

## 5. Advanced Topics

### Notifications

A notification is a message you can display to the user outside of your application's normal UI. When we 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.

## Creating a notification

The UI information and actions for a notification is specified in a NotificationCompat.Builder object. To create the notification itself, we call NotificationCompat.Builder.build(), which returns a Notification object containing your specifications. To issue the notification, you pass the Notification object to the system by calling NotificationManager.notify().

A Notification object *must* contain the following:

- A small icon, set by setSmallIcon()
- A title, set by setContentTypeTitle()
- Detail text, set by setContentTypeText()

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

## Toast

A toast provides simple feedback about an operation in a small popup. It only fills the amount of space required for the message and the current activity remains visible and interactive. For example, navigating away from an email before you send it triggers a "Draft saved" toast to let you know that you can continue editing later. Toasts automatically disappear after a timeout.



First, instantiate a `Toast` object with one of the `makeText()` methods. This method takes three parameters: the application `Context`, the text message, and the duration for the toast. It returns a properly initialized `Toast` object. You can display the toast notification with `show()`, as shown in the following example:

```
Context context = getApplicationContext();
CharSequence text = "Hello toast!";
int duration = Toast.LENGTH_SHORT;

Toast toast = Toast.makeText(context, text, duration);
toast.show();
```

This example demonstrates everything you need for most toast notifications. You should rarely need anything else. You may, however, want to position the toast differently or even use your own layout instead of a simple text message. The following sections describe how you can do these things.

You can also chain your methods and avoid holding on to the `Toast` object, like this:

```
Toast.makeText(context, text, duration).show();
```

## Positioning your Toast

A standard toast notification appears near the bottom of the screen, centered horizontally. You can change this position with the `setGravity(int, int, int)` method. This accepts three parameters: a `Gravity` constant, an x-position offset, and a y-position offset.

For example, if you decide that the toast should appear in the top-left corner, you can set the gravity like this:

```
toast.setGravity(Gravity.TOP|Gravity.LEFT, 0, 0);
```

If you want to nudge the position to the right, increase the value of the second parameter. To nudge it down, increase the value of the last parameter.

## BroadcastReceiver

A broadcast receiver (short receiver) is an Android component which allows you to register for system or application events. All registered receivers for an event are notified by the Android runtime once this event happens.

For example, applications can register for the `ACTION_BOOT_COMPLETED` system event which is fired once the Android system has completed the boot process.

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

There are following two important steps to make `BroadcastReceiver` works for the system broadcasted intents



- Creating the Broadcast Receiver

## Creating the Broadcast Receiver

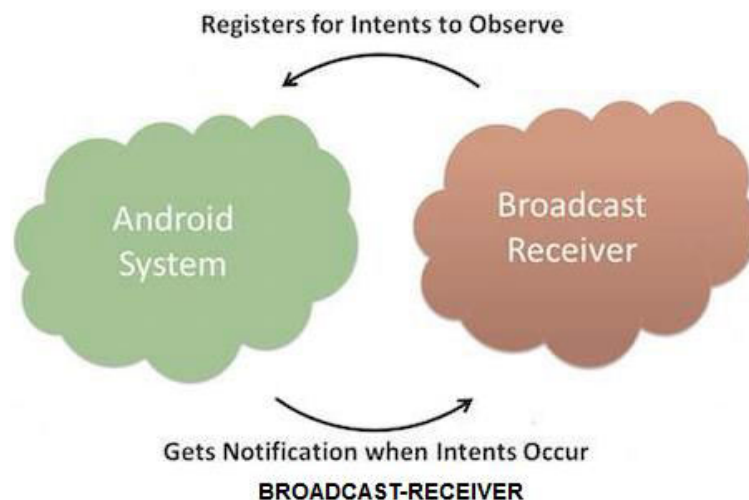
A broadcast receiver is implemented as a subclass of **BroadcastReceiver** class and overriding the `onReceive()` method where each message is received as a **Intent** object parameter.

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

- Registering Broadcast Receiver

## Registering Broadcast Receiver

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



## Threads and Handlers

### Thread

A Thread is a concurrent unit of execution. It has its own call stack for methods being invoked, their arguments and local variables. Each application has at least one thread running when it is started, the main thread, in the main "ThreadGroup". The runtime keeps its own threads in the system thread group.

There are two ways to execute code in a new thread.

We can either subclass Thread and overriding its `run()` method.

Or

Construct a new Thread and pass a Runnable to the constructor.

In either case, the `start()` method must be called to actually execute the new Thread.

