : They dictate the UI and handle the user interaction to the smart phone screen.
2	Services : They handle background processing associated with an application.
3	Broadcast Receivers : They handle communication between Android OS and applications.
4	Content Providers : They handle data and database management issues.

Activities

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

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

```
public class MainActivity extends Activity {  
}
```

Services

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

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

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

Broadcast Receivers

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

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

```
public class MyReceiver extends BroadcastReceiver {  
    public void onReceive(context,intent){}  
}
```

Content Providers

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

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

```
public class MyContentProvider extends ContentProvider {  
    public void onCreate(){}  
}
```

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

Additional Components

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

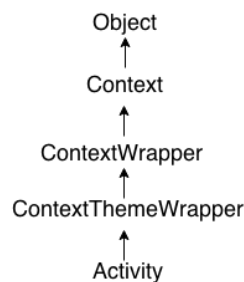
S.No	Components & Description
1	Fragments : Represents a portion of user interface in an Activity.
2	Views : UI elements that are drawn on-screen including buttons, lists forms etc.
3	Layouts :View hierarchies that control screen format and appearance of the views.
4	Intents :Messages wiring components together.
5	Resources :External elements, such as strings, constants and drawable pictures.
6	Manifest :Configuration file for the application.

Q2. EXPLAIN ABOUT ANDROID ACTIVITY LIFECYCLE?

Android Activity Lifecycle

An activity represents a single screen with a user interface just like window or frame of Java. Android activity is the subclass of ContextThemeWrapper class.

Android Activity Lifecycle is controlled by 7 methods of android.app.Activity class.



An activity is the single screen in android. It is like window or frame of Java. By the help of activity, you can place all your UI components or widgets in a single screen.

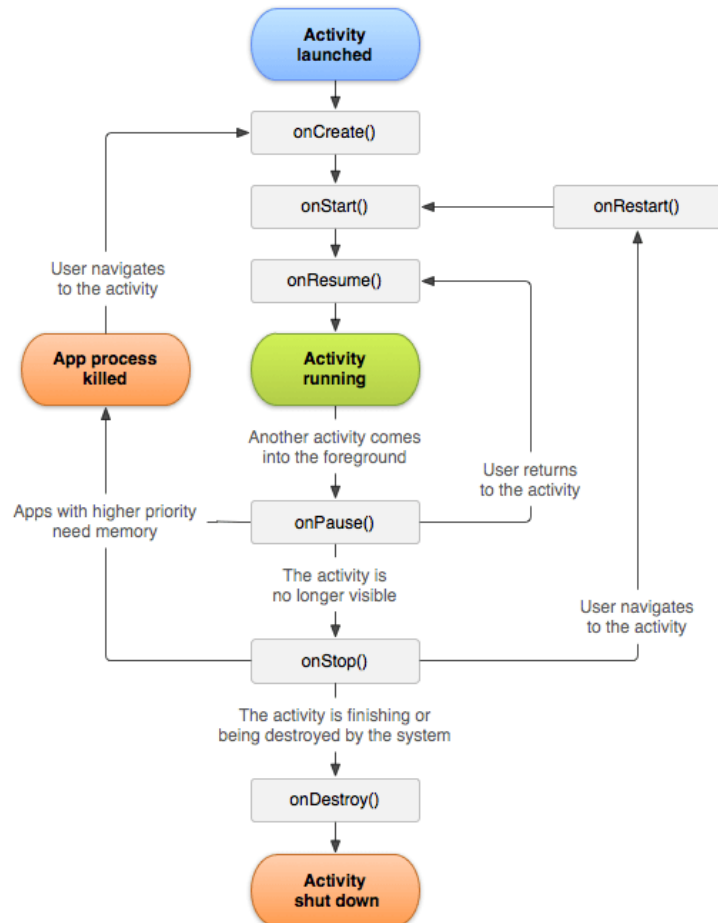
The 7 lifecycle method of Activity describes how activity will behave at different states.

Android Activity Lifecycle methods

Let's see the 7 lifecycle methods of android activity.

Method	Description
onCreate	called when activity is first created.
onStart	called when activity is becoming visible to the user.

onResume	called when activity will start interacting with the user.
onPause	called when activity is not visible to the user.
onStop	called when activity is no longer visible to the user.
onRestart	called after your activity is stopped, prior to start.
onDestroy	called before the activity is destroyed.



File: activity_main.xml

```

<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="example.javatpoint.com.activitylifecycle.MainActivity">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</android.support.constraint.ConstraintLayout>
  
```

File: MainActivity.java

```
package com.example.helloandroid;
import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.widget.TextView;
public class MainActivity extends Activity { //(1)
    @Override
    protected void onCreate(Bundle savedInstanceState) { //(2)
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main); //(3)
    }
    public boolean onCreateOptionsMenu(Menu menu) { //(4)
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.activity_main, menu);
        return true;
    }
}
```

(1) **Activity** is a java class that creates and default window on the screen where we can place different components such as Button, EditText, TextView, Spinner etc. It is like the Frame of Java AWT.

It provides life cycle methods for activity such as onCreate, onStop, onResume etc.

(2) The **onCreate method** is called when Activity class is first created.

(3) The **setContentView(R.layout.activity_main)** gives information about our layout resource. Here, our layout resources are defined in activity_main.xml file.

activity_main.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity" >
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="@string/hello_world" />
</RelativeLayout>
```

As you can see, a textview is created by the framework automatically. But the message for this string is defined in the strings.xml file. The **@string/hello_world** provides information about the textview message. The value of the attribute hello_world is defined in the strings.xml file.

strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">helloandroid</string>
    <string name="hello_world">Hello world!</string>
    <string name="menu_settings">Settings</string>
</resources>
```

Generated R.java file

It is the auto-generated file that contains IDs for all the resources of res directory. It is generated by aapt(Android Asset Packaging Tool). Whenever you create any component on activity_main, a corresponding ID is created in the R.java file which can be used in the Java Source file later.

File: R.java

```
package com.example.helloandroid;
public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int ic_launcher=0x7f020000;
    }
    public static final class id {
        public static final int menu_settings=0x7f070000;
    }
    public static final class layout {
        public static final int activity_main=0x7f030000;
    }
    public static final class menu {
        public static final int activity_main=0x7f060000;
    }
    public static final class string {
        public static final int app_name=0x7f040000;
        public static final int hello_world=0x7f040001;
        public static final int menu_settings=0x7f040002;
    }
    public static final class style {
        public static final int AppBaseTheme=0x7f050000;
        /** Application theme.
All customizations that are NOT specific to a particular API-level can go here.
        */
        public static final int AppTheme=0x7f050001;
    }
}
```

APK File

An apk file is created by the framework automatically. If you want to run the android application on the mobile, transfer and install it.

Resources

It contains resource files including activity_main, strings, styles etc.

Manifest file

It contains information about package including components such as activities, services, content providers etc.

Q3. WRITE SHORT NOTES ON DVM?

Dalvik Virtual Machine | DVM

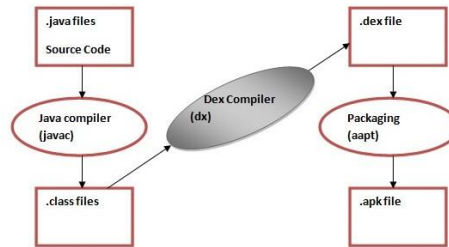
As we know the modern JVM is high performance and provides excellent memory management. But it needs to be optimized for low-powered handheld devices as well.

The **Dalvik Virtual Machine (DVM)** is an android virtual machine optimized for mobile devices. It optimizes the virtual machine for *memory*, *battery life* and *performance*.

Dalvik is a name of a town in Iceland. The Dalvik VM was written by Dan Bornstein.

The Dex compiler converts the class files into the .dex file that run on the Dalvik VM. Multiple class files are converted into one dex file.

Let's see the compiling and packaging process from the source file:



The **javac tool** compiles the java source file into the class file.

The **dx tool** takes all the class files of your application and generates a single .dex file. It is a platform-specific tool.

The **Android Assets Packaging Tool (aapt)** handles the packaging process.

Q5. WRITE 5 ABOUT ESSENTIAL ELEMENTS IN ANDROIDMANIFEST.XML FILE?

AndroidManifest.xml file in android

The **AndroidManifest.xml file** contains information of your package, including components of the application such as activities, services, broadcast receivers, content providers etc.

It performs some other tasks also:

- It is **responsible to protect the application** to access any protected parts by providing the permissions.
- It also **declares the android api** that the application is going to use.
- It **lists the instrumentation classes**. The instrumentation classes provides profiling and other informations. These informations are removed just before the application is published etc.

This is the required xml file for all the android application and located inside the root directory.

A simple AndroidManifest.xml file looks like this:

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.javatpoint.hello"
    android:versionCode="1"
    android:versionName="1.0" >

```

<uses-sdk

```

    android:minSdkVersion="8"
    android:targetSdkVersion="15" />

```

<application

```

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

```

<activity

```

    android:name=".MainActivity"
    android:label="@string/title_activity_main" >

```

<intent-filter>

```

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

```

```

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

```

</intent-filter>

</activity>

</application>

</manifest>

Elements of the AndroidManifest.xml file

The elements used in the above xml file are described below.

<manifest>

manifest is the root element of the AndroidManifest.xml file. It has **package** attribute that describes the package name of the activity class.

<application>

application is the subelement of the manifest. It includes the namespace declaration. This element contains several subelements that declares the application component such as activity etc.

The commonly used attributes of this element are **icon**, **label**, **theme** etc.

android:icon represents the icon for all the android application components.

android:label works as the default label for all the application components.

android:theme represents a common theme for all the android activities.

<activity>

activity is the subelement of application and represents an activity that must be defined in the AndroidManifest.xml file. It has many attributes such as label, name, theme, launchMode etc.

android:label represents a label i.e. displayed on the screen.

android:name represents a name for the activity class. It is required attribute.

<intent-filter>

intent-filter is the sub-element of activity that describes the type of intent to which activity, service or broadcast receiver can respond to.

<action>

It adds an action for the intent-filter. The intent-filter must have at least one action element.

Q6. WRITE SHORT NOTES ON ANDROID WIDGETS?

Android Widgets

There are given a lot of **android widgets** with simplified examples such as Button, EditText, AutoCompleteTextView, ToggleButton, DatePicker, TimePicker, ProgressBar etc.

Android widgets are easy to learn. The widely used android widgets with examples are given below:

Android Button : to perform event handling on button click.

Android Toast : Displays information for the short duration of time.

Custom Toast : We are able to customize the toast, such as we can display image on the toast for a short period of time

ToggleButton : It has two states ON/OFF.

CheckBox : for the application of simple food ordering checking with multiple options through checkbox

AlertDialog : AlertDialog displays a alert dialog containing the message with OK and Cancel buttons.

Spinner : Spinner displays the multiple options, but only one can be selected at a time.

AutoCompleteTextView : simple example of AutoCompleteTextView.

RatingBar : RatingBar displays the rating bar.

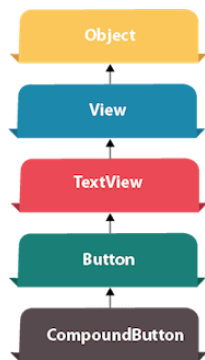
DatePicker : DatePicker displays the datepicker dialog that can be used to pick the date.

TimePicker : TimePicker displays the timepicker dialog that can be used to pick the time.

ProgressBar : ProgressBar displays progress task.

Q7. EXPLAIN DEVELOPING AN APPLICATION USING ANDROID BUTTON WITH LISTNER CLASS?

Android Button Example



Android Button represents a push-button. The `android.widget.Button` is subclass of `TextView` class and `CompoundButton` is the subclass of `Button` class.

There are different types of buttons in android such as `RadioButton`, `ToggleButton`, `CompoundButton` etc.

Android Button Example with Listener

Here, we are going to create two textfields and one button for sum of two numbers. If user clicks button, sum of two input values is displayed on the Toast.

We can perform action on button using different types such as calling listener on button or adding `onClick` property of button in activity's xml file.

```

button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        //code
    }
});

```

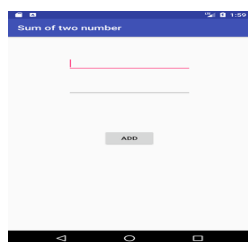
```

<Button
    android:onClick="methodName"
/>

```

Drag the component or write the code for UI in `activity_main.xml`

First of all, drag 2 textfields from the Text Fields palette and one button from the Form Widgets palette as shown in the following figure.



The generated code for the ui components will be like this:

File: `activity_main.xml`

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="example.javatpoint.com.sumoftwonumber.MainActivity">

    <EditText
        android:id="@+id/editText1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"

```



```

android:layout_centerHorizontal="true"
android:layout_marginTop="61dp"
android:ems="10"
android:inputType="number"
tools:layout_editor_absoluteX="84dp"
tools:layout_editor_absoluteY="53dp" />

```

<EditText

```

android:id="@+id/editText2"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_below="@+id/editText1"
android:layout_centerHorizontal="true"
android:layout_marginTop="32dp"
android:ems="10"
android:inputType="number"
tools:layout_editor_absoluteX="84dp"
tools:layout_editor_absoluteY="127dp" />

```

<Button

```

android:id="@+id/button"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_below="@+id/editText2"
android:layout_centerHorizontal="true"
android:layout_marginTop="109dp"
android:text="ADD"
tools:layout_editor_absoluteX="148dp"
tools:layout_editor_absoluteY="266dp" />

```

</RelativeLayout>

Activity class

Now write the code to display the sum of two numbers.

File: MainActivity.java

```

package example.javatpoint.com.sumoftwonumber;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {
    private EditText edittext1, edittext2;
    private Button buttonSum;

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

    public void addListenerOnButton() {

```

```

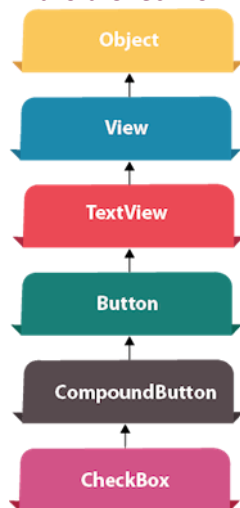
edittext1 = (EditText) findViewById(R.id.editText1);
edittext2 = (EditText) findViewById(R.id.editText2);
buttonSum = (Button) findViewById(R.id.button);

buttonSum.setOnClickListener(new View.OnClickListener() {
    public void onClick(View view) {
        String value1=edittext1.getText().toString();
        String value2=edittext2.getText().toString();
        int a=Integer.parseInt(value1);
        int b=Integer.parseInt(value2);
        int sum=a+b;
        Toast.makeText(getApplicationContext(),String.valueOf(sum), Toast.LENGTH_LONG).show();
    }
});
}
}

```

Q8. EXPLAIN DEVELOPING AN APPLICATION USING ANDROID CHECKBOX?

Android CheckBox Example



Android CheckBox is a type of two state button either checked or unchecked.

There can be a lot of usage of checkboxes. For example, it can be used to know the hobby of the user, activate/deactivate the specific action etc.

Android CheckBox class is the subclass of CompoundButton class.

Android CheckBox class

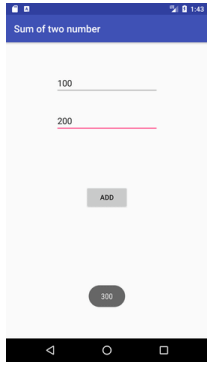
The android.widget.CheckBox class provides the facility of creating the CheckBoxes.

Methods of CheckBox class

There are many inherited methods of View, TextView, and Button classes in the CheckBox class. Some of them are as follows:

Method	Description
public boolean isChecked()	Returns true if it is checked otherwise false.
public void setChecked(boolean status)	Changes the state of the CheckBox.

Output:



Android CheckBox Example

activity_main.xml

Drag the three checkboxes and one button for the layout. Now the activity_main.xml file will look like this:

File: activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="example.javatpoint.com.checkbox.MainActivity">

    <CheckBox
        android:id="@+id/checkbox"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="144dp"
        android:layout_marginTop="68dp"
        android:text="Pizza"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <CheckBox
        android:id="@+id/checkbox2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="144dp"
        android:layout_marginTop="28dp"
        android:text="Coffee"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/checkbox" />

    <CheckBox
        android:id="@+id/checkbox3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="144dp"
        android:layout_marginTop="28dp"
        android:text="Burger"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/checkbox2" />

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="144dp"
        android:layout_marginTop="184dp"
        android:text="Order"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/checkbox3" />

</android.support.constraint.ConstraintLayout>
```

Activity class

Let's write the code to check which toggle button is ON/OFF.

File: MainActivity.java

```
package example.javatpoint.com.checkbox;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.CheckBox;
import android.widget.Toast;

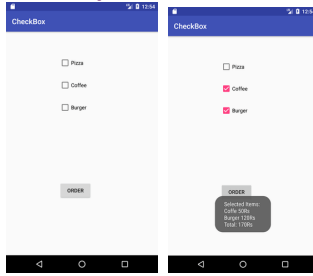
public class MainActivity extends AppCompatActivity {
    CheckBox pizza,coffe,burger;
    Button buttonOrder;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        addListenerOnButtonClick();
    }
    public void addListenerOnButtonClick(){
        //Getting instance of CheckBoxes and Button from the activity_main.xml file
        pizza=(CheckBox)findViewById(R.id.checkBox);
        coffe=(CheckBox)findViewById(R.id.checkBox2);
        burger=(CheckBox)findViewById(R.id.checkBox3);
        buttonOrder=(Button)findViewById(R.id.button);

        //Applying the Listener on the Button click
        buttonOrder.setOnClickListener(new View.OnClickListener(){

            public void onClick(View view) {
                int totalamount=0;
                StringBuilder result=new StringBuilder();
                result.append("Selected Items:");
                if(pizza.isChecked()){
                    result.append("\nPizza 100Rs");
                    totalamount+=100;
                }
                if(coffe.isChecked()){
                    result.append("\nCoffe 50Rs");
                    totalamount+=50;
                }
                if(burger.isChecked()){
                    result.append("\nBurger 120Rs");
                    totalamount+=120;
                }
                result.append("\nTotal: "+totalamount+"Rs");
                //Displaying the message on the toast
                Toast.makeText(getApplicationContext(), result.toString(), Toast.LENGTH_LONG).show();
            }

        });
    }
}
```

Output:



Q9. EXPLAIN DEVELOPING AN APPLICATION USING ANDROID CUSTOM CHECKBOX?

Android Custom CheckBox

Android provides facility to customize the UI of view elements rather than default.

You are able to create custom CheckBox in android. So, you can add some different images of checkbox on the layout.

Example of Custom CheckBox

In this example, we create both default as well as custom checkbox. Add the following code in activity_main.xml file.

activity_main.xml

File: activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="example.javatpoint.com.customcheckbox.MainActivity">
    <TextView
        android:id="@+id/textView1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:gravity="center_horizontal"
        android:textSize="25dp"
        android:text="Default Check Box"
        android:layout_alignParentTop="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true" />

    <CheckBox
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="New Check Box"
        android:id="@+id/checkBox"
        android:layout_below="@+id/textView1"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="46dp" />

    <CheckBox
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="New Check Box"
        android:id="@+id/checkBox2"
        android:layout_below="@+id/checkBox"
        android:layout_alignLeft="@+id/checkBox"
        android:layout_alignStart="@+id/checkBox" />
```


<View

```
android:layout_width="fill_parent"
android:layout_height="1dp"
android:layout_marginTop="200dp"
android:background="#B8B894"
android:id="@+id/viewStub" />
```

<CheckBox

```
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="CheckBox 1"
android:id="@+id/checkBox3"
android:button="@drawable/customcheckbox"
android:layout_below="@+id/viewStub"
android:layout_centerHorizontal="true"
android:layout_marginTop="58dp" />
```

<CheckBox

```
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="CheckBox 2"
android:id="@+id/checkBox4"
android:button="@drawable/customcheckbox"
android:layout_below="@+id/checkBox3"
android:layout_alignLeft="@+id/checkBox3"
android:layout_alignStart="@+id/checkBox3" />
```

<TextView

```
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:textAppearance="?android:attr/textAppearanceSmall"
android:textSize="25dp"
android:text="Custom Check Box"
android:id="@+id/textView"
android:layout_alignTop="@+id/viewStub"
android:layout_centerHorizontal="true" />
```

<Button

```
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Show Checked"
android:id="@+id/button"
android:layout_alignParentBottom="true"
android:layout_centerHorizontal="true" />
```

</RelativeLayout>

Now implement a selector in another file (checkbox.xml) under drawable folder which customizes the checkbox.

checkbox.xml

File: checkbox.xml

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:state_checked="true" android:drawable="@drawable/checked" />
```

```
<item android:state_checked="false" android:drawable="@drawable/unchecked"/>
```

1. </selector>

Activity class

File: MainActivity.java

```
package example.javatpoint.com.customcheckbox;
```

```
import android.support.v7.app.AppCompatActivity;
```

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

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

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

```
import android.widget.CheckBox;
```

```
import android.widget.Toast;
```

```
public class MainActivity extends AppCompatActivity {
```

```
    CheckBox cb1,cb2;
```

```
    Button button;
```

```
    @Override
```

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

```
        super.onCreate(savedInstanceState);
```

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

```
        cb1=(CheckBox)findViewById(R.id.checkBox3);
```

```
        cb2=(CheckBox)findViewById(R.id.checkBox4);
```

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

```
        button.setOnClickListener(new View.OnClickListener() {
```

```
            public void onClick(View v) {
```

```
                StringBuilder sb=new StringBuilder("");
```

```
                if(cb1.isChecked()){
```

```
                    String s1=cb1.getText().toString();
```

```
                    sb.append(s1);
```

```
                }
```

```
                if(cb2.isChecked()){
```

```
                    String s2=cb2.getText().toString();
```

```
                    sb.append("\n"+s2);
```

```
                }
```

```
                if(sb!=null && !sb.toString().equals("")){
```

```
                    Toast.makeText(getApplicationContext(), sb, Toast.LENGTH_LONG).show();
```

```
                }
```

```
            else{
```

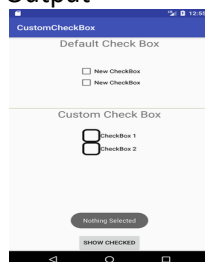
```
                Toast.makeText(getApplicationContext(), "Nothing Selected", Toast.LENGTH_LONG).show();
```

```
            }
```

```
        }    });
```

```
    } }
```

Output



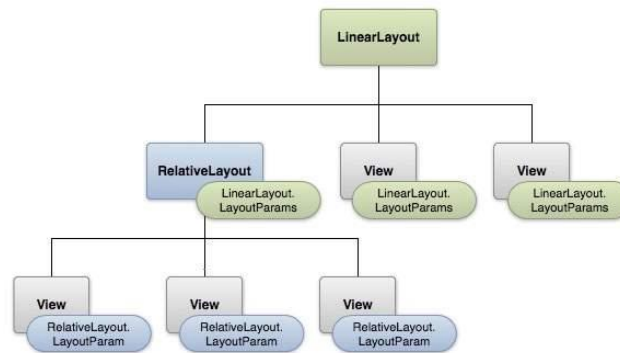
**Q10. EXPLAIN ABOUT USERINTERFACE LAYOUTS?HOW LAYOUTS CAN BE DESIGNED IN ANDROID?
OR EXPLAIN ABOUT VIEW OBJECT .HOW A VIEW OBJECT CAN FUNCTION IN ANDROID?**

Android - UI Layouts

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

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

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



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

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

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

    <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="I am a TextView" />

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

Layout params

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

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

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

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

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

```
</LinearLayout>
```

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

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

Android Layout Types

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

Sr.No	Layout & Description
1	<u>Linear Layout</u> : LinearLayout is a view group that aligns all children in a single direction, vertically or horizontally.
2	<u>Relative Layout</u> : RelativeLayout is a view group that displays child views in relative positions.
3	<u>Table Layout</u> : TableLayout is a view that groups views into rows and columns.
4	<u>Absolute Layout</u> : AbsoluteLayout enables you to specify the exact location of its children.
5	<u>Frame Layout</u> :The FrameLayout is a placeholder on screen that you can use to display a single view.
6	<u>List View</u> : ListView is a view group that displays a list of scrollable items.
7	<u>Grid View</u> : GridView is a ViewGroup that displays items in a two-dimensional, scrollable grid.

Layout Attributes

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

Sr.No	Attribute & Description
1	android:id : This is the ID which uniquely identifies the view.
2	android:layout_width : This is the width of the layout.
3	android:layout_height : This is the height of the layout
4	android:layout_marginTop : This is the extra space on the top side of the layout.
5	android:layout_marginBottom : This is the extra space on the bottom side of the layout.
6	android:layout_marginLeft : This is the extra space on the left side of the layout.
7	android:layout_marginRight : This is the extra space on the right side of the layout.
8	android:layout_gravity : This specifies how child Views are positioned.

9	android:layout_weight : This specifies how much of the extra space in the layout should be allocated to the View.
10	android:layout_x : This specifies the x-coordinate of the layout.
11	android:layout_y : This specifies the y-coordinate of the layout.
12	android:layout_width : This is the width of the layout.
13	android:paddingLeft : This is the left padding filled for the layout.
14	android:paddingRight : This is the right padding filled for the layout.
15	android:paddingTop : This is the top padding filled for the layout.
16	android:paddingBottom : This is the bottom padding filled for the layout.

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

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

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

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

Constant	Value	Description
top	0x30	Push object to the top of its container, not changing its size.
bottom	0x50	Push object to the bottom of its container, not changing its size.
left	0x03	Push object to the left of its container, not changing its size.
right	0x05	Push object to the right of its container, not changing its size.
center_vertical	0x10	Place object in the vertical center of its container, not changing its size.
fill_vertical	0x70	Grow the vertical size of the object if needed so it completely fills its container.
center_horizontal	0x01	Place object in the horizontal center of its container, not changing its size.
fill_horizontal	0x07	Grow the horizontal size of the object if needed so it completely fills its container.
center	0x11	Place the object in the center of its container in both the vertical and horizontal axis, not changing its size.
fill	0x77	Grow the horizontal and vertical size of the object if needed so it completely fills its container.
clip_vertical	0x80	Additional option that can be set to have the top and/or bottom edges of the child clipped to its container's bounds. The clip will be based on the vertical gravity: a top gravity will clip the bottom edge, a bottom gravity will clip the top edge, and neither will clip both edges.
clip_horizontal	0x08	Additional option that can be set to have the left and/or right edges of the child clipped to its container's bounds. The clip will be based on the horizontal gravity: a left gravity will clip the right edge, a right gravity will clip the left edge, and neither will clip both edges.

start	0x00800003	Push object to the beginning of its container, not changing its size.
end	0x00800005	Push object to the end of its container, not changing its size.

View Identification

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

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

Following is a brief description of @ and + signs –

- The at-symbol (@) at the beginning of the string indicates that the XML parser should parse and expand the rest of the ID string and identify it as an ID resource.
- The plus-symbol (+) means that this is a new resource name that must be created and added to our resources. To create an instance of the view object and capture it from the layout, use the following –

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

Q11. EXPLAIN ABOUT USERINTERFACE CONTROLS?HOW UI CONTROLS CAN BE USED IN ANDROID?

Input controls are the interactive components in user interface. Android provides a wide variety of controls you can use in your UI, such as buttons, text fields, seek bars, check box, zoom buttons, toggle buttons, and many more.



Android UI Controls

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

Sr.No.	UI Control & Description
1	<u>TextView</u> : This control is used to display text to the user.
2	<u>EditText</u> : EditText is a predefined subclass of TextView that includes rich editing capabilities.
3	<u>AutoCompleteTextView</u> : The AutoCompleteTextView is a view that is similar to EditText, except that it shows a list of completion suggestions automatically while the user is typing.
4	<u>Button</u> : A push-button that can be pressed, or clicked, by the user to perform an action.
5	<u>ImageButton</u> : An ImageButton is an AbsoluteLayout which enables you to specify the exact location of its children. This shows a button with an image (instead of text) that can be pressed or clicked by the user.
6	<u>CheckBox</u> :An on/off switch that can be toggled by the user. You should use check box when presenting users with a group of selectable options that are not mutually exclusive.
7	<u>ToggleButton</u> : An on/off button with a light indicator.
8	<u>RadioButton</u> :The RadioButton has two states: either checked or unchecked.
9	<u>RadioGroup</u> :A RadioGroup is used to group together one or more RadioButtons.
10	<u>ProgressBar</u> :The ProgressBar view provides visual feedback about some ongoing tasks, such as when you are performing a task in the background.
11	<u>Spinner</u> :A drop-down list that allows users to select one value from a set.
12	<u>TimePicker</u> :The TimePicker view enables users to select a time of the day, in either 24-hour mode or AM/PM mode.

13 DatePicker :The DatePicker view enables users to select a date of the day.

Creating UI Controls

Input controls are the interactive components in your app's user interface. Android provides a wide variety of controls you can use in your UI, such as buttons, text fields, seek bars, check box, zoom buttons, toggle buttons, and many more.

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

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

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

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

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

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

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

Q12. EXPLAIN ABOUT EVENT HANDLING? HOW EVENT HANDLING CAN BE DONE IN ANDROID?

Android - Event Handling

Events are a useful way to collect data about a user's interaction with interactive components of

Applications. Like button presses or screen touch etc. The Android framework maintains an event queue as first-in, first-out (FIFO) basis. You can capture these events in your program and take appropriate action as per requirements.

There are following three concepts related to Android Event Management –

- **Event Listeners** – An event listener is an interface in the View class that contains a single callback method. These methods will be called by the Android framework when the View to which the listener has been registered is triggered by user interaction with the item in the UI.
- **Event Listeners Registration** – Event Registration is the process by which an Event Handler gets registered with an Event Listener so that the handler is called when the Event Listener fires the event.
- **Event Handlers** – When an event happens and we have registered an event listener for the event, the event listener calls the Event Handlers, which is the method that actually handles the event.

Event Listeners & Event Handlers

Event Handler	Event Listener & Description
onClick()	OnClickListener() : This is called when the user either clicks or touches or focuses upon any widget like button, text, image etc. You will use onClick() event handler to handle such event.
onLongClick()	OnLongClickListener() : This is called when the user either clicks or touches or focuses upon any widget like button, text, image etc. for one or more seconds. You will use onLongClick() event handler to handle such event.

onFocusChange()	OnFocusChangeListener() : This is called when the widget loses its focus ie. user goes away from the view item. You will use onFocusChange() event handler to handle such event.
onKey()	OnFocusChangeListener() : This is called when the user is focused on the item and presses or releases a hardware key on the device. You will use onKey() event handler to handle such event.
onTouch()	OnTouchListener() : This is called when the user presses the key, releases the key, or any movement gesture on the screen. You will use onTouch() event handler to handle such event.
onMenuItemClick()	OnMenuItemClickListener() : This is called when the user selects a menu item. You will use onMenuItemClick() event handler to handle such event.
onCreateContextMenu()	onCreateContextMenuListener() : This is called when the context menu is being built(as the result of a sustained "long click")

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

Event Listeners Registration

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

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

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

An Example of Event Handling using Touch Mode Event

Touch Mode

Users can interact with their devices by using hardware keys or buttons or touching the screen. Touching the screen puts the device into touch mode. The user can then interact with it by touching the on-screen virtual buttons, images, etc. You can check if the device is in touch mode by calling the View class's `isInTouchMode()` method.

Focus

A view or widget is usually highlighted or displays a flashing cursor when it's in focus. This indicates that it's ready to accept input from the user.

- **isFocusable()** – it returns true or false
- **isFocusableInTouchMode()** – checks to see if the view is focusable in touch mode. (A view may be focusable when using a hardware key but not when the device is in touch mode)

- `android:focusUp="@id/button_1"`

onTouchEvent()

```
public boolean onTouchEvent(MotionEvent event){
    switch(event.getAction()){
        case TOUCH_DOWN:
            Toast.makeText(this,"you have clicked down Touch button",Toast.LENGTH_LONG).show();
            break();
    }
}
```

```

    case TOUCH_UP:
        Toast.makeText(this,"you have clicked up touch button",Toast.LENGTH_LONG).show();
        break;

    case TOUCH_MOVE:
        Toast.makeText(this,"you have clicked move touch button"Toast.LENGTH_LONG).show();
        break;
    }
    return super.onTouchEvent(event) ;
}

```

Event Handling Examples

Event Listeners Registration Using an Anonymous Inner Class

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

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

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

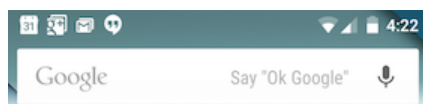
Step	Description
1	You will use Android studio IDE to create an Android application and name it as <i>myapplication</i> under a package <i>com.example.myapplication</i> as explained in the <i>Hello World Example</i> chapter.
2	Modify <i>src/MainActivity.java</i> file to add click event listeners and handlers for the two buttons defined.
3	Modify the default content of <i>res/layout/activity_main.xml</i> file to include Android UI controls.
4	No need to declare default string constants.Android studio takes care default constants.
5	Run the application to launch Android emulator and verify the result of the changes done in the application.

Q13. EXPLAIN ABOUT message Communication? HOW user can message or notifications can be done IN ANDROID? (OR) How alert message can be performed using Toast Class?

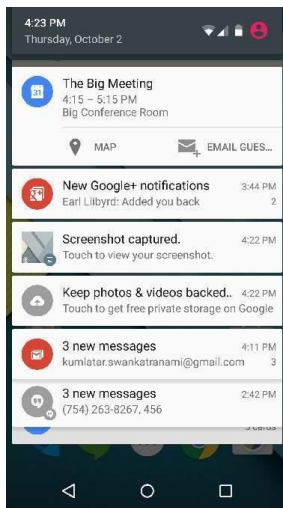
Android - Notifications

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

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



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



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

Create and Send Notifications

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

Step 1 - Create Notification Builder

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

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

Step 2 - Setting Notification Properties

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

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

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

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

Step 3 - Attach Actions

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

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

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

We take help of stack builder object which will contain an artificial back stack for the started Activity. This ensures that navigating backward from the Activity leads out of your application to the Home screen.

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

```
stackBuilder.addParentStack(ResultActivity.class);
```

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

Step 4 - Issue the notification

Finally, you pass the Notification object to the system by calling `NotificationManager.notify()` to send your notification. Make sure you call **NotificationCompat.Builder.build()** method on builder object before notifying it. This method combines all of the options that have been set and return a new **Notification** object.

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

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

The NotificationCompat.Builder Class

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

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

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

Example

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

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

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

```
package com.example.notificationdemo;

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

public class MainActivity extends Activity {
    Button b1;
    @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) {
        addNotification();
    }
});
}

private void addNotification() {
    NotificationCompat.Builder builder =
        new NotificationCompat.Builder(this)
            .setSmallIcon(R.drawable.abc)
            .setContentTitle("Notifications Example")
            .setContentText("This is a test notification");

    Intent notificationIntent = new Intent(this, MainActivity.class);
    PendingIntent contentIntent = PendingIntent.getActivity(this, 0, notificationIntent,
        PendingIntent.FLAG_UPDATE_CURRENT);
    builder.setContentIntent(contentIntent);

    // Add as notification
    NotificationManager manager = (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
    manager.notify(0, builder.build());
}
}

```

Following will be the content of **res/layout/notification.xml** file –

```

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

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="400dp"
        android:text="Hi, Your Detailed notification view goes here...." />
</LinearLayout>

```

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

```

package com.example.notificationdemo;

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

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

```


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

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

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Notification 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"
        android:layout_marginTop="48dp" />

    <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"
        android:layout_marginTop="42dp" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Notification"
        android:id="@+id/button"
        android:layout_marginTop="62dp"
        android:layout_below="@+id/imageButton"
        android:layout_centerHorizontal="true" />
</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="action_settings">Settings</string>
    <string name="app_name">tutorialspoint </string>
```



```
</resources>
```

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

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

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

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

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

        </activity>

        <activity android:name=".NotificationView"
            android:label="Details of notification"
            android:parentActivityName=".MainActivity">
            <meta-data
                android:name="android.support.PARENT_ACTIVITY"
                android:value=".MainActivity"/>
        </activity>

    </application>
</manifest>
```

Q14. EXPLAIN ABOUT RADIO BUTTONS? EXPLAIN WITH AN EXAMPLE HOW RADIO BUTTON CAN BE USED?

Android RadioButton

RadioButton is a two states button which is either checked or unchecked. If a single radio button is unchecked, we can click it to make checked radio button. Once a radio button is checked, it cannot be marked as unchecked by user.

RadioButton is generally used with *RadioGroup*. RadioGroup contains several radio buttons, marking one radio button as checked makes all other radio buttons as unchecked.

Example of Radio Button

In this example, we are going to implement single radio button separately as well as radio button in **RadioGroup**.

activity_main.xml

File: activity_main.xml

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

```
android:orientation="vertical"
tools:context="example.javatpoint.com.radiobutton.MainActivity">
```

<TextView

```
    android:id="@+id/textView1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="30dp"
    android:gravity="center_horizontal"
    android:textSize="22dp"
    android:text="Single Radio Buttons" />
```

```
<!-- Default RadioButtons -->
```

<RadioButton

```
    android:id="@+id/radioButton1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:text="Radio Button 1"
    android:layout_marginTop="20dp"
```

```
    android:textSize="20dp" />
```

<RadioButton

```
    android:id="@+id/radioButton2"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Radio Button 2"
    android:layout_marginTop="10dp"
    android:textSize="20dp" />
```

<View

```
    android:layout_width="fill_parent"
    android:layout_height="1dp"
    android:layout_marginTop="20dp"
    android:background="#B8B894" />
```

<TextView

```
    android:id="@+id/textView2"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="30dp"
    android:gravity="center_horizontal"
    android:textSize="22dp"
    android:text="Radio button inside RadioGroup" />
```

```
<!-- Customized RadioButtons -->
```

<RadioGroup

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/radioGroup">
```

<RadioButton

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

```

        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text=" Male"
        android:layout_marginTop="10dp"
        android:checked="false"
        android:textSize="20dp" />

```

<RadioButton

```

        android:id="@+id/radioFemale"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text=" Female"
        android:layout_marginTop="20dp"
        android:checked="false"
        android:textSize="20dp" />

```

</RadioGroup>

<Button

```

        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Show Selected"
        android:id="@+id/button"
        android:onClick="onclickbuttonMethod"
        android:layout_gravity="center_horizontal" />

```

</LinearLayout>

Activity class

File: MainActivity.java

package example.javatpoint.com.radiobutton;

import android.support.v7.app.AppCompatActivity;

import android.os.Bundle;

import android.view.View;

import android.widget.Button;

import android.widget.RadioButton;

import android.widget.RadioGroup;

import android.widget.Toast;

public class MainActivity **extends** AppCompatActivity {

Button button;

RadioButton genderradioButton;

RadioGroup radioGroup;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity_main);

radioGroup=(RadioGroup)findViewById(R.id.radioGroup);

}

public void onclickbuttonMethod(View v){

int selectedId = radioGroup.getCheckedRadioButtonId();

genderradioButton = (RadioButton) findViewById(selectedId);

if(selectedId==1){

Toast.makeText(MainActivity.this,"Nothing selected", Toast.LENGTH_SHORT).show();

}

else{

Toast.makeText(MainActivity.this,genderradioButton.getText(), Toast.LENGTH_SHORT).show();

```

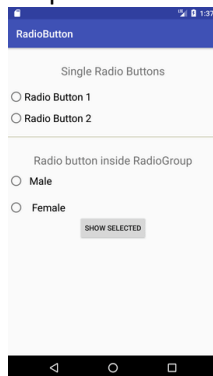
}

}

}

```

Output



Q15. EXPLAIN WITH AN EXAMPLE HOW EDIT TEXT CONTROLL CAN BE DONE IN ANDROID?

Android EditText with Examples

In android, **EditText** is a user interface control which is used to allow the user to enter or modify the text. While using **EditText** control in our android applications, we need to specify the type of data the text field can accept using the **inputType** attribute.

For example, if it accept plain text, then we need to specify the inputType as **"text"**. In case if **EditText** field is for password, then we need to specify the inputType as **"textPassword"**.

In android, **EditText** control is an extended version of **TextView** control with additional features and it is used to allow users to enter input values. In android, we can create **EditText** control in two ways either in XML layout file or create it in [Activity](#) file programmatically.

Create a EditText in Layout File

Following is the sample way to define **EditText** control in XML layout file in android application.

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <EditText
        android:id="@+id/txtSub"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Subject"
        android:inputType="text"/>
</LinearLayout>

```

If you observe above code snippet, here we defined **EditText** control to accept plain text by using inputType as **"text"** in xml layout file.

Create EditText Control in Activity File

In android, we can create **EditText** control programmatically in an [activity](#) file to allow users to enter text based on our requirements.

Following is the example of creating EditText control dynamically in an activity file.

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        LinearLayout linearLayout = (LinearLayout) findViewById(R.id.linearlayout);
        EditText et = new EditText(this);
        et.setHint("Subject");
        linearLayout.addView(et);
    }
}
```

Set the Text of Android EditText

In android, we can set the text of **EditText** control either while declaring it in **Layout** file or by using **setText()** method in Activity file.

Following is the example to set the text of TextView control while declaring it in XML Layout file.

```
<EditText
    android:id="@+id/editText1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Welcome to Tutlane" />
```

If you observe above example we used **android:text** property to the set required text for **EditText** control in XML Layout file.

Following is another way to set the text of **EditText** control programmatically in [activity](#) file using **setText()** method.

```
EditText et = (EditText)findViewById(R.id.editText1);
et.setText("Welcome to Tutlane");
```

If you observe above code snippet, we are finding the **EditText** control which we defined in XML layout file using **id** property and setting the text using **setText()** method.