### Some Thread Syntax:



### Thread()

Constructs a new `Thread` with no `Runnable` object and a newly generated name.

### Thread(Runnable runnable)

Constructs a new `Thread` with a `Runnable` object and a newly generated name.

### Thread(Runnable runnable, String threadName)

Constructs a new `Thread` with a `Runnable` object and name provided.

### Thread(String threadName)

Constructs a new `Thread` with no `Runnable` object and the name provided.

## Handler

Handler is part of the Android system's framework for managing threads. A Handler object receives messages and runs code to handle the messages. A Handler allows us to send and process Message and Runnable objects associated with a thread's MessageQueue.

There are two main uses for a Handler:

- to schedule messages and runnables to be executed at some point in the future
- to enqueue an action to be performed on a different thread than your own

Scheduling messages is accomplished with the following methods.

- `post(Runnable)`,
- `postAtTime(Runnable, long)`,
- `postDelayed(Runnable, long)`,
- `sendMessage(int)`,
- `sendMessage(Message)`,
- `sendMessageAtTime(Message, long)`, and
- `sendMessageDelayed(Message, long)`

The *post* versions allow you to enqueue Runnable objects to be called by the message queue when they are received; the *sendMessage* versions allow you to enqueue a Message object containing a bundle of data that will be processed by the Handler's `handleMessage(Message)` method (requiring that you implement a subclass of Handler). When posting or sending to a Handler, you can either allow the item to be processed as soon as the message queue is ready to do so, or specify a delay before it gets processed or absolute time for it to be processed. The latter two allow you to implement timeouts, ticks, and other timing-based behavior.

## Some Handler Syntax:

### Handler()

Default constructor associates this handler with the `Looper` for the current thread.

Use the provided `Looper` instead of the default one.



# AsyncTask

AsyncTask enables proper and easy use of the UI thread. This class allows to perform background operations and publish results on the UI thread without having to manipulate threads and/or handlers.

AsyncTask is designed to be a helper class around `Thread` and `Handler` and does not constitute a generic threading framework. AsyncTasks should ideally be used for short operations (a few seconds at the most.) If you need to keep threads running for long periods of time, it is highly recommended you use the various APIs provided by the `java.util.concurrent` package such as `Executor`, `ThreadPoolExecutor` and `FutureTask`.

An asynchronous task is defined by a computation that runs on a background thread and whose result is published on the UI thread. An asynchronous task is defined by 3 generic types, called `Params`, `Progress` and `Result`, and 4 steps, called `onPreExecute`, `doInBackground`, `onProgressUpdate` and `onPostExecute`.

## AsyncTask's generic types

---

The three types used by an asynchronous task are the following:

1. `Params`, the type of the parameters sent to the task upon execution.
2. `Progress`, the type of the progress units published during the background computation.
3. `Result`, the type of the result of the background computation.

Not all types are always used by an asynchronous task. To mark a type as unused, simply use the type `Void`:

```
private class MyTask extends AsyncTask<Void, Void, Void> { ... }
```

## The 4 steps

---

When an asynchronous task is executed, the task goes through 4 steps:

1. `onPreExecute()`, invoked on the UI thread before the task is executed. This step is normally used to setup the task, for instance by showing a progress bar in the user interface.
2. `doInBackground(Params...)`, invoked on the background thread immediately after `onPreExecute()` finishes executing. This step is used to perform background computation that can take a long time. The parameters of the asynchronous task are passed to this step. The result of the computation must be returned by this step and will be passed back to the last step. This step can also use `publishProgress(Progress...)` to publish one or more units of progress. These values are published on the UI thread, in the `onProgressUpdate(Progress...)` step.
3. `onProgressUpdate(Progress...)`, invoked on the UI thread after a call to `publishProgress(Progress...)`. The timing of the execution is undefined. This method is used to display any form of progress in the user interface while the background computation is still executing. For instance, it can be used to animate a progress bar or show logs in a text field.
4. `onPostExecute(Result)`, invoked on the UI thread after the background computation finishes. The result of the background computation is passed to this step as a parameter.

## Alarm Manager





Alarms (based on the `AlarmManager` class) gives a way to perform time-based operations outside the lifetime of your application. For example, we could use an alarm to initiate a long-running operation, such as starting a service once a day to download a weather forecast.

Alarms have these characteristics:

- They let us fire Intents at set times and/or intervals.
- We can use them in conjunction with broadcast receivers to start services and perform other operations.
- They operate outside of your application, so we can use them to trigger events or actions even when your app is not running, and even if the device itself is asleep.
- They help us to minimize your app's resource requirements. We can schedule operations without relying on timers or continuously running background services.