Get Text of Android EditText

In android, we can get the text of **EditText** control by using **getText()** method in [Activity](#) file.

Following is the example to get text of **EditText** control programmatically in [activity](#) file using **getText()** method.

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        EditText et = (EditText) findViewById(R.id.txtSub);
        String name = et.getText().toString();
    }
}
```

If you observe above code snippet, we are finding the **EditText** control which we defined in XML layout file using **id** property and getting the text of **EditText** control using **getText()** method.

Android EditText Attributes

The following are some of the commonly used attributes related to EditText control in android applications.

Attribute	Description
-----------	-------------

Attribute	Description
android:id	It is used to uniquely identify the control
android:gravity	It is used to specify how to align the text like left, right, center, top, etc.
android:text	It is used to set the text.
android:hint	It is used to display the hint text when text is empty
android:textColor	It is used to change the color of the text.
android:textColorHint	It is used to change the text color of hint text.
android:textSize	It is used to specify the size of the text.
android:textStyle	It is used to change the style (bold, italic, bolditalic) of text.
android:background	It is used to set the background color for edit text control
android:ems	It is used to make the textview be exactly this many ems wide.
android:width	It makes the TextView be exactly this many pixels wide.
android:height	It makes the TextView be exactly this many pixels tall.
android:maxLength	It is used to make the TextView be at most this many pixels wide.
android:minWidth	It is used to make the TextView be at least this many pixels wide.
android:textAllCaps	It is used to present the text in all CAPS
android:typeface	It is used to specify the Typeface (normal, sans, serif, monospace) for the text.
android:textColorHighlight	It is used to change the color of text selection highlight.
android:inputType	It is used to specify the type of text being placed in text fields.
android:fontFamily	It is used to specify the fontFamily for the text.
android:editable	If we set false, EditText won't allow us to enter or modify the text

Android EditText Control Example

Following is the example of using multiple **EditText** controls with different input types like password, phone, etc. in [LinearLayout](#) to build an android application.

Create a new android application using android studio and give names as **EditTextExample**. In case if you are not aware of creating an app in android studio check this article [Android Hello World App](#).

Now open an **activity_main.xml** file from **\res\layout** path and write the code like as shown below

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="40dp"
    android:orientation="vertical" android:id="@+id/linearlayout" >
    <EditText
        android:id="@+id/txtName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="25dp"
        android:ems="10"
        android:hint="Name"
        android:inputType="text"
        android:selectAllOnFocus="true" />
    <EditText
        android:id="@+id/txtPwd"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:ems="10"
        android:hint="Password 0 to 9"
        android:inputType="numberPassword" />
    <EditText
        android:id="@+id/txtEmail"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:ems="10"
        android:hint="Email"
        android:inputType="textEmailAddress" />
    <EditText
        android:id="@+id/txtDate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/editText3"
        android:ems="10"
        android:hint="Date"
        android:inputType="date" />
    <EditText
        android:id="@+id/txtPhone"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:ems="10"
        android:hint="Phone Number"
        android:inputType="phone"
        android:textColorHint="#FE8DAB"/>
    <Button
        android:id="@+id/btnSend"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```

        android:text="submit"
        android:textSize="16sp"
        android:textStyle="normal|bold" />
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/resultView"
    android:layout_marginTop="25dp"
    android:textSize="15dp"/>
</LinearLayout>

```

If you observe above code we created multiple **EditText** controls with different **inputTypes**, such as password, email address, date, phone number, plain text.

Once we are done with the creation of layout with required controls, we need to load the XML layout resource from our [activity onCreate\(\)](#) callback method, for that open main activity file **MainActivity.java** from **\java\com.tutlane.editttextexample** path and write the code like as shown below.

MainActivity.java

```

package com.tutlane.editttextexample;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
import org.w3c.dom.Text;

public class MainActivity extends AppCompatActivity {
    Button btnSubmit;
    EditText name, password, email, dob, phoneno;
    TextView result;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        name=(EditText)findViewById(R.id.txtName);
        password = (EditText)findViewById(R.id.txtPwd);
        email = (EditText)findViewById(R.id.txtEmail);
        dob = (EditText)findViewById(R.id.txtDate);
        phoneno= (EditText)findViewById(R.id.txtPhone);
        btnSubmit = (Button)findViewById(R.id.btnSend);
        result = (TextView)findViewById(R.id.resultView);
        btnSubmit.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                if (name.getText().toString().isEmpty() || password.getText().toString().isEmpty()
                || email.getText().toString().isEmpty() || dob.getText().toString().isEmpty()
                || phoneno.getText().toString().isEmpty()) {
                    result.setText("Please Fill All the Details");
                } else {
                    result.setText("Name - " + name.getText().toString() + "\n" + "Password - "
                    + password.getText().toString()
                    + "\n" + "E-Mail - " + email.getText().toString() + "\n" + "DOB - " + dob.getText().toString()
                    + "\n" + "Contact - " + phoneno.getText().toString());
                }
            }
        })
    }
}

```



```

    }
    });
}
}

```

If you observe above code we are calling our layout using **setContentView** method in the form of **R.layout.layout_file_name** in our [activity](#) file. Here our xml file name is **activity_main.xml** so we used file name **activity_main** and we are getting the text of our **EditText** controls whenever we click on button.

Generally, during the launch of our [activity](#), the **onCreate()** callback method will be called by the android framework to get the required layout for an activity.

Output of Android EditText Example

When we run the above example using the android virtual device (AVD) we will get a result like as shown below.



If you observe the above result, the system displaying an appropriate on-screen keyboard for each **EditText** control based on the defined **inputType** attribute and displayed a message if we click on the button without entering details in fields.

Once we enter details in all fields and click on Button we will get a result like as shown below.

This is how we can use **EditText** control in android applications to allow the user to enter or modify the text based on our requirements.

UNIT-III

Android UI Layouts (Linear, Relative, Frame, Table, ListView, GridView, WebView)

Q1. EXPLAIN ABOUT UI LAYOUTS IN DETAIL?

In android, **Layout** is used to define the user interface for an app or [activity](#) and it will hold the UI elements that will appear to the user. The user interface in the android app is made with a collection of **View** and **ViewGroup** objects. Generally, the android apps will contain one or more activities and each activity is one screen of the app. The activities will contain multiple UI components and those UI components are the instances of **View** and **ViewGroup** subclasses.

The **View** is a base class for all UI components in android and it is used to create an interactive UI components such as [TextView](#), [EditText](#), [Checkbox](#), [Radio Button](#), etc. and it responsible for event handling and drawing.

The **ViewGroup** is a subclass of **View** and it will act as a base class for **layouts** and **layouts parameters**. The **ViewGroup** will provide an invisible containers to hold other Views or ViewGroups and to define the layout properties.

In android, we can define a layouts in two ways, those are

- Declare UI elements in XML
- Instantiate layout elements at runtime

The android framework will allow us to use either or both of these methods to define our application's UI.

Declare UI Elements in XML

In android, we can create layouts same like web pages in HTML by using default [Views and ViewGroups](#) in the XML file. The layout file must contain only one root element, which must be a **View** or **ViewGroup** object. Once we define the root element, then we can add additional layout objects or widgets as child elements to build the View hierarchy that defines our layout.

Following is the example of defining a layout in an XML file (**activity_main.xml**) using [LinearLayout](#) to hold a [TextView](#), [EditText](#), and [Button](#).

```
<?xml version="1.0" encoding="utf-8"?>
<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:id="@+id/fstTxt"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Enter Name"
    />
    <EditText
        android:id="@+id/name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:ems="10">
    </EditText>
    <Button
        android:id="@+id/getName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Get Name" />
</LinearLayout>
```

We need to create a layout files in **/res/layout** project directory, then only the layout files will compile properly.

Load XML Layout File from an Activity

Once we are done with the creation of layout, we need to load the XML layout resource from our **activity onCreate()** callback method like as shown below

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

If you observe above code we are calling our layout using **setContentView** method in the form of **R.layout.layout_file_name**. Here our xml file name is **activity_main.xml** so we used file name **activity_main**.

Generally, during the launch of our **activity**, **onCreate()** callback method will be called by android framework to get the required layout for an **activity**.

Instantiate Layout Elements at Runtime

If we want to instantiate layout elements at runtime, we need to create own custom **View** and **ViewGroup** objects programmatically with required layouts.

Following is the example of creating a layout using **LinearLayout** to hold a **TextView**, **EditText** and **Button** in an **activity** using custom **View** and **ViewGroup** objects programmatically.

```
public class MainActivity extends AppCompatActivity {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        TextView textView1 = new TextView(this);  
        textView1.setText("Name:");  
        EditText editText1 = new EditText(this);  
        editText1.setText("Enter Name");  
        Button button1 = new Button(this);  
        button1.setText("Add Name");  
        LinearLayout linearLayout = new LinearLayout(this);  
        linearLayout.addView(textView1);  
        linearLayout.addView(editText1);  
        linearLayout.addView(button1);  
        setContentView(linearLayout);  
    }  
}
```

By creating a custom **View** and **ViewGroup** programmatically, we can define a layouts based on our requirements in android applications.

Width and Height

When we define a layout using XML file we need to set width and height for every **View** and **ViewGroup** element using **layout_width** and **layout_height** attributes.

Following is the example of setting width and height for **View** and **ViewGroup** elements in XML layout file.

```
<?xml version="1.0" encoding="utf-8"?>  
<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:id="@+id/fstTxt"
```

```

    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Enter Name" />
</LinearLayout>

```

If you observe above example, we used different values to set **layout_width** and **layout_height**, those are

- match_parent
- wrap_content

If we set value **match_parent**, then the **View** or **ViewGroup** will try to match with parent width or height.

If we set value **wrap_content**, then the **View** or **ViewGroup** will try to adjust its width or height based on the content.

Android Layout Attributes

In android, like **layout_width** and **layout_height** we have a different type of attributes available for **View** and **ViewGroup** objects to define the appearance of layouts based on our requirements.

The following are some of the common layout attributes used in the android application.

Attribute	Description
android:id	It is used to uniquely identify the view and ViewGroups
android:layout_width	It is used to define the width for View and ViewGroup elements in a layout
android:layout_height	It is used to define the height for View and ViewGroup elements in a layout
android:layout_marginLeft	It is used to define the extra space in the left side for View and ViewGroup elements in a layout
android:layout_marginRight	It is used to define the extra space in right side for View and ViewGroup elements in layout
android:layout_marginTop	It is used to define the extra space on top for View and ViewGroup elements in layout
android:layout_marginBottom	It is used to define the extra space in the bottom side for View and ViewGroup elements in a layout
android:paddingLeft	It is used to define the left side padding for View and ViewGroup elements in layout files
android:paddingRight	It is used to define the right side padding for View and ViewGroup elements in layout files
android:paddingTop	It is used to define padding for View and ViewGroup elements in layout files on top side
android:paddingBottom	It is used to define the bottom side padding for View and ViewGroup elements in layout files

Attribute	Description
android:layout_gravity	It is used to define how child Views are positioned

Android Layout Types

We have a different type of layouts available in android to implement user interface for our android applications with different designs based on our requirements.

Following are the commonly used layouts in android applications to implement required designs.

- [Linear Layout](#)
- [Relative Layout](#)
- [Frame Layout](#)
- [Table Layout](#)
- [Web View](#)
- [List View](#)
- [Grid View](#)

Android Linear Layout

In android, [LinearLayout](#) is a **ViewGroup** subclass which is used to render all child **View** instances one by one either in a horizontal direction or vertical direction based on the orientation property.

Android Relative Layout

In android, [RelativeLayout](#) is a **ViewGroup** which is used to specify the position of child **View** instances relative to each other (Child A to the left of Child B) or relative to the parent (Aligned to the top of a parent).

Android Frame Layout

In android, [FrameLayout](#) is a **ViewGroup** subclass which is used to specify the position of **View** instances it contains on the top of each other to display only a single View inside the FrameLayout.

Android Table Layout

In android, [TableLayout](#) is a **ViewGroup** subclass which is used to display the child View elements in rows and columns.

Android Web View

In android, [WebView](#) is a browser that is used to display the web pages as a part of our activity layout.

Android List View

In android, [ListView](#) is a **ViewGroup** which is used to display scrollable single column list of items.

Android Grid View

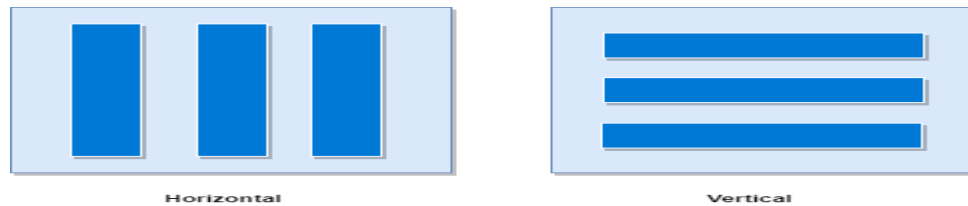
In android, [GridView](#) is a **ViewGroup** which is used to display items in a scrollable grid of columns and rows.

Android LinearLayout with Examples

In android, **LinearLayout** is a **ViewGroup** subclass which is used to render all child **View** instances one by one either in **Horizontal** direction or **Vertical** direction based on the **orientation** property.

In android, we can specify the linear layout orientation using **android:orientation** attribute.

Following is the pictorial representation of linear layout in android applications.



In **LinearLayout**, the child **View** instances arranged one by one, so the horizontal list will have only one row of multiple columns and vertical list will have one column of multiple rows.

Android LinearLayout Declaration

Following is the way we need to define the LinearLayout in android applications.

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

If you observe above code snippet, here we defined orientation as vertical, so this aligns all its child layout / views vertically.

Android LinearLayout Example

Following is the example of creating a **LinearLayout** with different controls in android application.

Create a new android application using android studio and give names as **LinearLayout**. In case if you are not aware of creating an app in android studio check this article [Android Hello World App](#).

Now open an **activity_main.xml** file from **\res\layout** path and write the code like as shown below

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="20dp"
    android:paddingRight="20dp"
    android:orientation="vertical" >
    <EditText
        android:id="@+id/txtTo"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="To"/>
    <EditText
        android:id="@+id/txtSub"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Subject"/>
```

```

<EditText
    android:id="@+id/txtMsg"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1"
    android:gravity="top"
    android:hint="Message"/>
<Button
    android:layout_width="100dp"
    android:layout_height="wrap_content"
    android:layout_gravity="right"
    android:text="Send"/>
</LinearLayout>

```

Once we are done with creation of layout, we need to load the XML layout resource from our [activity onCreate\(\)](#) callback method, for that open main activity file **MainActivity.java** from `\java\com.tutlane.linearlayout` path and write the code like as shown below.

MainActivity.java

```

package com.tutlane.linearlayout;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

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

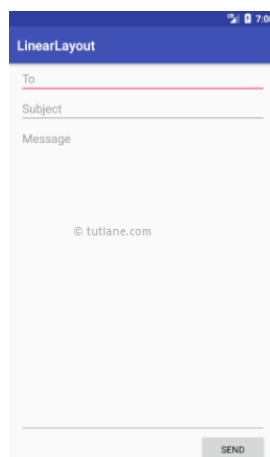
```

If you observe above code we are calling our layout using **setContentView** method in the form of **R.layout.layout_file_name**. Here our xml file name is **activity_main.xml** so we used file name **activity_main**.

Generally, during the launch of our [activity](#), the **onCreate()** callback method will be called by the android framework to get the required layout for an activity.

Output of Android LinearLayout Example

When we run above example using the android virtual device (AVD) we will get a result like as shown below.



Layout Weight Attribute

If you observe the above example we used layout **weight** attribute (`android:layout_weight`) in child view. Actually, this attribute is used by child views to specify how much space the **View** should occupy on the screen.

If we assign a larger weight value to the child view, then it will expand to fill any remaining space in the parent view.

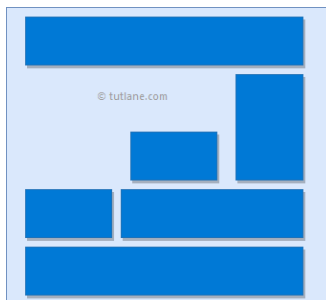
If you observe above example, we used three text fields and we assigned weight value to only one text field. The two text fields without weight will occupy only the area required for its content and the other text field with weight value will expand to fill the remaining space after all three fields measured.

This is how we can use **LinearLayout** in android applications to render all **View** instances one by one either in **Horizontal** direction or **Vertical** direction based on the **orientation** property.

Android RelativeLayout with Examples

In android, **RelativeLayout** is a **ViewGroup** which is used to specify the position of child **View** instances relative to each other (Child **A** to the left of Child **B**) or relative to the parent (Aligned to the top of parent).

Following is the pictorial representation of relative layout in android applications.



In android, **RelativeLayout** is very useful to design user interface because by using relative layout we can eliminate the nested view groups and keep our layout hierarchy flat, which improves the performance of application.

Android Positioning Views in Relative Layout

As we discussed, in **RelativeLayout** we need to specify the position of child views relative to each other or relative to the parent. In case if we didn't specify the position of child views, by default all child views are positioned to top-left of the layout.

Following are the some of most useful layout properties available to views in RelativeLayout.

Attribute	Description
layout_alignParentTop	If it specified "true", the top edge of view will match the top edge of the parent.
layout_alignParentBottom	If it specified "true", the bottom edge of view will match the bottom edge of parent.
layout_alignParentLeft	If it specified "true", the left edge of view will match the left edge of parent.
layout_alignParentRight	If it specified "true", the right edge of view will match the right edge of the parent.
layout_centerInParent	If it specified "true", the view will be aligned to the centre of parent.
layout_centerHorizontal	If it specified "true", the view will be horizontally centre aligned within its parent.
layout_centerVertical	If it specified "true", the view will be vertically centre aligned within its parent.
layout_above	It accepts another sibling view id and places the view above the specified view id.
layout_below	It accepts another sibling view id and places the view below the specified view id.
layout_toLeftOf	It accepts another sibling view id and places the view left of the specified view id.
layout_toRightOf	It accepts another sibling view id and places the view right of the specified view id.
layout_toStartOf	It accepts another sibling view id and places the view to start of the specified view id.
layout_toEndOf	It accepts another sibling view id and places the view to the end of the specified view id.

Android RelativeLayout Example

Following is the example of creating a **RelativeLayout** with different controls in android application.

Create a new android application using android studio and give names as **RelativeLayout**

Now open an **activity_main.xml** file from **\res\layout** path and write the code like as shown below

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="10dp"
    android:paddingRight="10dp">
    <Button
        android:id="@+id/btn1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:text="Button1" />
    <Button
        android:id="@+id/btn2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_centerVertical="true"
        android:text="Button2" />
    <Button
        android:id="@+id/btn3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_centerVertical="true"
        android:text="Button3" />

    <Button
        android:id="@+id/btn4"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:text="Button4" />
    <Button
        android:id="@+id/btn5"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBottom="@+id/btn2"
        android:layout_centerHorizontal="true"
        android:text="Button5" />
    <Button
        android:id="@+id/btn6"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_above="@+id/btn4"
        android:layout_centerHorizontal="true"
        android:text="Button6" />
    <Button
        android:id="@+id/btn7"
        android:layout_width="wrap_content"
```

```
android:layout_height="wrap_content"
android:layout_toEndOf="@+id/btn1"
android:layout_toRightOf="@+id/btn1"
android:layout_alignParentRight="true"
android:text="Button7" />
```

</RelativeLayout>

Once we are done with creation of layout, we need to load the XML layout resource from our [activity onCreate\(\)](#) callback method, for that open main activity file **MainActivity.java** from `\java\com.tutlane.relativelayout` path and write the code like as shown below.

MainActivity.java

```
package com.tutlane.linearlayout;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

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

If you observe above code we are calling our layout using **setContentView** method in the form of **R.layout.layout_file_name**. Here our xml file name is **activity_main.xml** so we used file name **activity_main**.

Generally, during the launch of our [activity, onCreate\(\)](#) callback method will be called by the android framework to get the required layout for an [activity](#).

Output of Android RelativeLayout Example

When we run above example using the android virtual device (AVD) we will get a result like as shown below.

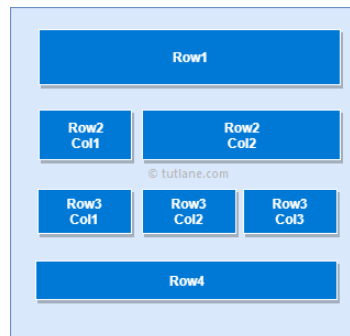


This is how we can use RelativeLayout in android applications based on our requirements.

Android TableLayout with Examples

In android, **TableLayout** is a **ViewGroup** subclass that is used to display the child View elements in rows and columns.

Following is the pictorial representation of table layout in android applications.



In android, TableLayout will position its children elements into rows and columns and it won't display any border lines for rows, columns or cells.

The TableLayout in android will work same as the HTML table and the table will have as many columns as the row with the most cells. The TableLayout can be explained as **<table>** and TableRow is like **<tr>** element.

Android TableLayout Example

Following is the example of creating a **TableLayout** with different controls in android application.

Create a new android application using android studio and give names as **TableLayout**.

Now open an **activity_main.xml** file from **\res\layout** path and write the code like as shown below

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginTop="100dp"
    android:paddingLeft="10dp"
    android:paddingRight="10dp" >
    <TableRow android:background="#0079D6" android:padding="5dp">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="UserId" />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="User Name" />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Location" />
    </TableRow>
    <TableRow android:background="#DAE8FC" android:padding="5dp">
        <TextView
```

```

        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="1" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Suresh Dasari" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Hyderabad" />
</TableRow>
<TableRow android:background="#DAE8FC" android:padding="5dp">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="2" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Rohini Alavala" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Guntur" />
</TableRow>
<TableRow android:background="#DAE8FC" android:padding="5dp">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="3" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Trishika Dasari" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Guntur" />
</TableRow>
</TableLayout>

```

Once we are done with creation of layout, we need to load the XML layout resource from our [activity onCreate\(\)](#) callback method, for that open main activity file **MainActivity.java** from **\java\com.tutlane.tablelayout** path and write the code like as shown below.

MainActivity.java

```
package com.tutlane.linearlayout;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

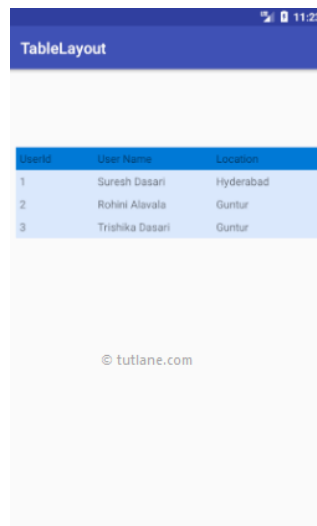
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

If you observe above code, we are calling our layout using **setContentView** method in the form of **R.layout.layout_file_name**. Here our xml file name is **activity_main.xml** so we used file name **activity_main**.

Generally, during the launch of our [activity](#), **onCreate()** callback method will be called by android framework to get the required layout for an [activity](#).

Output of Android TableLayout Example

When we run above example using the android virtual device (AVD) we will get a result like as shown below.



Userid	User Name	Location
1	Suresh Dasari	Hyderabad
2	Rohini Alavala	Guntur
3	Trishika Dasari	Guntur

© tutlane.com

This is how we can use the table layout in android applications based on our requirements.

Android FrameLayout with Examples

In android, **FrameLayout** is a **ViewGroup** subclass that is used to specify the position of **View** instances it contains on the top of each other to display only single **View** inside the FrameLayout.

In simple manner, we can say FrameLayout is designed to block out an area on the screen to display a single item.

Following is the pictorial representation of frame layout in android applications.



In android, **FrameLayout** will act as a placeholder on the screen and it is used to hold a single child view.

In FrameLayout, the child views are added in a stack and the most recently added child will show on the top. We can add multiple children views to FrameLayout and control their position by using gravity attributes in FrameLayout.

Android FrameLayout Example

Following is the example of creating a **FrameLayout** with different controls in android application.

Create a new android application using android studio and give names as **FrameLayout**.

Now open an **activity_main.xml** file from **\res\layout** path and write the code like as shown below

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <ImageView
        android:id="@+id/imgvw1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:scaleType="centerCrop"
        android:src="@drawable/flimg" />
    <TextView
        android:id="@+id/txtvw1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="40dp"
        android:background="#4C374A"
        android:padding="10dp"
        android:text="Grand Palace, Bangkok"
        android:textColor="#FFFFFF"
        android:textSize="20sp" />
    <TextView
        android:id="@+id/txtvw2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="right|bottom"
```

```

        android:background="#AA000000"
        android:padding="10dp"
        android:text="21/Aug/2017"
        android:textColor="#FFFFFF"
        android:textSize="18sp" />
</FrameLayout>

```

If you observe above code we used **ImageView** to show the image (**flimg**) from drawable folder in framelayout. So add your image to **drawable** folder and replace **@drawable/flimg** path with your image path.

Once we are done with the creation of layout, we need to load the XML layout resource from our **activity onCreate()** callback method, for that open main activity file **MainActivity.java** from **\java\com.tutlane.framelayout** path and write the code like as shown below.

MainActivity.java

```

package com.tutlane.linearlayout;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

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

```

If you observe above code, we are calling our layout using **setContentView** method in the form of **R.layout.layout_file_name**. Here our xml file name is **activity_main.xml** so we used file name **activity_main**.

Generally, during the launch of our **activity**, the **onCreate()** callback method will be called by the android framework to get the required layout for an **activity**.

Output of Android FrameLayout Example

When we run the above example using the android virtual device (AVD) we will get a result like as shown below.



This is how we can use frame layout in android applications based on our requirements.

Android ListView with Examples

In android, **ListView** is a **ViewGroup** that is used to display the list of scrollable of items in multiple rows and the list items are automatically inserted to the list using an **adapter**.

Generally, the adapter pulls data from sources such as an array or database and converts each item into a result view and that's placed into the list.

Following is the pictorial representation of listview in android applications.



Android Adapter

In android, **Adapter** will act as an intermediate between the data sources and adapter views such as [ListView](#), [GridView](#) to fill the data into adapter views. The adapter will hold the data and iterates through an items in data set and generate the views for each item in the list.

Generally, in android we have a different types of adapters available to fetch the data from different data sources to fill the data into adapter views, those are

Adapter	Description
ArrayAdapter	It will expects an Array or List as input.
CurosrAdapter	It will accepts an instance of cursor as an input.
SimpleAdapter	It will accepts a static data defined in the resources.
BaseAdapter	It is a generic implementation for all three adapter types and it can be used for ListView , GridView or Spinners based on our requirements

Android ListView Example

Following is the example of creating a **ListView** using **arrayadapter** in android application.

Create a new android application using android studio and give names as **ListView**.

Now open an **activity_main.xml** file from **\res\layout** path and write the code like as shown below

activity_main.xml

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

Once we are done with creation of layout, now we will bind data to our **ListView** using **ArrayAdapter**, for that open main activity file **MainActivity.java** from **\java\com.tutlane.listview** path and write the code like as shown below.

MainActivity.java

```
package com.tutlane.listview;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.ArrayAdapter;
import android.widget.ListView;

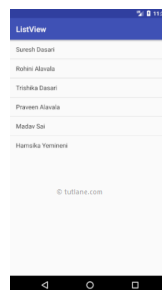
public class MainActivity extends AppCompatActivity {
    private ListView mListView;
    private ArrayAdapter aAdapter;
    private String[] users = { "Suresh Dasari", "Rohini Alavala", "Trishika Dasari", "Praveen Alavala", "Madav Sai", "Hamsika Yemineni" };
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mListView = (ListView) findViewById(R.id.userlist);
        aAdapter = new ArrayAdapter(this, android.R.layout.simple_list_item_1, users);
        mListView.setAdapter(aAdapter);
    }
}
```

If you observe above code, we are binding static array (**users**) details to **ListView** using **ArrayAdapter** and calling our layout using **setContentView** method in the form of **R.layout.layout_file_name**. Here our xml file name is **activity_main.xml** so we used file name **activity_main**.

Generally, during the launch of our [activity](#), onCreate() callback method will be called by the android framework to get the required layout for an [activity](#).

Output of Android ListView Example

When we run above example using android virtual device (AVD) we will get a result like as shown below.



This is how we can bind data to ListView using ArrayAdapter in android applications based on our requirements.

Android ListView with Custom Adapter Example

In previous example, we learned a simple way to bind data to **ListView** using **ArrayAdapter** in the android application. Now we will see how to create our own custom adapter and bind data to ListView with example.

For this, we need to create our own custom adapter class by extending with the **BaseAdapter** class and create a class that will contain parameters for list row items.

Now create a new android application using an android studio and give names as **ListView**. Open an **activity_main.xml** file from \res\layout path and write the code like as shown below

activity_main.xml

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

```

android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:orientation="vertical" >
<ListView
    android:id="@+id/user_list"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:dividerHeight="1dp" />
</LinearLayout>

```

Now we need to create a layout for **listview** row items, for that right click on **layouts** folder à select New à Layout resource file à Give name as **list_row.xml** and click **OK**. Now open newly created file (**list_row.xml**) and write the code like as shown below

list_row.xml

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:padding="5dip" >
    <TextView
        android:id="@+id/name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textStyle="bold"
        android:textSize="17dp" />
    <TextView
        android:id="@+id/designation"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/name"
        android:layout_marginTop="7dp"
        android:textColor="#343434"
        android:textSize="14dp" />
    <TextView
        android:id="@+id/location"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBaseline="@+id/designation"
        android:layout_alignBottom="@+id/designation"
        android:layout_alignParentRight="true"
        android:textColor="#343434"
        android:textSize="14dp" />
</RelativeLayout>

```

Now we need to create a custom class (**ListItem.java**) to represent each row in the list, for that right click on **java** folder à select New à Java Class à Give name as **ListItem.java** and click **OK**. Open **ListItem.java** file and write the code like as shown below

ListItem.java

```

package com.tutlane.listview;
public class ListItem {
    private String name;
    private String designation;
    private String location;
    public String getName() {
        return name;
    }
}

```

```

    }
    public void setName(String name) {
        this.name = name;
    }
    public String getDesignation() {
        return designation;
    }
    public void setDesignation(String designation) {
        this.designation = designation;
    }
    public String getLocation() {
        return location;
    }
    public void setLocation(String location) {
        this.location = location;
    }
}

```

Now we need to create a custom adapter (**CustomListAdapter.java**) and extend it by using **BaseAdapter**. In case if we are extending our class by using **BaseAdapter**, we need to override following methods from **BaseAdapter** class.

Method	Description
getCount()	It will return total number of rows count in listview
getItem()	It is used to specify the object data of each row
getItemId()	It returns the id of each row item
getView()	It is used to return a view instance that represents a single row in ListView item.

To create custom adapter right-click on **java** folder à select New à Java Class à Give name as **CustomListAdapter.java** and click **OK**.

Open **CustomListAdapter.java** file and write the code like as shown below

CustomListAdapter.java

```

package com.tutlane.listview;
import android.content.Context;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.BaseAdapter;
import android.widget.TextView;

import java.util.ArrayList;
public class CustomListAdapter extends BaseAdapter {
    private ArrayList<ListItem> listData;
    private LayoutInflater inflater;
    public CustomListAdapter(Context aContext, ArrayList<ListItem> listData) {
        this.listData = listData;
        inflater = LayoutInflater.from(aContext);
    }
    @Override
    public int getCount() {
        return listData.size();
    }
}

```

If you observe above class we are extending our custom adapter by using **BaseAdapter** and we override all **BaseAdapter** methods in our custom adapter.

Open main activity file (**MainActivity.java**) and write the code like as shown below.

```
package com.tutlane.listview;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ListView;
import android.widget.Toast;

import java.util.ArrayList;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```

setContentView(R.layout.activity_main);
ArrayList userList = getListData();
final ListView lv = (ListView) findViewById(R.id.user_list);
lv.setAdapter(new CustomListAdapter(this, userList));
lv.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> a, View v, int position, long id) {
        ListItem user = (ListItem) lv.getItemAtPosition(position);
        Toast.makeText(MainActivity.this, "Selected : " + " " + user.getName()+" , "+ user.getLocation(),
Toast.LENGTH_SHORT).show();
    }
});
}
private ArrayList getListData() {
    ArrayList<ListItem> results = new ArrayList<>();
    ListItem user1 = new ListItem();
    user1.setName("Suresh Dasari");
    user1.setDesignation("Team Leader");
    user1.setLocation("Hyderabad");
    results.add(user1);
    ListItem user2 = new ListItem();
    user2.setName("Rohini Alavala");
    user2.setDesignation("Agricultural Officer");
    user2.setLocation("Guntur");
    results.add(user2);
    ListItem user3 = new ListItem();
    user3.setName("Trishika Dasari");
    user3.setDesignation("Chartered Accountant");
    user3.setLocation("Guntur");
    results.add(user3);
    return results;
}
}

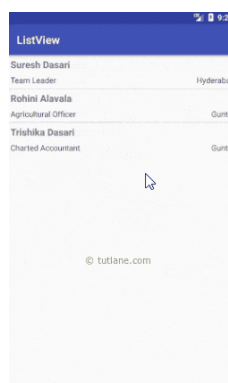
```

If you observe above code we are building and binding data to ListView using our custom adapter and calling our layout using **setContentView** method in the form of **R.layout.layout_file_name**. Here our xml file name is **activity_main.xml** so we used file name **activity_main**.

Generally, during the launch of our [activity](#), **onCreate()** callback method will be called by android framework to get the required layout for an [activity](#).

Output of Android Custom ListView Example

When we run the above example using an android virtual device (AVD) we will get a result like as shown below.



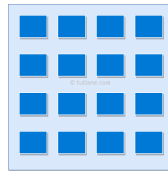
This is how we can bind data to ListView using custom adapter in android applications based on our requirements.

Android GridView with Examples

In android, **Grid View** is a **ViewGroup** that is used to display items in a two dimensional, scrollable grid and grid items are automatically inserted to the gridview layout using a list adapter.

Generally, the adapter pulls data from sources such as an array or database and converts each item into a result view and that's placed into the list.

Following is the pictorial representation of GridView in android applications.



Android Adapter

In android, **Adapter** will act as an intermediate between the data sources and adapter views such as [ListView](#), [Gridview](#) to fill the data into adapter views. The adapter will hold the data and iterates through items in the data set and generate the views for each item in the list.

Generally, in android we have a different types of adapters available to fetch the data from different data sources to fill the data into adapter views, those are

Adapter	Description
ArrayAdapter	It will expect an Array or List as input.
CurosrAdapter	It will accept an instance of a cursor as an input.
SimpleAdapter	It will accept a static data defined in the resources.
BaseAdapter	It is a generic implementation for all three adapter types and it can be used for ListView , Gridview or Spinners based on our requirements

Android GridView Example

Following is the simple example showing user details using **GridView** and showing the position of a particular image when clicking on it in android applications.

Create a new android application using android studio and give names as **GridView**.

Once we create an application, add some sample images to project **/res/drawable** directory to show the images in GridView.

Now open an **activity_main.xml** file from **/res/layout** path and write the code like as shown below

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<GridView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gridview"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:columnWidth="110dp"
    android:numColumns="auto_fit"
    android:verticalSpacing="10dp"
    android:horizontalSpacing="10dp"
    android:stretchMode="columnWidth"
    android:gravity="center" />
```


Once we are done with creation of layout, we need to create a custom adapter (**ImageAdapter.java**) by extending it using **BaseExtender** to show all the items in the grid, for that right click on **java** folder à Give name as **ImageAdapter.java** and click **OK**.

Open **ImageAdapter.java** file and write the code like as shown below

ImageAdapter.java

```
package com.tutlane.gridview;
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;
    public ImageAdapter(Context c) {
        mContext = c;
    }
    public int getCount() {
        return thumbImages.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 = new ImageView(mContext);
        imageView.setLayoutParams(new GridView.LayoutParams(200, 200));
        imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
        imageView.setPadding(8, 8, 8, 8);
        imageView.setImageResource(thumbImages[position]);
        return imageView;
    }
    // Add all our images to arraylist
    public Integer[] thumbImages = {
        R.drawable.img1, R.drawable.img2,
        R.drawable.img3, R.drawable.img4,
        R.drawable.img5, R.drawable.img6,
        R.drawable.img7, R.drawable.img8,
        R.drawable.img1, R.drawable.img2,
        R.drawable.img3, R.drawable.img4,
        R.drawable.img5, R.drawable.img6,
        R.drawable.img7, R.drawable.img8,
        R.drawable.img1, R.drawable.img2,
        R.drawable.img3, R.drawable.img4,
        R.drawable.img5
    };
}
```

If you observe above code we referred some images, actually those are the sample images which we added in **/res/drawable** directory.

Now we will bind images to our **GridView** using our custom adapter (**ImageAdapter.java**), for that open main activity file **MainActivity.java** from **\java\com.tutlane.gridview** path and write the code like as shown below.

MainActivity.java

```
package com.tutlane.gridview;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.Toast;

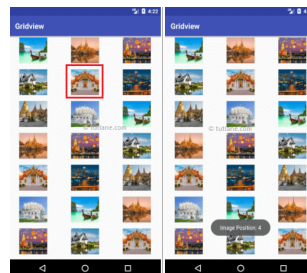
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        GridView gv = (GridView) findViewById(R.id.gvDetails);
        gv.setAdapter(new ImageAdapter(this));
        gv.setOnItemClickListener(new AdapterView.OnItemClickListener() {
            public void onItemClick(AdapterView<?> parent, View v, int position, long id) {
                Toast.makeText(MainActivity.this, "Image Position: " + position, Toast.LENGTH_SHORT).show();
            }
        });
    }
}
```

If you observe above code, we are binding image details to **GridView** using our custom adapter (**ImageAdapter.java**) and calling our layout using **setContentView** method in the form of **R.layout.layout_file_name**. Here our xml file name is **activity_main.xml** so we used file name **activity_main**.

Generally, during the launch of our [activity](#), **onCreate()** callback method will be called by android framework to get the required layout for an [activity](#).

Output of Android GridView Example

When we run above example using the android virtual device (AVD) we will get a result like as shown below.



This is how we can bind images to GridView using Adapter in android applications based on our requirements.

Android GridView Details Activity Example

In above example, we implemented an image gallery using gridview in android application. Now we will extend the functionality of above example to show the selected grid image in full screen.

Now we need to create a new layout (**image_details.xml**) file in project **/res/layout** directory to show the image details, for that right click on the layouts folder à select New à Layout resource file à Give name as **image_details.xml** and click **OK**. Now open newly created file (**image_details.xml**) and write the code like as shown below.

image_details.xml

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

Now we need to create a custom activity (**FullImageActivity.java**) to show the image details in our newly created layout (**image_details.xml**) file, for that right click on java folder à select New à Java Class à Give name as **FullImageActivity.java** and click **OK**.