**Note:** For timing operations that are guaranteed to occur *during* the lifetime of your application, instead consider using the `Handler` class in conjunction with `Timer` and `Thread`. This approach gives Android better control over system resources.

## Alarm Types:

There are two general clock types for alarms: "elapsed real time" and "real time clock" (RTC). Elapsed real time uses the "time since system boot" as a reference, and real time clock uses UTC (wall clock) time. This means that elapsed real time is suited to setting an alarm based on the passage of time (for example, an alarm that fires every 30 seconds) since it isn't affected by time zone/locale. The real time clock type is better suited for alarms that are dependent on current locale.

Both types have a "wakeup" version, which says to wake up the device's CPU if the screen is off. This ensures that the alarm will fire at the scheduled time. This is useful if your app has a time dependency—for example, if it has a limited window to perform a particular operation. If you don't use the wakeup version of your alarm type, then all the repeating alarms will fire when your device is next awake.

Here is the list of types:

- `ELAPSED_REALTIME` —Fires the pending intent based on the amount of time since the device was booted, but doesn't wake up the device. The elapsed time includes any time during which the device was asleep.
- `ELAPSED_REALTIME_WAKEUP` —Wakes up the device and fires the pending intent after the specified length of time has elapsed since device boot.
- `RTC` —Fires the pending intent at the specified time but does not wake up the device.
- `RTC_WAKEUP` —Wakes up the device to fire the pending intent at the specified time.

## Networking in Android

Android lets your application connect to the internet or any other local network and allows you to perform network operations.

A device can have various types of network connections. This chapter focuses on using either a Wi-Fi or a mobile network connection.



## Checking Network Connection

Before you perform any network operations, you must first check that are you connected to that network or internet e.t.c. For this android provides **ConnectivityManager** class. You need to instantiate an object of this class by calling **getSystemService()** method. Its syntax is given below –

```
ConnectivityManager check =(ConnectivityManager)
this.context.getSystemService(Context.CONNECTIVITY_SERVICE);
```

Once you instantiate the object of **ConnectivityManager** class, you can use **getAllNetworkInfo** method to get the information of all the networks. This method returns an array of **NetworkInfo**. So you have to receive it like this.

```
NetworkInfo[] info =check.getAllNetworkInfo();
```

The last thing you need to do is to check **Connected State** of the network. Its syntax is given below –

```
for(int i=0;i<info.length;i++){
if(info[i].getState()==NetworkInfo.State.CONNECTED){
    Toast.makeText(context,"Internet is connected
    Toast.LENGTH_SHORT).show();
}
}
```

Apart from this connected states, there are other states a network can achieve. They are listed below:

Sr.No	State
1	Connecting
2	Disconnected
3	Disconnecting
4	Suspended
5	Unknown

## Performing Network Operations

After checking that you are connected to the internet, you can perform any network operation. Here we are fetching the html of a website from a url.

Android provides **HttpURLConnection** and **URL** class to handle these operations. You need to instantiate an object of **URL** class by providing the link of website. Its syntax is as follows –

```
String link ="http://www.google.com";
URL url=new URL(link);
```

After that you need to call **openConnection** method of **url** class and receive it in a **HttpURLConnection** object.

After that you need to call the **connect** method of **HttpURLConnection** class.

```
HttpURLConnection conn =(HttpURLConnection)url.openConnection();
conn.connect();
```





And the last thing you need to do is to fetch the HTML from the website. For this you will use **InputStream** and **BufferedReader** class. Its syntax is given below –

```
InputStream is=conn.getInputStream();
BufferedReader reader =newBufferedReader(newInputStreamReader(is,"UTF-8"));
StringwebPage="",data="";

while((data =reader.readLine())!=null){
webPage+= data +"\n";
}
```

Apart from this connect method, there are other methods available in `HttpURLConnection` class. They are listed below –

Sr.No	Method & description
1	<b>disconnect()</b> This method releases this connection so that its resources may be either reused or closed
2	<b>getRequestMethod()</b> This method returns the request method which will be used to make the request to the remote HTTP server
3	<b>getResponseCode()</b> This method returns response code returned by the remote HTTP server
4	<b>setRequestMethod(String method)</b> This method Sets the request command which will be sent to the remote HTTP server
5	<b>usingProxy()</b> This method returns whether this connection uses a proxy server or not