Open **FullImageActivity.java** file and write the code like as shown below

FullImageActivity.java

```
package com.tutlane.gridview;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.widget.ImageView;
public class FullImageActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.image_details);
        // Get intent data
        Intent i = getIntent();
        // Get Selected Image Id
        int position = i.getExtras().getInt("id");
        ImageAdapter imageAdapter = new ImageAdapter(this);
        ImageView imageView = (ImageView) findViewById(R.id.full_image_view);
        imageView.setImageResource(imageAdapter.thumbImages[position]);
    }
}
```

Now we need to include our newly created activity file (**FullImageActivity.java**) in **AndroidManifest.xml** file like as shown below. For that, open **AndroidManifest.xml** file and write the code like as shown below

AndroidManifest.xml

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        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>
```

```

    </activity>
    <!-- FullImageActivity -->
    <activity android:name=".FullImageActivity"></activity>
</application>
</manifest>

```

Now we need to modify gridview image click function in main activity file (**MainActivity.java**) to get image details and show it in new activity.

Open main activity file (**MainActivity.java**) and write the code like as shown below.

MainActivity.java

```

package com.tutlane.gridview;
import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.GridView;

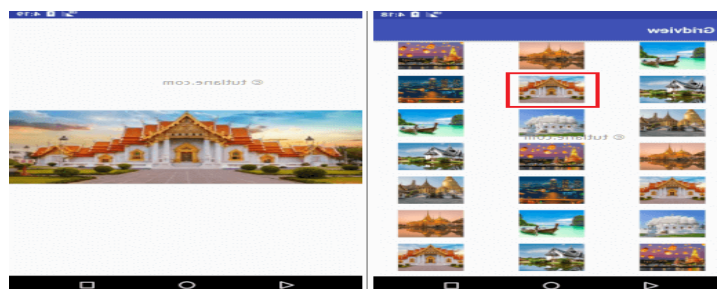
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        GridView gv = (GridView) findViewById(R.id.gvDetails);
        gv.setAdapter(new ImageAdapter(this));
        gv.setOnItemClickListener(new AdapterView.OnItemClickListener() {
            public void onItemClick(AdapterView<?> parent, View v, int position, long id) {
                // Sending image id to FullScreenActivity
                Intent i = new Intent(getApplicationContext(), FullImageActivity.class);
                // passing array index
                i.putExtra("id", position);
                startActivity(i);
            }
        });
    }
}

```

If you observe above code, we are getting the selected image details on image click and sending those details to our newly created activity file to show the image in full screen.

Output of Android GridView Details Activity Example

When we run above example using android virtual device (AVD) we will get a result like as shown below.



This is how we can build image gallery in gridview and show the selected image in android applications based on our requirements.

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. In other words, the `onCreate()` method of the activity is fired whenever there is a change in screen orientation.

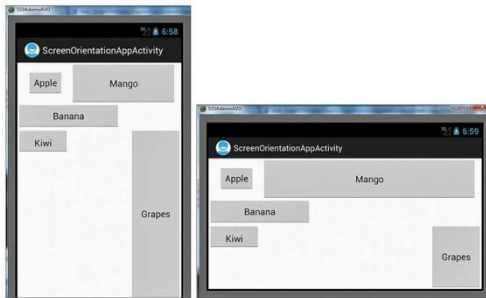
Portrait mode is longer in height and smaller in width, whereas landscape mode is wider but smaller in height. Being wider, landscape mode has more empty space on the right side of the screen. At the same time, some of the controls don't appear because of the smaller height. Thus, controls need to be laid out differently in the two screen orientations because of the difference in the height and width of the two orientations.

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. Let's examine this method by creating an Android project called `ScreenOrientationApp`. To lay out the controls at locations relative to the four edges of the screen, write the code in the layout file `activity_screen_orientation_app.xml` as shown in Listing 3.15.



Listing 3.15 The Layout file `activity_screen_orientation_app.xml` on Laying Out Controls Relative to the Four Edges of the Screen

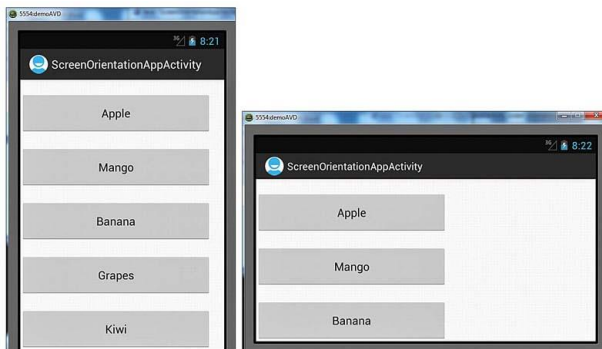
```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <Button
        android:id="@+id/Apple"
        android:text="Apple"
        android:layout_width="wrap_content"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

```

Listing 3.15 shows five Button controls arranged in a RelativeLayout container. The controls are aligned relative to the edges of the container or in relation to each other. Let's keep the activity file ScreenOrientationAppActivity.java unchanged with the default code, as shown in Listing 3.16.



Listing 3.16 Default Code in the Java Activity File ScreenOrientationAppActivity.java

```

package com.androidunleashed.screenorientationapp;
import android.app.Activity;
import android.os.Bundle;

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

```

```
}  
}
```

When the application is run while in the default Portrait mode, the controls appear as shown in Figure 3.16 (left). Because the five Button controls are placed in relation to the four edges of the container and in relation to each other, none of the Button controls disappear if the screen is rotated to Landscape mode, as shown in Figure 3.16 (right). To switch between portrait mode and landscape mode on the device emulator, press the Ctrl+F11 keys.



Figure 3.16 (left) Controls in Portrait mode, and (right) the controls in Landscape mode

Now that we understand the concept of adapting to screen orientation through anchoring controls, let's have a look at another approach.

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. To understand this, let's write code as shown in Listing 3.17 for laying out the controls for portrait mode in the default layout file `activity_screen_orientation_app.xml` (found in the `res/layout` folder).

Listing 3.17 The Layout File `activity_screen_orientation_app.xml` on Laying Out Controls in Portrait Mode

```
<LinearLayout 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="300dp"  
        android:layout_height="wrap_content"  
        android:padding="20dp"  
        android:layout_marginTop="20dp" />  
    <Button  
        android:id="@+id/Mango"  
        android:text="Mango"  
        android:layout_width="300dp"  
        android:layout_height="wrap_content"  
        android:padding="20dp"  
        android:layout_marginTop="20dp" />  
    <Button  
        android:id="@+id/Banana"  
        android:text="Banana"  
        android:layout_width="300dp"  
        android:layout_height="wrap_content"  
        android:padding="20dp"  
        android:layout_marginTop="20dp" />  
    <Button  
        android:id="@+id/Grapes"
```



```

        android:text="Grapes"
        android:layout_width="300dip"
        android:layout_height="wrap_content"
        android:padding="20dip"
        android:layout_marginTop="20dip" />
    <Button
        android:id="@+id/Kiwi"
        android:text="Kiwi"
        android:layout_width="300dip"
        android:layout_height="wrap_content"
        android:padding="20dip"
        android:layout_marginTop="20dip" />
</LinearLayout>

```

In Listing 3.17, we can see that five Button controls are vertically arranged in a LinearLayout container, one below the other. This vertical arrangement makes a few of the Button controls disappear when the screen is in Landscape mode.

If we run the application without defining the layout for the Landscape mode, we find the controls arranged in Portrait mode, as shown in [Figure 3.17](#) (left). But when we switch the screen orientation to Landscape, we find the last two Button controls have disappeared, as shown in [Figure 3.17](#) (right). This is because in Landscape mode, the screen becomes wider but shorter in height.



Figure 3.17 (left) Controls in Portrait mode, and (right) some controls disappear in Landscape mode.

To use the blank space on the right side of the screen in Landscape mode, we need to define another layout file, `activity_screen_orientation_app.xml`, created in the `res/layout-land` folder. The `layout-land` folder has to be created manually inside the `res` folder. Right-click on the `res` folder in the Package Explorer window and select the New, Folder option. A dialog box opens, asking for the name for the new folder. Assign the name `layout-land` to the new folder, and click the Finish button. Copy the `activity_screen_orientation_app.xml` file from the `res/layout` folder and paste it into `res/layout-land` folder. Modify the `activity_screen_orientation_app.xml` file in the `res/layout-land` folder so as to arrange the controls in Landscape mode. The code in the newly created `activity_screen_orientation_app.xml` is modified as shown in Listing 3.18.

Listing 3.18 The Layout File `activity_screen_orientation_app.xml` in the `res/layout-land` Folder

```

<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="250dp"
        android:layout_height="wrap_content"
        android:padding="20dip"
        android:layout_marginTop="20dip" />
    <Button
        android:id="@+id/Mango"

```



```

        android:text="Mango"
        android:layout_width="250dp"
        android:layout_height="wrap_content"
        android:padding="20dp"
        android:layout_marginTop="20dp"
        android:layout_toRightOf="@id/Apple" />
<Button
    android:id="@+id/Banana"
    android:text="Banana"
    android:layout_width="250dp"
    android:layout_height="wrap_content"
    android:padding="20dp"
    android:layout_marginTop="20dp"
    android:layout_below="@id/Apple" />
<Button
    android:id="@+id/Grapes"
    android:text="Grapes"
    android:layout_width="250dp"
    android:layout_height="wrap_content"
    android:padding="20dp"
    android:layout_marginTop="20dp"
    android:layout_below="@id/Apple"
    android:layout_toRightOf="@id/Banana" />
<Button
    android:id="@+id/Kiwi"
    android:text="Kiwi"
    android:layout_width="250dp"
    android:layout_height="wrap_content"
    android:padding="20dp"
    android:layout_marginTop="20dp"
    android:layout_below="@id/Banana" />
</RelativeLayout>

```

In this code block, we can see that, to fill up the blank space on the right side of the screen, the Mango and Grapes button controls are set to appear to the right of the Apple and Banana button controls. We can also detect the screen orientation via Java code. Let's modify the activity file ScreenOrientationAppActivity.java to display a toast message when the screen switches between landscape mode and portrait mode. The code written in the Java activity file ScreenOrientationAppActivity.java is shown in Listing 3.19.

Listing 3.19 Code Written in the Java Activity File ScreenOrientationAppActivity.java

```

package com.androidunleashed.screenorientationapp;

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

public class ScreenOrientationAppActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_screen_orientation_app);
        if(getResources().getDisplayMetrics().widthPixels>getResources().getDisplayMetrics().
            heightPixels)

```

```

{
    Toast.makeText(this,"Screen switched to Landscape mode",Toast.LENGTH_SHORT).show();
}
else
{
    Toast.makeText(this,"Screen switched to Portrait mode",Toast.LENGTH_SHORT).show();
}
}
}

```

Now, when we run the application, the controls appear in portrait mode as shown in [Figure 3.18](#) (left) and in landscape mode as shown in [Figure 3.18](#) (right). We can see that none of the Button controls are now hidden in landscape mode.



[Figure 3.18](#) (left) Controls in Portrait mode, and (right) all controls are visible in Landscape mode.

Unit – IV

Using Selection Widgets and Debugging

As the name suggests, selection widgets refers to the group controls that display a list of choices from which users select items. To constrain users to enter the correct data type or within a specific range and also to show the valid values, lists and drop-down lists are commonly used in applications. Lists and drop-down lists are called ListView and Spinner controls in Android. Besides ListView and Spinner, Two more selection widgets in this chapter: AutoCompleteTextView and GridView.

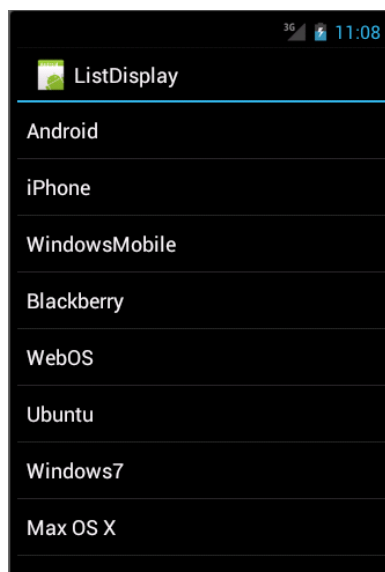
Q1. WRITE ABOUT ANDROID LISTVIEW CONTROL OR HOW A SCROLLABLE LIST CAN BE CREATED USING LIST VIEW CONTROL.EXPLAIN WITH AN EXAMPLE?

Android **ListView** is a view which contains the group of items and displays in a scrollable list. ListView is implemented by importing *android.widget.ListView* class. ListView is a default scrollable which does not use other scroll view.

ListView uses Adapter classes which add the content from data source (such as string array, array, database etc) to ListView. Adapter bridges data between an *AdapterViews* and other Views (ListView, ScrollView etc).

ListView is widely used in android applications. A very common example of ListView is your phone contact book, where you have a list of your contacts displayed in a ListView and if you click on it then user information is displayed.

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.



List View

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, grid view etc.

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

Android provides several subclasses of Adapter that are useful for retrieving different kinds of data and building views for an AdapterView (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	android:id :: This is the ID which uniquely identifies the layout.
2	android:divider :: This is drawable or color to draw between list items.
3	android:dividerHeight ::This specifies height of the divider. This could be in px, dp, sp, in, or mm.
4	android:entries ::Specifies the reference to an array resource that will populate the ListView.
5	android:footerDividersEnabled :: When set to false, the ListView will not draw the divider before each footer view. The default value is true.
6	android:headerDividersEnabled :: When set to false, the ListView will not draw the divider after each header view. The default value is true.

ArrayAdapter

We can use this adapter when our 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 we have an array of strings we want to display in a ListView, initialize a new **ArrayAdapter** using a constructor to specify the layout for each string and the string array –

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

Here are arguments for this constructor –

- First argument **this** is the application context. Most of the case, keep it **this**.
- Second argument will be layout defined in XML file and having **TextView** for each string in the array.
- Final argument is an array of strings which will be populated in the text view.

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

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

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

Example

Following is the example which will take through simple steps to show how to create our own Android application using ListView. Follow the following steps to modify the Android application we created in *Hello World Example* chapter –

Step	Description
1	We will use Android Studio IDE to create an Android application and name it as <i>ListDisplay</i> under a package <i>com.example.ListDisplay</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 ListView content with the self explanatory attributes.
3	No need to change string.xml, Android studio takes care of default string constants.
4	Create a Text View file <i>res/layout/activity_listview.xml</i> . This file will have setting to display all the list items. So you can customize its fonts, padding, color etc. using this file.
6	Run the application to launch Android emulator and verify the result of the changes done in the application.

Following is the content of the modified main activity file **src/com.example.ListDisplay/ListDisplay.java**. This file can include each of the fundamental life cycle methods.

```
package com.example.ListDisplay;  
import android.os.Bundle;  
import android.app.Activity;  
import android.view.Menu;  
import android.widget.ArrayAdapter;  
import android.widget.ListView;
```

```

public class ListDisplay extends Activity {
    // Array of strings...
    String[] mobileArray = {"Android","IPhone","WindowsMobile","Blackberry",
        "WebOS","Ubuntu","Windows7","Max OS X"};
    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);
    }
}

```

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

```

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

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

</LinearLayout>

```

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

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">ListDisplay</string>
    <string name="action_settings">Settings</string>
</resources>

```

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


```

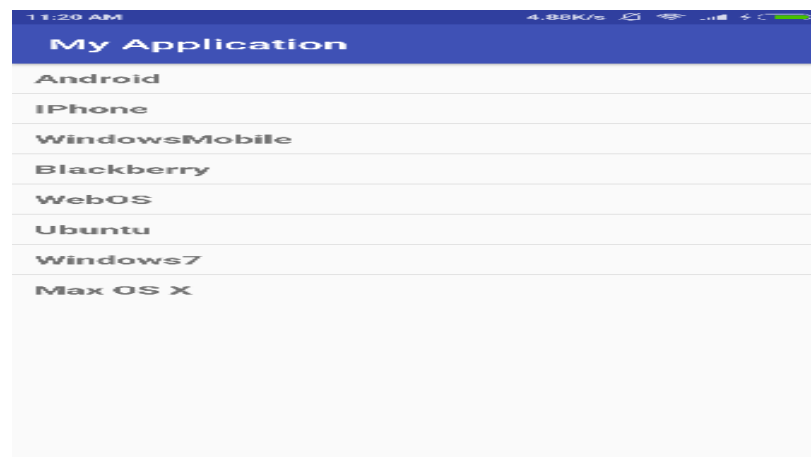
<?xml version="1.0" encoding="utf-8"?>
<!-- Single List Item Design -->

<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/label"

```

```
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:padding="10dip"
android:textSize="16dip"
android:textStyle="bold" >
</TextView>
```

Let's try to run our modified **Hello World!** application we just modified. I assume we had created our **AVD** while doing environment set-up. To run the app from Android studio, open one of our project's activity files and click Run  icon from the tool bar. Android studio installs the app on our AVD and starts it and if everything is fine with our set-up and application, it will display following Emulator window –



Q2. WRITE ABOUT ANDROID LISTVIEW CONTROL OR HOW A SCROLLABLE LIST CAN BE CREATED USING ADAPTER CONTROL.EXPLAIN WITH AN EXAMPLE?

Adapters

Android provides a framework of adapters (also known as data adapters) that are used to provide a data source for displaying content in the form of choices in selection widgets; that is, they help create child elements for the selection widgets. The data source refers to the content, including elements in arrays and data in database tables. The Adapters serve two purposes. First, they provide the data source for a selection widget, and second, they convert individual elements of data into specific Views to be displayed inside the selection widget. The second purpose is important, as it casts the data to suit the selection widget environment, overriding its default behavior, and also enables us to format the data in the available formats. Android provides many basic Adapters such as ListAdapter, ArrayAdapter, and CursorAdapter. We can also create our own Adapter.

Populating ListView Through the ArrayAdapter

The ArrayAdapter is one of the adapters provided by Android that provides data sources (child elements) to selection widgets and also casts the data into specific view(s) to be displayed inside the selection widgets. In this section, we learn to create

an ArrayAdapter and use it to populate the ListView control. An ArrayAdapter can be created through string resource, as well as through string arrays defined in Java code.

Create a new Android project called ListViewDemo1. Again, in this application, we use two controls, ListView and TextView, where ListView displays the items assigned to it through ArrayAdapter and the TextView displays the item that is selected by the user from the ListView. Define the two controls by writing the code shown in [Listing 5.4](#) into the layout file `activity_list_view_demo1.xml`.

Listing 5.4. The Layout File `activity_list_view_demo1.xml` After Adding the ListView and TextView Controls

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <ListView
        android:id="@+id/fruits_list"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:drawSelectorOnTop="false"/>
    <TextView
        android:id="@+id/selectedopt"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
</LinearLayout>
```

Next, we need to write code into the Java activity file `ListViewDemo1Activity.java` to serve the following purposes:

- Create an ArrayAdapter through a string array and assign it to the ListView for displaying items
- Display the item selected from the ListView in the TextView

The code written into the activity file `ListViewDemo1Activity.java` is shown in [Listing 5.5](#).

Listing 5.5. Code Written into the Java Activity File `ListViewDemo1Activity.java`

```
package com.androidunleashed.listviewdemo1;
import android.app.Activity;
import android.os.Bundle;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.TextView;
import android.widget.ListView;
import android.widget.ArrayAdapter;
```

```

import android.widget.AdapterView;
import android.view.View;

public class ListViewDemo1Activity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_list_view_demo1);
        final String[] fruits={"Apple", "Mango", "Orange", "Grapes", "Banana"};
        final TextView selectedOpt=(TextView)findViewById(R.id.selectedopt);
        ListView fruitsList = (ListView)findViewById(R.id.fruits_list);
        final ArrayAdapter<String> arrayAdpt= new ArrayAdapter<String>(this,
            android.R.layout.simple_list_item_1, fruits);
        fruitsList.setAdapter(arrayAdpt);
        fruitsList.setOnItemClickListener(new OnItemClickListener(){
            public void onItemClick(AdapterView<?> parent, View v, int position,long id){
                selectedOpt.setText("You have selected "+fruits[position]);
            }
        });
    }
}

```

An ArrayAdapter is the simplest of the adapters and acts as the data source for the selection widgets ListView, GridView, and so on. An ArrayAdapter makes use of the TextView control to represent the child Views in a View. In the code shown in [Listing 5.5](#), an ArrayAdapter is created through the following code:

```

ArrayAdapter<String> arrayadpt=new ArrayAdapter<String> (this,
android.R.layout.simple_list_item_1, fruits);

```

This constructor creates an ArrayAdapter called arrayAdpt that can display the elements of the specified array, fruits, via the TextView control.

The ArrayAdapter constructor consists of the following:

- **this (the current context)**—As the Activity is an extension of the Context class, we use the current instance as the context.
- **android.R.layout.simple_list_item_1**—Points to a TextView defined by the Android SDK that will be used for displaying each item in the ListView. The elements of the array that is specified next needs to be wrapped or cast in a view before being assigned to any selection widget for display. So, the android.R.layout.simple_list_item_1 simply turns the strings defined in the string array into a TextView for displaying them in a ListView.
- **array**—The data source—an array of strings for the ListView.

We can see that the ListView and TextView controls from the layout files are accessed and mapped to the objects fruitsList and selectedOpt, respectively. The arrayAdpt ArrayAdapter containing the elements of the fruits array

in TextView form is assigned to the ListView control for displaying choices to the user. The OnItemClickListener interface is implemented via an anonymous class that implements a callback method, onItemClick(). The reference of an anonymous class is passed to the fruitsList ListView to invoke the callback method onItemClick() when any of the items in ListView is clicked. In the onItemClick() method, the item selected in the ListView is displayed via the TextView control selectedOpt. When we run the application, the list of items is displayed via ListView, and the item selected from the ListView is displayed via the TextView control, as shown in Figure 5.2.

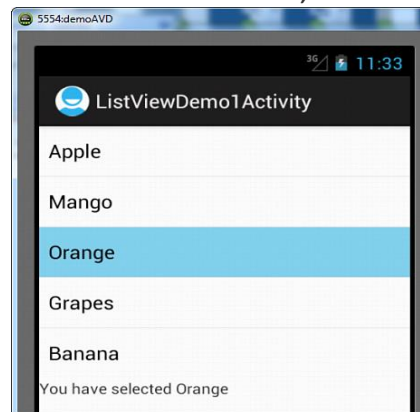


Figure 5.2. Options displayed through Java code in the ListView control and the selected option from ListView displayed through the TextView control

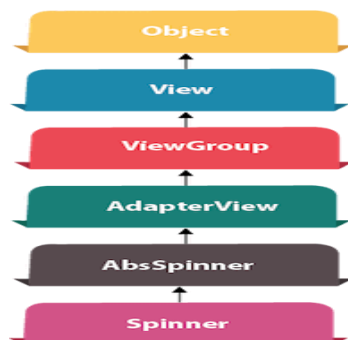
Now that we understand how to create a ListView through an Activity base class, let's create ListView by extending the ListActivity class.

Q3. WRITE ABOUT ANDROID SPINNER CONTROL OR HOW A DROP-DOWN LIST CAN BE CREATED USING SPINNER CONTROL. EXPLAIN WITH AN EXAMPLE?

USING THE SPINNER CONTROL

Android Spinner is like the combobox of AWT or Swing. It can be used to display the multiple options to the user in which only one item can be selected by the user. Android spinner is like the drop down menu with multiple values from which the end user can select only one value.

Android spinner is associated with AdapterView. So you need to use one of the adapter classes with spinner.



Android Spinner class is the subclass of AbsSpinner class.

Android Spinner Example

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com
/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="example.javatpoint.com.spinner.MainActivity">

    <Spinner
        android:id="@+id/spinner"
        android:layout_width="149dp"
        android:layout_height="40dp"
        android:layout_marginBottom="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.502"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.498" />
</android.support.constraint.ConstraintLayout>
```

MainActivity.java

```
package example.javatpoint.com.spinner;
import android.support.v7.app.AppCompatActivity;
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;

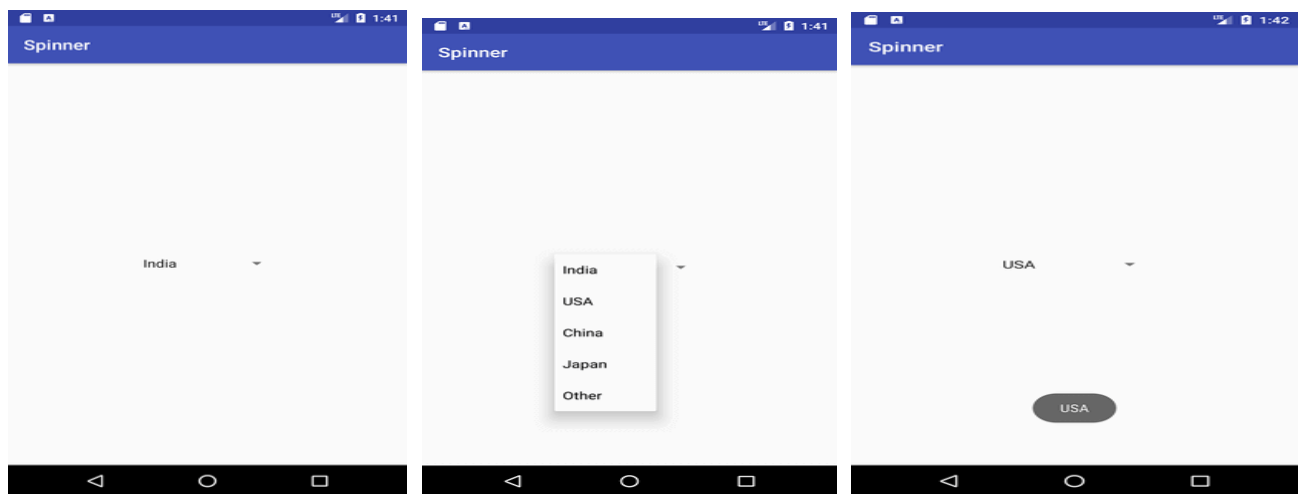
public class MainActivity extends AppCompatActivity implements
    AdapterView.OnItemClickListener {
    String[] country = { "India", "USA", "China", "Japan", "Other"};
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        //Getting the instance of Spinner and applying OnItemSelectedListener on it
        Spinner spin = (Spinner) findViewById(R.id.spinner);
        spin.setOnItemSelectedListener(this);
        ArrayAdapter aa = new ArrayAdapter(this, android.R.layout.simple_spinner_item, country);
```

```

aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
spin.setAdapter(aa);
}
//Performing action onItemSelected and onNothing selected
public void onItemSelected(AdapterView<?> arg0, View arg1, int position, long id) {
    Toast.makeText(getApplicationContext(),country[position] , Toast.LENGTH_LONG).show();
}
public void onNothingSelected(AdapterView<?> arg0) {
    // TODO Auto-generated method stub
}
}
}

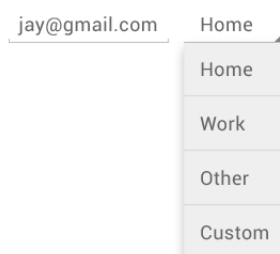
```

Output:



Q4. WRITE ABOUT ANDROID SPINNER CONTROL USING ARRAY ADAPTER? OR HOW A DROP-DOWN LIST CAN BE CREATED USING SPINNER CONTROL THROUGH ARRAY ADAPTER.EXPLAIN WITH AN EXAMPLE?

Spinners provide a quick way to select one value from a set. In the default state, a spinner shows its currently selected value. Touching the spinner displays a dropdown menu with all other available values, from which the user can select a new one.



We can add a spinner to layout with the [Spinner](#) object. We should usually do so in XML layout with a `<Spinner>` element.

For example:

```
<Spinner
    android:id="@+id/planets_spinner"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

To populate the spinner with a list of choices, you then need to specify a [SpinnerAdapter](#) in your [Activity](#) or [Fragment](#) source code.

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com
/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="example.javatpoint.com.spinner.MainActivity">
    <Spinner
        android:id="@+id/spinner"
        android:layout_width="149dp"
        android:layout_height="40dp"
        android:layout_marginBottom="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.502"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.498" />
</android.support.constraint.ConstraintLayout>
```

MainActivity.java

```
package example.javatpoint.com.spinner;
import android.support.v7.app.AppCompatActivity;
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;

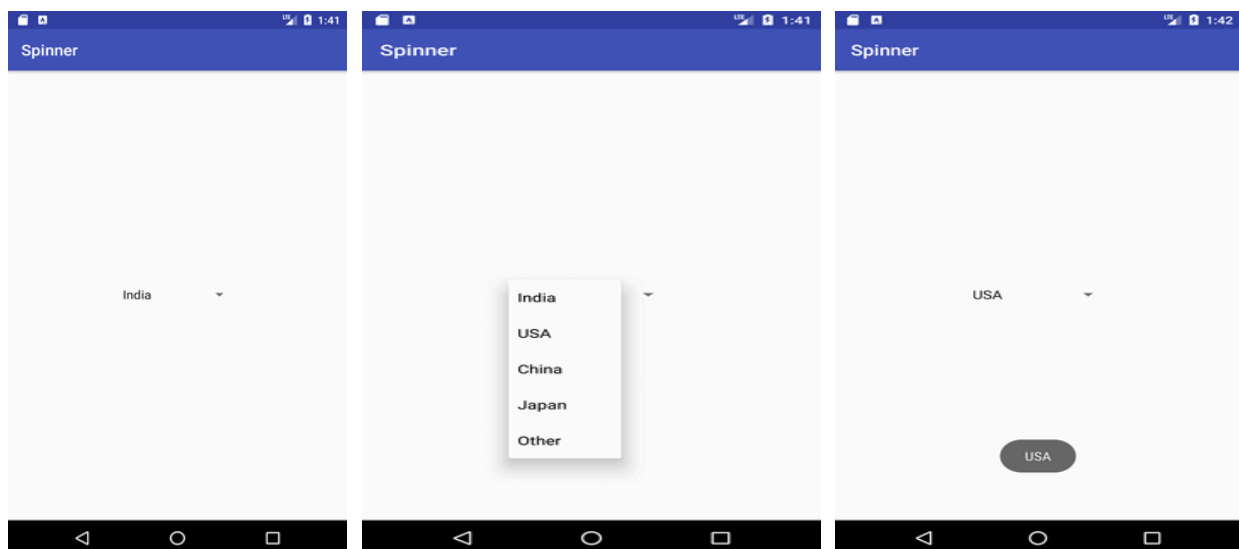
public class MainActivity extends AppCompatActivity implements
    AdapterView.OnItemClickListener {
    String[] country = { "India", "USA", "China", "Japan", "Other"};
    protected void onCreate(Bundle savedInstanceState) {
```

```

super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
//Getting the instance of Spinner and applying OnItemSelectedListener on it
Spinner spin = (Spinner) findViewById(R.id.spinner);
spin.setOnItemSelectedListener(this);
//Creating the ArrayAdapter instance having the country list
ArrayAdapter aa = new ArrayAdapter(this,android.R.layout.simple_spinner_item,country);
aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
//Setting the ArrayAdapter data on the Spinner
spin.setAdapter(aa);
}
//Performing action onItemSelected and onNothing selected
public void onItemSelected(AdapterView<?> arg0, View arg1, int position, long id) {
    Toast.makeText(getApplicationContext(),country[position] , Toast.LENGTH_LONG).show();
}
public void onNothingSelected(AdapterView<?> arg0) {
    // TODO Auto-generated method stub
}
}

```

Output:



Q5. WRITE ABOUT ANDROID GRID-VIEW CONTROL OR HOW A TEXT OR IMAGE CAN BE SHOWN IN GRID VIEW.EXPLAIN WITH AN EXAMPLE?

USING THE GRIDVIEW CONTROL

The GridView control is a ViewGroup used to display text and image data in the form of a rectangular, scrollable grid. To display data in the grid, we first define a GridView control in the XML layout, and then bind the data that we want to be displayed to it using the ArrayAdapter.

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**

As the name suggests, **ViewGroup** is a view that contains other views known as child views. The **ViewGroup** class is a base class for layout managers that are used to contain and arrange several views. **ListView**, **GridView**, and other container controls are good examples of **ViewGroups**.

Let's create a new Android project called **GridViewApp**. In this we display certain strings arranged in a rectangular grid. When a user selects any of the strings, its name is displayed. That is, we require two controls in this application: a **GridView** control for arranging strings and a **TextView** control for displaying the string selected by the user.

GridView Attributes

Following are the important attributes specific to **GridView** –

Different attributes used in **GridView**.

The number of rows displayed through **GridView** is dependent on the number of elements supplied by the attached adapter. The size and number of columns is controlled through the following attributes:

- **android:numColumns**—Defines the number of columns. If we supply a value, **auto_fit**, Android computes the number of columns based on available space.
- **android:verticalSpacing** and **android:horizontalSpacing**—Define the amount of whitespace between the items in the grid.
- **android:columnWidth**—Defines the width of each column.
- **android:stretchMode**—The attribute determines whether the column can stretch or expand to take up the available space. The valid values for this attribute are
- **none**—Does not allow columns to stretch or expand
- **columnWidth**—Makes the columns take up all available space
- **spacingWidth**—Makes the whitespace between columns take up all available space

File: activity_main.xml

```
<GridView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gridView1"
    android:numColumns="auto_fit"
    android:gravity="center"
    android:columnWidth="50dp"
    android:stretchMode="columnWidth"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >
</GridView>
```


MainActivity.java

```
package com.example.gridview;
```

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

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

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

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

```
import android.widget.AdapterView;
```

```
import android.widget.AdapterView.OnItemClickListener;
```

```
import android.widget.ArrayAdapter;
```

```
import android.widget.GridView;
```

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

```
import android.widget.Toast;
```

```
public class MainActivity extends Activity {
```

```
    GridView gridView;
```

```
    static final String[] numbers = new String[] {
```

```
        "1", "2", "3", "4", "5",
```

```
        "6", "7", "8", "9", "10",};
```

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

```
        super.onCreate(savedInstanceState);
```

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

```
        gridView = (GridView) findViewById(R.id.gridView1);
```

```
        ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,  
            android.R.layout.simple_list_item_1, numbers);
```

```
        gridView.setAdapter(adapter);
```

```
        gridView.setOnItemClickListener(new OnItemClickListener() {
```

```
            public void onItemClick(AdapterView<?> parent, View view, int position,long id) {
```

```
                Toast.makeText(getApplicationContext(),((TextView) view).getText(), Toast.LENGTH_  
LONG).show();
```

```
            }
```

```
        });
```

```
    }
```

```
    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;
```

```
    }
```

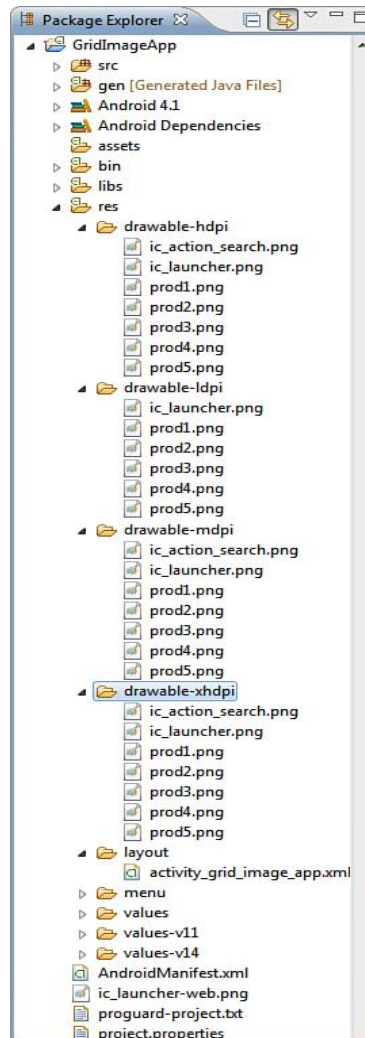
```
}
```

Q6. WRITE ABOUT ANDROID GRID-VIEW CONTROL OR HOW IMAGES CAN BE DISPLAYED IN GRID VIEW.EXPLAIN WITH AN EXAMPLE?

Displaying Images in GridView

To display content in the GridView control, we use Adapters, which provide the content to display to the controls. The content can be fetched from the sources such as arrays, databases, or other data sources.

Let's create a new application called GridImageApp. Assuming the image filenames that we want to display through the GridView control are prod1.png, prod2.png, prod3.png, prod4.png, and prod5.png, copy them into the four res/drawable folders. Our project in the Package Explorer window appears as shown.



The Package Explorer window showing the images copied to theres/drawable folders

In this application, we want a message to be displayed showing the image number of the picture displayed via the GridView control. Our application is therefore going to have two controls: aTextView control for displaying the selected image number and a GridView control for displaying images in a grid. After we add the TextView and GridView controls, activity_grid_image_app.xml appears .

The Layout File activity_grid_image_app.xml After Adding the TextView and GridView Controls

```
<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:id="@+id/selectedopt"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="List of Products " />
    <GridView android:id="@+id/grid"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:verticalSpacing="2dip"
        android:horizontalSpacing="2dip"
        android:numColumns="auto_fit"
        android:columnWidth="100dip"
        android:stretchMode="columnWidth"
        android:gravity="center" />
</LinearLayout>
```

To display images in the GridView control and also tell us which image is selected by the user, write the code into the activity file GridImageAppActivity.java.

Code Written into the Java Activity File GridImageAppActivity.java

```
package com.androidunleashed.gridimageapp;

import android.app.Activity;
import android.os.Bundle;
import android.widget.GridView;
import android.view.View;
import android.widget.ImageView;

import android.content.Context;
import android.widget.BaseAdapter;
import android.widget.AdapterView;
import android.widget.TextView;
import android.view.ViewGroup;

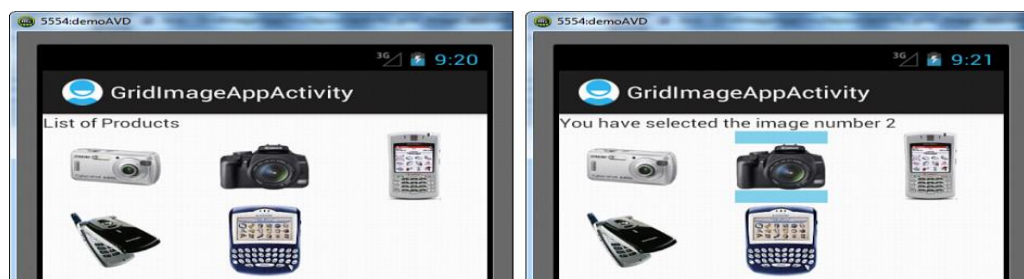
public class GridImageAppActivity extends Activity implements
    AdapterView.OnItemClickListener {
    TextView selectedOpt;
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_grid_image_app);
        selectedOpt=(TextView) findViewById(R.id.selectedopt);
        GridView g=(GridView) findViewById(R.id.grid);
        g.setAdapter(new ImageAdapter(this));
        g.setOnItemClickListener(this);
    }
    public void onItemClick(AdapterView<?> parent, View v, int position, long id) {
        int p=position+1;
```

```

        selectedOpt.setText("You have selected the image number "+p);
    }
    public class ImageAdapter extends BaseAdapter {
        private Context context;
        Integer[] images = {
            R.drawable.prod1,
            R.drawable.prod2,
            R.drawable.prod3,
            R.drawable.prod4,
            R.drawable.prod5
        };
        public ImageAdapter(Context c) {
            context = c;
        }
        public int getCount() {
            return images.length;
        }
        public Object getItem(int position) {
            return position;
        }
        public long getItemId(int position) {
            return position;
        }
        public View getView(int position, View convertView, ViewGroup parent) {
            ImageView imageView = new ImageView(context);
            imageView.setImageResource(images[position]);
            imageView.setLayoutParams(new GridView.LayoutParams(100, 120));
            return imageView;
        }
    }
}

```

The TextView and GridView controls defined in activity_grid_image_app.xml with the IDs `selectedopt` and `grid`, respectively, are fetched and mapped to the objects `selectedOpt` and `g`, respectively.



Images displayed in a GridView control (left), and the selected image number displayed via a TextView (right)

Q7. WRITE ABOUT ANDROID VIEW PAGER CONTROL OR HOW A IMAGE GALLERY CAN BE SHOWN USING VIEW PAGERCONTROL.EXPLAIN WITH AN EXAMPLE?

CREATING AN IMAGE GALLERY USING THE VIEWPAGER CONTROL

The ViewPager control (android.support.v4.view.ViewPager) helps in showing data, which may be text, image, and so on, in the form of pages with the horizontal swiping behavior. That is, the pages of data being displayed can be flipped left and right. To identify and to keep track of the pages being displayed through ViewPager, a key object is associated with each of them. The ViewPager needs a data adapter to define and load the data for each page. The data adapter that is used to define the data for each page to be displayed through the ViewPager control is the PagerAdapter (android.support.v4.view.PagerAdapter) class.

While implementing the PagerAdapter, we must override the following methods:

- **instantiateItem(View container, int position)**—Used for creating and instantiating the page and adding it to the container. Using the LayoutInflater service, the method also inflates the appropriate layout and adds it to the specified container.

Syntax:

```
public Object instantiateItem(View container, int position)
```

- **container**—Represents the container in which the page has to be displayed.
- **position**—Represents the position of the page to be instantiated.
- **destroyItem(View container, int position, Object object)**—Used for removing the page of the specified position from the container.
- **isViewFromObject(View view, Object object)**—Determines whether the specified page is associated with a specific key object.
- **getCount()**—Defines the size of the paging range, that is, the count of the number of the pages.

The position of the pages is zero based by default; that is, the first page to the left is in position 0, the next page to the right is position 1, and so on. We can also set the initial position of the pager through the setCurrentItem() method.

To listen to the change in state of the selected page, we need to define a class that extends SimpleOnPageChangeListener. When a page from the ViewPager is selected, the callback method onPageSelected() is called.

Lets when an image is selected from the gallery, the selected image number is displayed. Let's name this application ViewPagerApp. The first step is to copy the images that we want to display through the gallery into the res/drawable folders of the application. Assuming we have the files named prod1.png, prod2.png, prod3.png, prod4.png, and prod5.png on our local disk drive, copy them into the res/drawable folders of our project.

After copying the images, we define two controls, TextView and ViewPager, in the layout file activity_view_pager_app.xml. The TextView control displays a message with the image number selected from the image gallery. The ViewPager control

displays the images in a horizontally scrolling list. After we define the TextView and ViewPager control, activity_view_pager_app.xml appears .

The Layout File activity_view_pager_app.xml After Defining the TextView and ViewPager Controls

```
<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:id="@+id/selectedopt"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Image Gallery "
        android:gravity="center"
        android:textStyle="bold" />
    <android.support.v4.view.ViewPager
        android:id="@+id/viewpager"
        android:layout_width="match_parent"
        android:layout_height="100dip"
        android:layout_marginTop="25dip" />
</LinearLayout>
```

We can see that the TextView and ViewPager controls are assigned the IDs selectedopt and viewpager, respectively. To make the image gallery appear at a distance of 25dip from the top of the LinearLayout container, the android:layout_marginTop attribute is set to 25dip. To constrain the height of the images being displayed to 100dip, the android:layout_height is set to 100dip.

To display images in the ViewPager control and to display the image number selected by the user, write the code into the activity file ViewPagerAppActivity.java.

Code Written into the Java Activity File ViewPagerAppActivity.java

```
package com.androidunleashed.viewpagerapp;

import android.os.Bundle;
import android.app.Activity;
import android.support.v4.view.ViewPager;
import android.support.v4.view.PagerAdapter;
import android.support.v4.view.ViewPager.SimpleOnPageChangeListener;
import android.widget.TextView;
import android.view.View;
import android.widget.ImageView;

public class ViewPagerAppActivity extends Activity {
    public TextView selectedOpt;
```

```

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_view_pager_app);
    selectedOpt=(TextView) findViewById(R.id.selectedopt);
    ViewPager viewPager = (ViewPager) findViewById(R.id.viewpager);
    viewPager.setAdapter(new ImageAdapter());
    viewPager.setOnPageChangeListener(new PageListener());
}
public class ImageAdapter extends PagerAdapter {
    Integer[] images = {
        R.drawable.prod1,
        R.drawable.prod2,
        R.drawable.prod3,
        R.drawable.prod4,
        R.drawable.prod5
    };
    public Object instantiateItem(View container, int position) {
        ImageView view = new ImageView(ViewPagerAppActivity.this);
        view.setImageResource(images[position]);
        ((ViewPager) container).addView(view, 0);
        return view;
    }
    public int getCount() {
        return images.length;
    }
    public void destroyItem(View arg0, int arg1, Object arg2) {
        ((ViewPager) arg0).removeView((View) arg2);
    }
    public boolean isViewFromObject(View arg0, Object arg1) {
        return arg0 == ((View) arg1);
    }
}
private class PageListener extends SimpleOnPageChangeListener{
    public void onPageSelected(int position) {
        selectedOpt.setText("You have selected the page number "+position);
    }
} }

```

The TextView and ViewPager controls defined in activity_view_pager_app.xml with the IDsselectedopt and viewpager, respectively, are fetched and mapped to the respective objectsselectedOpt and viewPager.