## 6. Graphics and Multimedia

### Graphics



If one talks about standard components that we can be used in UI, this is good but it is not enough when we want to develop a game or an app that requires graphic contents. Android SDK provides a set of API for drawing custom 2D and 3Dgraphics. When we write an app that requires graphics, we should consider how intensive the graphic usage is. In other words,there could be an app that uses quite static graphics without complex effects and there could be other app that uses intensivegraphical effects like games.According to this usage, there are different techniques we can adopt:

- **Canvas and Drawable:** In this case, we can extend the existing UI widgets so that we can customize their behavior or we cancreate custom 2D graphics using the standard method provided by the Canvasclass.
- **Hardware acceleration:** We can use hardware acceleration when drawing with the Canvas API. This is possible fromAndroid 3.0.

Hardware acceleration is enabled by default if your Target API level is  $\geq 14$ , but can also be explicitly enabled. If your application uses only standard views and Drawables, turning it on globally should not cause any adverse drawing effects. However, because hardware acceleration is not supported for all of the 2D drawing operations, turning it on might affect some of your custom views or drawing calls. Problems usually manifest themselves as invisible elements, exceptions, or wrongly rendered pixels. To remedy this, Android gives you the option to enable or disable hardware acceleration at multiple levels.

## Controlling Hardware Acceleration

You can control hardware acceleration at the following levels:

- Application
- Activity
- Window
- View

### *Application level*

In your Android manifest file, add the following attribute to the <application> tag to enable hardware acceleration for your entire application:

```
<applicationandroid:hardwareAccelerated="true" ...>
```

### *Activity level*

If your application does not behave properly with hardware acceleration turned on globally, you can control it for individual activities as well. To enable or disable hardware acceleration at the activity level, you can use the `android:hardwareAccelerated` attribute for the <activity> element.

The following example enables hardware acceleration for the entire application but disables it for one activity:



```
<applicationandroid:hardwareAccelerated="true">
    <activity ... />
    <activityandroid:hardwareAccelerated="false"/>
</application>
```

## Determining if a View is Hardware Accelerated

It is sometimes useful for an application to know whether it is currently hardware accelerated, especially for things such as custom views. This is particularly useful if your application does a lot of custom drawing and not all operations are properly supported by the new rendering pipeline.

There are two different ways to check whether the application is hardware accelerated:

- `View.isHardwareAccelerated()` returns `true` if the View is attached to a hardware accelerated window.
- `Canvas.isHardwareAccelerated()` returns `true` if the Canvas is hardware accelerated

If you must do this check in your drawing code, use `Canvas.isHardwareAccelerated()` instead of `View.isHardwareAccelerated()` when possible. When a view is attached to a hardware accelerated window, it can still be drawn using a non-hardware accelerated Canvas. This happens, for instance, when drawing a view into a bitmap for caching purposes.

- **OpenGL:** Android supports OpenGL natively using NDK. This technique is very useful when we have an app that uses intensively graphic contents (i.e games).

Android includes support for high performance 2D and 3D graphics with the Open Graphics Library (OpenGL®), specifically, the OpenGL ES API. OpenGL is a cross-platform graphics API that specifies a standard software interface for 3D graphics processing hardware. OpenGL ES is a flavor of the OpenGL specification intended for embedded devices. Android supports several versions of the OpenGL ES API:

- OpenGL ES 1.0 and 1.1 - This API specification is supported by Android 1.0 and higher.
- OpenGL ES 2.0 - This API specification is supported by Android 2.2 (API level 8) and higher.
- OpenGL ES 3.0 - This API specification is supported by Android 4.3 (API level 18) and higher.
- OpenGL ES 3.1 - This API specification is supported by Android 5.0 (API level 21) and higher.



The easiest way to use 2D graphics is extending the View class and overriding the onDraw method. We can use this technique when we do not need a graphics intensive app.

In this case, we can use the Canvas class to create 2D graphics. This class provides a set of methods starting with draw that can be used to draw different shapes like:

- lines
- circle
- rectangle
- oval
- picture
- arc

For example let us suppose we want to draw a rectangle. We create a custom view and then we override onDraw method. Here we draw the rectangle:

```
public class TestView extends View {  
    public TestView(Context context) {  
        super(context);  
    }  
    public TestView(Context context, AttributeSet attrs, int defStyle) {  
        super(context, attrs, defStyle);  
    }  
    public TestView(Context context, AttributeSet attrs) {  
        super(context, attrs);  
    }  
    @Override  
    protected void onDraw(Canvas canvas) {  
        super.onDraw(canvas);  
        Paint p = new Paint();  
        p.setColor(Color.GREEN);  
        p.setStrokeWidth(1);  
    }  
}
```

## Animation

**Animation** is the process of creating the illusion of motion and shape change by means of the rapid display of a sequence of static images that minimally differ from each other. The illusion—as in motion pictures in general—is thought to rely on the phi phenomenon. Animators are artists who specialize in the creation of animation.