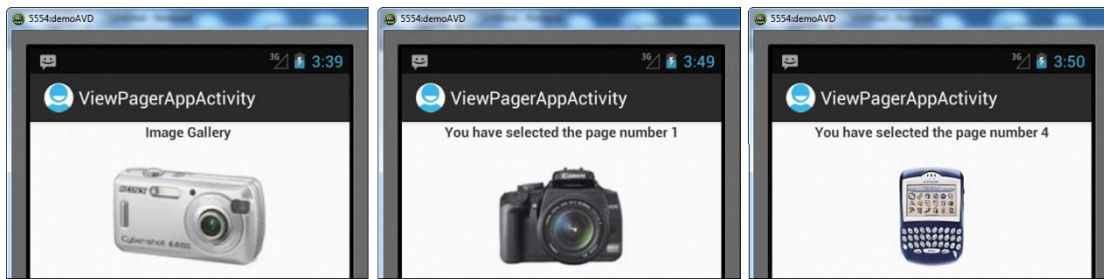


Image displayed in the gallery control on application startup (left), hidden images displayed on scrolling the gallery (middle), and the selected image number displayed via a TextView (right)

Q8. Explain HOW AND WHAT IS THE USE OF THE DEBUGGING TOOL? OR EXPLAIN DALVIK DEBUG MONITOR SERVICE(DDMS)?

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.

Figure 5.11 shows the DDMS tool window.

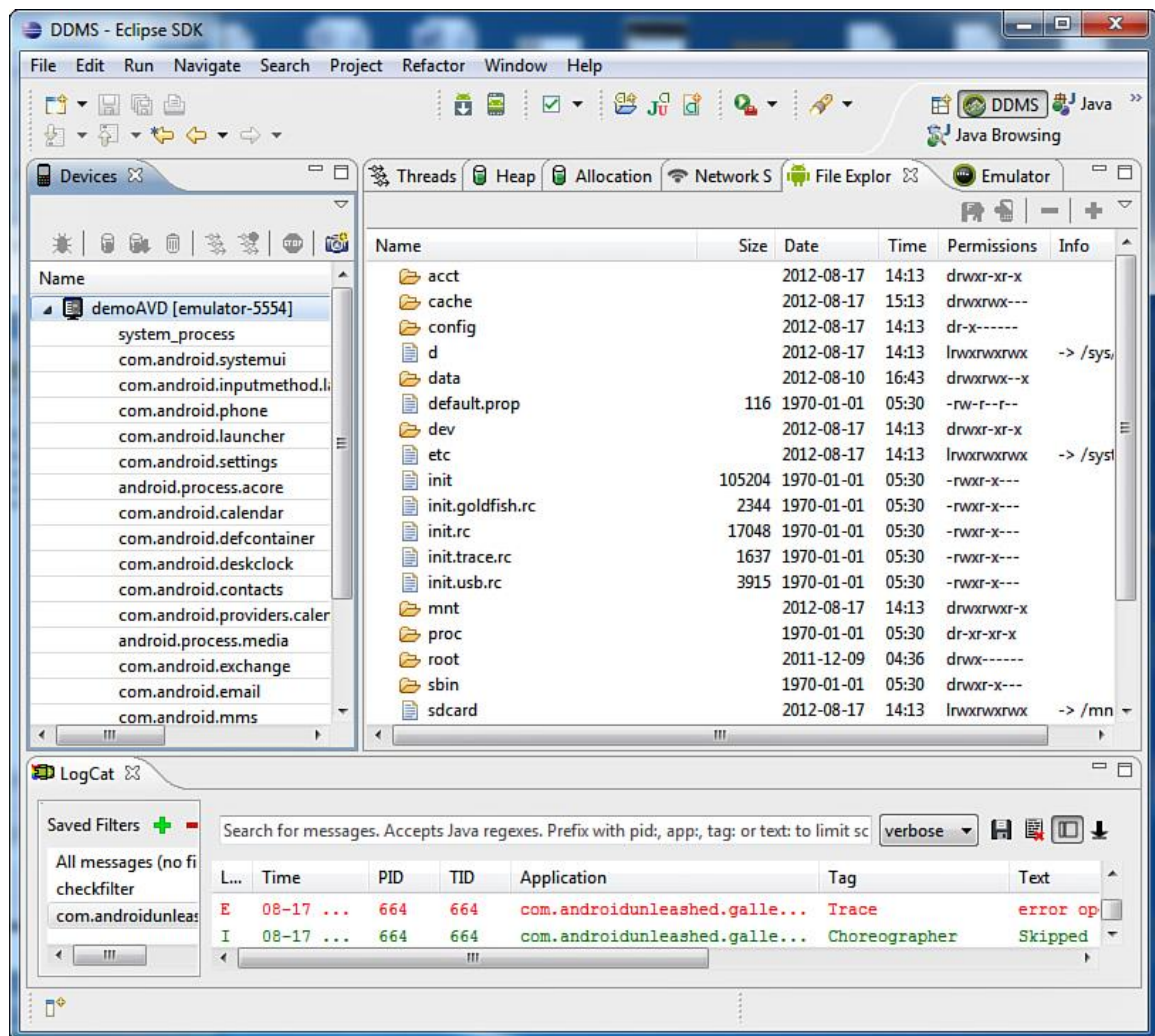


Figure 5.11. The DDMS tool window

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). The VMs associated with each device or AVD also is displayed. Selecting a VM displays its information in the right pane. In the Devices tab, you see some icons, described here:

- **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.

If the application is running and its output is being displayed through the device/emulator, clicking the Screen Capture icon displays the Device Screen Capture dialog box, as shown in [Figure 5.12](#) (left). The text, Capturing, tells us that the output of the application or image being displayed in the device/emulator is in the process of being captured. Once the image is captured, it is displayed as shown in [Figure 5.12](#) (right).

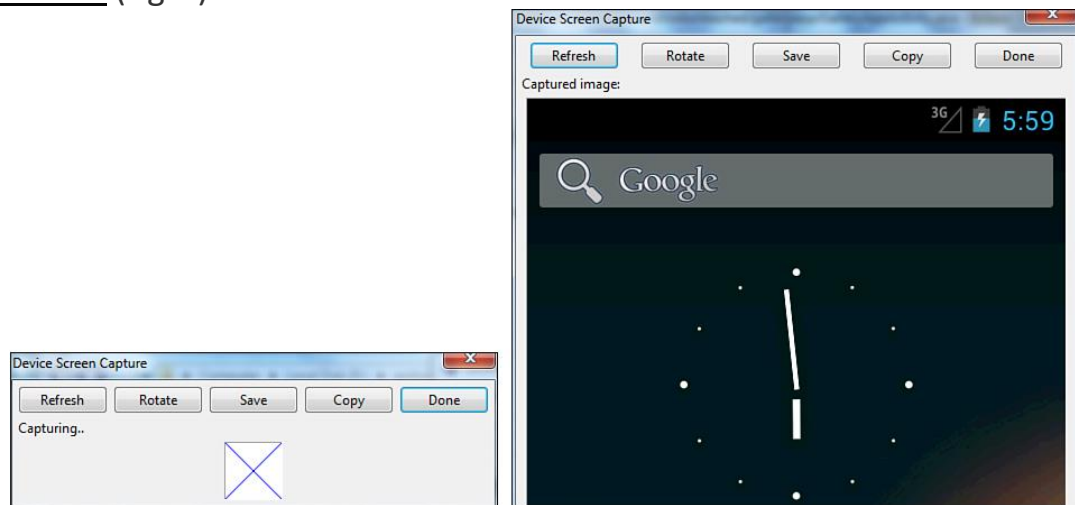


Figure 5.12. Image shown in the device/emulator is being captured (left), and the captured image of the device/emulator displayed (right)

The meaning of the buttons shown at the top in the Device Screen Capture dialog box is shown here:

- **Refresh**—Updates the captured image.
- **Rotate**—With each click of this button, the captured image rotates 90 degrees in the counterclockwise direction.
- **Save**—Saves the captured image as a .png file.
- **Copy**—Copies the captured image to the clipboard.
- **Done**—Closes the Device Screen Capture dialog.

Back to DDMS, on the right pane (refer to [Figure 5.11](#)), we find the following tabs:

- **Threads**—Displays information about the threads within each process, as shown in [Figure 5.13](#) (left). The following information about the threads is displayed:
 - **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). Select the Cause GC button to begin the garbage collection process. The object types and the size of memory allocated to them are displayed. After we select an object type, a bar graph is displayed, showing the number of objects allocated for a particular memory size in bytes (see [Figure 5.13](#)—right).

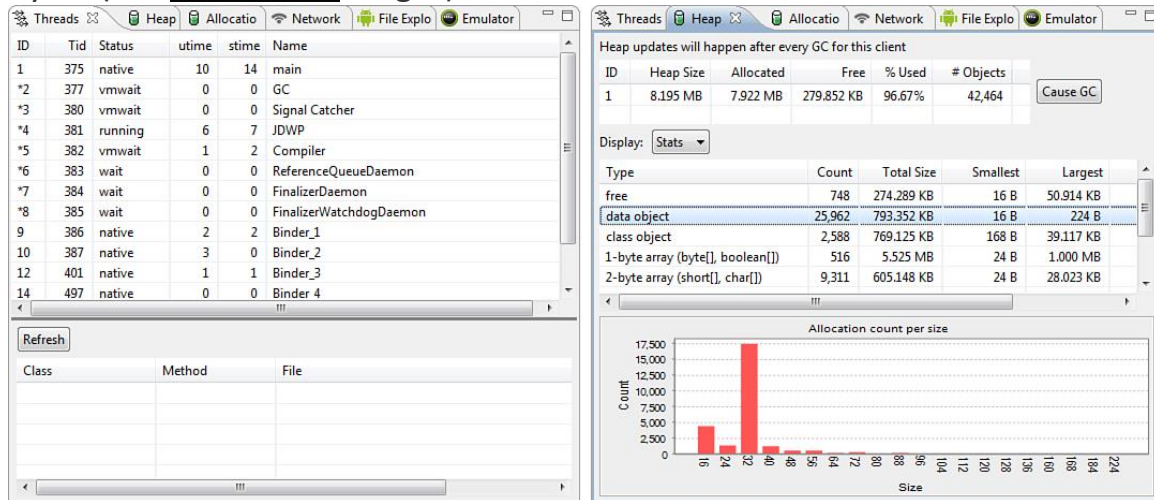


Figure 5.13. The Threads tab, displaying information about running threads (left), and the Heap tab displaying heap information of the current process (right)

- **Allocation Tracker**—Tracks the objects allocated to an application.
- **Network Statistics**—Helps us in getting information regarding network usage of our application, that is, when our app made network requests, speed of data transfer—and other related information.
- **File Explorer**—Displays the file system on the device.

We can view and delete files on the device/emulator through this tab. We can even push or pull files from the device using the two icons, Pull a file from the device and Push a file onto the device, that are shown at the top. To copy a file from the device, select the file in the File Explorer and click the Pull a file from the device button. The Get Device File dialog box opens up, prompting us to specify the path and filename where we want to store the pulled device file. Similarly, to copy a file to the device, click the Push file onto the device button in the File Explorer tab.

Right of the File Explorer tab is the Emulator Control tab that can be used to simulate incoming phone calls, SMS messages, or GPS coordinates. To simulate an incoming phone call, select the Voice option, provide the incoming phone number, and click the Call button, as shown in [Figure 5.15](#) (left). In the emulator, an incoming call appears, prompting the user to answer the call in [Figure 5.15](#) (right). The

incoming call can be ended either by clicking the End button in the emulator or by clicking the Hang Up button in the Emulator Control tab.

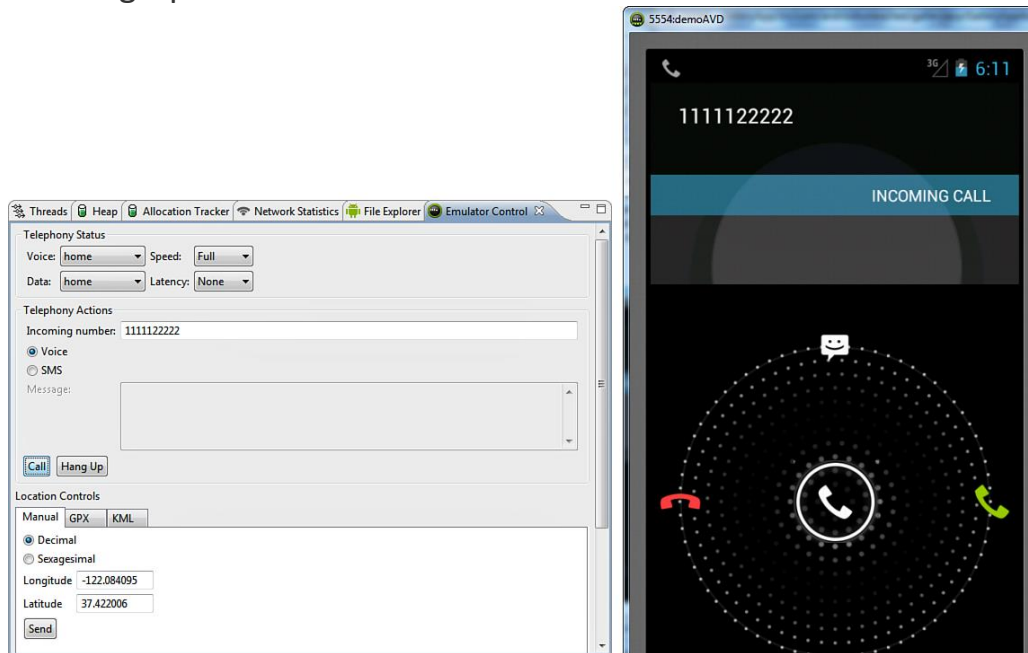


Figure 5.15. Simulating an incoming phone call through the Emulator Control tab (left), and an incoming phone call appears on the Android emulator (right).

To simulate an SMS message, select the SMS option in the Emulator Control tab, provide the phone number, write the message, and click the Send button, as shown in [Figure 5.16](#) (left). In the emulator, an incoming SMS notification appears at the top (see [Figure 5.16](#)—right). We can simulate GPS coordinates (longitude and latitude values) manually, via the GPX file or KML file through the Emulator Control tab. Remember, only GPX 1.1 files are supported.

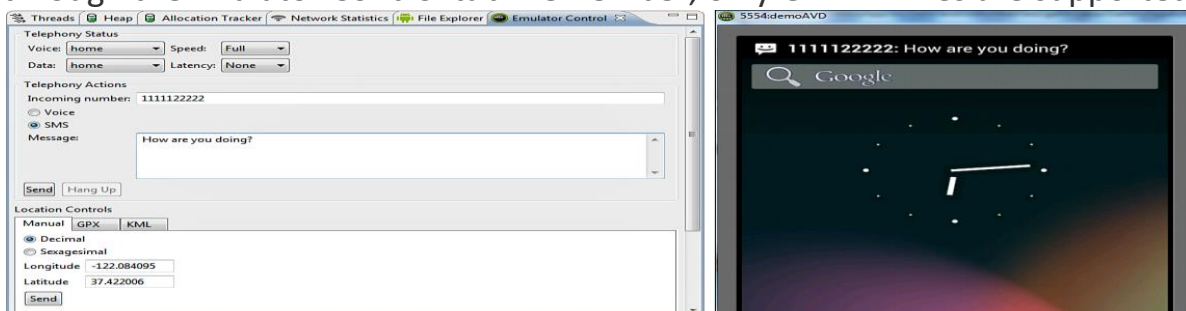


Figure 5.16. Simulating SMS via the Emulator Control tab (left), and incoming SMS notification displayed at the top in the Android emulator (right)

The bottom pane of the DDMS is used to show the log of the processes on the selected device or AVD. The pane is meant for performing debugging and tracing tasks. The LogCat tab shows all messages of the device, including exceptions and those placed in the application to see the intermediate results. We can also set up filters to watch filtered log information. The Console tab displays the messages related to the starting of the activity.

Q9. WRITE ABOUT ANDROID DEBUGGING APPLICATIONS? OR EXPLAIN MOST CRITICAL TASK IN SOFTWARE DEVELOPMENT DEBUGGING?

The most critical and essential task in software development—debugging.

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.

We can place as many breakpoints as we want in our application. The statements just perform simple multiplication and display log messages. Only the statements in bold are newly added; the rest of the code is the same.

Code Added to the Java Activity File HelloWorldAppActivity.java

```
package com.androidunleashed.helloworldapp;
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;
import android.util.Log;

public class HelloWorldAppActivity extends Activity {

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello_world_app);
        TextView mesg = (TextView)findViewById(R.id.message);
        mesg.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 (see [Figure 5.17](#)).

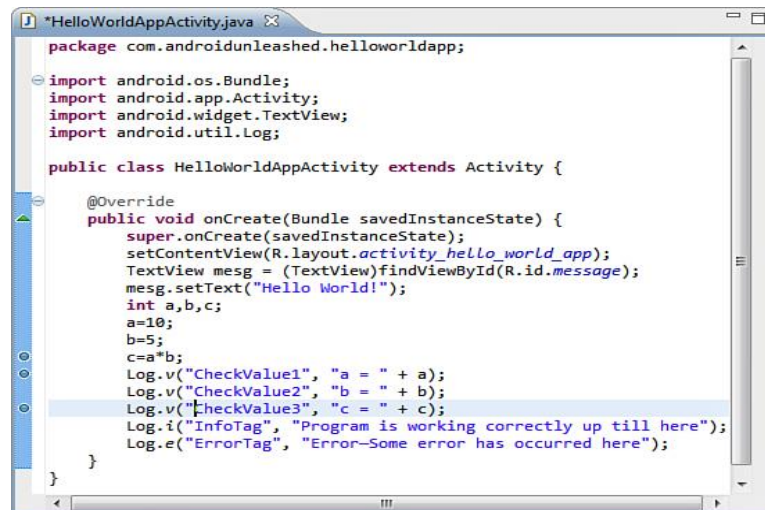


Figure 5.17. Activity file displaying the statements where breakpoints are inserted

To stop execution at the breakpoints, don't run the application; instead debug it by either pressing F11, selecting Run, Debug, or right-clicking the project in Package Explorer and selecting Debug As, Android Application. During debugging, the application pauses when the first breakpoint is reached. At the breakpoints, we can highlight variables to see their values and execute certain expressions. When the application reaches a breakpoint for the first time, a window pops up asking whether we want to switch to the Debug perspective, as shown in [Figure 5.18](#). To prevent this window from appearing again, check the Remember my decision check box and click Yes.

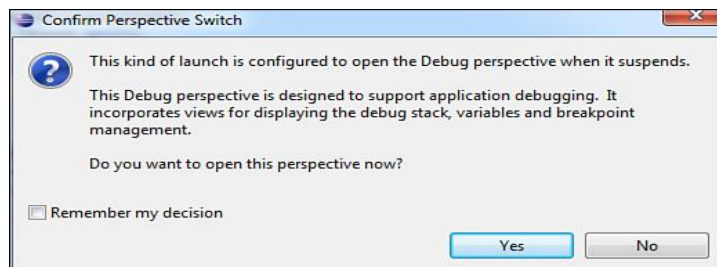


Figure 5.18. Dialog prompting to switch to the Debug perspective

USING THE DEBUG PERSPECTIVE

When the application switches from the Java to the Debug perspective, we see the callback stack, console, code view, and variables, as shown in [Figure 5.19](#).

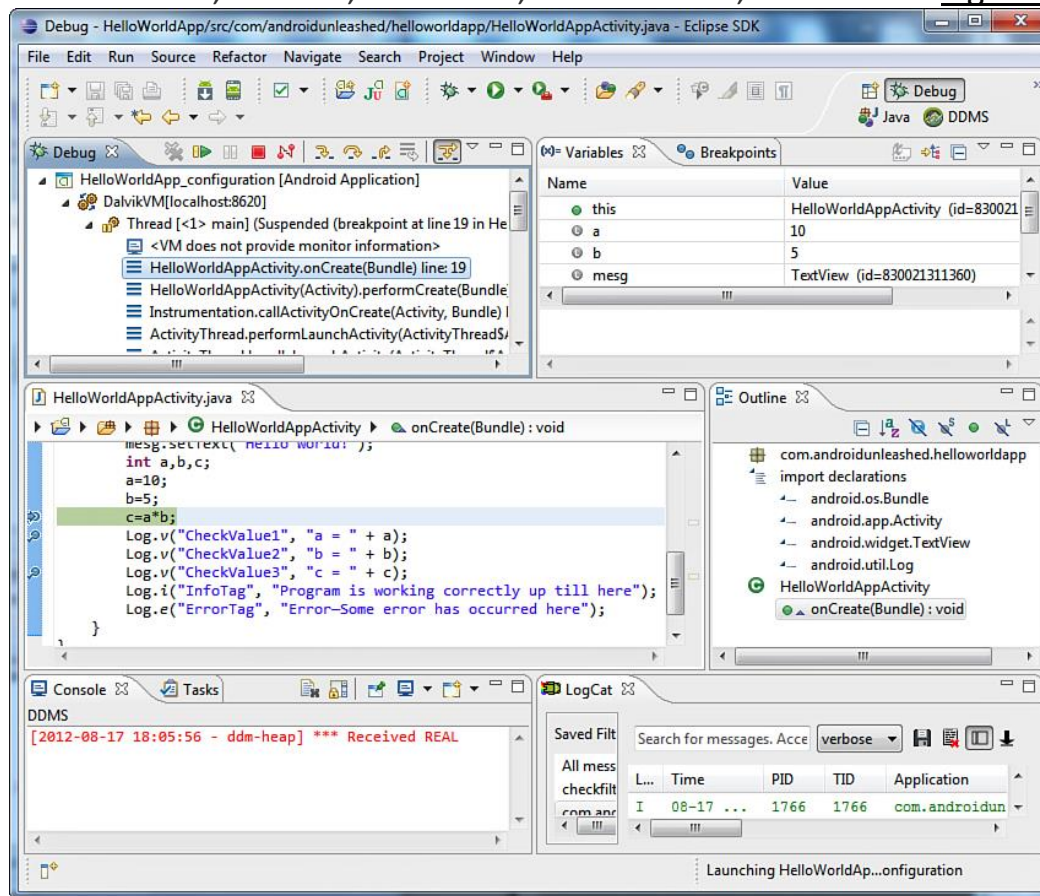


Figure 5.19. The Debug perspective window showing different panes

The following panes are visible by default in Debug perspective:

- **Debug**—On the top left, this pane displays the application being debugged, along with its currently running threads.
- **Variables**—Displays values of the variables at the specific breakpoints.
- **Breakpoints**—Lists all the breakpoints inserted in the code.
- **Editor**—At the middle left, this pane displays the application code pointing to the breakpoint where the application is currently suspended.
- **Outline**—At the center right, this pane lists the imports, classes, methods, and variables used in the application. When we select an item in the Outline pane, the matching source code in the Editorpane is highlighted.
- **Console**—At the bottom left, the pane displays the status of emulator/device activity, such as the launching of activities.
- **LogCat**—At the bottom right, this pane displays the system log messages.

Debug Pane

The Debug pane displays debug session information in a tree hierarchy.

Code Added to the Java Activity File HelloWorldAppActivity.java

```
package com.androidunleashed.helloworldapp;
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;
import android.util.Log;

public class HelloWorldAppActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello_world_app);
        TextView mesg = (TextView)findViewById(R.id.message);
        mesg.setText("Hello World!");
        int a,b,c;
        a=10;
        b=5;
        c=a*b;
        callExcep();
        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");
    }
    public void callExcep() {
        throw new RuntimeException("RuntimeException testing");
    }
}
```

We see that a method, `callExcep()`, that throws a `RuntimeException` is defined. When we don't insert Java Exception Breakpoint, if the application is run, it simply crashes when the exception is thrown. To suspend the thread execution when the exception occurs, select the Add a Java Exception Breakpoint button from the Breakpoints pane. We can then examine the variables and logging messages that might have thrown the exception. After we select the Java Exception Breakpoint button, a dialog box showing all of the available exceptions is displayed, as shown in [Figure 5.25](#) (left).

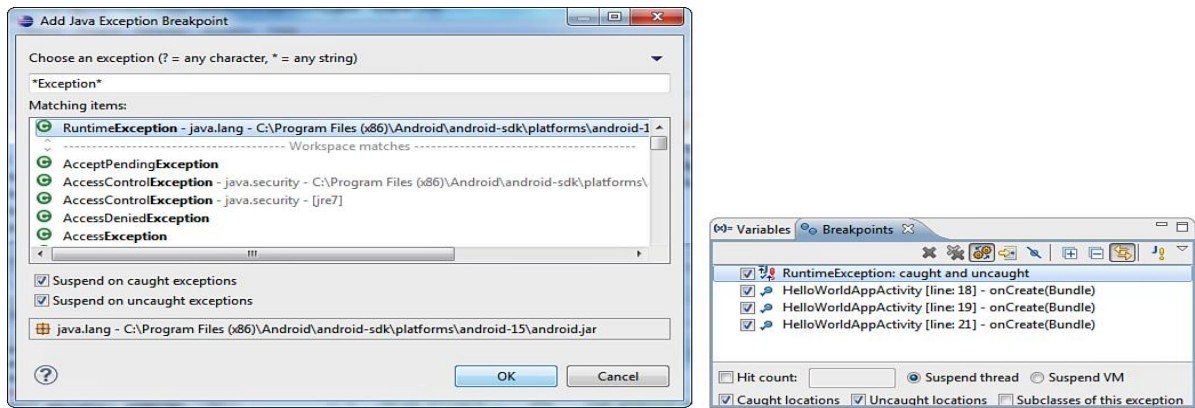


Figure 5.25. Dialog box for adding a Java Exception Breakpoint (left), and the Breakpoints pane showing the RuntimeException (right)

Type the name of the exception we want to catch or select it from the list. Two check boxes are displayed at the bottom so we can choose whether we want to suspend the thread execution on caught or uncaught exceptions. Select Suspend on caught exceptions to suspend the thread execution at locations where the exception is thrown and is caught by a catch clause. Similarly, select Suspend on uncaught exception if we want to suspend thread execution at locations where the exception is thrown but is uncaught. Let's select RuntimeException from the list of available exceptions. Keep the two check boxes selected (by default) and select OK to add an Exception Breakpoint to the application. An exception, RuntimeException caught and uncaught, is added and displayed in the Breakpoints pane, as shown in [Figure 5.25](#) (right). Now the thread is suspended on the statement that throws the exception, allowing us to examine the variables and the LogCat logging pane to see what went wrong.

Q10. Explain HOW INFORMATION IS FETCHED USING DIALOGS AND FRAGMENTS?

Displaying and Fetching Information Using Dialogs and Fragments

A dialog is a smaller window that pops up to interact with the user. It can display important messages and can even prompt for some data. Once the interaction with the dialog is over, the dialog disappears, allowing the user to continue with the application. Fragments, as the name suggests, enable us to fragment or divide our Activities into encapsulated reusable modules, each with its own user interface, making our application suitable to different screen sizes. That is, depending on the available screen size, we can add or remove fragments in our application.

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 DatePicker control.
- **ProgressDialog**—A dialog that displays a ProgressBar control showing the progress of a designated operation. We learned to work with the ProgressBar control in [Chapter 4, “Utilizing Resources and Media.”](#)
- **TimePickerDialog**—A dialog that enables you to set and select a time with a TimePicker control.

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. In Android, the dialogs are called asynchronously; that is, the dialogs are displayed and the main thread that invokes the dialogs returns and continues executing the rest of the application. The rest of the code continues to execute in the background and also allows users to simultaneously interact with the dialog. That means the dialogs in Android are modal in nature. If the dialog is open, users can interact only with the options and controls in the dialog until it is closed. While the user interacts with the dialog, the parent activity resumes its normal execution for efficiency.

Each dialog window is defined within the activity where it will be used. A dialog window can be created once and displayed several times. It can also be updated dynamically.

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. If we do not want to display a dialog, we can remove it from the activity dialog pool by passing its dialog identifier to the removeDialog() method.

The `onCreateDialog()` method is called only once while creating the dialog for the first time, whereas the `onPrepareDialog()` method is called each time the `showDialog()` method is called, allowing the activity to update the dialog before displaying it to the user.

By overriding the `onCreateDialog` method, we specify dialogs that will be created when `showDialog()` is called. Several dialog window types are available in the Android SDK, such as `AlertDialog`, `DatePickerDialog`, and `TimePickerDialog`, that we can readily use in an application. **All the dialog windows are created by extending the `Dialog` class.**

AlertDialog

An `AlertDialog` is a popular method of getting feedback from the user. This pop-up dialog remains there until closed by the user and hence is used for showing critical messages that need immediate attention or to get essential feedback before proceeding further.

The simplest way to construct an `AlertDialog` is to use the static inner class `AlertDialog.Builder` that offers a series of methods to configure an `AlertDialog`.

This example creates a new `AlertDialog.Builder` object called `alertDialog`:

```
AlertDialog.Builder alertDialog = new AlertDialog.Builder(this);
```

In this example, `this` refers to the context, that is, the current activity created here. We can add a title, icon, and message to the `alertDialog` object that we want to display in the dialog. We can define buttons and controls for user interaction to display in the dialog. We can also register event listeners with the dialog buttons for handling events. All these tasks can be easily accomplished through the methods provided by the `AlertDialog.Builder` subclass.

Methods of the `AlertDialog.Builder` Subclass

The methods of the `AlertDialog.Builder` subclass that we can use to configure the `AlertDialog` are

- **`setTitle()` and `setIcon()`**—For specifying the text and icon to appear in the title bar of the dialog box.
- **`setMessage()`**—For displaying a text message in the dialog box.
- **`setPositiveButton()`, `setNeutralButton()`, and `setNegativeButton()`**—For configuring the following three buttons:
 - **Positive button**—Represents the OK button.
 - **Negative button**—Represents the Cancel button.
 - **Neutral button**—Represents a button to perform a function other than OK or Cancel.

Through these three methods, we can set the three buttons to appear in the dialog and also define their location in the dialog box. We can also define the captions and actions of these buttons.

Let's create an Android application to see how AlertDialog is displayed. Name the project AlertDialogApp. In this, we want to display a Button control that, when clicked, displays the AlertDialog. So, first we need to define a Button control in the layout file activity_alert_dialog_app.xml, which appears as shown.

The Layout File activity_alert_dialog_app.xml After Adding the ButtonControl

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/click_btn"
        android:text="Click for Alert Dialog" />
</LinearLayout>
```

To display an AlertDialog, we use the AlertDialog.Builder subclass to create a Builder object. Thereafter, we configure the dialog with a title, message, and buttons with the Builder object. We then define actions for the respective buttons, if any. Finally, the dialog is built and shown on the screen through the Builder object. To do all this, the code into the AlertDialogAppActivity.java Java activity file is as shown in [Listing 6.2](#).

Listing 6.2. Code Written into the Java Activity File AlertDialogAppActivity.java

```
package com.androidunleashed.alertdialogapp;
import android.app.Activity;
import android.os.Bundle;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.view.View;
import android.app.AlertDialog;
import android.content.DialogInterface;

public class AlertDialogAppActivity extends Activity implements OnClickListener {
    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);
    }
    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();
    }
}

```

When click_btn is clicked, the onClick() callback method is executed.

In onClick() a Builder called alertDialog is created. To display the alert dialog on the screen, we provide the Context—the current activity—to the builder object. We then set the icon and title for the dialog box. The title and text of the dialog are set to Alert window and This is an alert, respectively.

After running the application, we see a Button control with the caption Click for Alert Dialog, as shown in [Figure 6.1](#) (left). When we select the Button control, an AlertDialog is displayed with the title Alert window showing the message This is an alert, as shown in [Figure 6.1](#) (right).

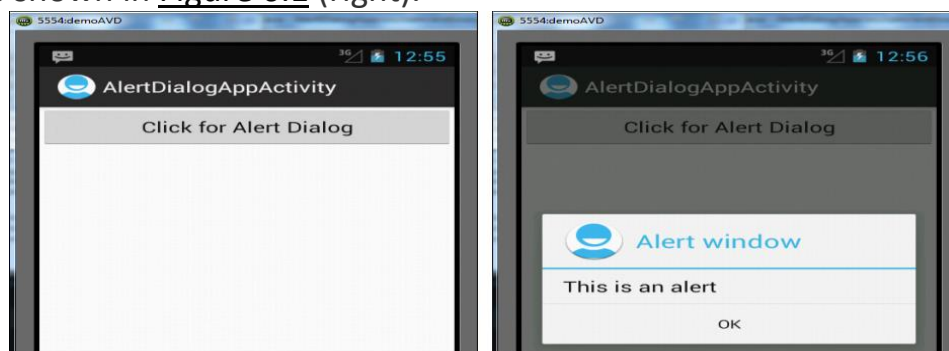


Figure 6.1. Button with the caption Click for Alert Dialog displayed on application startup (left), and the AlertDialog appears on selecting the Button control (right)

Besides showing essential or critical messages to the user that require immediate action, the AlertDialog can also be used for getting input from the user. Let's see how.

Getting Input via the Dialog Box

We modify our current Android project AlertDialogApp to get input from the user. We make the following changes to the application:

- Dynamically create an EditText control and set it as part of the AlertDialog to prompt the user for input.
- Add a TextView control to the layout file to display the data entered by the user in AlertDialog.

To make it more specific, our application asks the user to input a name through AlertDialog, and when the user selects the OK button after entering a name, a welcome message is displayed through the TextView control defined in the layout file. We also add a Cancel button to the AlertDialog, allowing the user to cancel the operation, which terminates the dialog.

We don't have to defining the EditText control in the layout file, as it will be created dynamically with Java code in the activity file. The only thing that we need to define in main.xml is a TextView control that will be used for displaying a Welcome message on the screen.

The code shown in [Listing 6.3](#) is added to the layout file for defining the TextView control. Only the code in bold is newly added.

Listing 6.3. The Layout File activity_alert_dialog_app.xml After Adding the TextViewControl

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/click_btn"
        android:text="Click for Alert Dialog"/>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/response"/>
</LinearLayout>
```

We can see that the newly added code defines a response TextView control in the layout file. Next we add code to the Java Activity file AlertDialogAppActivity.java to do the following tasks:

- Dynamically create an EditText control and set it as the content of the AlertDialog.
 - Access the TextView control from the layout file main.xml and map it to a TextView object.
 - Fetch the name entered by the user in the EditText control and assign it to the TextView object for displaying a welcome message.
 - Register an event listener for the Cancel button. Recall that the purpose of the Cancel button is to cancel the operation and terminate the AlertDialog.
- To perform all these tasks, the code shown in Listing 6.4 is added to AlertDialogAppActivity.java. Only the code in bold is newly added.

Listing 6.4. Code Written into the Java Activity File AlertDialogAppActivity.java

```
package com.androidunleashed.alertdialogapp;
import android.app.Activity;
import android.os.Bundle;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
import android.view.View;
import android.app.AlertDialog;
import android.content.DialogInterface;

public class AlertDialogAppActivity extends Activity implements OnClickListener {
    TextView resp;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_alert_dialog_app);
        resp = (TextView)this.findViewById(R.id.response);
        Button b = (Button)this.findViewById(R.id.click_btn);
        b.setOnClickListener(this);
    }

    public void onClick(View v) {
        AlertDialog.Builder alertDialog = new AlertDialog.Builder(this);
        alertDialog.setTitle("Alert window");
        alertDialog.setIcon(R.drawable.ic_launcher);
        alertDialog.setMessage("Enter your name ");
        final EditText username = new EditText(this);
        alertDialog.setView(username);
        alertDialog.setPositiveButton("OK", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int buttonId) {
                String str = username.getText().toString();
                resp.setText("Welcome "+str+ "!");
                return;
            }
        });
    }
}
```

```

        alertDialog.setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int buttonId) {
                return;
            }
        });
        alertDialog.show();
    }
}

```

We can see here that the TextView control from the layout file is mapped to the TextView object resp. The message set in the AlertDialog is Enter your name. The dynamically created EditText control username is set to appear as content in the AlertDialog, allowing users to enter a name. When a user selects the OK button after entering a name, a Welcome message is displayed, along with the entered name via the TextView object resp. No action is performed when Cancel is clicked; we simply terminate the AlertDialog by returning back to the main activity.

After running the application, we see a Button control with the caption Click for Alert Dialog. When clicked, it displays an AlertDialog with the title Alert window that shows the message Enter your name. There is also an EditText control prompting the user to enter a name, as shown in [Figure 6.2](#) (left).

After the user enters a name in EditText and click the OK button, a welcome message is displayed via the TextView control, as shown in [Figure 6.2](#) (right).

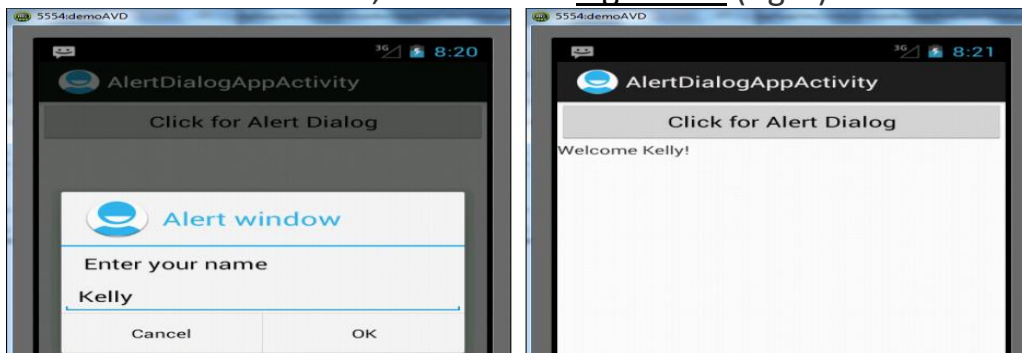


Figure 6.2. Getting input from the user via the AlertDialog (left), and the data entered by the user displayed through TextView (right)

Q9. Explain HOW DATEPICKER DIALOG CAN BE FUNCTIONED. EXPLAIN WITH EXAMPLE?

DatePickerDialog

DatePickerDialog is used to see and modify the date. We can supply the day, month, and year values to its constructor to initialize the date initially displayed through this dialog. The constructor includes a callback listener to inform the current Context when the date has been set or changed. To initialize the current date to the dialog, we use

a Calendar instance. To try DatePickerDialog, let's create a new Android project and name it DatePickerApp. The application contains a TextView and a Button control. When clicked, the Button control displays the DatePickerDialog, and the TextView control displays the date set by the user.

To define the Button and TextView control, let's write the code shown in [Listing 6.5](#) into the layout file activity_date_picker_app.xml.

Listing 6.5. The Layout File activity_date_picker_app.xml After Adding the TextView and Button Controls

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView android:id="@+id/datevw"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <Button android:id="@+id/date_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Set the Date" />
</LinearLayout>
```

We can see that a TextView and a Button control with the IDs datevw and date_button, respectively, are defined in the layout file. The caption Set the Date is set to display in the Button control.

To add action to the application, we need to write some Java code into the activity file DatePickerAppActivity.java. The code in the activity file does the following:

- Access the system's current date through the Calendar instance.
- Display the current system date in the TextView control.
- Display the DatePickerDialog, initialized to display the current system date when the Button control is clicked.
- Access the date set by the user in the DatePickerDialog when its Set button is clicked and display it through the TextView control.

To perform all the preceding tasks, the code shown in [Listing 6.6](#) is written into the Java activity file DatePickerAppActivity.java.

Listing 6.6. Code Written into the Java Activity File DatePickerAppActivity.java

```
package com.androidunleashed.datepickerapp;
```

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

```

import android.widget.TextView;
import android.widget.Button;
import java.util.Calendar;
import android.app.DatePickerDialog;
import android.view.View.OnClickListener;
import android.view.View;
import android.widget.DatePicker;

public class DatePickerAppActivity extends Activity {
    private TextView dispDate;
    private int yr, mon, dy;
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_date_picker_app);
        dispDate = (TextView) findViewById(R.id.datevw);
        Button dateButton = (Button) findViewById(R.id.date_button);
        final Calendar c = Calendar.getInstance();
        yr = c.get(Calendar.YEAR);
        mon = c.get(Calendar.MONTH);
        dy = c.get(Calendar.DAY_OF_MONTH);
        dispDate.setText("Current date is: "+(mon+1)+"-"+dy+"-"+yr);
        dateButton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                new DatePickerDialog(DatePickerAppActivity.this, dateListener, yr,
mon, dy).show();
            }
        });
    }

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

```

When a user clicks a button, we want the DatePickerDialog to be invoked so a date can be selected. A date_button Button is captured from the layout and mapped to a Button object dateButton. A setOnClickListener event listener is associated with

the Button so that when it is clicked, the event handler, that is, the callback method, `onClick()`, is invoked. In the `onClick()` method, a new instance of a `DatePickerDialog` is created using the `DatePickerDialog` constructor and is displayed on the screen.

We want to initialize the `DatePickerDialog` to today's date, so we use the `Calendar` class to set the `DatePickerDialog` control to today's date each time the dialog is shown. An instance of `Calendar` is then created, initially set to the current date. The current year, month, and day are fetched from the `Calendar` instance and passed to the `DatePickerDialog` constructor to initialize it to display the current date. The constructor also includes a callback listener to inform the current `Context` when the date is set or changed.

The `DatePickerDialog` provides a callback listener, `OnDateChangeListener` or `OnDateSetListener`, that listens for when the user has finished setting the date. This occurs when the user clicks the Set button in the `DatePickerDialog`. The `onDateSet()` method is called when the date is set or changed, and we use it to display the set date through the `TextView`.

Note that the month value is zero-based. January is considered month 0 and December month 11. To display the correct month, the `mon` variable, which carries the month number of the selected month, is incremented by 1 before being displayed through the `TextView`.

After running the application, we see a Button control with the caption Set the Date (see [Figure 6.3](#)—top left). When clicked, it displays a `DatePickerDialog` showing today's date, as shown in [Figure 6.3](#) (top right). We can change the day, month, and year as desired by scrolling them in an up or down direction (see [Figure 6.3](#)—bottom left). After we set the date in the `DatePickerDialog`, when the Done button is clicked, the currently set date is displayed via the `TextView` control, as shown in [Figure 6.3](#) (bottom right).

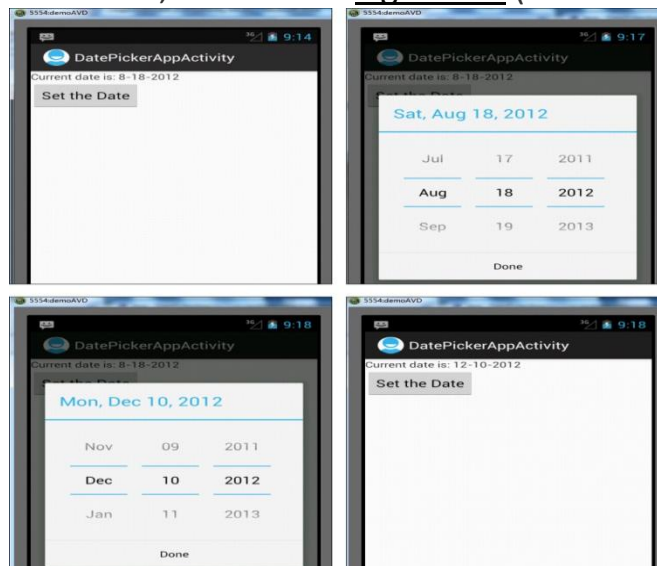


Figure 6.3. A `TextView` displaying the current date with a Button control (top left); the `DatePicker` dialog after clicking the Button control (top right); changing the day, month, and year displayed through the `DatePicker` dialog (bottom left); and displaying the date selected from the `DatePicker` dialog in the `TextView` control (bottom right)

Q10. Explain HOW TIME PICKER DIALOG CAN BE FUNCTIONED. EXPLAIN WITH EXAMPLE?

TimePickerDialog

The TimePickerDialog allows us to set or select time through the built-in Android TimePickerView. We can set the values of hour and minute with values of hour ranging from 0 through 23 and minutes from 0 through 59. The dialog provides a callback listener, OnTimeChangeListener or OnTimeSetListener, which tells us when a time is changed or set by the user.

Again, we create a new Android project, called TimePickerApp, to see how TimePickerDialog works. In this application, we use two controls, TextView and a Button, where the TextView control displays the current system time and the new time set by the user. The Button control is used to invoke the TimePickerDialog; when the Button control is clicked, the TimePickerDialog appears. To define TextView and Button, write the code shown in [Listing 6.7](#) into the layout file `activity_time_picker_app.xml`.

Listing 6.7. The Layout File `activity_time_picker_app.xml` After Adding the TextView and Button Controls

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView android:id="@+id/timevw"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <Button android:id="@+id/time_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Set the Time" />
</LinearLayout>
```

We can see here that the TextView and Button controls are defined with the IDs `timevw` and `time_button`, respectively. The caption on the Button control is Set the Time.

Next, we need to write code into the Java activity file `TimePickerAppActivity.java` to perform the following tasks:

- Invoke the TimePickerDialog when the Button control is clicked.
- Display the current system time in the TextView control.
- Use the Calendar instance to initialize TimePickerDialog to display the current system time.
- Display the newly set time in the TextView control.

To perform these tasks, the code shown in [Listing 6.8](#) is written into the `TimePickerAppActivity.java` file.

Listing 6.8. Code Written into the Java Activity File TimePickerAppActivity.java

```
package com.androidunleashed.timepickerapp;
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;
import android.widget.Button;
import java.util.Calendar;
import android.app.TimePickerDialog;
import android.view.View.OnClickListener;
import android.view.View;
import android.widget.TimePicker;

public class TimePickerAppActivity extends Activity {
    private TextView dispTime;
    private int h, m;
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_time_picker_app);
        dispTime = (TextView) findViewById(R.id.timevw);
        Button timeButton = (Button) findViewById(R.id.time_button);
        final Calendar c = Calendar.getInstance();
        h = c.get(Calendar.HOUR_OF_DAY);
        m = c.get(Calendar.MINUTE);
        dispTime.setText("Current time is: "+h+": "+m);
        timeButton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                new TimePickerDialog(TimePickerAppActivity.this, timeListener,
h,m,true).show();
            }
        });
    }

    private TimePickerDialog.OnTimeSetListener timeListener = new
    TimePickerDialog.OnTimeSetListener() {
        public void onTimeSet(TimePicker view, int hour, int minute) {
            h = hour;
            m = minute;
            dispTime.setText("Current time is: "+h+": "+m);
        }
    };
}
```

In this application, the timePickerDialog is displayed when the Button is selected. The Button with the text Set the Time and the ID time_button is captured from the layout

and mapped to aButton object timeButton. A setOnClickListener event listener is attached to the button so that when it is clicked, the event handler callback method onClick() is invoked. In onClick(), a new instance of a TimePickerDialog is created using the TimePickerDialog constructor and is displayed on the screen.

To initialize the TimePickerDialog so it displays the current system time, we use the Calendar class. An instance of Calendar class is created that is initially set to be the current system time. The current hour and minute values are fetched from the Calendar instance and passed to the TimePickerDialog constructor to initialize it, which sets it to display current system time. We also pass a Boolean value, true, to the constructor to indicate that we want to display the 24-hour clock and not the 12-hour clock that displays AM/PM. The constructor also includes a callback listener to inform the current Context (i.e., the current activity) when the time is set or changed in the TimePickerDialog.

The TimePickerDialog provides a callback listener, OnTimeChangeListener or OnTimeSetListener, that listens for when the user is finished setting the time by selecting the Done button. The onTimeSet() method is called when the time is set or changed and we use it to display the selected time through the TextView.

After running the application, we see a Button control with the caption Set the Time (see [Figure 6.4](#)—top left), which, when clicked, displays a TimePickerDialog showing the current time, as shown in [Figure 6.4](#) (top right). The current hour and minute are displayed, and by scrolling them in an up or down direction, we can change them as desired (see [Figure 6.4](#)—bottom left). After we set the desired time in the TimePickerDialog and select the Done button, the currently set time is displayed via the TextView control, as shown in [Figure 6.4](#) (bottom right).

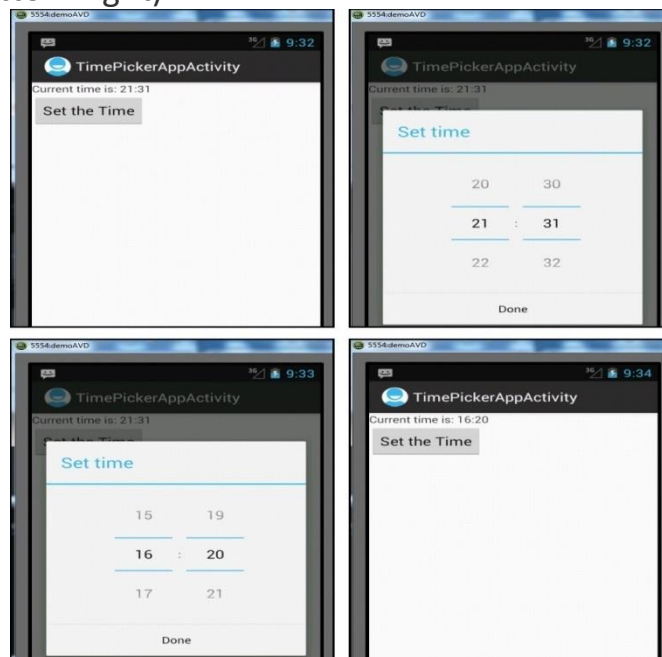


Figure 6.4. The TextView displaying the current time with a Button control (top left), the TimePicker dialog appears after selecting the Button control (top right), changing the

hour and minutes displayed via the TimePicker dialog (bottom left), and displaying the time selected from the TimePicker dialog in the TextView control (bottom right)

Q12. Explain how about combining the two Controls datePickerdialog and timepickerdialog in one application?

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. So, let's write the code shown in Listing 6.9 into the layout file activity_date_time_picker_app.xml to define a TextView and two Button controls.

Listing 6.9. The Layout File activity_date_time_picker_app.xml After Adding the TextView and Button controls

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

We can see here that the TextView and the two Button controls are defined with the IDs datetimevw, date_button, and time_button, respectively. The captions for the two Button controls are Set Date and Set Time, respectively.

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.
- To perform these tasks, the code shown in Listing 6.10 is written into DateTimePickerAppActivity.java.

Listing 6.10. Code Written into the Java Activity File DateTimePickerAppActivity.java

```
package com.androidunleashed.datetimepickerapp;
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;
import android.widget.Button;
import java.util.Calendar;
import android.app.TimePickerDialog;
import android.app.DatePickerDialog;
import android.view.View.OnClickListener;
import android.view.View;
import android.widget.TimePicker;
import android.widget.DatePicker;

public class DateTimePickerAppActivity extends Activity {
    private TextView dateTimeView;
    private Calendar c;
    private int h, m, yr, mon, dy;
    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 current
time 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, int monthOfYear, int dayOf-
Month)
    {
        yr = year;
        mon = monthOfYear;
        dy = dayOfMonth;
        dateTimeView.setText("Current date is "+ (mon+1)+"-"+dy+"-"+yr+" and
current time is: "+h+": "+m);
    }
};

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

```

The respective listeners, OnDateSetListener and OnTimeSetListener, invoke their callback methods, onDateSet() and onTimeSet(), when the Done button in the DatePickerDialog or TimePickerDialog is selected by the user.

The two callback methods access the newly set date and time and display them through the TextView control.

After we run the application, the system's current date and time are displayed through the TextView control. Two Button controls with the captions Set Date and Set Time are

displayed in [Figure 6.5\(left\)](#). When the Set Date button is clicked, the DatePickerDialog showing the system's current date is displayed, as shown in [Figure 6.5 \(middle\)](#). If we scroll in an up or down direction, the day, month, and year can be changed as desired. After we set the desired date, the currently set date and time are displayed via the TextView control, as shown in [Figure 6.5 \(right\)](#).

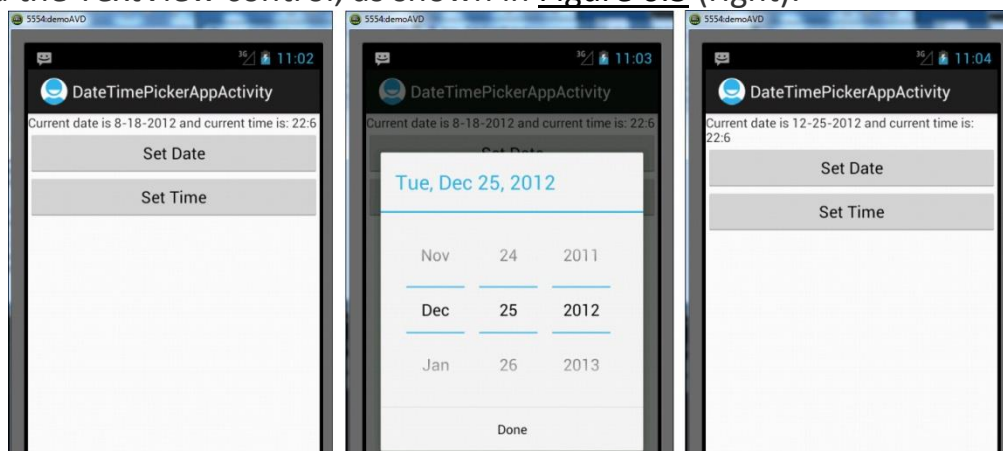


Figure 6.5. The TextView displaying the current date and time and two Buttoncontrols (left), the DatePicker dialog appears when the Set Date button is clicked (middle), and the date selected from the DatePicker dialog displayed in the TextView(right)

Similarly, when the Set Time button is clicked, the TimePickerDialog initialized to the system's current time is displayed, as shown in [Figure 6.6 \(left\)](#). If we scroll in an up or down direction, the hour and minute can be changed as desired, as shown in [Figure 6.6 \(middle\)](#). After we set the desired time in the TimePickerDialog, the currently set date and time are displayed via the TextView control, as shown in [Figure 6.6 \(right\)](#).

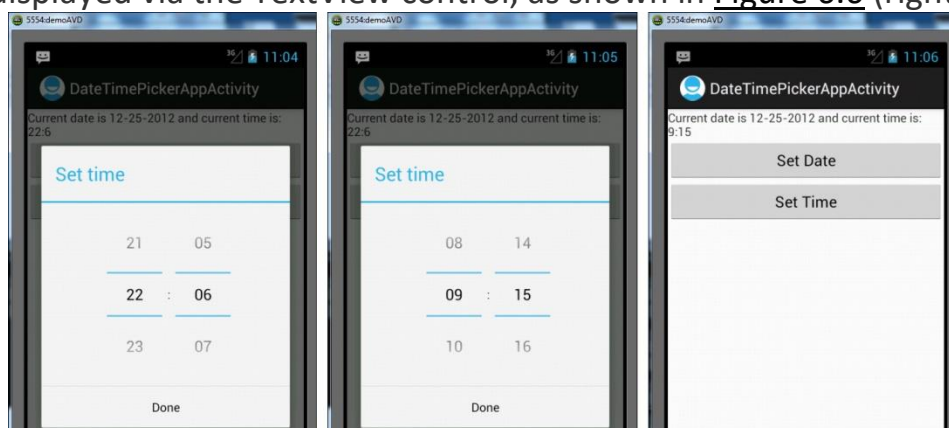


Figure 6.6. The TimePicker dialog appears when the Set Time button is clicked (left), changing the hour and minutes in the TimePicker dialog (middle), and the time selected from the TimePicker dialog displayed in the TextView (right)

We can also format the date and time. Let's modify the DateTimePickerAppActivity.java file to appear as shown in [Listing 6.11](#).

Listing 6.11. Code in the Java Activity File DateTimePickerAppActivity.java

```
package com.androidunleashed.datetimepickerapp;
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;
import android.widget.Button;
import java.util.Calendar;
import android.app.TimePickerDialog;
import android.app.DatePickerDialog;
import android.view.View.OnClickListener;
import android.view.View;
import android.widget.TimePicker;
import android.widget.DatePicker;
import java.text.DateFormat;

public class DateTimePickerAppActivity extends Activity {
    private TextView dateTimeView;
    private Calendar c;
    DateFormat dateTimeFormat = DateFormat.getDateTimeInstance();

    @Override
    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();
        dateTimeView.setText(dateTimeFormat.format(c.getTime()));
        dateButton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                new DatePickerDialog(DateTimePickerAppActivity.this, dateListener,c.
get(Calendar.YEAR), c.get(Calendar.MONTH), c.get(Calendar.DAY_OF_MONTH)).show();
            }
        });
        timeButton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                new TimePickerDialog(DateTimePickerAppActivity.this, timeListener,
c.get(Calendar.HOUR_OF_DAY), c.get(Calendar.MINUTE),true).show();
            }
        });
    }
}
```

```

private DatePickerDialog.OnDateSetListener dateListener = new DatePickerDialog.
OnDateSetListener() {
    public void onDateSet(DatePicker view, int year, int monthOfYear, int dayOf-
Month)
    {
        c.set(Calendar.YEAR,year);
        c.set(Calendar.MONTH,monthOfYear);
        c.set(Calendar.DAY_OF_MONTH,dayOfMonth);
        dateTimeView.setText(DateTimeFormat.format(c.getTime()));
    }
};

private TimePickerDialog.OnTimeSetListener timeListener = new
TimePickerDialog.OnTimeSetListener() {
    public void onTimeSet(TimePicker view, int hour, int minute) {
        c.set(Calendar.HOUR_OF_DAY, hour);
        c.set(Calendar.MINUTE, minute);
        dateTimeView.setText(DateTimeFormat.format(c.getTime()));
    }
};
}

```

After we run the application, the formatted date and time are displayed, as shown in [Figure 6.7](#).

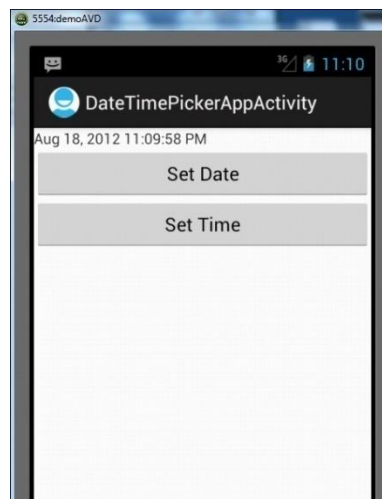


Figure 6.7. The formatted date and time

Q12. Explain HOW FRAGMENTS CAN FUNCTION OR HOW SCREEN ORIENTATION CAN BE SET USING FRAGMENTS. EXPLAIN WITH EXAMPLE?

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.

When developing an application, we need to arrange Views in such a way that the user can view everything in both landscape and portrait mode. If we don't organize the Views with this in mind, problems arise if the user switches modes while running an application. One solution to this problem is one we have already seen—designing an individual layout for each device or screen mode. This solution is time consuming. Another solution is implementing fragments in the application.

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. Let's assume that we have two fragments, Fragment1 and Fragment2, each having its own set of Views. If the screen size is small, we can create two activities, each having a single fragment, and display one activity at a time. If the device screen is able to accommodate views of both Fragment1 and Fragment2, these can be embedded into a single activity to fill up the screen.

A fragment is like a subactivity with its own life cycle and view hierarchy. We can add or remove fragments while the activity is running. Remember that the fragments exist within the context of an activity, and so cannot be used without one.

Tip

Fragments are self-contained and can be used in multiple activities.

To create a fragment, we need to extend the Fragment class and implement several life cycle callback methods, similar to an activity.

The Life Cycle of a Fragment

The life cycle of a fragments is affected by the activity's life cycle in which it is embedded. That is, when the activity is paused, all the fragments in it are paused. Similarly, if an activity is destroyed, all of its fragments are destroyed, as well. 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. The method is used to initialize the items of the fragment that we want to retain when the fragment is resumed after it is paused or stopped. For example, a fragment can save the state into a Bundle object that the activity can use in the `onCreate()` callback while re-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 attached to 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.

A fragment also has a bundle associated with it that serves as its initialization arguments. Like an activity, a fragment can be saved and later automatically restored by the system.

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.

To define a ListView control in the first fragment, the code shown in [Listing 6.12](#) is written into the XML file, fragment1.xml.

Listing 6.12. Code Written into the XML File fragment1.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:background="#0000FF" >
    <ListView
        android:id="@+id/fruits_list"
        android:layout_width="match_parent"
```

```

        android:layout_height="match_parent"
        android:drawSelectorOnTop="false"/>
</LinearLayout>

```

We can see here that a ListView selection widget is defined with the ID `fruits_list`. For distinguishing the two fragments, the background of this fragment is set to blue. To define a TextView control for the second fragment, the code shown in [Listing 6.13](#) is written into the XML file `fragment2.xml`.

Listing 6.13. Code Written into the XML File `fragment2.xml`

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView
        android:id="@+id/selectedopt"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Please select a fruit" />
</LinearLayout>

```

We can see that a `selectedopt` TextView control is defined and is set to display Please select a fruit. Each fragment has a Java class that loads its UI from the XML file, so for the two fragments, we need to add two Java classes to our application. Add `Fragment1Activity.java` and `Fragment2Activity.java` to the `com.androidunleashed.fragmentsapp` package of the application. The code shown in [Listing 6.14](#) is written into the Java class file of the first fragment, `Fragment1Activity.java`.

Listing 6.14. Code Written into the Java Class File `Fragment1Activity.java`

```

package com.androidunleashed.fragmentsapp;
import android.app.Fragment;
import android.os.Bundle;
import android.view.ViewGroup;
import android.view.View;
import android.view.LayoutInflater;
import android.widget.ListView;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.TextView;

public class Fragment1Activity extends Fragment {

```



```

    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(){
            public void onItemClick(AdapterView<?> parent, View v, int position,
long id)
            {
                TextView selectedOpt = (TextView) getActivity().findViewById(R.
id.selectedopt);
                selectedOpt.setText("You have selected "+((TextView) v).getText().
toString());
            }
        });
        return vw;
    }
}

```

We can see that the Java class for the fragment extends the Fragment base class. To access and draw the UI for the fragment, the onCreateView() method is overridden.

In the onCreateView() method, a LayoutInflater object is used to inflate the UI—the ListView control we defined in the fragment1.xml file.

The OnItemClickListener interface is implemented via an anonymous class that implements a callback method, onItemClick(). The reference to the anonymous class is passed to the ListView, to invoke the callback method onItemClick() when any item in the ListView is clicked. In the onItemClick() method, the item selected in the ListView is displayed via the TextView control selectedOpt.

To load the UI of the second fragment from the XML file fragment2.xml, write the code shown in [Listing 6.15](#) into the Java class file of the second fragment Fragment2-Activity.java.

Listing 6.15. Code Written into the Java Class File Fragment2Activity.java

```

package com.androidunleashed.fragmentsapp;
import android.app.Fragment;
import android.os.Bundle;
import android.view.ViewGroup;
import android.view.View;
import android.view.LayoutInflater;

```



```

public class Fragment2Activity extends Fragment {
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState) {
        return inflater.inflate(R.layout.fragment2, container, false);
    }
}

```

Like the Java class of the first fragment, this class also extends the Fragment base class. The `onCreateView()` method is overridden where a `LayoutInflater` object is used to inflate the `TextView` control we defined in the `fragment2.xml` file.

To accommodate both the fragments in the application, the code shown in [Listing 6.16](#) is written into the layout file `activity_fragments_app.xml`.

Listing 6.16. The Layout File `activity_fragments_app.xml` After Adding the Two Fragments

```

<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.Fragment1Activity"
        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>

```

Here we can see that the two fragments are added to the activity through the `<fragment>` elements. The fragments are assigned the IDs `fragment1` and `fragment2`, respectively. The fragments are set to refer to their respective Java class through the `android:name` attribute. The first fragment refers to its Java class file `Fragment1Activity`, which was placed in the `com.androidunleashed.fragmentsapp` package. The orientation of the container `LinearLayout`, was set to `horizontal`, so both the fragments appear beside each other. We don't have to write any code into `FragmentsAppActivity.java`. We can leave the default code unchanged, as shown in [Listing 6.17](#).

Listing 6.17. Code Written into the Java Activity File FragmentsAppActivity.java

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

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

After we run the application, the two UIs defined in Fragment1 and Fragment2 appear side by side. The ListView of Fragment1 displays the list of items, and Fragment2 displays the TextView asking the user to Please select a fruit, as shown in [Figure 6.8](#) (left). After a fruit has been selected from the ListView, its name is displayed through the TextView, as shown in [Figure 6.8](#)(right).

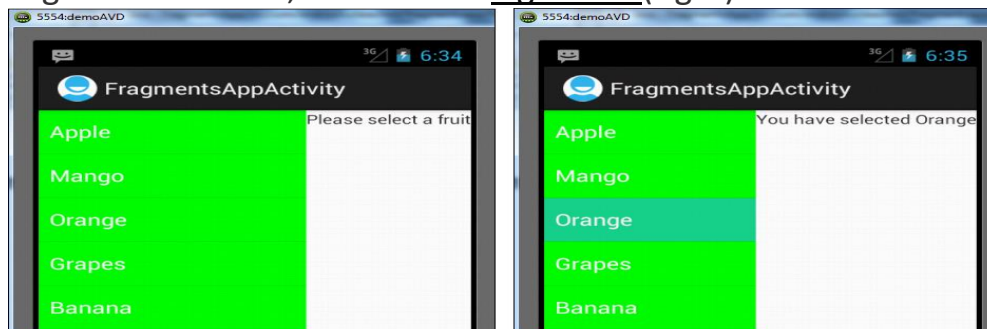


Figure 6.8. ListView and TextView controls displayed via two fragments (left), and the TextView of the second fragment, showing the item selected from the ListView of the first fragment (right)

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`.

FragmentManager

As the name suggests, 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 `FragmentManager` transaction required to add, remove, and replace fragments. To access the `FragmentManager`, the method used is `getFragmentManager()`, as shown here:

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

To perform fragment transactions, we use the instance of the `FragmentManager` as shown here:

```
FragmentManager fragmentManager = getSupportFragmentManager();
FragmentManager.beginTransaction();
```

A new `FragmentManager` is created using the `beginTransaction()` method of the `FragmentManager`. The following code shows how to add a fragment:

```
FragmentManager fragmentManager = getSupportFragmentManager();
FragmentManager.beginTransaction();
FragmentManager.fragment = new Fragment1Activity();
fragmentManager.add(R.id.fragment_container, fragment, "TAG1");
fragmentManager.commit();
```

Here the `Fragment1Activity` is the Java class of the fragment, which is also used to load the UI of the fragment from its XML file. We assume that the `fragment_container` is the ID of the container that exists in the layout file where we want to put our fragment. Usually `LinearLayout` or `FrameLayout` is used as the `fragment_container`. The `TAG1` refers to the unique ID to identify and access the fragment. The `commit()` method is used to apply the changes.

Before we add a fragment, it is a wise idea to check whether it already exists by modifying the code as shown here:

```
FragmentManager fragmentManager = getSupportFragmentManager();
FragmentManager.beginTransaction();
if(null==fragmentManager.findFragmentByTag(TAG1)){
    Fragment1Activity fragment = new Fragment1Activity();
    fragmentManager.add(R.id.fragment_container, fragment, "TAG1");
}
fragmentManager.commit();
```

We can see that the `findFragmentByTag()` method of the `FragmentManager` checks to see whether any fragment with the given tag exists. One more method that can be used to identify a fragment is `findFragmentById()`. The `findFragmentById()` method is used to identify the fragment that is added to the Activity layout. Otherwise, `findFragmentByTag()` is preferred. The `Fragment1Activity` is a Java class meant for loading the Views defined in the fragment's layout file.

To replace the fragment or content being displayed in the `fragment_container` with the View from another fragment, we use the `replace()` method of the `FragmentManager` as shown here:

```
fragmentManager.replace(R.id.fragment_container, fragment2, "TAG2");
```

In this statement, the Views of fragment2 replace the content being displayed in thefragment_container of the Activity layout. To remove a fragment, we identify it either through thefindFragmentById() or findFragmentByTag() methods and then use the remove() method ofFragmentTransaction. The following code identifies the fragment via the findFragmentById()method and then removes it:

```
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
Fragment fragment = fragmentManager.findFragmentById(R.id.fragment);
fragmentTransaction.remove(fragment);
fragmentTransaction.commit();
```

Here we assume that a fragment with the ID fragment exists in the Activity. To identify the fragment through the findFragmentByTag() method, the statement can be replaced by the following:

```
Fragment fragment = fragmentManager.findFragmentByTag(TAG1);
```

Communicating Between Fragments

We can also pass information among the fragments. The two methods provided by the Fragment class to enable communication between fragments are setArguments() and getArguments(). The setArguments() method stores a Bundle in the fragment, whereas the getArguments() method retrieves the Bundle to fetch the passed information.

The following code passes information from fragment 1 to fragment 2. We assume that fragmentis the ID of the fragment container that exists in the layout file where we want to display fragment 2.

```
final Fragment2Activity frag2 = new Fragment2Activity();    #1
final Bundle args = new Bundle();                          #2
String selectedItem="Text to send to fragment 2";          #3
if(null==fragmentManager.findFragmentByTag(FRAG2)){        #4
    args.putString("item", selectedItem);                  #5
    frag2.setArguments(args);                              #6
    fragmentTransaction.replace(R.id.fragment, frag2);     #7
    String tag = null;                                     #8
    fragmentTransaction.addToBackStack(tag);               #9
    fragmentTransaction.commit();
}
```

Statement #1, Fragment2Activity, represents the Java class of the fragment 2. A Java class instance called frag2 is created. Statement #2 creates a Bundle object called args, and a string, selectedItem, is defined in #3 that we want to pass to fragment 2. Statement #4 checks to see whether fragment 2 doesn't already exist in the layout.

The selectedItem variable is saved in the Bundle object args under the key item in #5. The Bundle object args is stored in fragment 2 in #6. Through statement #7, fragment 2 replaces the View in the fragment container of the layout file. The statements #8 and #9 are meant for navigating to the previous fragment as discussed next.

Navigating to Previous Fragments

The Activity stack keeps track of previous Activities. When we press the back button, the Activities in the Activity stack pop up, making their Views visible. In other words, the Activity stack enables us to navigate back to previous screens by using the back button. The same concept is applicable to fragments as well. To add the FragmentTransaction to the back stack, we need to call the addToBackStack() method of FragmentTransaction before calling the commit() method.

In the code shown previously, fragment 2 replaces fragment 1, which was being displayed in the fragment container of the layout file. fragment 1 is added to the back stack, making its Views invisible. Pressing the back button then reverses the previous FragmentTransaction and returns the View of the earlier fragment, fragment 1.

Retrieving Content Passed Through Bundle

We can access the content passed to the fragment via the Bundle that was saved through the setArguments() method. The getArguments() method accesses the Bundle that may be passed to the fragment. The following code accesses the Bundle object passed to the fragment. It also accesses the content passed under the item key and assigns it to the String selectedItem:

```
String selectedItem="";
@Override
public void onCreate(Bundle state) {
    super.onCreate(state);
    if (null == state) state = getArguments();
    if (null != state){
        selectedItem = state.getString("item");
    }
}
```

Saving and Restoring the State of Fragments

Fragments can also save their state to be restored later, just like Activities. The callback methods meant for this purpose are onSaveInstanceState() and onRestoreInstanceState().

The onSaveInstanceState() Callback

The onSaveInstanceState() callback is used for saving the status of the fragment into a Bundle object, which is then used while restoring the fragment. The following code saves

the status of the fragment. It saves the value of the selectedItem variable into the Bundle under theselectedfruit key.

```
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    super.onSaveInstanceState(savedInstanceState);
    savedInstanceState.putString("selectedfruit", selectedItem);
}
```

The onRestoreInstanceState() Callback

The onRestoreInstanceState() callback is for restoring the fragment to its earlier saved state. It uses the content in the Bundle that was used in the onSaveInstanceState() for getting the previously saved content of the fragment. The following code restores the fragment status. It retrieves the value under the selectedfruit key from the Bundle that was saved while callingonSaveInstanceState():

```
@Override
public void onRestoreInstanceState(Bundle savedInstanceState){
    super.onRestoreInstanceState(savedInstanceState);
    selectedItem = savedInstanceState.getString("selectedfruit");
}
```

To understand how fragments are created programmatically, we create an application similar to ourFragmentsApp application showing a ListView and TextView in landscape mode and only aListView in portrait mode. When an item from the ListView is selected, its name is displayed through a TextView. Let's create a new Android project called FragmentCodeApp. Add two layout files, fragment1.xml and fragment2.xml, to the res/layout folder. The fragment1.xml file contains the code to display a ListView control, and fragment2.xml contains code to display aTextView control. The code in fragment1.xml is the same as that as shown in [Listing 6.12](#). Similarly, the code written into fragment2.xml is the same as that shown in [Listing 6.13](#). Write the code shown in [Listing 6.23](#) into the main layout file of the application,activity_fragment_code_app.xml.

Listing 6.23. Code in the Layout File activity_fragment_code_app.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >
    <LinearLayout
        android:id="@+id/fragment1"
        android:layout_weight="1"
        android:layout_width="wrap_content"
```

```

        android:layout_height="match_parent" />
<LinearLayout
    android:id="@+id/fragment2"
    android:layout_weight="0"
    android:layout_width="wrap_content"
    android:layout_height="match_parent" />
</LinearLayout>

```

We can see that two LinearLayout elements are added to the layout file instead of the fragments. This is because we are adding fragments dynamically through code. The Views of the desired fragments are displayed through these LinearLayout containers. The LinearLayout elements are assigned the IDs fragment1 and fragment2, respectively, to identify them in the Java code.

To load the Views of the two layout files defined in fragment1.xml and fragment2.xml, add two Java class files

called Fragment1Activity.java and Fragment2Activity.java to the com.androidunleashed.fragmentcodeapp package of the application. To load the Views defined in fragment2.xml, the code that is written into Fragment2Activity.java is the same as that shown in [Listing 6.15](#). To load the Views defined in fragment1.xml, write the code shown in [Listing 6.24](#) into the Java class file Fragment1Activity.java. Only the code shown in bold is new; the rest of the code is the same as [Listing 6.20](#).

Listing 6.24. Code Written into the Java Class File Fragment1Activity.java

```

package com.androidunleashed.fragmentcodeapp;

import android.view.View;
import android.view.LayoutInflater;
import android.app.Fragment;
import android.os.Bundle;
import android.view.ViewGroup;
import android.widget.ListView;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.TextView;
import android.content.Intent;
import android.app.FragmentManager;

public class Fragment1Activity extends Fragment{
    protected static final String FRAG2 = "2";
    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);
    final FragmentManager fragmentManager = getFragmentManager();
    fruitsList.setOnItemClickListener(new OnItemClickListener(){
        @Override
        public void onItemClick(AdapterView<?> parent, View v, int position,
long id){
            if(null!=fragmentManager.findFragmentByTag(FRAG2)){
                TextView selectedOpt = (TextView) getActivity().findViewById(R.
id.selectedopt);
                selectedOpt.setText("You have selected "+((TextView)
v).getText().toString());
            } else {
                Intent intent = new Intent(getActivity().getApplicationCon-
text(), ShowItemActivity.class);
                intent.putExtra("item", ((TextView) v).getText().toString());
                startActivity(intent);
            }
        }
    });
    return vw;
}
}

```

We want only the View of the first fragment, ListView, displayed in portrait mode, and when an item from the ListView is selected, the name of the selected item is displayed on the next screen. The new screen is created through a new activity. So, add a Java class file called ShowItemActivity.java to the com.androidunleashed.fragmentcodeapp package of the application. In the ShowItemActivity.java file, write the code as shown in [Listing 6.21](#). As said earlier, to inform about the newly added activity file to the application, we need to write a statement as shown in [Listing 6.22](#) in the AndroidManifest.xml file. To the main activity file of the application FragmentCodeAppActivity.java, write the code as shown in [Listing 6.25](#).

Listing 6.25. Code Written into the Java Activity File FragmentCodeAppActivity.java

```
package com.androidunleashed.fragmentcodeapp;

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

public class FragmentCodeAppActivity extends Activity {
    private static final String FRAG1 = "1";
    private static final String FRAG2 = "2";

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_fragment_code_app);
        FragmentManager fragmentManager = getFragmentManager();
        FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
        if (getResources().getConfiguration().orientation == Configuration.ORIENTA-
TION_LANDSCAPE)
        {
            fragmentTransaction.add(R.id.fragment1, new Fragment1Activity(), FRAG1);
            fragmentTransaction.add(R.id.fragment2, new Fragment2Activity(), FRAG2);
        }
        else
        {
            if(null!=fragmentManager.findFragmentByTag(FRAG2))
                fragmentTransaction.remove(fragmentManager.findFragmentByTag(FRAG2));
            fragmentTransaction.add(R.id.fragment1, new Fragment1Activity(), FRAG1);
        }
        fragmentTransaction.commit();
    }
}
```

After we run the application, the output of the application is the same as that shown earlier in [Figures 6.9](#) and [6.10](#).

Tip

Besides making a class extend a Fragment base class, the fragments are also created by making the class extend certain subclasses of the Fragment base class. Such fragments are known as *specialized fragments*.

CREATING SPECIAL FRAGMENTS

After understanding the procedure to create simple fragments, we learn to create specialized fragments such as list fragments, dialog fragments, and preference fragments. To create these, we extend from the following subclasses of the Fragment base class:

- ListFragment
- DialogFragment
- PreferenceFragment

Creating a ListFragment

A ListFragment is a fragment that contains a built-in ListView that can be set to display items from a specified data source. The data source can be an array or a cursor. To understand ListFragments, let's create an application consisting of a ListView and a TextView. The ListView displays some items to choose from. The item selected from the ListView is displayed through a TextView. In this application, the ListView is displayed via a ListFragment, and the TextView is displayed via a simple fragment. The item selected from the ListView in the ListFragment is displayed through the TextView in the simple fragment. Let's name the new Android project ListFragApp. We first create a fragment to hold the TextView control. So, let's add an XML file called fragment2.xml to the res/layout folder of our project. [Listing 6.26](#) shows how to define a TextView control in fragment2.xml.

Listing 6.26. Code in the XML File fragment2.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView
        android:id="@+id/selectedopt"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Please select a fruit" />
</LinearLayout>
```

We can see that a TextView control with the ID selectedopt is defined in a LinearLayout container. The initial text assigned to the TextView control is Please select a fruit. This TextView control is assigned text through Java code to indicate the item selected from the ListView.

To load the UI of the fragment from fragment2.xml, we need to create a Java class file. So, add a Java class file called Fragment2Activity.java under the com.androidunleashed.listfragapp package. Write the code as shown in [Listing 6.27](#) into the Java class file, Fragment2Activity.java.

Listing 6.27. Code Written into the Java Class File of the Second FragmentFragment2Activity

```
package com.androidunleashed.listfragapp;

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

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

We can see that the Java class extends the Fragment base class. The onCreateView() method is overridden when a LayoutInflater object is used to inflate the TextView control UI that we defined in the fragment2.xml file.

We use ListFragment to display the ListView control. As I said earlier, the ListFragment already contains a ListView so we don't need to define a UI for this fragment. We can directly add a Java class file that extends the ListFragment class. In this Java class file, we write code to define the items to be displayed through the ListView of the ListFragment and also to display the item selected from the ListView through the TextView of the Fragment2. So, add a Java class called Fragment1Activity.java to the project and write the code shown in [Listing 6.28](#) into it.

Listing 6.28. Code Written into the Java Class for the First FragmentFragment1Activity.java

```
package com.androidunleashed.listfragapp;

import android.app.ListFragment;
import android.os.Bundle;
import android.widget.ArrayAdapter;
import android.view.View;
import android.widget.ListView;
import android.widget.TextView;

public class Fragment1Activity extends ListFragment {
    final String[] fruits={"Apple", "Mango", "Orange", "Grapes", "Banana"};
```

```

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ArrayAdapter<String> arrayAdpt = new ArrayAdapter<String>(getActivity(),
android.R.layout.simple_list_item_1, fruits);
    setListAdapter(arrayAdpt);
}

```

```

@Override
public void onItemClick(AdapterView l, View v, int position, long id) {
    TextView selectedOpt = (TextView) getActivity().findViewById(R.
id.selectedopt);
    selectedOpt.setText("You have selected "+((TextView) v).getText().
toString());
}
}

```

As expected, the Java class extends the ListFragment base class to create a ListFragment. To display content through the ListView of the ListFragment, an array called fruits is defined and fruit names are assigned to it. In the onCreate() method, an ArrayAdapter object called arrayadpt is defined to display the elements of the fruits array in the simple_list_item_1 mode. When we use the setListAdapter() method, the content in the ArrayAdapter object, arrayadpt, is assigned to the ListView for display. As expected, the onItemClick() method is invoked when any of the fruits displayed through the ListView control is selected. In this method, we display the name of the selected fruit through the TextView control that we defined in fragment2.xml.

To accommodate both the fragments in the application, code is written into activity_list_frag_app.xml as shown in [Listing 6.29](#).