Animation can be recorded on either analogue media, such as a flip book, motion picture film, video tape, or on digital media, including formats such as animated GIF, Flash animation or digital video. To display animation, a digital camera, computer, or projector are used along with new technologies that are produced.



Animation creation methods include the traditional animation creation method and those involving stop motion animation of two and three-dimensional objects, such as paper cutouts, puppets and clay figures. Images are displayed in a rapid succession, usually 24, 25, 30, or 60 frames per second.

Animations can add subtle visual cues that notify users about what's going on in your app and improve their mental model of your app's interface. Animations are especially useful when the screen changes state, such as when content loads or new actions become available. Animations can also add a polished look to your app, which gives your app a higher quality feel.

### Crossfading Two Views

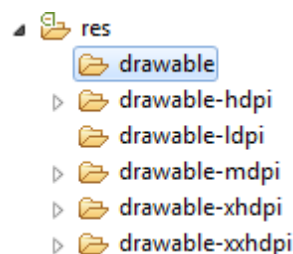
Crossfade animations (also known as dissolve) gradually fade out one UI component while simultaneously fading in another. This animation is useful for situations where you want to switch content or views in your app. Crossfades are very subtle and short but offer a fluid transition from one screen to the next. When you don't use them, however, transitions often feel abrupt or hurried.

### Using ViewPager for Screen Slides

Screen slides are transitions between one entire screen to another and are common with UIs like setup wizards or slideshows. The images slides with a ViewPager provided by the support library. ViewPagers can animate screen slides automatically. Here's what a screen slide looks like that transitions from one screen of content to the next.

## Drawable

In Android, a Drawable is a graphical object that can be shown on the screen. From API point of view all the Drawable objects derive from *Drawable* class. They have an important role in Android programming and we can use XML to create them. They differ from standard widgets because they are not interactive, meaning that they do not react to user touch. Images, colors, shapes, objects that change their aspect according to their state, object that can be animated are all drawable objects. In Android under res directory, there is a sub-dir reserved for Drawable, it is called *res/drawable*.



Under the drawabledir we can add binary files like images or XML files.

We can create several directories according to the screen density we want to support. These directories have a name like *drawable-<>*. This is very useful when we use images; in this case, we have to create several image versions: for example, we can create an image for the high dpi screen or another one for medium dpi screen.

Once we have our file under drawable directory, we can reference it, in our class, using

*R.drawable.file\_name*

While it is very easy add a binary file to one of these directory, it is a matter of copy and paste, if we want to use a XML file we have to create it.



There are several types of drawable:

- Bitmap
- Nine-patch
- State list
- Level list
- Transition drawable
- Inset drawable
- Clip drawable
- Scale drawable
- Shape drawable

### Shape drawable

This is a generic shape. Using XML we have to create a file with shape element as root. This element has an attribute called

`android:shape`

where we define the type of shape like rectangle, oval, line and ring. We can customize the shape using child elements like:

For example, let us suppose we want to create an oval with solid background color. We create a XML file called for example *oval.xml*:

```
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="oval" >
    <solid android:color="#FF0000" />
    <size
        android:height="100dp"
        android:width="120dp" />
</shape>
```

### Multi touch and gestures

Multi-touch gesture happens when more than one finger touches the screen at the same time. Android allows us to detect these gestures.

Android system generates the following touch events whenever multiple fingers touch the screen at the same time.

Sr.No	Event & description
1	<b>ACTION_DOWN</b> For the first pointer that touches the screen. This starts the gesture.
2	<b>ACTION_POINTER_DOWN</b> For extra pointers that enter the screen beyond the first.
3	<b>ACTION_MOVE</b> A change has happened during a press gesture.





4	<b>ACTION_POINTER_UP</b> Sent when a non-primary pointer goes up.
5	<b>ACTION_UP</b> Sent when the last pointer leaves the screen.