Listing 6.29. The activity_list_frag_app.xml Layout File After Adding Two Fragments

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >
    <fragment
        android:name="com.androidunleashed.listfragapp.Fragment1Activity"
        android:id="@+id/fragment1"
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="match_parent" />
    <fragment
        android:name="com.androidunleashed.listfragapp.Fragment2Activity"

```

```

        android:id="@+id/fragment2"
        android:layout_weight="0"
        android:layout_width="wrap_content"
        android:layout_height="match_parent" />
</LinearLayout>

```

We can see that the fragment1 and fragment2 fragments are added to the activity through the<fragment> elements. The fragments are set to refer to their respective Java classes through theandroid:name attribute. We don't have to write any code into the Java activity file of the applicationListFragAppActivity.java. We leave the default code in the activity file unchanged, as shown in[Listing 6.30](#).

Listing 6.30. Default Code in the Java Activity File ListFragAppActivity.java

```

package com.androidunleashed.listfragapp;

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

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

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

```

After running the application, we see the two fragments side-by-side, as shown in [Figure 6.11](#) (left). TheListView on the left side appears through the ListFragment. The content in the ListView is displayed via the Java class file of the ListFragment, Fragment1Activity.java. The item selected from the ListView is displayed through the TextView defined in Fragment2, as shown in[Figure 6.11](#) (right).

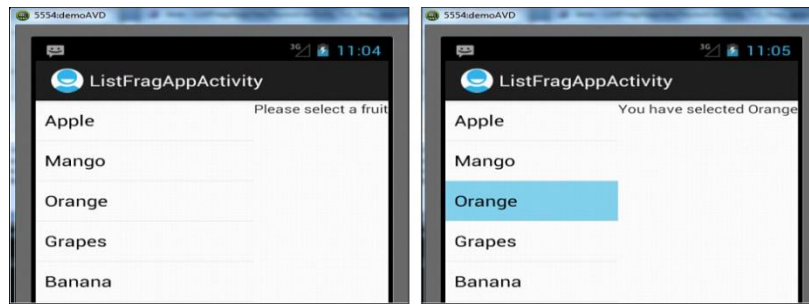


Figure 6.11. The ListView displayed via ListFragment (left), and the Item selected from the ListView of ListFragment, displayed via the TextView of the second fragment (right)

Using a DialogFragment

In Android, dialogs are asynchronous. Synchronous dialogs are those in which the activity suspends its execution until a dialog is dismissed. While the user is interacting with the dialog, no further execution takes place. Asynchronous dialogs are those in which activity continues its normal execution, and at the same time users can interact with the dialog. The activity accesses user interaction with the dialog by implementing callback methods. The dialogs in Android are modal in nature; while a dialog is open, users cannot access any other part of the application. The benefit of calling dialogs asynchronously is that it not only increases code efficiency, but also provides us with the capability to dismiss the dialog through code.

We can display a DialogFragment by extending the DialogFragment base class, which in turn is derived from the Fragment class. To demonstrate DialogFragment, let's create a new Android project called DialogFragApp. In this project, we use two fragments. One is used to show a DialogFragment, and the other displays a TextView. The user's interaction with the DialogFragment is conveyed through the TextView control in the second fragment. The selected button in the DialogFragment is displayed via the TextView control in the second fragment.

Before beginning the creation of DisplayFragment, let's first define the UI of the simple fragment that consists of a TextView. To do so, add an XML file called fragment2.xml to the res/layout folder. Write the code shown in [Listing 6.31](#) into the fragment2.xml file.

Listing 6.31. Code Written into the XML File fragment2.xml

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

```

        android:text="Select Open Dialog Button" />
    </LinearLayout>

```

We can see that a TextView control is defined inside a LinearLayout container. The TextView is assigned the selectedopt ID and is initialized to display the text Select Open Dialog Button. This TextView is used to display the option selected by the user in the DialogFragment.

To load the UI of the fragment from fragment2.xml, a Java class file called Fragment2Activity.java is added to the project. Write the code shown in [Listing 6.32](#) into the Java file Fragment2Activity.java.

Listing 6.32. Code Written into the Java Class for the Second Fragment Fragment2Activity.java

```

package com.androidunleashed.dialogfragapp;

import android.app.Fragment;
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);
    }
}

```

The Java class extends the Fragment base class. The onCreateView() method is overridden when a LayoutInflater object is used to inflate the TextView control UI that we defined in the fragment2.xml file.

To accommodate the fragment defined in fragment2.xml, we need to write code into the layout file activity_dialog_frag_app.xml, as shown in [Listing 6.33](#).

Listing 6.33. The Layout File activity_dialog_frag_app.xml After Adding a Fragment and a Button

```

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

```

```

        android:name="com.androidunleashed.dialogfragapp.Fragment2Activity"
        android:id="@+id/fragment2"
        android:layout_weight="0"
        android:layout_width="wrap_content"
        android:layout_height="match_parent" />
<Button
    android:id="@+id/dialog_button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Open Dialog" />
</LinearLayout>

```

A Button control is defined because we want the DialogFragment to appear only when a button is selected in the application. Both the Fragment and Button controls are nested inside the LinearLayout container. The Fragment is assigned the ID fragment2 and is set to refer to its Java class through the android:name attribute. The Button control is assigned the ID dialog_button, and the caption as Open Dialog. fragment2 is meant to display a TextView to show the option selected by the user in the DialogFragment.

Now it's time to write code to show a DialogFragment. As stated earlier, to show DialogFragment, a Java class needs to extend the DialogFragment class. Let's add a Java class called Fragment1Activity.java under the package com.androidunleashed.dialogfragapp. To display a DialogFragment, write the code shown in [Listing 6.34](#) into the Fragment1Activity.java file.

Listing 6.34. Code Written into the Java Class File of the First FragmentFragment1Activity.java

```

package com.androidunleashed.dialogfragapp;
import android.app.DialogFragment;
import android.app.Fragment;
import android.os.Bundle;
import android.app.Dialog;
import android.app.AlertDialog;
import android.content.DialogInterface;

public class Fragment1Activity extends DialogFragment{
    static Fragment1Activity newInstance(String title) {
        Fragment1Activity fragment = new Fragment1Activity();
        Bundle args = new Bundle();
        args.putString("title", title);
        fragment.setArguments(args);
        return fragment;
    }
}

```



```

@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
    String title = getArguments().getString("title");
    Dialog diag = new AlertDialog.Builder(getActivity())
        .setIcon(R.drawable.ic_launcher)
        .setTitle(title)
        .setPositiveButton("OK", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int whichButton) {
                ((DialogFragAppActivity) getActivity()).PositiveButton();
            }
        })
        .setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int whichButton) {
                ((DialogFragAppActivity) getActivity()).NegativeButton();
            }
        })
        .create();
    return diag;
}
}

```

We can see that to create the DialogFragment, the Java class extends the DialogFragment class. The newInstance() method is used to create a new instance of the fragment. The title of the DialogFragment is passed to this method as an argument, which in turn is stored in the Bundle object and is associated with the fragment that is returned by this method.

To create the view hierarchy of the DialogFragment, the onCreateDialog() method of the DialogFragment class is overridden, and a Bundle object carrying the title of the fragment and other information, if any, is passed to it. In the onCreateDialog() method, an alert dialog builder is used to create a dialog object. In the beginning of this chapter, we learned that AlertDialog is a dialog window that displays a message with optional buttons. In the onCreateDialog() method, an AlertDialog with two buttons, OK and Cancel, is created, and the title that has to be displayed in the fragment is obtained from the title argument saved in the Bundle object. An OnClickListener() is associated with the two buttons OK and Cancel, which results in invoking the respective onClick() method when the respective button is clicked. When OK is selected, thePositiveButton() method from the activity is called. Similarly, when Cancel is selected, theNegativeButton() method from the activity is called. The method returns the created AlertDialog.

In the Java activity file, we need to write code to invoke the DialogFragment. The code must be written to take the necessary action when OK or Cancel is selected from the DialogFragment. The code written into the Java activity file DialogFragAppActivity.java is shown in [Listing 6.35](#).

Listing 6.35. Code Written into the Java Activity File DialogFragAppActivity.java

```
package com.androidunleashed.dialogfragapp;

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

public class DialogFragAppActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_dialog_frag_app);
        Button dialogButton = (Button)findViewById(R.id.dialog_button);
        dialogButton.setOnClickListener(new Button.OnClickListener(){
            @Override
            public void onClick(View arg0) {
                Fragment1Activity dialogFragment = Fragment1Activity.
newInstance("Continue Processing?");
                dialogFragment.show(getFragmentManager(), "Dialog Fragment Example");
            }
        });
    }

    public void PositiveButton() {
        TextView selectedOpt = (TextView)findViewById(R.id.selectedopt);
        selectedOpt.setText("You have selected OK button");
    }

    public void NegativeButton() {
        TextView selectedOpt = (TextView) findViewById(R.id.selectedopt);
        selectedOpt.setText("You have selected Cancel button");
    }
}
```

We want the DialogFragment to appear when the Button is selected from the application. So, we see that the dialogButton Button control is captured from the layout file and is mapped to the Button object dialogButton. An OnClickListener is associated with the Button control, and the onClick() callback method is called if the Button control is selected from the application. In the onClick() method, the DialogFragment is created by

creating a dialogFragment instance of theFragment1Activity Java class file, and the title of the DialogFragment is passed to it asContinue processing.

The DialogFragment is made visible by calling its show() method. The show() method adds the fragment to the given FragmentManager. The code also defines the two methods,PositiveButton() and NegativeButton(), which are invoked when OK and Cancel from theDialogFragment are selected. In both the PositiveButton() and NegativeButton() methods, the selectedOpt TextView control that we defined in fragment2.xml is accessed and mapped to the TextView object selectedOpt. When OK is selected, a message,You have selected OK button, appears in the TextView through the selectedOpt instance. Similarly, when Cancel from the DialogFragment is selected, a message,You have selected Cancel button, appears in the TextView of the second fragment.

After we run the application, a TextView and a Button control are displayed, as shown in Figure 6.12(left). The TextView is displayed through fragment2.xml. The TextView displays the initial textSelect the Open Dialog Button, directing the user to select the Open Dialog button. After we click the Open Dialog button, a dialog fragment with the title Continue Processing opens, showing two buttons, OK and Cancel, as shown in Figure 6.12 (middle). After we click the OK button from the DialogFragment, a message,You have selected OK button, is shown through theTextView control, as shown in Figure 6.12 (right).

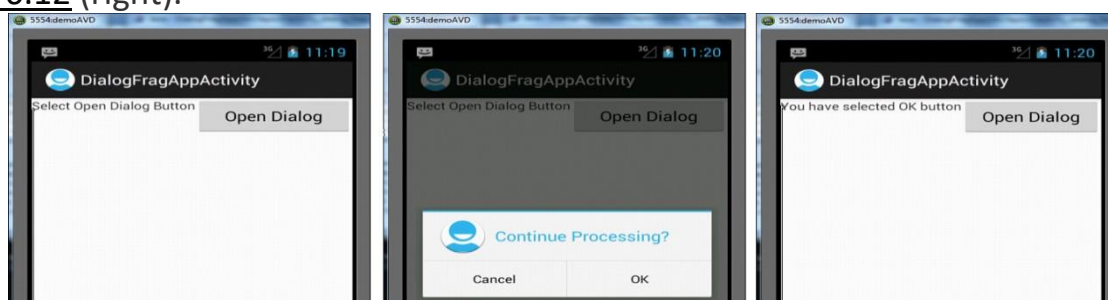


Figure 6.12. The TextView and Button displayed on application startup (left), theDialogFragment appears after clicking the button (middle), and the TextView showing that the DialogFragment OK button was clicked (right)

After we click the Open Dialog button again, the DialogFragment opens up once more. This time, if we select Cancel from the DialogFragment, the TextView displays the message You have selected Cancel button, as shown in Figure 6.13.

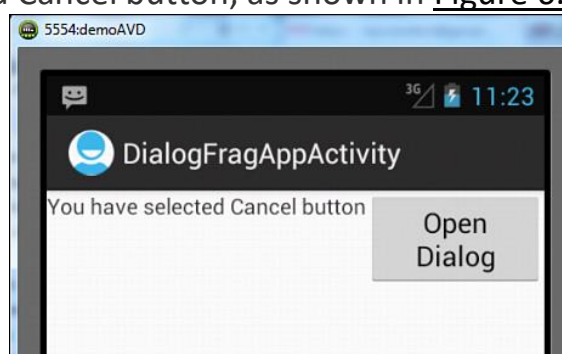


Figure 6.13. The TextView showing that the DialogFragment Cancel button was clicked

Using PreferenceFragment

PreferenceFragment is a fragment that enables users to configure and personalize an application. The PreferenceFragment can contain several Preference Views that help in uniformly setting application preferences with minimum effort. [Table 6.1](#) shows the list of Preference Views that can be displayed via a PreferenceFragment.

Table 6.1. Preference Views That Can Be Displayed in PreferenceFragments

Preference View	Description
PreferenceScreen	The root element of the XML used to define a preference screen
CheckBoxPreference	Displays a simple check box that returns <code>true</code> when checkedh otherwise returns <code>false</code>
ListPreference	Displays a list of radio buttons allowing the user to select one
EditTextPreference	Displays a dialog with an <code>EditText</code> control allowing the user to enter text
RingtonePreference	Displays radio buttons indicating the ringtones available for selection
PreferenceCategory	Used in grouping related preferences in categories
Preference	A custom preference that acts like a <code>Button</code> control

To understand how application preferences are set, let's create a new Android project calledPreferenceFragApp. There are two ways of displaying Preference Views in aPreferenceFragment: through an XML file and through code. We prefer the XML approach, so we first add a folder called xml to the res folder. Inside the res/xml folder, we add an XML file calledpreferences.xml. This file contains the Preference Views we want to display to the user to configure the application. The options selected by the user in Preference Views persist in the application. The code written into the preferences.xml file is shown in [Listing 6.36](#).

Listing 6.36. Code Written into XML File preferences.xml

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android" >
  <PreferenceCategory android:title="Category 1">
    <CheckBoxPreference
      android:title="Pizza"
      android:defaultValue="false"
      android:key="Pizakey" />
    <EditTextPreference android:key="Namekey"
      android:title="Enter your name: "
      android:dialogTitle="Enter your information">
    </EditTextPreference>
  </PreferenceCategory>
```

```

<PreferenceCategory android:title="Category 2">
    <RingtonePreference android:showDefault="true"
        android:key="Audio" android:title="Select sound"
        android:ringtoneType="notification">
    </RingtonePreference>
    <ListPreference android:title="Fruits List "
        android:key="fruits_list"
        android:entries="@array/fruits"
        android:entryValues="@array/fruitselected"
        android:dialogTitle="Choose a fruit">
    </ListPreference>
</PreferenceCategory>
<Preference
    android:title="Submit"
    android:key="submitPref" />
</PreferenceScreen>

```

We can see that the Preference Views are shown in two categories: Category 1 and Category 2. Category 1 includes two Preference Views: a CheckBoxPreference and an EditTextPreference. Category 2 includes the RingtonePreference and ListPreference. Every Preference View needs to have an android:key value that is used to identify and access its value. The android:title attribute is used to assign initial text to the Preference View, and the android:defaultValue attribute is used to assign a default value to the Preference View.

The CheckBoxPreference displays a check box as its UI element, and it stores a value in Boolean form—either true or false. The value true is stored when the check box in CheckBoxPreference is selected, and false when the check box is not selected. The default value false is assigned to the CheckBoxPreference using the android:defaultValue attribute.

The EditTextPreference is assigned the Namekey key, and the title Enter your name: appears as the text of the Preference View. When the EditTextPreference is selected, a dialog titled Enter your information is displayed, asking the user to enter information. When the user clicks OK, the entered information is saved to the preference store.

The RingtonePreference opens a dialog box showing the list of ringtones, allowing the user to select a default ringtone or silent mode. The key assigned to the RingtonePreference is Audio, and the dialog box is assigned the title Select sound. The android:ringtoneType attribute helps in determining the list of ringtones to be displayed. Valid values for android:ringtoneType attribute are ringtone, notification, alarm, and all.

The ListPreference shows a dialog box listing a set of preferences in the form of radio buttons, allowing the user to select one of them. The dialog box is titled Choose a fruit and is assigned the key fruits_list. The android:entries attribute assigns a string array named fruits to the ListPreference to show the list of preferences. That is, the elements in the fruits array display text for the radio buttons displayed via the ListPreference.

The `android:entryValues` attribute defines another array, `fruitselected`, to hold the values of the elements defined in the `fruitsarray`. The `android:entryValues` attribute represents an array that stores the values corresponding to the radio button selected by the user.

The `<Preference>` elements display a Submit button in the `PreferenceFragment` that users click after selecting the desired preferences from Preference Views to either store the preferences or perform another action. The Submit button is assigned the key `submitPref`, which is used to identify it in the Java code.

Next, we need to define two arrays in the `strings.xml` resource file: one to display text for the radio button in the `ListPreference` and the second to store the values of the corresponding elements in the first array. After we define the two arrays, the `strings.xml` file appears as shown in [Listing 6.37](#).

Listing 6.37. The Strings Resource File `strings.xml` After Defining the Two Arrays

```
<resources>
  <string name="app_name">PreferenceFragApp</string>
  <string name="menu_settings">Settings</string>
  <string name="title_activity_preference_frag_app">PreferenceFragAppActivity</string>
  <string-array name="fruits">
    <item>Apple</item>
    <item>Mango</item>
    <item>Orange</item>
    <item>Grapes</item>
    <item>Banana</item>
  </string-array>
  <string-array name="fruitselected">
    <item>You have selected Apple</item>
    <item>You have selected Mango</item>
    <item>You have selected Orange</item>
    <item>You have selected Grapes</item>
    <item>You have selected Banana</item>
  </string-array>
</resources>
```

The elements in the `fruits` array are used to display text for the radio buttons shown in the `ListPreference`, and the elements in the `fruitselected` array show the values that are returned if the corresponding elements in the `fruits` array are selected.

To load the Preference Views defined in `preferences.xml`, a Java class file called `PrefActivity.java` is added to the project. Write the code shown in [Listing 6.38](#) into the Java class file `PrefActivity.java`.

Listing 6.38. Code Written into PreferenceFragment PrefActivity.java

```
package com.androidunleashed.preferencefragapp;

import android.os.Bundle;
import android.app.Activity;
import android.preference.Preference;
import android.preference.Preference.OnPreferenceClickListener;
import android.preference.PreferenceFragment;

public class PrefActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getFragmentManager().beginTransaction().replace(android.R.id.content, new
PrefsFragment()).commit();
    }

    public static class PrefsFragment extends PreferenceFragment {
        @Override
        public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            addPreferencesFromResource(R.xml.preferences);
            Preference submitPref = (Preference) findPreference("submitPref");
            submitPref.setOnPreferenceClickListener(new OnPreferenceClickListener()
{
                public boolean onPreferenceClick(Preference preference) {
                    getActivity().finish();
                    return true;
                }
            });
        }
    }
}
```

To create the PreferenceFragment, a Java class called PrefsFragment is defined that extends the PreferenceFragment class. The addPreferencesFromResource() method is called to load the Preference Views in the PreferenceFragment from the XML file preferences.xml. The Submit button defined in the preferences.xml file through the <Preference> element is accessed and mapped to the Preference object submitPref. An OnPreferenceClickListenablevent handler is added to the submitPref object. Its callback method, onPreferenceClick(), is implemented, which executes when the submitPref Button is clicked. In the onPreferenceClick() method, we finish by closing the PreferenceFragment and returning to PreferenceFragActivity.java to take necessary action on the selected preferences. Through the Java activity

file PreferenceFragAppActivity.java, we display the preferences selected by the user via TextView controls.

To display the options selected from the Preference Views shown in the PreferenceFragment, we need to define four TextView controls in the layout file activity_preference_frag.xml. After defining the four TextView controls activity_preference_frag_app.xml appears as shown in [Listing 6.39](#).

Listing 6.39. The Layout File activity_preference_frag_app.xml After Adding the FourTextView Controls

```
<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:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/pizza"/>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/name"/>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/ringtone"/>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/fruit"/>
</LinearLayout>
```

We can see that the four TextView controls are assigned the IDs pizza, name, ringtone, and fruit. The TextView controls are vertically arranged inside the LinearLayout container. The pizza TextView is used to indicate whether the user has checked the check box in the CheckBoxPreference. The name TextView is used to display the name entered by the user in the EditTextPreference. The ringtone TextView is used to display the type of ring tone selected by the user in the RingtonePreference. The fruit TextView is used to display the fruit selected by the user in the ListPreference.

To display the PreferenceFragment and show the preferences selected by the user, we need to write the code shown in [Listing 6.40](#) into the main activity file PreferenceFragAppActivity.java.

Listing 6.40. Code Written into the Main Activity File PreferenceFragAppActivity.java

```
package com.androidunleashed.preferencefragapp;
import android.app.Activity;
import android.os.Bundle;
import android.content.Intent;
import android.preference.PreferenceManager;
import android.content.SharedPreferences;
import android.widget.TextView;

public class PreferenceFragAppActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_preference_frag_app);
        startActivity(new Intent(this, PrefActivity.class));
    }

    @Override
    public void onResume() {
        super.onResume();
        SharedPreferences prefs=PreferenceManager.getDefaultSharedPreferences(this);
        TextView pizza=(TextView)findViewById(R.id.pizza);
        TextView name=(TextView)findViewById(R.id.name);
        TextView ringtone=(TextView)findViewById(R.id.ringtone);
        TextView fruit=(TextView)findViewById(R.id.fruit);
        if(Boolean.valueOf(prefs.getBoolean("Pizzakey", false)))
            pizza.setText("You have selected Pizza");
        else
            pizza.setText("");
        ringtone.setText("The ringtone selected is "+prefs.getString("Audio",
"Silent"));
        name.setText("The name entered is "+prefs.getString("Namekey", ""));
        String selectedFruit = prefs.getString("fruits_list", "Apple");
        fruit.setText(selectedFruit);
    }
}
```

To display the PreferenceFragment, its activity class, PrefActivity.class, is started. To show the preferences selected by the user in the PreferenceFragment, the TextView controls defined in the layout file main.xml are accessed and mapped to the TextView objects. The pizza, name, ringtone, and fruit TextViews are mapped to the TextView objects pizza, name, ringtone, and fruit, respectively.

To find the options selected in the Preference Views, a SharedPreferences object called prefs is created. To read the value of CheckBoxPreference, we access the shared preferences and call the getBoolean() method, passing the key of the CheckBoxPreference to it. When the CheckBoxPreference Pizza key is passed to the getBoolean() method of the SharedPreferences instance, it returns true or false, indicating whether the check box in CheckBoxPreference is checked.

Thereafter, EditTextPreference is accessed by passing its Name key to the getString() method of the SharedPreferences instance. Similarly, the RingtonePreference and ListPreference are accessed by passing their keys, Audio and fruits_list, to the getString() method of the SharedPreferences instance. The preferences selected by the user in the Preference Views are displayed via TextView controls.

To make the newly added activity PrefActivity.java visible to Android, it is declared in AndroidManifest.xml by adding the following statement in it:

```
<activity android:name=".PrefActivity" android:label="@string/app_name" />
```

After running the application, we see the Preference Views defined in Category 1 and Category 2. The CheckBoxPreference check box is unchecked by default. When the check box is selected, it is checked, as shown in [Figure 6.14](#) (left). When the EditTextPreference with the text enter your name: is selected, a dialog box titled Enter your information pops up. We can enter a name or cancel the operation by selecting Cancel. Let's enter Troy as shown in [Figure 6.14](#) (middle), then click OK to go back to the PreferenceFragment. When Select sound is clicked and which represents RingtonePreference, a dialog box prompting the user to select a ringtone type is opened, as shown in [Figure 6.14](#) (right).

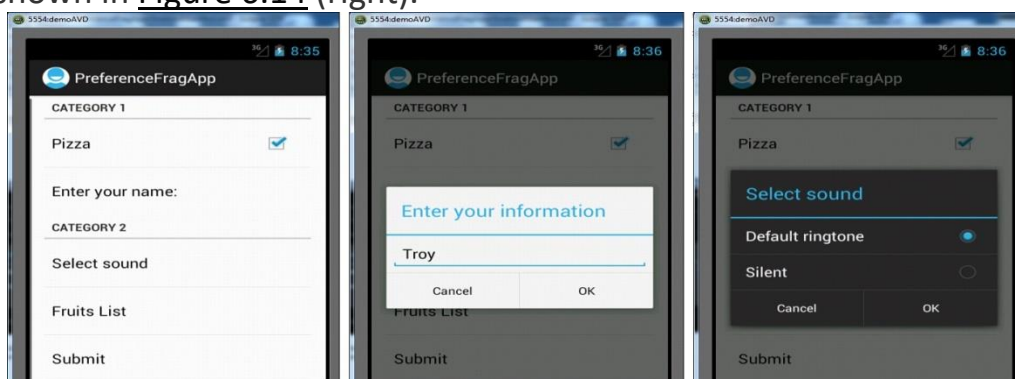


Figure 6.14. A PreferenceFragment showing different Preference Views (left), the EditTextPreference prompting for information (middle), and the RingtonePreference prompting to select a ringtone type (right)

Let's select a Default ringtone followed by clicking the OK button to return to the PreferenceFragment. After we select the ListPreference represented by Fruits List, a dialog box titled Choose a fruit opens up showing several fruits in the form of radio buttons, as shown in [Figure 6.15](#) (left). Let's select Orange. On selecting a fruit, we automatically

return to thePreferenceFragment. Finally, we click the Submit button at the bottom of thePreferenceFragment to close the fragment and display the selected preferences. All the selected preferences are shown through the TextView controls, as shown in Figure 6.15 (right).

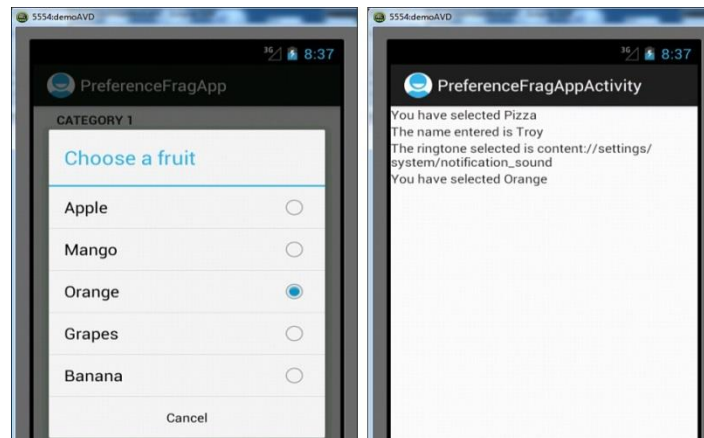


Figure 6.15. The ListPreference showing selectable fruits in the form of radio buttons (left) and all the selected preferences displayed via TextView controls (right)

Unit 5: Building Menus and Storing Data: Creating Interface Menus and Action Bars, Menus and Their Types, Creating Menus Through XML, Creating Menus Through Coding, Applying a Context Menu to a List View, Using the Action Bar, Replacing a Menu with the Action Bar, Creating a Tabbed Action Bar, Creating a Drop-Down List Action Bar Using Databases: Using the SQLiteOpenHelperclasss, Accessing Databases with the ADB, Creating a Data Entry Form, Communicating with SMS and Emails: Understanding Broadcast Receivers, Using the Notification System, Sending SMS Messages with Java Code, Receiving SMS Messages, Sending Email, Working With Telephony Manager.

Q1. EXPLAIN CREATING INTERFACE MENUS. WRITE ABOUT DIFFERENT TYPES OF MENUS IN ANDROID?

Creating Interface Menus and Action Bars:

Menus are a common user interface component in many types of applications. To provide a familiar and consistent user experience, we should use the [Menu](#) APIs to present user actions and other options in our activities. **Android Option Menus** are the primary menus of android. They can be used for settings, search, delete item etc.

Beginning with Android 3.0 (API level 11), Android-powered devices are no longer required to provide a dedicated *Menu* button. With this change, Android apps should migrate away from a dependence on the traditional 6-item menu panel and instead provide an app bar to present common user actions.

Although the design and user experience for some menu items have changed, the semantics to define a set of actions and options is still based on the [Menu](#) APIs.

Three fundamental types of menus or action presentations on all versions of Android:

Options menu and app bar

The [options menu](#) is the primary collection of menu items for an activity. It's where we should place actions that have a global impact on the app, such as "Search," "Compose email," and "Settings."

Context menu and contextual action mode

A context menu is a [floating menu](#) that appears when the user performs a long-click on an element. It provides actions that affect the selected content or context frame.

The [contextual action mode](#) displays action items that affect the selected content in a bar at the top of the screen and allows the user to select multiple items.

Popup menu

A popup menu displays a list of items in a vertical list that's anchored to the view that invoked the menu. It's good for providing an overflow of actions that relate to specific content or to provide options for a second part of a command. Actions in a popup menu should **not** directly affect the corresponding content—that's what contextual actions are for. Rather, the popup menu is for extended actions that relate to regions of content in your activity.

Creating an Options Menu



Figure 1. Options menu in the Browser.

The options menu is where you should include actions and other options that are relevant to the current activity context, such as "Search," "Compose email," and "Settings."

Where the items in our options menu appear on the screen depends on the version for which we developed our application:

If we developed our application for **Android 2.3.x (API level 10) or lower**, the contents of our options menu appear at the top of the screen when the user presses the *Menu* button, as shown in figure 1. When opened, the first visible portion is the icon menu, which holds up to six menu items. If our menu includes more than six items, Android places the sixth item and the rest into the overflow menu, which the user can open by selecting *More*.

If we developed application for **Android 3.0 (API level 11) and higher**, items from the options menu are available in the app bar. By default, the system places all items in the action overflow, which the user can reveal with the action overflow icon on the right side of the app bar (or by pressing the device *Menu* button, if available). To enable quick access to important actions, it can promote a few items to appear in the app bar by adding `android:showAsAction="ifRoom"` to the corresponding `<item>` elements

You can declare items for the options menu from either your [Activity](#) subclass or a [Fragment](#) subclass. If both your activity and fragment(s) declare items for the options menu, they are combined in the UI. The activity's items appear first, followed by those of each fragment in the order in which each fragment is added to the activity. If necessary, you can re-order the menu items with the `android:orderInCategory` attribute in each `<item>` you need to move.

To specify the options menu for an activity, override [onCreateOptionsMenu\(\)](#) (fragments provide their own [onCreateOptionsMenu\(\)](#) callback). In this method, you can inflate your menu resource ([defined in XML](#)) into the [Menu](#) provided in the callback.

```
public boolean onCreateOptionsMenu(Menu menu) {  
    MenuInflater inflater = getMenuInflater();  
    inflater.inflate(R.menu.game_menu, menu);  
    return true;  
}
```

We can also add menu items using [add\(\)](#) and retrieve items with [findItem\(\)](#) to revise their properties with [MenuItem](#) APIs.

Handling click events

When the user selects an item from the options menu (including action items in the app bar), the system calls your activity's `onOptionsItemSelected()` method. This method passes the `MenuItem` selected. We can identify the item by calling `getItemId()`, which returns the unique ID for the menu item (defined by the `android:id` attribute in the menu resource or with an integer given to the `add()` method). We can match this ID against known menu items to perform the appropriate action.

```
public boolean onOptionsItemSelected(MenuItem item) {  
    // Handle item selection  
    switch (item.getItemId()) {  
        case R.id.new_game:  
            newGame();  
            return true;  
        case R.id.help:  
            showHelp();  
            return true;  
        default:  
            return super.onOptionsItemSelected(item);  
    }  
}
```

Creating Contextual Menus

Android context menu appears when user press long click on the element. It is also known as floating menu. A contextual menu offers actions that affect a specific item or context frame in the UI. You can provide a context menu for any view, but they are most often used for items in a `ListView`, `GridView`, or other view collections in which the user can perform direct actions on each item. It doesn't support item shortcuts and icons.

There are two ways to provide contextual actions:



- In a **floating context menu**. A menu appears as a floating list of menu items (similar to a dialog) when the user performs a long-click (press and hold) on a view that declares support for a context menu. Users can perform a contextual action on one item at a time.
- In the **contextual action mode**. This mode is a system implementation of `ActionMode` that displays a *contextual action bar* at the top of the screen with action items that affect the selected item(s). When this mode is active, users can perform an action on multiple items at once (if your app allows it).

Creating a floating context menu

To provide a floating context menu:

1. Register the `View` to which the context menu should be associated by calling `registerForContextMenu()` and pass it the `View`.

If your activity uses a `ListView` or `GridView` and you want each item to provide the same context menu, register all items for a context menu by passing the `ListView` or `GridView` to `registerForContextMenu()`.

2. Implement the `onCreateContextMenu()` method in your `Activity` or `Fragment`.

When the registered view receives a long-click event, the system calls your `onCreateContextMenu()` method. This is where you define the menu items, usually by inflating a menu resource.

```
public void onCreateContextMenu(ContextMenu menu, View v,
                               ContextMenuInfo menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.context_menu, menu);
}
```

1. `MenuInflater` allows you to inflate the context menu from a [menu resource](#). The callback method parameters include the `View` that the user selected and a `ContextMenu.ContextMenuInfo` object that provides additional information about the item selected. If your activity has several views that each provide a different context menu, you might use these parameters to determine which context menu to inflate.
2. Implement `onContextItemSelected()`

```
3. public boolean onContextItemSelected(MenuItem item) {
    AdapterContextMenuInfo info = (AdapterContextMenuInfo) item.getMenuInfo();
    switch (item.getItemId()) {
        case R.id.edit:
            editNote(info.id);
            return true;
        case R.id.delete:
            deleteNote(info.id);
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}
```

4. The `getItemId()` method queries the ID for the selected menu item, which you should assign to each menu item in XML using the `android:id` attribute, as shown in the section about [Defining a Menu in XML](#).
5. When you successfully handle a menu item, return `true`. If you don't handle the menu item, you should pass the menu item to the superclass implementation. If your activity includes fragments, the activity receives this callback first. By calling the superclass when unhandled, the system passes the event to the respective callback method in each fragment, one at a time (in the order each fragment was added) until `true` or `false` is returned. (The default implementation for [Activity](#) and `android.app.Fragment` return `false`, so you should always call the superclass when unhandled.)

Using the contextual action mode

The contextual action mode is a system implementation of [ActionMode](#) that focuses user interaction toward performing contextual actions. When a user enables this mode by selecting an item, a *contextual action bar* appears at the top of the screen to present actions the user can perform on the currently selected item(s). While this mode is enabled, the user can select multiple items (if you allow it), deselect items, and continue to navigate within the activity (as much as you're willing to allow). The action mode is disabled and the contextual action bar disappears when the user deselects all items, presses the BACK button, or selects the *Done* action on the left side of the bar.

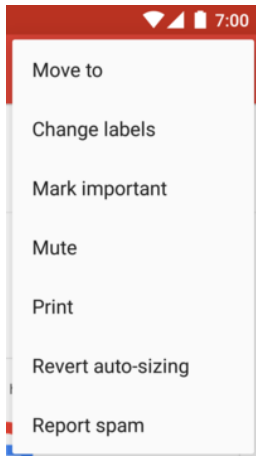
.....

Creating a Popup Menu

A [PopupMenu](#) is a modal menu anchored to a [View](#). Android Popup Menu displays the menu below the anchor text if space is available otherwise above the anchor text. It disappears if you click outside the popup menu.

The `android.widget.PopupMenu` is the direct subclass of `java.lang.Object` class. It's useful for:

- Providing an overflow-style menu for actions that *relate to* specific content
 - Providing a second part of a command sentence (such as a button marked "Add" that produces a popup menu with different "Add" options).
 - Providing a drop-down similar to [Spinner](#) that does not retain a persistent selection.



If we [define our menu in XML](#), here's how we can show the popup menu:

1. Instantiate a [PopupMenu](#) with its constructor, which takes the current application [Context](#) and the [View](#) to which the menu should be anchored.
2. Use [MenuInflater](#) to inflate your menu resource into the [Menu](#) object returned by `PopupMenu.getMenu()`.
3. Call `PopupMenu.show()`.

```
4. <ImageButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/ic_overflow_holo_dark"
    android:contentDescription="@string/descr_overflow_button"
    android:onClick="showPopup" />
```

The activity can then show the popup menu like this:

```
public void showPopup(View v) {
    PopupMenu popup = new PopupMenu(this, v);
    MenuInflater inflater = popup.getMenuInflater();
    inflater.inflate(R.menu.actions, popup.getMenu());
    popup.show();
}
```

In API level 14 and higher, we can combine the two lines that inflate the menu with [PopupMenu.inflate\(\)](#).

The menu is dismissed when the user selects an item or touches outside the menu area. We can listen for the dismiss event using [PopupMenu.OnDismissListener](#).

Using checkable menu items

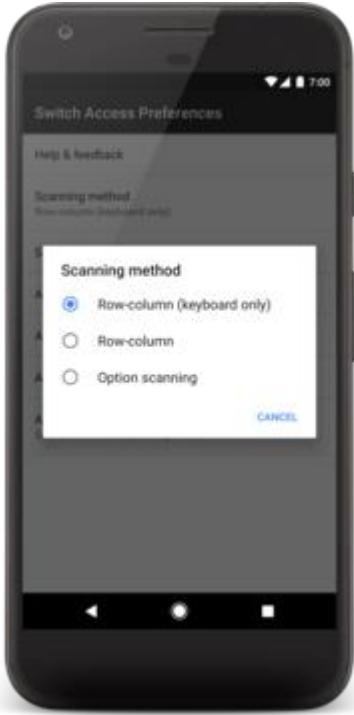


Figure 5. Screenshot of a submenu with checkable items.

A menu can be useful as an interface for turning options on and off, using a checkbox for stand-alone options, or radio buttons for groups of mutually exclusive options. Figure 5 shows a submenu with items that are checkable with radio buttons.

We can define the checkable behavior for individual menu items using the `android:checkable` attribute in the `<item>` element, or for an entire group with the `android:checkableBehavior` attribute in the `<group>` element. For example, all items in this menu group are checkable with a radio button:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <group android:checkableBehavior="single">
    <item android:id="@+id/red"
          android:title="@string/red" />
    <item android:id="@+id/blue"
          android:title="@string/blue" />
  </group>
</menu>
```

The `android:checkableBehavior` attribute accepts either:

- Single : Only one item from the group can be checked (radio buttons)
- All : All items can be checked (checkboxes)
- None : No items are checkable

We can apply a default checked state to an item using the `android:checked` attribute in the `<item>` element and change it in code with the `setChecked()` method.

When a checkable item is selected, the system calls our respective item-selected callback method (such as `onOptionsItemSelected()`). It is here that we must set the state of the checkbox, because a checkbox or radio button does not change its state automatically. We can query the current state of the item (as it was before the user selected it) with `isChecked()` and then set the checked state with `setChecked()`.

```

public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.vibrate:
        case R.id.dont_vibrate:
            if (item.isChecked()) item.setChecked(false);
            else item.setChecked(true);
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}

```

Q2. EXPLAIN DEFINING A MENU THROUGH XML?

Defining a Menu in XML

For all menu types, Android provides a standard XML format to define menu items. Instead of building a menu in our activity's code, We should define a menu and all its items in an XML [menu resource](#). We can then inflate the menu resource (load it as a [Menu](#) object) in our activity or fragment.

Using a menu resource is a good practice for a few reasons:

- It's easier to visualize the menu structure in XML.
- It separates the content for the menu from your application's behavioral code.
- It allows you to create alternative menu configurations for different platform versions, screen sizes, and other configurations by leveraging the [app resources](#) framework.

To define the menu, create an XML file inside our project's `res/menu/` directory and build the menu with the following elements:

`<menu>`

Defines a [Menu](#), which is a container for menu items. A `<menu>` element must be the root node for the file and can hold one or more `<item>` and `<group>` elements.

`<item>`

Creates a [MenuItem](#), which represents a single item in a menu. This element may contain a nested `<menu>` element in order to create a submenu.

`<group>`

An optional, invisible container for `<item>` elements. It allows us to categorize menu items so they share properties such as active state and visibility.

Here's an example menu named `game_menu.xml`:

```

<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/new_game"
        android:icon="@drawable/ic_new_game"
        android:title="@string/new_game"
        android:showAsAction="ifRoom"/>
    <item android:id="@+id/help"
        android:icon="@drawable/ic_help"
        android:title="@string/help" />
</menu>

```

The `<item>` element supports several attributes we can use to define an item's appearance and behavior. The items in the above menu include the following attributes:

`android:id` : A resource ID that's unique to the item, which allows the application to recognize the item when the user selects it.

`android:icon` : A reference to a drawable to use as the item's icon.

`android:title` : A reference to a string to use as the item's title.

`android:showAsAction` : Specifies when and how this item should appear as an action item in the app bar.

To use the menu in our activity, we need to inflate the menu resource (convert the XML resource into a programmable object) using `MenuInflater.inflate()`.

Q3. EXPLAIN CREATING MENU THROUGH JAVA CODE?

Creating Menus Through Coding:

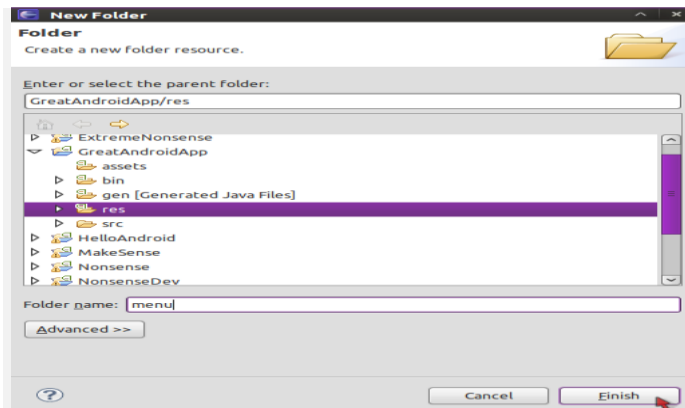
To implement an options menu for an Activity in an [Android app](#), a few fairly straightforward steps are required.

Step 1: Open an Activity Class

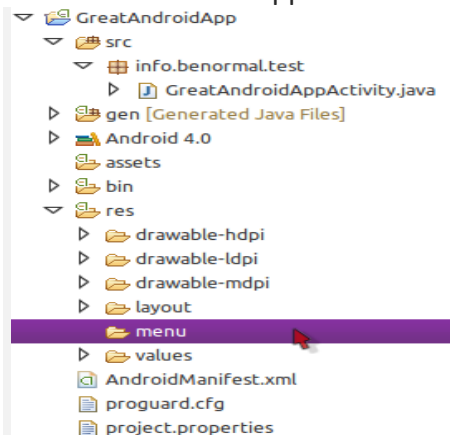
Select your application package and choose “File”, “New”, then “Class” and enter a name of your choice. Remember to make your class extend the Activity class and add it to the application Manifest.

Step 2: Create a Resources Folder

The “res” folder holds all of your application resources. To create a menu, you need a menu folder, so create one inside the “res” folder by selecting it and choosing “File”, “New”, then “Folder” and entering “menu” as the name.



our new folder will appear within the “res” directory:



Step 3: Create a [Menu](#) XML File

Choose the folder and create a new file by selecting “File”, “New”, then “File” and entering a name. You can choose any filename you like, for example “my_options_menu.xml”.

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

Step 4: Add Items to our Menu

You can add one or more items to your options menu depending on the needs of your own project. Add an item for each menu option using the following syntax:

```
<item android:id="@+id/about"
android:title="About" />
<item android:id="@+id/help"
```

```
android:title="Help" />
```

Step 5: Create Icons for our Menu Items

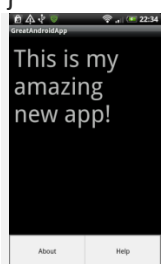
Once you have your icons in their folders, you can alter your menu item XML to include them as follows:

```
<item android:id="@+id/about"
android:icon="@drawable/about"
android:title="About" />
<item android:id="@+id/help"
android:icon="@drawable/help"
android:title="Help" />
```

Step 6: Inflate our Menu Resource

Add the following method to your Java code, inside the class declaration and after the "onCreate" method:

```
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.my_options_menu, menu);
    return true;
}
```



Step 7: Detect User Interaction

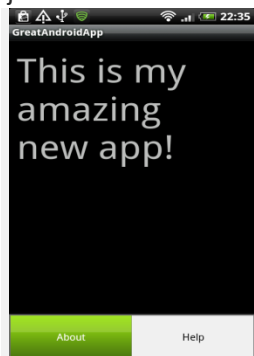
Add the following method outline after the "onCreateOptionsMenu" method:

```
public boolean onOptionsItemSelected(MenuItem item) {
    //respond to menu item selection
}
```

Step 8: Respond to Menu Item Selection

Add a switch statement to your method using the following sample syntax:

```
switch (item.getItemId()) {
    case R.id.about:
        startActivity(new Intent(this, About.class));
        return true;
    case R.id.help:
        startActivity(new Intent(this, Help.class));
        return true;
    default:
        return super.onOptionsItemSelected(item);
}
```



Q4. EXPLAIN HOW CONTEX MENU IS CREATING USING LISTVIEW?

Android Context Menu Example

Android context menu appears when user press long click on the element. It is also known as floating menu. It affects the selected content while doing action on it. It doesn't support item shortcuts and icons.

A **contextual actionbar** is a **menu** that is activated when user **long press** on a specific item.

The **contextual menu** can be applied to almost all views but it is usually used with **ListView**.

A contextual menu offers actions that affect a specific item or context frame in the UI. You can provide a context menu for any view, but they are most often used for items in a [ListView](#), [GridView](#), or other view collections in which the user can perform direct actions on each item.

There are two ways to provide contextual actions:

- In a floating context menu. A menu appears as a floating list of menu items (similar to a dialog) when the user performs a long-click (press and hold) on a view that declares support for a context menu. Users can perform a contextual action on one item at a time.
- In the contextual action mode. This mode is a system implementation of Action Mode that displays a *contextual action bar* at the top of the screen with action items that affect the selected item(s). When this mode is active, users can perform an action on multiple items at once (if your app allows it).

We can distinguish two different type of contextual menu:

- *Floating menu*
- *Contextual action mode (ActionMode)*

The floating menu is used with Android version lower than 3.0 (API level 11). It is essentially a menu that appears when an user long click on an ListView item. ContextMenu support has been in android since API level 1. It's an extension of menu and implements Menu interface. It's actually an interface itself.

To create Android ListView context menu, we have first to define a ActionMode.Callback interface. This interface is called when an user long clicks on an ListView item. The code looks like:

```
private ActionMode.Callback modeCallBack = new ActionMode.Callback()
{
    public boolean onPrepareActionMode(ActionMode mode, Menu menu)
    return false;
}

public void onDestroyActionMode(ActionMode mode) {
    mode = null;
}

public boolean onCreateActionMode(ActionMode mode, Menu menu) {
    return true;
}

public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
}
};
```

Let's see the simple example of context menu in android.

activity_main.xml

Drag one listview from the palette, now the xml file will look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="example.javatpoint.com.contextmenu.MainActivity">

    <ListView
        android:layout_width="368dp"
        android:layout_height="495dp"
        android:id="@+id/listView"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.0"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</android.support.constraint.ConstraintLayout>
```

Create a separate menu_main.xml file in menu directory for menu items.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/call"
        android:title="Call" />
    <item android:id="@+id/sms"
        android:title="SMS" />
</menu>
```

MainActivity.java

```
package example.javatpoint.com.contextmenu;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.ContextMenu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {
```

```

ListView listView;
String contacts[]={"Ajay","Sachin","Sumit","Tarun","Yogesh"};
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    listView=(ListView)findViewById(R.id.listView);
    ArrayAdapter<String> adapter=new ArrayAdapter<String>(this,android.R.layout.simple_list_item
_1,contacts);
    listView.setAdapter(adapter);
    // Register the ListView for Context menu
    registerForContextMenu(listView);
}
@Override
public void onCreateContextMenu(ContextMenu menu, View v, ContextMenu.ContextMenuInfo menu
Info)
{
    super.onCreateContextMenu(menu, v, menuInfo);
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu_main, menu);
    menu.setHeaderTitle("Select The Action");
}
@Override
public boolean onContextItemSelected(MenuItem item){
    if(item.getItemId()==R.id.call){
        Toast.makeText(getApplicationContext(),"calling code",Toast.LENGTH_LONG).show();
    }
    else if(item.getItemId()==R.id.sms){
        Toast.makeText(getApplicationContext(),"sending sms code",Toast.LENGTH_LONG).show();
    }else{
        return false;
    }
    return true;
}
}

```

Q4. EXPLAIN THE ROLE OF ANDROID ACTION BAR. OR EXPLAIN HOW ACTION BARS CAN BE USED IN CREATING MENUS.?

Android Action Bar

The action bar is an important design element, usually at the top of each screen in an app, that provides a consistent familiar look between Android apps. It is used to provide better user interaction and experience by supporting easy navigation through tabs and drop-down lists. It also provides a space for the app or activity's identity, thus enabling the user to know their location in the app, and easy access to the actions that can be performed.

The *toolbar* (formerly known as action bar) is represented as of Android 5.0 via the `Toolbar` view group. It can be freely positioned into your layout file. It can display the activity title, icon, actions which can be triggered, additional views and other interactive items. It can also be used for navigation in your application.

Before Android 5.0 the location of the toolbar (actionbar) was hard coded to the top of the activity. It is possible to disable the toolbar via the used theme, but the default Android themes have it enabled.

The following screenshot shows the toolbar of the Google+ Android application with interactive items and a navigation bar. On top it also indicates that the user can open a navigation bar on the side of the application.

The action bar was introduced in Android 3.0, although support for older versions can be achieved by using the [Android Support Library](#). Before its release, the Options Menu was usually used to provide the actions and functionality that are now put on the action bar. The action bar is included by default in all activities for apps with a `minSdkVersion` of 11. You can disable it and opt to only use the options menu, but for better user experiences it's better to use the action bar as it is visible to the user, while the options menu needs the user to request it and the user might not be aware of its existence in the first place.

Setting up the Action Bar

To start off, we are going to create a new project. Make sure to select a minimum SDK version of 11 or above. When we run our project, the action bar will be included at the top of our app's screen. It is included in all activities that use or inherit from the [Theme.Holo](#) theme – which is the default when the `minSdkVersion` is set to 11 or greater. A typical action bar is shown in the following figure.



The action bar consists of:

- App icon – This is used to identify your app with a logo or icon.
- View control – This can also be used to identify the app or the specific activity the user is on by the title. If your app has different views, it can also be used to display these and allow for easy switching between views.
- Action buttons – These are used to display the most important and/or often used actions. If there isn't enough space to show all of the action buttons, those that don't fit are automatically moved to the action overflow.
- Action overflow – This is used for the lesser used actions.

Adding Actions to the Action Bar

To add actions to the action bar, create a XML file in the `res/menu` directory where you will define each action. It is possible to define the actions in Java code, but you will write less code if you use XML. The contents of `res/menu/main_activity_bar.xml` are shown below.

Creating actions in the toolbar

Entries in the toolbar are typically called *actions*. While it is possible to create entries in the action bar via code, it is typically defined in an XML resource file.

Each menu definition is contained in a separate file in the `res/menu` folder. The Android tooling automatically creates a reference to menu item entries in the R file, so that the menu resource can be accessed.

An activity adds entries to the action bar in its `onCreateOptionsMenu()` method.

The `showAsAction` attribute allows you to define how the action is displayed. For example, the `ifRoom` attribute defines that the action is only displayed in the action bar if there is sufficient screen space available.

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

    <item
        android:id="@+id/action_refresh"
        android:orderInCategory="100"
```



```

    android:showAsAction="always"
    android:icon="@drawable/ic_action_refresh"
    android:title="Refresh"/>
<item
    android:id="@+id/action_settings"
    android:title="Settings">
</item>
</menu>

```

Next, add the string literals to `res/values/strings.xml`, as shown below.

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">ActionBar</string>
    <string name="action_settings">Settings</string>
    <string name="action_search">Search</string>
    <string name="action_record">Record Video</string>
    <string name="action_save">Save</string>
    <string name="action_label">Add Label</string>
    <string name="action_play">Play Video</string>
    <string name="hello_world">Hello world!</string>
</resources>

```

The icon attribute takes a resource ID for an image which will appear on the action bar with or without the title. To display the action title add `withText` to `showAsAction`. For example you could use `android:showAsAction="ifRoom|withText"` to indicate that if there is room on the bar for the action button and text, they should both be shown. To force an action to always be displayed, use `always` on `showAsAction`. However, this is not advisable as it might cause undesirable layout effects on smaller screens. If you must, limit it to one or two items.

- The title will be used in the overflow if there isn't enough space on the action bar for the action item.
- It might not be obvious to the user what the action item does just from its icon alone and so providing a title enables them to long press it to reveal a tool-tip that displays the title.
- The title provides accessibility for sight-impaired users, as the screen reader can read the menu item's title.

Next, we need to implement the `onCreateOptionsMenu()` callback method in our activity. This inflates the menu resource into the given `Menu` object for use in the action bar.

The code for this function is shown below.

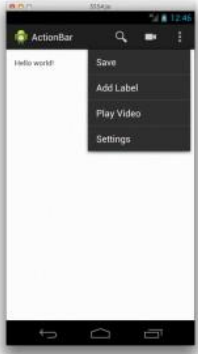
```

public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();

    inflater.inflate(R.menu.main_activity_bar, menu);
    return super.onCreateOptionsMenu(menu);
}

```

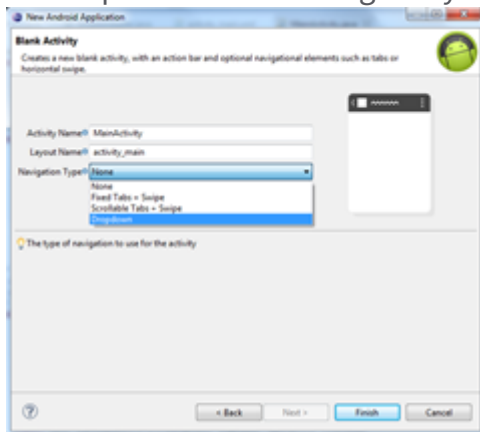
Run the project and you should see something similar to the following figure. Some action buttons appear on the action bar while the rest can be seen on the expanded action overflow. On changing to landscape view, the action bar automatically adapts to the new width and displays more actions according to the guidelines given in the XML file.



Q5. EXPLAIN CREATING A DROP-DOWN LIST ACTION BAR?

Creating a Drop-Down List Action Bar

Android ActionBar supports different navigation types within activities. *Eclipse + ADT* and *Android studio* both support this kind of navigation and we can find everything we need to implement it. To enable the **drop down navigation** in our Activity we can create a new project and we reach the last step before confirming everything we have:



To enable the drop down navigation we have to get the ActionBar reference and then set the navigation type we want, in our case `NAVIGATION_MODE_LIST`. If we look at the code generated we notice:

```
[java]// Set up the action bar to show a dropdown list.
final ActionBar actionBar = getActionBar();
actionBar.setDisplayShowTitleEnabled(false);
actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);
```

Android ActionBar: Implementing the navigation item

We need to create a list of items corresponding to the action we want to support. We have to create a menu with different items. The drop down list is implemented like a Spinner. Spinner is an Android view that displays one item at a time and the user can select one item among a list of items. So to create our drop down navigation list we need simply populate a spinner. To populate a spinner we need an Adapter, in our case a simple `ArrayAdapter`, so we have:

```
ArrayList<String> itemList = new ArrayList<String>();
itemList.add("Section 1");
itemList.add("Section 2");
ArrayAdapter<String> aAdpt = new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1,
android.R.id.text1, itemList);
```

It uses `getActionBarThemedContextCompat`. Looking at this method code we find out that it returns **this** for all the version below Ice Cream Sandwich and another value otherwise.

Now we have to assign our adapter to the ActionBar, we can do it using:

```
actionBar.setListNavigationCallbacks(aAddpt, this);
```

Implementing ActionBar.OnNavigationListener

As when user selects an item we implements ActionBar.OnNavigationListener that has only one method to override:

```
public boolean onNavigationItemSelected(int position, long id) {
```

```
// Our logic
```

Now we have to activate the right view according to the item selected by user. Usually we can use Fragment to enable the right UI.

Running the code we have:



Q6. EXPLAIN ABOUT ROLE OF SQLITE OPEN HELPER CLASS. OR EXPLAIN HOW MENUS ARE CAREATED USINJG SQLITE OPEN HELPERCLASS?

Using the SQLiteOpenHelperclasses:

For maximum control over local data, developers can use SQLite directly by leveraging [SQLiteOpenHelper](#) for executing SQL requests and managing a local database.

SQLite is an **open-source relational database** i.e. used to perform database operations on android devices such as storing, manipulating or retrieving persistent data from the database.

It is embedded in android by default. So, there is no need to perform any database setup or administration task.

The android.database.sqlite.SQLiteOpenHelper class is used for database creation and version management. For performing any database operation, we have to provide the implementation of **onCreate()** and **onUpgrade()** methods of SQLiteOpenHelper class.

Example of android SQLite database

package example.javatpoint.com.sqlitetutorial;

```

public class Contact {
    int _id;
    String _name;
    String _phone_number;
    public Contact(){ }
    public Contact(int id, String name, String _phone_number){
        this._id = id;
        this._name = name;
        this._phone_number = _phone_number;
    }

    public Contact(String name, String _phone_number){
        this._name = name;
        this._phone_number = _phone_number;
    }
    public int getID(){
        return this._id;
    }

    public void setID(int id){
        this._id = id;
    }

    public String getName(){
        return this._name;
    }

    public void setName(String name){
        this._name = name;
    }

    public String getPhoneNumber(){
        return this._phone_number;
    }

    public void setPhoneNumber(String phone_number){
        this._phone_number = phone_number;
    }
}

```

DatabaseHandler.java

Now, let's create the database handler class that extends SQLiteOpenHelper class and provides the implementation of its methods.

```
package example.javatpoint.com.sqlitetutorial;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import java.util.ArrayList;
import java.util.List;

public class DatabaseHandler extends SQLiteOpenHelper {
    private static final int DATABASE_VERSION = 1;
    private static final String DATABASE_NAME = "contactsManager";
    private static final String TABLE_CONTACTS = "contacts";
    private static final String KEY_ID = "id";
    private static final String KEY_NAME = "name";
    private static final String KEY_PH_NO = "phone_number";

    public DatabaseHandler(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
        //3rd argument to be passed is CursorFactory instance
    }

    // Creating Tables
    @Override
    public void onCreate(SQLiteDatabase db) {
        String CREATE_CONTACTS_TABLE = "CREATE TABLE " + TABLE_CONTACTS + "("
            + KEY_ID + " INTEGER PRIMARY KEY," + KEY_NAME + " TEXT,"
            + KEY_PH_NO + " TEXT" + ")";
        db.execSQL(CREATE_CONTACTS_TABLE);
    }

    // Upgrading database
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // Drop older table if existed
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_CONTACTS);

        // Create tables again
        onCreate(db);
    }

    // code to add the new contact
    void addContact(Contact contact) {
        SQLiteDatabase db = this.getWritableDatabase();

        ContentValues values = new ContentValues();
        values.put(KEY_NAME, contact.getName()); // Contact Name
        values.put(KEY_PH_NO, contact.getPhoneNumber()); // Contact Phone

        // Inserting Row
        db.insert(TABLE_CONTACTS, null, values);
        //2nd argument is String containing nullColumnHack
        db.close(); // Closing database connection
    }

    // code to get the single contact
    Contact getContact(int id) {
        SQLiteDatabase db = this.getReadableDatabase();
```

```

Cursor cursor = db.query(TABLE_CONTACTS, new String[] { KEY_ID,
    KEY_NAME, KEY_PH_NO }, KEY_ID + "=?",
    new String[] { String.valueOf(id) }, null, null, null, null);
if (cursor != null)
    cursor.moveToFirst();

Contact contact = new Contact(Integer.parseInt(cursor.getString(0)),
    cursor.getString(1), cursor.getString(2));
// return contact
return contact;
}

// code to get all contacts in a list view
public List<Contact> getAllContacts() {
    List<Contact> contactList = new ArrayList<Contact>();
    // Select All Query
    String selectQuery = "SELECT * FROM " + TABLE_CONTACTS;

    SQLiteDatabase db = this.getWritableDatabase();
    Cursor cursor = db.rawQuery(selectQuery, null);

    // looping through all rows and adding to list
    if (cursor.moveToFirst()) {
        do {
            Contact contact = new Contact();
            contact.setID(Integer.parseInt(cursor.getString(0)));
            contact.setName(cursor.getString(1));
            contact.setPhoneNumber(cursor.getString(2));
            // Adding contact to list
            contactList.add(contact);
        } while (cursor.moveToNext());
    }

    // return contact list
    return contactList;
}

// code to update the single contact
public int updateContact(Contact contact) {
    SQLiteDatabase db = this.getWritableDatabase();

    ContentValues values = new ContentValues();
    values.put(KEY_NAME, contact.getName());
    values.put(KEY_PH_NO, contact.getPhoneNumber());

    // updating row
    return db.update(TABLE_CONTACTS, values, KEY_ID + " = ?",
        new String[] { String.valueOf(contact.getID()) });
}

// Deleting single contact
public void deleteContact(Contact contact) {
    SQLiteDatabase db = this.getWritableDatabase();
    db.delete(TABLE_CONTACTS, KEY_ID + " = ?",
        new String[] { String.valueOf(contact.getID()) });
    db.close();
}

// Getting contacts Count
public int getContactsCount() {
    String countQuery = "SELECT * FROM " + TABLE_CONTACTS;
    SQLiteDatabase db = this.getReadableDatabase();

```

```

        Cursor cursor = db.rawQuery(countQuery, null);
        cursor.close();

        // return count
        return cursor.getCount();
    }
}

```

MainActivity.java

```

package example.javatpoint.com.sqlitetutorial;

```

```

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;
import java.util.List;

```

```

public class MainActivity extends AppCompatActivity {

```

```

    @Override

```

```

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        DatabaseHandler db = new DatabaseHandler(this);

```

```

        // Inserting Contacts

```

```

        Log.d("Insert: ", "Inserting ..");
        db.addContact(new Contact("Ravi", "9100000000"));
        db.addContact(new Contact("Srinivas", "9199999999"));
        db.addContact(new Contact("Tommy", "9522222222"));
        db.addContact(new Contact("Karthik", "9533333333"));

```

```

        // Reading all contacts

```

```

        Log.d("Reading: ", "Reading all contacts..");
        List<Contact> contacts = db.getAllContacts();

```

```

        for (Contact cn : contacts) {
            String log = "Id: " + cn.getID() + " ,Name: " + cn.getName() + " ,Phone: " +
                cn.getPhoneNumber();

            // Writing Contacts to log
            Log.d("Name: ", log);
        }
    }
}

```

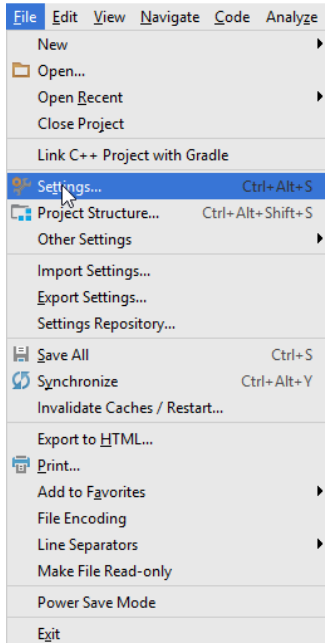
Q7. EXPLAIN HOW DATABASES CAN BE ACCESSED USING ADB?

Accessing Databases with the ADB,

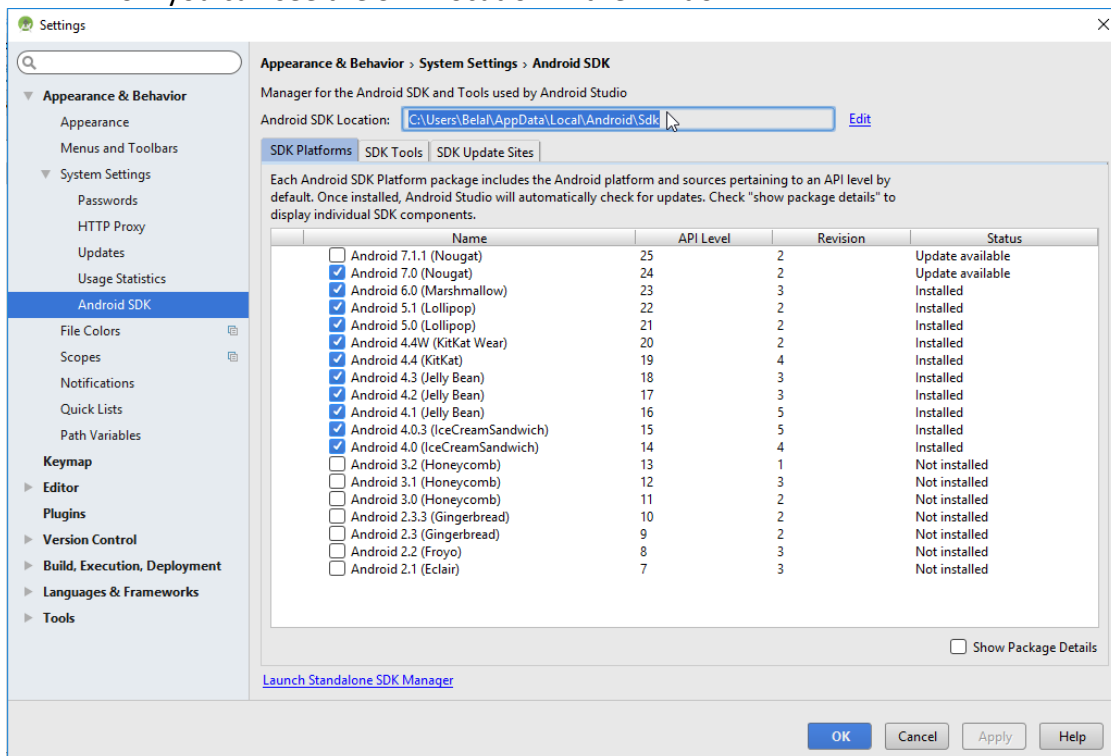
Sometimes while developing applications we need to see the database. Android has SQLite Database and in Android Studio there is no direct option to view the database.

Finding SDK Location

- First you need to find the SDK Location of your system. For this go to **File -> Settings**.



- Now you can see the SDK Location in the window.



- Copy the path.

Going to the SDK Location in Command Prompt

- Now open command prompt and write the following command.

2 cd C:\Users\Belal\AppData\Local\Android\Sdk

3

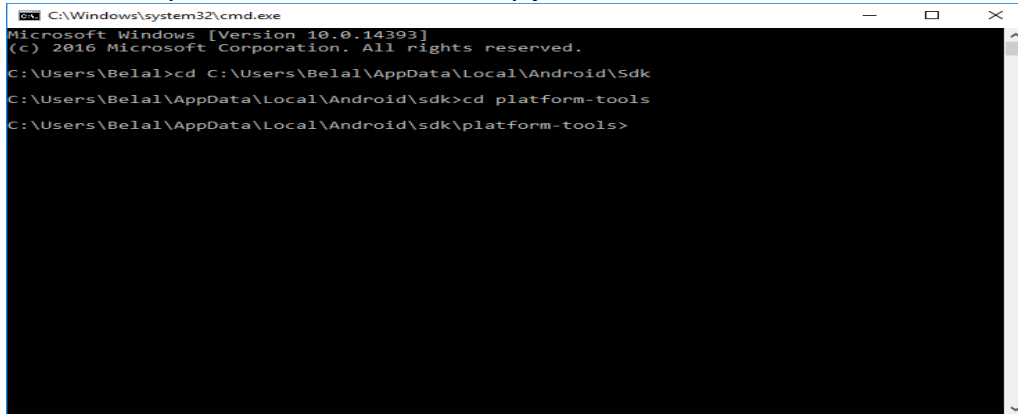
- Again run the following command.

1

2 cd platform-tools

3

- Now you are inside the directory **platform-tools**.



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Belal>cd C:\Users\Belal\AppData\Local\Android\Sdk
C:\Users\Belal\AppData\Local\Android\sdk>cd platform-tools
C:\Users\Belal\AppData\Local\Android\sdk\platform-tools>
```

Enabling ADB Root

- To access database we have to enable adb root. For this simply run the following command.

1

2 adb root

3

Listing Devices

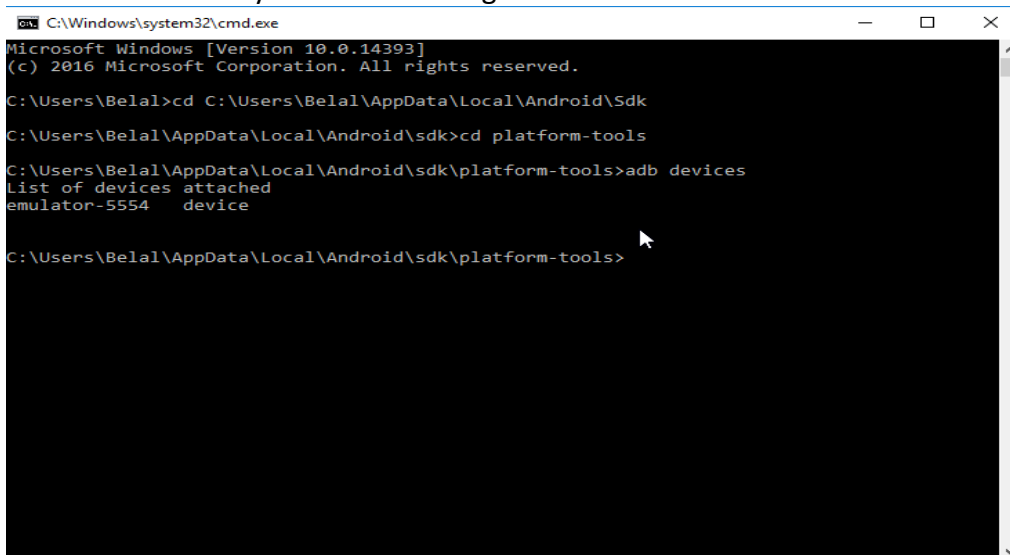
- Now we will list out all the running emulators and devices. For this run the following command.

1

2 adb devices

3

- It will show you all the running devices.



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Belal>cd C:\Users\Belal\AppData\Local\Android\Sdk
C:\Users\Belal\AppData\Local\Android\sdk>cd platform-tools
C:\Users\Belal\AppData\Local\Android\sdk\platform-tools>adb devices
List of devices attached
emulator-5554    device

C:\Users\Belal\AppData\Local\Android\sdk\platform-tools>
```

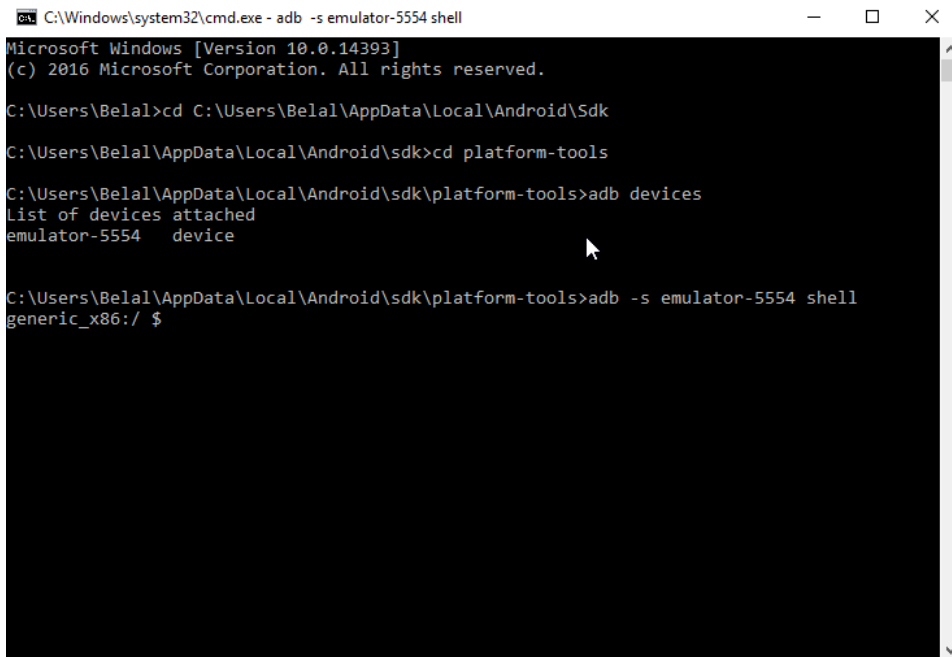
- As you can see I have only one devices for now.

Connecting with Device Shell

- Run the following command to connect a shell to your device.

1
2 adb -s emulator-5554 shell
3

- Dont forget to change the **emulator's name** in the above command.
- And after running the command you will be connected to the device.



```
C:\Windows\system32\cmd.exe - adb -s emulator-5554 shell
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Belal>cd C:\Users\Belal\AppData\Local\Android\Sdk
C:\Users\Belal\AppData\Local\Android\sdk>cd platform-tools
C:\Users\Belal\AppData\Local\Android\sdk\platform-tools>adb devices
List of devices attached
emulator-5554    device

C:\Users\Belal\AppData\Local\Android\sdk\platform-tools>adb -s emulator-5554 shell
generic_x86:/ $
```

Going to Database Folder of the Device

- Run again the following command to access database folder.

1
2 cd data/data/net.simplifiedcoding.androidmysqlsync/databases
3

- Don't forget to change **net.simplifiedcoding.androidmysqlsync** with the package name of the application that you want to access.

Connecting to SQLite Database

- To connect to SQLite Database run the following command.

1
2 sqlite3 NamesDB
3

- In the above command **NamesDB** is my database name you have to change it with your database name.

```
C:\Windows\system32\cmd.exe - adb -s emulator-5554 shell

C:\Users\Belal>cd C:\Users\Belal\AppData\Local\Android\Sdk\platform-tools

C:\Users\Belal\AppData\Local\Android\sdk\platform-tools>adb root

C:\Users\Belal\AppData\Local\Android\sdk\platform-tools>adb devices
List of devices attached
emulator-5554    device

C:\Users\Belal\AppData\Local\Android\sdk\platform-tools>adb -s emulator-5554 shell
generic_x86:/ # cd data/data/net.simplifiedcoding.androidmysqlsync/databases
qlite3 NamesDB.db
SQLite version 3.9.2 2015-11-02 18:31:45
Enter ".help" for usage hints.
sqlite>
```

- Bingo! We are connected with the database.
- Now you can run the command **.help** for the **SQLite Cheat Sheet** to view or modify databases.

Q8. EXPLAIN CREATING DATA ENTRY FORM IN ANDROID?

Creating a Data Entry Form

Android applications often rely upon data supplied by users. Android use of a number of the most common controls used to collect data from the user, including:

- The EditText control
- The Spinner control
- The Checkbox control
- The Button control

We design and implement a form within your Android application which allows the user to supply important feedback to the developer. The user is given a number of options for submitting different types of feedback. This feedback can then be sent to the developer as an email. The form you create will ultimately look like this:

The screenshot shows an Android application interface with a title bar 'Message Launcher'. Below the title bar, the text 'Enter feedback details to send to the developer:' is displayed. The form consists of several input fields: 'Your Name' (with an orange border), 'Your Email', 'Praise' (with a dropdown arrow), and a large text area for 'Feedback Details...'. At the bottom, there is a checkbox labeled 'Would you like an email response?' and a 'Send Feedback' button. The status bar at the top right shows the time as 6:13 PM.

Step 1: Designing the Form

First, you need to give some thought to what kind of data you want to collect from the user. The form may have any number of fields. Consider the types of data you want to collect and choose the appropriate type of control. For example:

- To collect text input, use EditText controls
- To limit the user to a fixed set of responses, use Spinner controls, similar to a drop-down menu
- To collect boolean (yes/no) input, use CheckBox controls
- To allow the user to trigger events, use Button controls

For this tutorial, you will be designing a feedback form. This form collects five pieces of data from the user:

- The user's name (a string)
- The user's email (a string)
- The type of feedback (options: Praise, Gripe, Suggestion or Bug)
- The feedback message (a string)
- Whether or not the user wants an email response (a boolean)

Step 2: Creating the Layout Resource

Begin by creating a layout resource for the form screen. The form will have a bunch of fields, which could span more than a single screen (depending on the device screen size), so you should consider wrapping the entire form within a ScrollView control to enable scrollbars.

The ScrollView control must have exactly one child view, so consider which layout control is most appropriate for the form you want to create. Forms are often contained within a vertically oriented LinearLayout control, so that the form fields cascade down the page vertically, one after another. This also helps the user's focus move from field to field naturally.

A simple form layout resource might look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/ScrollView01"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:scrollbars="vertical">
<LinearLayout
android:layout_width="fill_parent"
android:orientation="vertical"
android:layout_height="fill_parent">

<!--Put form controls here-->

</LinearLayout>
</ScrollView>
```

Step 3: Add a TextView Control (Form Description)

Next, you need to add a TextView control within the LinearLayout control. The TextView control called TextViewTitle displays the form description and purpose to the user. This control displays a string resource called @string/feedbacktitle, which must be defined within the /res/values/strings.xml string resource file. Here is the XML to add to your form layout resource file:

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

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/feedbacktitle"
        android:textSize="10pt">
</TextView>
```

Step 4: Add an EditText Control (Name)

Now you need to add your first EditText control just below the TextView control you just created. This EditText control called EditTextName acts as a form field for the user's name. You can use the hint attribute to supply a string to display in the EditText control when it's empty (e.g. "Type your name here..."). You can also set the inputType attribute of the EditText control to apply name entering logic. Here is the XML to add to your form layout resource file:

```
<EditText
    android:id="@+id/EditTextName"
    android:layout_height="wrap_content"
    android:hint="@string/feedbackname"
    android:inputType="textPersonName"
    android:layout_width="fill_parent">
</EditText>
```

Step 5: Add another EditText Control (Email)

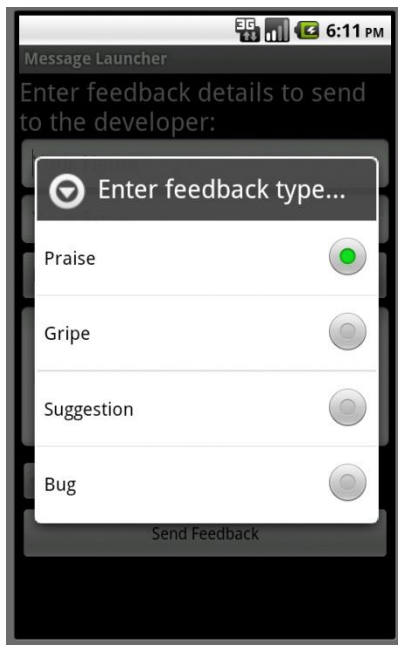
Next, you need to add your second EditText control just below the EditText control called EditTextName. This EditText control called EditTextEmail acts as a form field for the user's email address. Again, set the hint attribute to supply a string to display in the EditText control when it's empty. This time, set the inputType attribute of the EditText control to textEmailAddress, which will make entering emails easier on the user.

Here is the XML to add to your form layout resource file:

```
<EditText
    android:id="@+id/EditTextEmail"
    android:layout_height="wrap_content"
    android:hint="@string/feedbackemail"
    android:inputType="textEmailAddress"
    android:layout_width="fill_parent">
</EditText>
```

Step 6: Add a Spinner Control (Feedback Type)

Next, you need to add a Spinner control just below the EditText control you just created. This Spinner control called SpinnerFeedbackType allows the user to select the type of feedback from a fixed list of options (Praise, Gripe, Suggestion, or Bug).



First, you need to define these choices as individual string resources in the `strings.xml` resource file.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!--Other string resources also defined in this file... -->
    <string name="feedbacktype1">Praise</string>
    <string name="feedbacktype2">Gripe</string>
    <string name="feedbacktype3">Suggestion</string>
    <string name="feedbacktype4">Bug</string>
</resources>
```

Next, create a string array resource using the individual string resources as follows in `/res/values/arrays.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="feedbacktypelist">
        <item>@string/feedbacktype1</item>
        <item>@string/feedbacktype2</item>
        <item>@string/feedbacktype3</item>
        <item>@string/feedbacktype4</item>
    </string-array>
</resources>
```

Now you are ready to configure the Spinner control in your form layout. Begin by supplying the prompt attribute, which will provide a helpful string at the top of the Spinner control. Next, specify the list of string choices using the `entries` attribute—specifically, set the `entries` attribute to the string array you just defined: `@array/feedbacktypelist`.

Here is the XML to add to your form layout resource file:

```
<Spinner
    android:id="@+id/SpinnerFeedbackType"
    android:layout_height="wrap_content"
    android:prompt="@string/feedbacktype"
    android:layout_width="fill_parent"
    android:entries="@array/feedbacktypelist">
</Spinner>
```

Step 7: Add a Multi-Line EditText Control (Feedback)

Next, you need to add one more EditText control just below the Spinner control. This EditText control called EditTextFeedbackBody acts as a form field for the feedback text. Again, set the hint attribute to supply a string to display in the EditText control when it's empty. This time you want to give the user ample space to write praise, gripes, suggestions, or describe bugs in the application. Therefore, you may want to set the inputType attribute of the EditText control to textMultiLine and specify the number of lines to draw using the lines attribute.

Here is the XML to add to your form layout resource file:

```
<EditText
    android:id="@+id/EditTextFeedbackBody"
    android:layout_height="wrap_content"
    android:hint="@string/feedbackbody"
    android:inputType="textMultiLine"
    android:lines="5"
    android:layout_width="fill_parent">
</EditText>
```

Step 8: Add a CheckBox Control

Next, you need to add a CheckBox control just below the EditText control you just created. This CheckBox control called CheckBoxResponse allows the user to choose whether or not they want to request an email response from the app developer. You can use the text attribute to supply a string to display next to the CheckBox control.

Here is the XML to add to your form layout resource file:

```
<CheckBox
    android:id="@+id/CheckBoxResponse"
    android:layout_height="wrap_content"
    android:text="@string/feedbackresponse"
    android:layout_width="fill_parent">
</CheckBox>
```

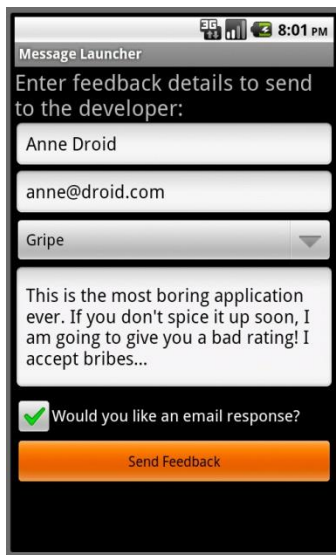
Step 9: Add a Button Control

Finally, you are ready to finish off the form with a Button control. If you want to have a button with text on it, use the Button control; if you prefer a button with a picture on it, use an ImageButton control instead. We will use a Button control here. First, set the text on the Button control using the text attribute. Next, you can easily register a click handler (as opposed to registering it programmatically in your Activity) for your Button control using the onClick attribute.

Here is the XML to add to your form layout resource file:

```
<Button
    android:id="@+id/ButtonSendFeedback"
    android:layout_height="wrap_content"
    android:text="@string/feedbackbutton"
    android:onClick="sendFeedback"
    android:layout_width="fill_parent">
</Button>
```

Excellent! You've finished designing your form. Now, all you need to do is implement the sendFeedback() method in your Activity.



Step 10: Implement a Button click handler

In the Button control, you specified the onClick attribute as sendFeedback. Now you will need to implement a method called sendFeedback() within your Activity class. For example:

```
public void sendFeedback(View button) {
    // Do click handling here
}
```

Step 11: Reading Input from EditText Controls

Now that your form is designed and the controls have been implemented, you next need to collect the form data from the individual fields when the Button control is clicked.

For an EditText control, you use the getText() method.

```
final EditText nameField = (EditText)
findViewById(R.id.EditTextName);
String name = nameField.getText().toString();
```

```
final EditText emailField = (EditText)
findViewById(R.id.EditTextEmail);
String email = emailField.getText().toString();
```

```
final EditText feedbackField = (EditText)
findViewById(R.id.EditTextFeedbackBody);
String feedback = feedbackField.getText().toString();
```

Step 12: Reading Input From Spinner Controls

Your form included a Spinner control. You use the getSelectedItem() method to read the data from this form control.