So in order to detect any of the above mention event , you need to override **onTouchEvent()** method and check these events manually. Its syntax is given below –

```
public boolean onTouchEvent(MotionEvent ev){
    final int actionPeformed=ev.getAction();
    switch(actionPeformed){
        case MotionEvent.ACTION_DOWN:{
            break;
        }
        case MotionEvent.ACTION_MOVE:{
            break;
        }
    }
    return true;
}
```

In these cases, you can perform any calculation you like. For example zooming , shrinking e.t.c. In order to get the co-ordinates of the X and Y axis, you can call **getX()** and **getY()** method. Its syntax is given below:

```
final float x =ev.getX();
final float y =ev.getY();
```

Apart from these methods, there are other methods provided by this MotionEvent class for better dealing with multitouch. These methods are listed below:

Sr.No	Method & description
1	<b>getAction()</b> This method returns the kind of action being performed
2	<b>getPressure()</b> This method returns the current pressure of this event for the first index
3	<b>getRawX()</b> This method returns the original raw X coordinate of this event
4	<b>getRawY()</b> This method returns the original raw Y coordinate of this event
5	<b>getSize()</b> This method returns the size for the first pointer index
6	<b>getSource()</b> This method gets the source of the event



7	<b>getXPrecision()</b> This method return the precision of the X coordinates being reported
8	<b>getYPrecision()</b> This method return the precision of the Y coordinates being reported

## Multimedia

The Android SDK provides a set of APIs to handle multimedia files, such as audio, video and images. Moreover, the SDK provides other API sets that help developers to implement interesting graphics effects, like animations and so on.

The modern smart phones and tablets have an increasing storage capacity so that we can store music files, video files, images. etc. Not only the storage capacity is important, but also the high definition camera makes it possible to take impressive photos. In this context, the Multimedia API plays an important role.

### Multimedia API

Android supports a wide list of audio, video and image formats. You can give a look here to have an idea; just to name a few formats supported:

#### Audio

- MP3
- MIDI
- Vorbis (es: mkv)

#### Video

- H.263
- MPEG-4 SP

#### Images

- JPEG
- GIF
- PNG

All the classes provided by the Android SDK that we can use to add multimedia capabilities to our apps are under the

*android.media package.*

In this package, the heart class is called MediaPlayer. This class has several methods that we can use to play audio and video file stored in our device or streamed from a remote server. This class implements a state machine with well-defined states and we have to know them before playing a file. Simplifying the state diagram, as shown in the official documentation, we can define these macro-states:

- **Idle state:** When we create a new instance of the MediaPlayer class.
- **Initialization state:** This state is triggered when we use `setDataSource` to set the information source that MediaPlayer has to use.
- **Prepared state:** In this state, the preparation work is completed. We can enter in this state calling `prepare` method or `prepareAsync`. In the first case after the method returns the state



moves to Prepared. In the async way, we have to implement a listener to be notified when the system is ready and the state moves to Prepared. We have to keep in mind that when calling the prepare method, the entire app could hang before the method returns because the method can take a long time before it completes its work, especially when data is streamed from a remote server. We should avoid calling this method in the main thread because it might cause a ANR (Application Not Responding) problem. Once the MediaPlayer is in prepared state we can play our file, pause it or stop it.

- **Completed state:** The end of the stream is reached.

We can play a file in several ways:

```
// Raw audio file as resource
MediaPlayer mp = MediaPlayer.create(this, R.raw.audio_file);

// Local file
MediaPlayer mp1 = MediaPlayer.create(this, Uri.parse("file:///..."));

// Remote file
MediaPlayer mp2 = MediaPlayer.create(this, Uri.parse("http://website.com"));
```

or we can use setDataSource in this way:

```
MediaPlayer mp3 = new MediaPlayer();
mp3.setDataSource("http://www.website.com");
```

Once we have created our MediaPlayer we can “prepare” it:

```
mp3.prepare();
```

and finally we can play it:

```
mp3.start();
```

## Using Android Camera

If we want to add to our apps the capability to take photos using the integrated smart phone camera, then the best way is to use an Intent. For example, let us suppose we want to start the camera as soon as we press a button and show the result in our app.

In the onCreate method of our Activity, we have to setup a listener of the Button and when clicked to fire the intent:

```
Button b = (Button) findViewById(R.id.btn1);
b.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // Here we fire the intent to start the camera
        Intent i = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
        startActivityForResult(i, 100);
    }
});
```



```
}  
});
```

In the `onActivityResult` method, we retrieve the picture taken and show the result:

```
@Override  
protected void onActivityResult(int requestCode, int resultCode, Intent data) {  
// This is called when we finish taking the photo  
Bitmap bmp = (Bitmap) data.getExtras().get("data");  
iv.setImageBitmap(bmp);  
}
```

## 7. Packaging and Monetizing

### Data Management

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

Your data storage options are the following:

1. Shared Preferences: Store private primitive data in key-value pairs.
2. Internal Storage: Store private data on the device memory.
3. External Storage: Store public data on the shared external storage.
4. SQLite Databases: Store structured data in a private database.
5. Network Connection: Store data on the web with your own network server.