```
final Spinner feedbackSpinner = (Spinner)
findViewById(R.id.SpinnerFeedbackType);
String feedbackType = feedbackSpinner.getSelectedItem().toString();
```

In this case, the selected item in the Spinner control is the String chosen by the user of the selected item.

Step 13: Reading Input from CheckBox Controls

Finally, your form included a CheckBox control. In this case, the result is just a flag to tell your application if the box was checked or not.


```
final CheckBox responseCheckbox = (CheckBox)
findViewById(R.id.CheckBoxResponse);
boolean bRequiresResponse = responseCheckbox.isChecked();
```

You can use this Boolean value however you want in your app.

Step 14: Generate the Appropriate Email Details

Now that you've got all your form data, you're ready to craft a message. Simply process all the data fields and build an appropriate feedback message. For example, you might use some fields in the message subject, and others in the message body. You can use format strings to help build the appropriate strings, the specifics of which will be discussed in an upcoming quick tip.



Q9. EXPLAIN HOW SMS AND EMAILS CAN BE PROCESSED IN ANDROID?

Communicating with SMS and Emails:

Communication with text messages is a popular way to send and receive information via cellular phones. Communicating through text messages not only consumes fewer network resources but also reduces network congestion, making it an inexpensive mode of communication. Moreover, .

Android smartphones can send and receive messages to or from any other phone that supports Short Message Service (SMS). You have two choices for *sending* SMS messages:

- Use an implicit [Intent](#) to launch a messaging app with the [ACTION_SENDTO](#) intent action.
 - This is the simplest choice for sending messages. The user can add a picture or other attachment in the messaging app, if the messaging app supports adding attachments.
 - Your app doesn't need code to request permission from the user.
 - If the user has multiple SMS messaging apps installed on the Android phone, the App chooser will appear with a list of these apps, and the user can choose which one to use. (Android smartphones will have at least one, such as Messenger.)
 - The user can change the message in the messaging app before sending it.

- The user navigates back to your app using the **Back** button.
- Send the SMS message using the [sendTextMessage\(\)](#) method or other methods of the [SmsManager](#) class.
- This is a good choice for sending messages from your app without having to use another installed app.
- Your app must ask the user for permission before sending the SMS message, if the user hasn't already granted permission.
- The user stays in your app during and after sending the message.
- You can manage SMS operations such as dividing a message into fragments, sending a multipart message, get carrier-dependent configuration values, and so on.

To *receive* SMS messages, use the [onReceive\(\)](#) method of the [BroadcastReceiver](#) class.

Task 1. Launch a messaging app to send a message

In this task you create an app called PhoneMessaging, a new version of the PhoneCallDial app from a previous lesson. The new app launches a messaging app with an implicit intent, and passes a fixed phone number and a message entered by the user.

The user can tap the messaging icon in your app to send the message. In the messaging app launched by the intent, the user can tap to send the message, or change the message or the phone number before sending the message. After sending the message, the user can navigate back to your app using the **Back** button.

1.1 Modify the app and layout

1. Copy the [PhoneCallDial](#) project folder, rename it to **PhoneMessaging**, and refactor it to populate the new name throughout the app project. (See the [Appendix](#) for instructions on copying a project.)
2. Add an icon for the messaging button by following these steps:
 - a. Select drawable in the Project: Android view and choose **File > New > Vector Asset**.
 - b. Click the Android icon next to "Icon:" to choose an icon. To find a messaging icon, choose **Communication** in the left column.
 - c. Select the icon, click **OK**, click **Next**, and then click **Finish**.

Add the following EditText to the existing layout after the `phone_icon` ImageButton:

```
...
<ImageButton
    android:id="@+id/phone_icon"
    ... />

<EditText
    android:id="@+id/sms_message"
    android:layout_width="200dp"
    android:layout_height="wrap_content"
    android:layout_below="@id/number_to_call"
    android:layout_marginTop="@dimen/activity_vertical_margin"
    android:layout_marginRight="@dimen/activity_horizontal_margin"
    android:hint="Enter message here"
    android:inputType="textMultiLine"/>
```

You will use the `android:id sms_message` to retrieve the message in your code. You can use `@dimen/activity_horizontal_margin` and `@dimen/activity_vertical_margin` for the EditText margins because they are already defined in the `dimens.xml` file. The EditText view uses the `android:inputType` attribute set to `"textMultiLine"` for entering multiple lines of text.

After adding hard-coded strings and dimensions, extract them into resources:

`android:layout_width="@dimen/edittext_width"`: The width of the EditText message (200dp).

`android:hint="@string/enter_message_here":` The hint for the EditText ("Enter message here").

Add the following ImageButton to the layout after the above EditText:

```
<ImageButton
    android:id="@+id/message_icon"
    android:contentDescription="Send a message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/activity_vertical_margin"
    android:layout_toRightOf="@id/sms_message"
    android:layout_toEndOf="@id/sms_message"
    android:layout_below="@id/phone_icon"
    android:src="@drawable/ic_message_black_24dp"
    android:onClick="smsSendMessage"/>
```

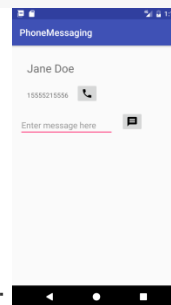
You will use the `android:id` `message_icon` to refer to the ImageButton for launching the messaging app. Use the vector asset you added previously (such as `ic_message_black_24dp` for a messaging icon) for the ImageButton.

After adding the hard-coded string for the `android:contentDescription` attribute, extract it into the resource `send_a_message`.

The `smsSendMessage()` method referred to in the `android:onClick` attribute remains highlighted until you create this method in the MainActivity, which you will do in the next step.

Click `smsSendMessage` in the `android:onClick` attribute, click the red light bulb that appears, and then select **Create smsSendMessage(View) in 'MainActivity'**. Android Studio automatically creates the `smsSendMessage()` method in MainActivity as public, returning void, with a View parameter. This method is called when the user taps the `message_icon` ImageButton.

```
public void smsSendMessage(View view) {
}
```



Your app's layout should now look like the following figure:

1.2 Edit the onClick method in MainActivity

Inside the `smsSendMessage()` method in MainActivity, get the phone number from the `number_to_call` TextView, and concatenate it with the `smsto:` prefix (as in `smsto:14155551212`) to create the phone number URI string `smNumber`:

```
...
TextView textView = (TextView) findViewById(R.id.number_to_call);
// Use format with "smsto:" and phone number to create smNumber.
String smNumber = String.format("smsto: %s",
                                textView.getText().toString());
...
```

Get the string of the message entered into the EditText view:

```
...
// Find the sms_message view.
EditText smsEditText = (EditText) findViewById(R.id.sms_message);
// Get the text of the SMS message.
String sms = smsEditText.getText().toString();
...
```

Create an implicit intent (`smsIntent`) with the intent action `ACTION_SENDTO`, and set the phone number and text message as intent data and extended data, using `setData()` and `putExtra`:

```

...
// Create the intent.
Intent smsIntent = new Intent(Intent.ACTION_SENDTO);
// Set the data for the intent as the phone number.
smsIntent.setData(Uri.parse(smsNumber));
// Add the message (sms) with the key ("sms_body").
smsIntent.putExtra("sms_body", sms);
...

```

The [putExtra\(\)](#) method needs two strings: the key identifying the type of data ("sms_body") and the data itself, which is the text of the message (sms). For more information about common intents and the putExtra() method, see [Common Intents: Text Messaging](#).

Add a check to see if the implicit intent resolves to a package (a messaging app). If it does, send the intent with startActivity(), and the system launches the app. If it does not, log an error.

```

...
// If package resolves (target app installed), send intent.
if (smsIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(smsIntent);
} else {
    Log.e(TAG, "Can't resolve app for ACTION_SENDTO Intent");
}
...

```

The full method should now look like the following:

```

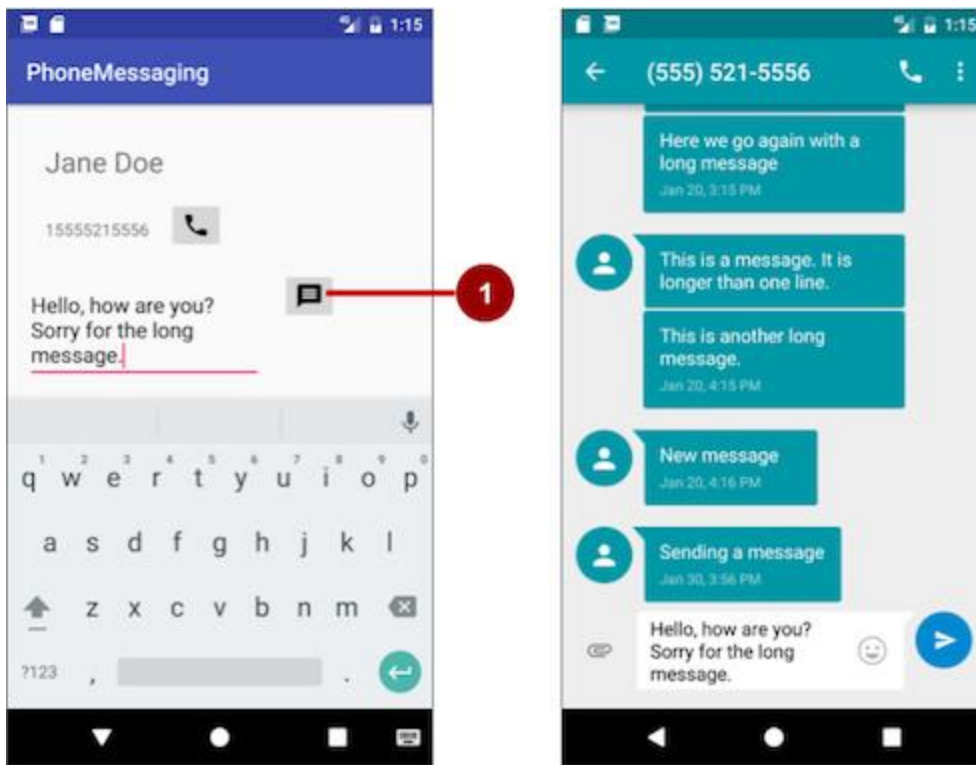
public void smsSendMessage(View view) {
    TextView textView = (TextView) findViewById(R.id.number_to_call);
    // Use format with "smsto:" and phone number to create smsNumber.
    String smsNumber = String.format("smsto: %s",
                                     textView.getText().toString());

    // Find the sms_message view.
    EditText smsEditText = (EditText) findViewById(R.id.sms_message);
    // Get the text of the sms message.
    String sms = smsEditText.getText().toString();
    // Create the intent.
    Intent smsIntent = new Intent(Intent.ACTION_SENDTO);
    // Set the data for the intent as the phone number.
    smsIntent.setData(Uri.parse(smsNumber));
    // Add the message (sms) with the key ("sms_body").
    smsIntent.putExtra("sms_body", sms);
    // If package resolves (target app installed), send intent.
    if (smsIntent.resolveActivity(getPackageManager()) != null) {
        startActivity(smsIntent);
    } else {
        Log.d(TAG, "Can't resolve app for ACTION_SENDTO Intent");
    }
}
}

```

1.3 Run the app

1. Run the app on either an emulator or a device.
2. Enter a message, and tap the messaging icon (marked "1" in the left side of the figure below). The messaging app appears, as shown on the right side of the figure below.



3. Use the **Back** button to return to the PhoneMessaging app. You may need to tap or click it more than once to leave the SMS messaging app.

Solution code

Android Studio project: [PhoneMessaging](#)

Task 2. Send an SMS message from within an app

In this task you will copy the [PhoneCallingSample](#) app from the lesson on making a phone call, rename and refactor it to **SmsMessaging**, and modify its layout and code to create an app that enables a user to enter a phone number, enter an SMS message, and send the message from within the app.

In the first step you will add the code to send the message, but the app will work only if you first turn on SMS permission manually for the app in Settings on your device or emulator.

In subsequent steps you will do away with setting this permission manually by requesting SMS permission from the app's user if it is not already set.

2.1 Create the app and layout and add permission

1. Copy the [PhoneCallingSample](#) project folder, rename it to **SmsMessaging**, and refactor it to populate the new name throughout the app project. (See the [Appendix](#) for instructions on copying a project.)
2. Open **strings.xml** and change the app_name string resource to "SMS Messaging".
3. Add the android.permission.SEND_SMS permission to the **AndroidManifest.xml** file, and remove the CALL_PHONE and READ_PHONE_STATE permissions for phone use, so that you have only one permission:
4.

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

Sending an SMS message is permission-protected. Your app can't use SMS without the SEND_SMS permission line in AndroidManifest.xml. This permission line enables a setting for the app in the Settings app that gives the user the choice of allowing or disallowing use of SMS. (In the next task you will add a way for the user to grant that permission from within the app.)

5. Add a messaging icon as you did in the previous task, and remove the phone icon from the drawable folder.
 6. Open `activity_main.xml` and edit the EditText view and replace the `android:layout_margin` attribute with the following:
- ```
7. ...
8. android:layout_marginTop="@dimen/activity_vertical_margin"
9. android:layout_marginRight="@dimen/activity_horizontal_margin"
10. ...
```

You can use `@dimen/activity_horizontal_margin` and `@dimen/activity_vertical_margin` because they are already defined in the `dimens.xml` file.

11. Add the following EditText to the layout after the first EditText (for an image of the layout, see the figure at the end of these steps):

```
12. ...
13. <EditText
14. android:id="@+id/sms_message"
15. android:layout_width="@dimen/edittext_width"
16. android:layout_height="wrap_content"
17. android:layout_below="@id/editText_main"
18. android:layout_margin="@dimen/activity_horizontal_margin"
19. android:hint="Enter message here"
20. android:inputType="textMultiLine"/>
```

You will use the `android:id` attribute to `sms_message` to identify it as the EditText for the message. The EditText view uses the `android:inputType` attribute set to `"textMultiLine"` for entering multiple lines of text.

21. After adding the hard-coded string "Enter message here" for the `android:hint` attribute, extract it into the text resource `"enter_message_here"`.
22. Change the `android:layout_below` attribute for the `button_retry` Button to refer to the `sms_message` EditText view. The Button should appear below the SMS message in the layout if it becomes visible:

```
23. android:layout_below="@id/sms_message"
```

The `button_retry` Button is set to invisible. It appears only if the app detected that telephony is not enabled, or if the user previously denied phone permission when the app requested it.

24. Replace the `phone_icon` ImageButton from the existing layout with the following:

```
25. <ImageButton
26. android:id="@+id/message_icon"
27. android:contentDescription="Send a message"
28. android:layout_width="wrap_content"
29. android:layout_height="wrap_content"
30. android:layout_marginTop="@dimen/activity_vertical_margin"
31. android:layout_toRightOf="@id/sms_message"
32. android:layout_toEndOf="@id/sms_message"
33. android:layout_below="@id/editText_main"
34. android:src="@drawable/ic_message_black_24dp"
35. android:visibility="visible"
36. android:onClick="smsSendMessage"/>
```

You will use the `android:id` `message_icon` in your code to refer to the ImageButton for sending the message. Use the vector asset you added previously (such as `ic_message_black_24dp` for a messaging icon) for the ImageButton. Make sure you include the `android:visibility` attribute set to `"visible"`. You will control the visibility of this ImageButton from your code.

37. After adding a hard-coded string for the `android:contentDescription` attribute, extract it to the `send_a_message` string resource.

The `smsSendMessage()` method referred to in the `android:onClick` attribute for the ImageButton remains highlighted until you create this method in the MainActivity, which you will do in the next step.

38. Click `smsSendMessage` in the `android:onClick` attribute, click the red light bulb that appears, and then select **Create `smsSendMessage(View)` in 'MainActivity'**. Android Studio automatically creates the `smsSendMessage()` method in MainActivity as public, returning void, with a view parameter. This method is called when the user taps the `message_icon` ImageButton.

```
39. public void smsSendMessage(View view) {
40. }
```

Your app's layout should look like the following figure (the `button_retry` Button is invisible):



# SMS Messaging

Enter a phone number

Enter message here



## 2.2 Edit the onClick method in MainActivity

1. Open MainActivity and find the new `smsSendMessage()` method you created in the last step.
2. Add statements to the method to get the string for the phone number from the `editText_main` view, and get the string for the SMS message from the `sms_message` view:

```
3. public void smsSendMessage(View view) {
4. EditText editText = (EditText) findViewById(R.id.editText_main);
5. // Set the destination phone number to the string in editText.
6. String destinationAddress = editText.getText().toString();
7. // Find the sms_message view.
8. EditText smsEditText = (EditText) findViewById(R.id.sms_message);
9. // Get the text of the sms message.
10. String smsMessage = smsEditText.getText().toString();
11. ...
12. }
```
13. Declare additional string and `PendingIntent` parameters for the `sendTextMessage()` method, which will send the message (`destinationAddress` is already declared as the string for the phone number to receive the message):
  - `scAddress`: A string for the service center address, or `null` to use the current default SMSC. A Short Message Service Center (SMSC) is a network element in the mobile telephone network. The mobile network operator usually presets the correct service center number in the default profile of settings stored in the device's SIM card.
  - `smsMessage`: A string for the body of the message to send.
  - `sentIntent`: A `PendingIntent`. If not `null`, this is broadcast when the message is successfully sent or if the message failed.
  - `deliveryIntent`: A `PendingIntent`. If not `null`, this is broadcast when the message is delivered to the recipient.
  - ...
  - // Set the service center address if needed, otherwise null.
  - `String scAddress = null;`
  - // Set pending intents to broadcast
  - // when message sent and when delivered, or set to null.
  - `PendingIntent sentIntent = null, deliveryIntent = null;`
  - ...
14. Use the `SmsManager` class to create `smsManager`, which automatically imports `android.telephony.SmsManager`, and use `sendTextMessage()` to send the message:

```
15. ...
16. // Use SmsManager.
17. SmsManager smsManager = SmsManager.getDefault();
18. smsManager.sendTextMessage
19. (destinationAddress, scAddress, smsMessage,
20. sentIntent, deliveryIntent);
21. ...
```

The full method should now look like the following:

```
public void smsSendMessage(View view) {
 EditText editText = (EditText) findViewById(R.id.editText_main);
 // Set the destination phone number to the string in editText.
 String destinationAddress = editText.getText().toString();
 // Find the sms_message view.
 EditText smsEditText = (EditText) findViewById(R.id.sms_message);
 // Get the text of the SMS message.
 String smsMessage = smsEditText.getText().toString();
 // Set the service center address if needed, otherwise null.
 String scAddress = null;
 // Set pending intents to broadcast
 // when message sent and when delivered, or set to null.
 PendingIntent sentIntent = null, deliveryIntent = null;
 // Use SmsManager.
 SmsManager smsManager = SmsManager.getDefault();
 smsManager.sendTextMessage
 (destinationAddress, scAddress, smsMessage,
 sentIntent, deliveryIntent);
}
```

If you run the app now, on either a device or an emulator, the app may crash depending on whether the device or emulator has been previously set to allow the app to use SMS. In some versions of Android, this permission is turned on by default. In other versions, this permission is turned off by default.

To set the app's permission on a device or emulator instance, choose **Settings > Apps > SMS Messaging > Permissions**, and turn on the SMS permission for the app. Since the user can turn on or off SMS permission at any time, you need to add a check in your app for this permission, and request it from the user if necessary. You will do this in the next step.

## 2.3 Check for and request permission for SMS

Your app must always get permission to use anything that is not part of the app itself. In [Step 2.1](#) you added the following permission to the AndroidManifest.xml file:

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

This permission enables a permission setting in the Settings app for your app. The user can allow or disallow this permission at any time from the Settings app. You can add code to request permission from the user if the user has turned off SMS permission for the app. Follow these steps:

1. At the top of MainActivity, below the class definition, change the global constant for the MY\_PERMISSIONS\_REQUEST\_CALL\_PHONE request code to the following:
2. `private static final int MY_PERMISSIONS_REQUEST_SEND_SMS = 1;`  
When a result returns in the activity, it will contain the MY\_PERMISSIONS\_REQUEST\_SEND\_SMS requestCode so that your code can identify it.
3. Remove the constant declarations for `mTelephonyManager` and `MyPhoneCallListener`.
4. Remove the `isTelephonyEnabled()` method, and remove all of the code in the `onCreate()` method that starts with the `mTelephonyManager` assignment, leaving only the first two lines:
5. `@Override`
6. `protected void onCreate(Bundle savedInstanceState) {`
7.  `super.onCreate(savedInstanceState);`
8.  `setContentView(R.layout.activity_main);`
9. `}`
10. Refactor/rename the existing `disableCallButton()` method to `disableSmsButton()` and edit the method to do the following:
  - a. Display a toast to notify the user that SMS usage is disabled.
  - b. Find and then set the `smsButton` (the message icon) to be invisible so that the user can't send a message.
  - c. Set the **Retry** button to be visible, so that the user can restart the activity and allow permission.
- d. `private void disableSmsButton() {`
- e.  `Toast.makeText(this, "SMS usage disabled", Toast.LENGTH_LONG).show();`
- f.  `ImageButton smsButton = (ImageButton) findViewById(R.id.message_icon);`
- g.  `smsButton.setVisibility(View.INVISIBLE);`
- h.  `Button retryButton = (Button) findViewById(R.id.button_retry);`
- i.  `retryButton.setVisibility(View.VISIBLE);`
- j. `}`  
Extract a string resource (`sms_disabled`) for the hard-coded string "SMS usage disabled" in the toast statement.
11. Refactor/rename the existing `enableCallButton()` method to `enableSmsButton()` to set the SMS message icon button to be visible:
12. `private void enableSmsButton() {`
13.  `ImageButton smsButton = (ImageButton) findViewById(R.id.message_icon);`
14.  `smsButton.setVisibility(View.VISIBLE);`
15. `}`
16. Modify the existing `retryApp()` method in MainActivity to remove the call to `enableCallButton()`.
17. In MainActivity, rename and refactor the `checkForPhonePermission()` method to `checkForSmsPermission()`, and change the code to the following:
18. `private void checkForSmsPermission() {`
19.  `if (ActivityCompat.checkSelfPermission(this,`
20.  `Manifest.permission.SEND_SMS) !=`

```

21. PackageManager.PERMISSION_GRANTED) {
22. Log.d(TAG, getString(R.string.permission_not_granted));
23. // Permission not yet granted. Use requestPermissions().
24. // MY_PERMISSIONS_REQUEST_SEND_SMS is an
25. // app-defined int constant. The callback method gets the
26. // result of the request.
27. ActivityCompat.requestPermissions(this,
28. new String[]{Manifest.permission.SEND_SMS},
29. MY_PERMISSIONS_REQUEST_SEND_SMS);
30. } else {
31. // Permission already granted. Enable the SMS button.
32. enableSmsButton();
33. }
34. }

```

Use [checkSelfPermission\(\)](#) to determine whether your app has been granted a particular permission by the user. If permission has *not* been granted by the user, use the [requestPermissions\(\)](#) method to display a standard dialog for the user to grant permission.

When your app calls `requestPermissions()`, the system shows a standard dialog for each permission to the user, as shown in the figure below.



# SMS Messaging

Enter a phone number



Allow **SMS Messaging** to send and view SMS messages?



Don't ask again

35. When the user responds to the request permission dialog, the system invokes your app's `onRequestPermissionsResult()` method, passing it the user response. Find the `onRequestPermissionsResult()` method you created for the previous version of this app. Your implementation of `onRequestPermissionsResult()` already uses a `switch` statement based on the value of `requestCode`. A case for checking phone permission is already implemented using `MY_PERMISSIONS_REQUEST_CALL_PHONE`. Replace `MY_PERMISSIONS_REQUEST_CALL_PHONE` with `MY_PERMISSIONS_REQUEST_SEND_SMS`, and replace `CALL_PHONE` with `SEND_SMS`. The switch block should now look like the following:

```
...
switch (requestCode) {
 case MY_PERMISSIONS_REQUEST_SEND_SMS: {
 if (permissions[0].equalsIgnoreCase(
 Manifest.permission.SEND_SMS)
 && grantResults[0] ==
 PackageManager.PERMISSION_GRANTED) {
 // Permission was granted. Enable sms button.
 enableSmsButton();
 } else {
 // Permission denied.
 Log.d(TAG, getString(R.string.failure_permission));
 Toast.makeText(this,
 getString(R.string.failure_permission),
 Toast.LENGTH_LONG).show();
 // Disable the sms button.
 disableSmsButton();
 }
 }
}
```

If the user allows the permission request, the message button is re-enabled with `enableSmsButton()` in case it was made invisible by lack of permission. If the user denies the permission requests, your app should take appropriate action. For example, your app might disable the functionality that depends on a specific permission and show a dialog explaining why it could not perform it. For now, log a debug message, display a toast to show that permission was not granted, and disable the message button with `disableSmsButton()`.

36. In the `onCreate()` method of `MainActivity`, add a call to the `checkForSmsPermission()` method:

```
37. @Override
38. protected void onCreate(Bundle savedInstanceState) {
39. super.onCreate(savedInstanceState);
40. setContentView(R.layout.activity_main);
41. checkForSmsPermission();
42. }
43. Remove the callNumber() method and the MyPhoneCallListener inner class (including
 the onCallStateChanged() method, as you are no longer using the Telephony Manager).
44. Remove the onDestroy() method since you are no longer using a listener.
45. Since the user might turn off SMS permission while the app is still running, add a check for SMS
 permission in the smsSendMessage() method after setting the sentIntent but before using
 the SmsManager class:
46. ...
47. PendingIntent sentIntent = null, deliveryIntent = null;
48. // Check for permission first.
49. checkForSmsPermission();
50. // Use SmsManager.
51. ...
```

## 2.4 Run the app and test permissions

1. Run your app. Enter a phone number (or the emulator port number if using emulators), and enter the message to send. Tap the messaging icon to send the message.
2. After running the app, choose **Settings > Apps > SMS Messaging > Permissions** and turn off SMS permission for the app.

3. Run the app again. You should see the SMS permission request dialog as shown below.



# SMS Messaging

Enter a phone number



Allow **SMS Messaging** to send and view SMS messages?



Don't ask again



4. Click **Deny**. In the app's UI, the message icon button no longer appears, and a **Retry** button appears, as shown below.



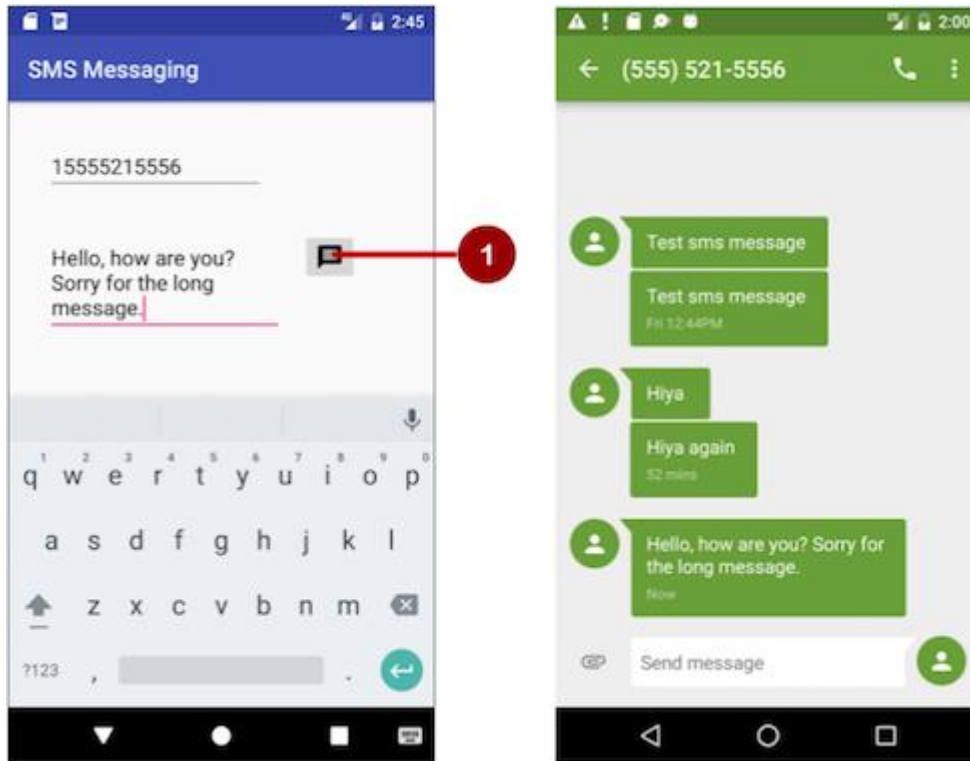
# SMS Messaging

Enter a phone number

Enter message here

**RETRY**

5. Click **Retry**, and then click **Allow** for SMS permission.
6. Test the app's ability to send a message:
  - a. Enter a phone number.
  - b. Enter a message.
  - c. Tap the messaging icon.



## Task 3. Receive SMS messages with a broadcast receiver

To receive SMS messages, use the [onReceive\(\)](#) method of the [BroadcastReceiver](#) class. The Android framework sends out system broadcasts of events such as receiving an SMS message, containing intents that are meant to be received using a BroadcastReceiver. You need to add the `RECEIVE_SMS` permission to your app's `AndroidManifest.xml` file.

### 3.1 Add permission and create a broadcast receiver

To add `RECEIVE_SMS` permission and create a broadcast receiver, follow these steps:

1. Open the `AndroidManifest.xml` file and add the `android.permission.RECEIVE_SMS` permission below the other permission for SMS use:
 

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

Receiving an SMS message is permission-protected. Your app can't receive SMS messages without the `RECEIVE_SMS` permission line in `AndroidManifest.xml`.
4. Select the package name in the Project:Android: view and choose **File > New > Other > Broadcast Receiver**.
5. Name the class "MySmsReceiver" and make sure "Exported" and "Enabled" are checked.

The "Exported" option allows your app to respond to outside broadcasts, while "Enabled" allows it to be instantiated by the system.

6. Open the AndroidManifest.xml file again. Note that Android Studio automatically generates a `<receiver>` tag with your chosen options as attributes:

```
7. <receiver
8. android:name=
9. "com.example.android.smsmessaging.MySmsReceiver"
10. android:enabled="true"
11. android:exported="true">
12. </receiver>
```

## 3.2 Register the broadcast receiver

In order to receive any broadcasts, you must register for specific broadcast intents. In the [Intent documentation](#), under "Standard Broadcast Actions", you can find some of the common broadcast intents sent by the system. In this app, you use the `android.provider.Telephony.SMS_RECEIVED` intent. Add the following inside the `<receiver>` tags to register your receiver:

```
<receiver
 android:name="com.example.android.smsmessaging.MySmsReceiver"
 android:enabled="true"
 android:exported="true">
 <intent-filter>
 <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
 </intent-filter>
</receiver>
```

## 3.3 Implement the onReceive() method

Once the `BroadcastReceiver` intercepts a broadcast for which it is registered (`SMS_RECEIVED`), the intent is delivered to the receiver's `onReceive()` method, along with the context in which the receiver is running.

1. Open `MySmsReceiver` and add under the class declaration a string constant `TAG` for log messages and a string constant `pdu_type` for identifying PDUs in a bundle:

```
2. public class MySmsReceiver extends BroadcastReceiver {
3. private static final String TAG =
4. MySmsReceiver.class.getSimpleName();
5. public static final String pdu_type = "pdus";
6. ...
```

7. Delete the default implementation inside the supplied `onReceive()` method.

8. In the blank `onReceive()` method:

- a. Add the `@TargetApi` annotation for the method, because it performs a different action depending on the build version.

- b. Retrieve a map of extended data from the intent to a `bundle`.

- c. Define the `msgs` array and `strMessage` string.

- d. Get the format for the message from the `bundle`.

```
e. @TargetApi(Build.VERSION_CODES.M)
```

```
f. @Override
```

```
g. public void onReceive(Context context, Intent intent) {
```

```
h. // Get the SMS message.
```

```
i. Bundle bundle = intent.getExtras();
```

```
j. SmsMessage[] msgs;
```

```
k. String strMessage = "";
```

```
l. String format = bundle.getString("format");
```

```
m. ...
```

As you enter `SmsMessage[]`, Android Studio automatically imports `android.telephony.SmsMessage`.

9. Retrieve from the bundle one or more pieces of data in the protocol data unit (PDU) format, which is the industry-standard format for an SMS message:

```
10. ...
11. // Retrieve the SMS message received.
12. Object[] pdus = (Object[]) bundle.get(pdu_type);
```

13. ...  
14. If there are messages (pdus), check for Android version 6.0 (Marshmallow) and newer versions. You will use this boolean to check if your app needs the deprecated signature [createFromPdu\(byte\[\] pdu\)](#) for earlier versions of Android:

```
15. ...
16. if (pdus != null) {
17. // Check the Android version.
18. boolean isVersionM = (Build.VERSION.SDK_INT >=
19. Build.VERSION_CODES.M);
20. ...
21. Initialize the msgs array, and use its length in the for loop:
22. ...
23. // Fill the msgs array.
24. msgs = new SmsMessage[pdus.length];
25. for (int i = 0; i < msgs.length; i++) {
26. // Check Android version and use appropriate createFromPdu.
27. if (isVersionM) {
28. // If Android version M or newer:
29. msgs[i] =
30. SmsMessage.createFromPdu((byte[]) pdus[i], format);
31. } else {
32. // If Android version L or older:
33. msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
34. }
35. }
```

Use [createFromPdu\(byte\[\] pdu, String format\)](#) to fill the msgs array for Android version 6.0 (Marshmallow) and newer versions. For earlier versions of Android, use the deprecated signature [createFromPdu\(byte\[\] pdu\)](#).

36. Build the strMessage to show in a toast message:

- Get the originating address using the `getOriginatingAddress()` method.
- Get the message body using the `getMessageBody()` method.
- Add an ending character for an end-of-line.

```
d. ...
e. // Build the message to show.
f. strMessage += "SMS from " + msgs[i].getOriginatingAddress();
g. strMessage += " :" + msgs[i].getMessageBody() + "\n";
h. ...
```

37. Log the resulting strMessage and display a toast with it:

```
38. ...
39. // Log and display the SMS message.
40. Log.d(TAG, "onReceive: " + strMessage);
41. Toast.makeText(context, strMessage, Toast.LENGTH_LONG).show();
42. ...
```

The complete `onReceive()` method is shown below:

```
@TargetApi(Build.VERSION_CODES.M)
@Override
public void onReceive(Context context, Intent intent) {
 // Get the SMS message.
 Bundle bundle = intent.getExtras();
 SmsMessage[] msgs;
 String strMessage = "";
 String format = bundle.getString("format");
 // Retrieve the SMS message received.
 Object[] pdus = (Object[]) bundle.get("pdu_type");
 if (pdus != null) {
 // Check the Android version.
 boolean isVersionM =
 (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M);
 // Fill the msgs array.
 msgs = new SmsMessage[pdus.length];
 for (int i = 0; i < msgs.length; i++) {
 // Check Android version and use appropriate createFromPdu.
 if (isVersionM) {
 // If Android version M or newer:
 msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i], format);
 } else {
```

```

 // If Android version L or older:
 msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
 }
 // Build the message to show.
 strMessage += "SMS from " + msgs[i].getOriginatingAddress();
 strMessage += " : " + msgs[i].getMessageBody() + "\n";
 // Log and display the SMS message.
 Log.d(TAG, "onReceive: " + strMessage);
 Toast.makeText(context, strMessage, Toast.LENGTH_LONG).show();
}
}
}

```

### 3.4 Run the app and send a message

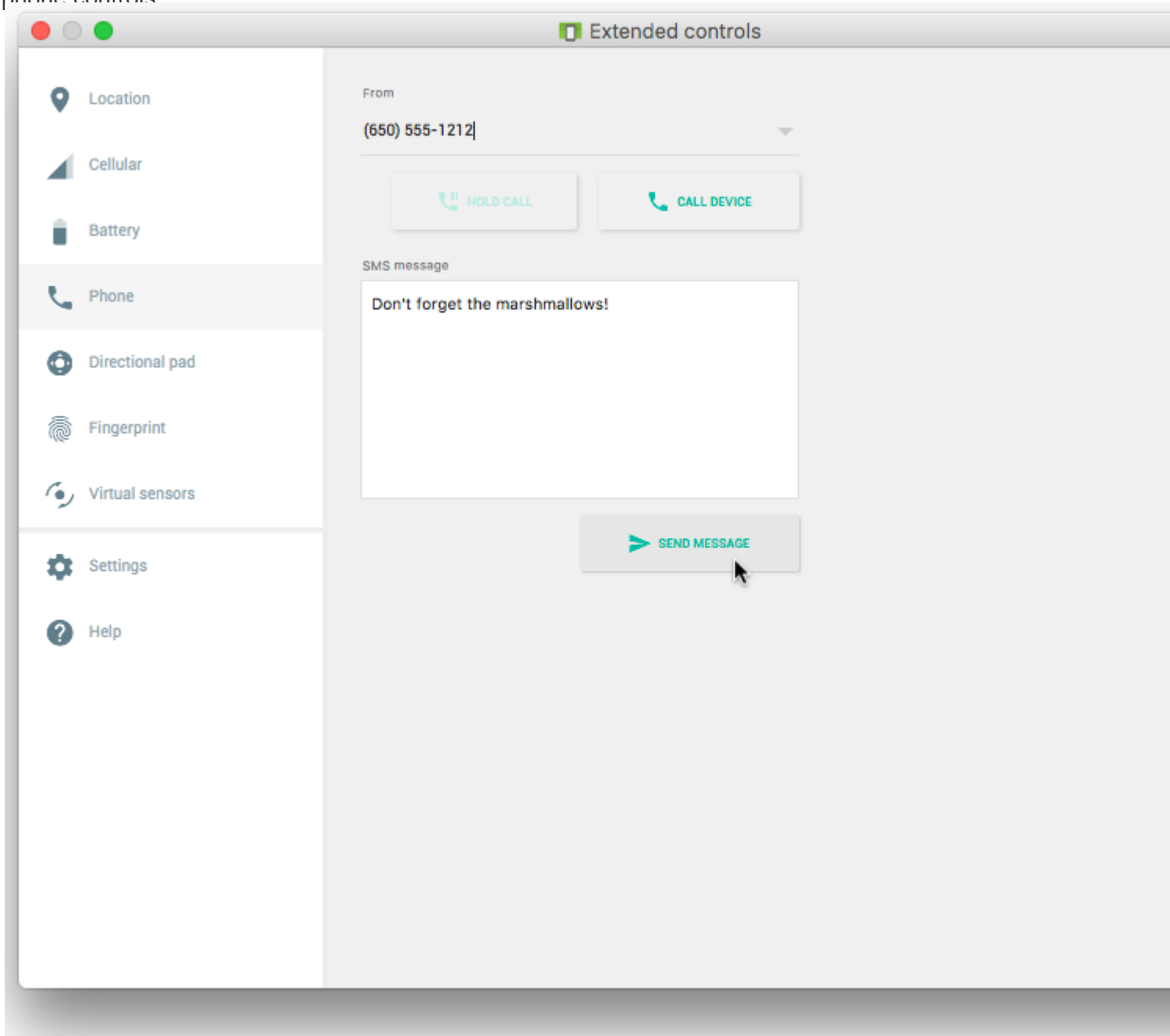
Run the app on a device. If possible, have someone send you an SMS message from a different device.

You can also receive an SMS text message when testing on an emulator. Follow these steps:

1. Run the app on an emulator.
2. Click the ... (More) icon at the bottom of the emulator's toolbar on the right side, as shown in the figure below:



3. The extended controls for the emulator appear. Click **Phone** in the left column to see the extended phone controls:



4. You can now enter a message (or use the default "marshmallows" message) and click **Send Message** to have the emulator send an SMS message to itself.

5. The emulator responds with a notification about receiving an SMS message. The app should also display a toast message showing the message and its originating address, as shown below:





Messenger • now ▾

(650) 555-1212

Don't forget the marshmallows!

REPLY

---

Enter message here

---

## Solution Code

Android Studio project: [SmsMessaging](#)

## Coding challenge

**Note:** All coding challenges are optional.

**Challenge:** Create a simple app with one button, **Choose Picture and Send**, that enables the user to select an image from the Gallery and send it as a Multimedia Messaging Service (MMS) message. After tapping the button, a choice of apps may appear, including the Messenger app. The user can select the Messenger app, and select an existing conversation or create a new conversation, and then send the image as a message.

The following are hints:

- To access and share an image from the Gallery, you need the following permission in the AndroidManifest.xml file:

```
<uses-permission
 android:name="android.permission.READ_EXTERNAL_STORAGE" />
```
- To enable the above permission, follow the model shown previously in this chapter to check for the READ\_EXTERNAL\_STORAGE permission, and request permission if necessary.
- Use the following intent for picking an image:

```
Intent galleryIntent = new Intent(Intent.ACTION_PICK,
 android.provider.MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
startActivityForResult(galleryIntent, IMAGE_PICK);
```
- Override the [onActivityResult](#) method to retrieve the intent result, and use `getData()` to get the Uri of the image in the result:

```
protected void onActivityResult
 (int requestCode, int resultCode, Intent imageReturnedIntent) {
 ...
 Uri mSelectedImage = imageReturnedIntent.getData();
}
```
- Set the image's Uri, and use an intent with ACTION\_SEND, `putExtra()`, and `setType()`:

```
Intent smsIntent = new Intent(Intent.ACTION_SEND);
smsIntent.putExtra(Intent.EXTRA_STREAM, mSelectedImage);
smsIntent.setType("image/*");
```
- Android Studio emulators can't pass MMS messages to and from each other. You must test this app on real Android devices.
- For more information about sending multimedia messages, see [Sending MMS with Android](#).