Android provides a way for you to expose even your private data to other applications — with a content provider. A content provider is an optional component that exposes read/write access to your application data, subject to whatever restrictions you want to impose.

#### 1. Shared Preferences



The `SharedPreferences` class provides a general framework that allows you to save and retrieve persistent key-value pairs of primitive data types. You can use `SharedPreferences` to save any primitive data: booleans, floats, ints, longs, and strings. This data will persist across user sessions (even if your application is killed).

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

- `getSharedPreferences()` - Use this if you need multiple preferences files identified by name, which you specify with the first parameter.
- `getPreferences()` - Use this if you need only one preferences file for your Activity. Because this will be the only preferences file for your Activity, you don't supply a name.

To write values:

- Call `edit()` to get a `SharedPreferences.Editor`.
- Add values with methods such as `putBoolean()` and `putString()`.
- Commit the new values with `commit()`.

To read values, use `SharedPreferences` methods such as `getBoolean()` and `getString()`.

## 2. Internal Storage

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

To create and write a private file to the internal storage:

- Call `openFileOutput()` with the name of the file and the operating mode. This returns a `FileOutputStream`.
- Write to the file with `write()`.
- Close the stream with `close()`.

To read a file from internal storage:

- Call `openFileInput()` and pass it the name of the file to read. This returns a `FileInputStream`.
- Read bytes from the file with `read()`.
- Then close the stream with `close()`.

## 3. External Storage

Every Android-compatible device supports a shared "external storage" that you can use to save files. This can be a removable storage media (such as an SD card) or an internal (non-removable) storage. Files saved to the external storage are world-readable and can be modified by the user when they enable USB mass storage to transfer files on a computer.

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

For example:

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



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

#### 4. Databases

Android provides full support for SQLite databases. Any databases you create will be accessible by name to any class in the application, but not outside the application.

The recommended method to create a new SQLite database is to create a subclass of `SQLiteOpenHelper` and override the `onCreate()` method, in which you can execute a SQLite command to create tables in the database.

For example:

You can then get an instance of your `SQLiteOpenHelper` implementation using the constructor you've defined. To write to and read from the database, call `getWritableDatabase()` and `getReadableDatabase()`, respectively. These both return.

```
public class DictionaryOpenHelper extends SQLiteOpenHelper {

    private static final int DATABASE_VERSION = 2;
    private static final String DICTIONARY_TABLE_NAME = "dictionary";
    private static final String DICTIONARY_TABLE_CREATE =
        "CREATE TABLE " + DICTIONARY_TABLE_NAME + " (" +
        KEY_WORD + " TEXT, " +
        KEY_DEFINITION + " TEXT);";

    DictionaryOpenHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

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

You can then get an instance of your `SQLiteOpenHelper` implementation using the constructor you've defined. To write to and read from the database, call `getWritableDatabase()` and `getReadableDatabase()`, respectively. These both return a `SQLiteDatabase` object that represents the database and provides methods for SQLite operations.

You can execute SQLite queries using the `SQLiteDatabase` `query()` methods, which accept various query parameters, such as the table to query, the projection, selection, columns, grouping, and others. For complex queries, such as those that require column aliases, you should use `SQLiteQueryBuilder`, which provides several convenient methods for building queries.

Every SQLite query will return a `Cursor` that points to all the rows found by the query. The `Cursor` is always the mechanism with which you can navigate results from a database query and read rows and columns.

Database debugging

The Android SDK includes a `sqlite3` database tool that allows you to browse table contents, run SQL commands, and perform other useful functions on SQLite databases.





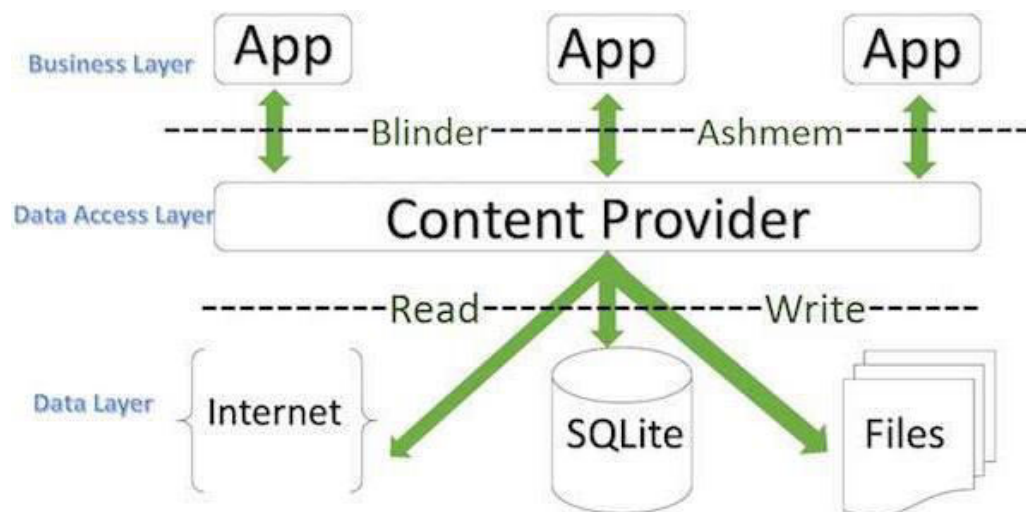
## Content Provider Classes

Content providers are one of the primary building blocks of Android applications, providing content to applications. They encapsulate data and provide it to applications through the single `ContentResolver` interface.

A content provider is only required if you need to share data between multiple applications. For example, the contacts data is used by multiple applications and must be stored in a content provider. If you don't need to share data amongst multiple applications you can use a database directly via `SQLiteDatabase`.

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

When a request is made via a `ContentResolver` the system inspects the authority of the given URI and passes the request to the content provider registered with the authority. The content provider can interpret the rest of the URI however it wants. The `UriMatcher` class is helpful for parsing URIs.



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

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 My Application extends ContentProvider {  
      
}
```

Requests to `ContentResolver` are automatically forwarded to the appropriate `ContentProvider` instance, so subclasses don't have to worry about the details of cross-process calls.

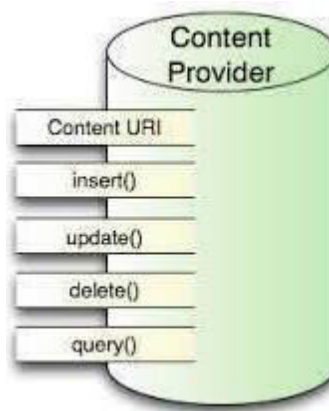
### Creating Content Provider

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

- First of all you need to create a Content Provider class that extends the *`ContentProviderbase`* class.



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



## Signing and Exporting an App

Android requires that all apps be digitally signed with a certificate before they can be installed. Android uses this certificate to identify the author of an app, and the certificate does not need to be signed by a certificate authority. Android apps often use self-signed certificates. The app developer holds the certificate's private key.

You can sign an app in debug or release mode. You sign your app in debug mode during development and in release mode when you are ready to distribute your app. The Android SDK generates a certificate to sign apps in debug mode. To sign apps in release mode, you need to generate your own certificate.

### Signing in Debug Mode

In debug mode, you sign your app with a debug certificate generated by the Android SDK tools. This certificate has a private key with a known password, so you can run and debug your app without typing the password every time you make a change to your project.

Android Studio signs your app in debug mode automatically when you run or debug your project from the IDE.

You can run and debug an app signed in debug mode on the emulator and on devices connected to your development machine through USB, but you cannot distribute an app signed in debug mode.

By default, the *debug* configuration uses a debug keystore, with a known password and a default key with a known password. The debug keystore is located in `$HOME/.android/debug.keystore`, and is created if not present. The debug build type is set to use this debug SigningConfig automatically.

### Signing in Release Mode



In release mode, you sign your app with your own certificate:

- *Create a keystore.* A **keystore** is a binary file that contains a set of private keys. You must keep your keystore in a safe and secure place.
- *Create a private key.* A **private key** represents the entity to be identified with the app, such as a person or a company.
- Add the signing configuration to the build file for the app module:

```
...
android {
    ...
    defaultConfig { ... }
    signingConfigs {
        release {
            storeFile file("myreleasekey.keystore")
            storePassword "password"
            keyAlias "MyReleaseKey"
            keyPassword "password"
        }
    }
    buildTypes {
        release {
            ...
            signingConfig signingConfigs.release
        }
    }
}
...
```

- Invoke the assembleRelease build task from Android Studio.

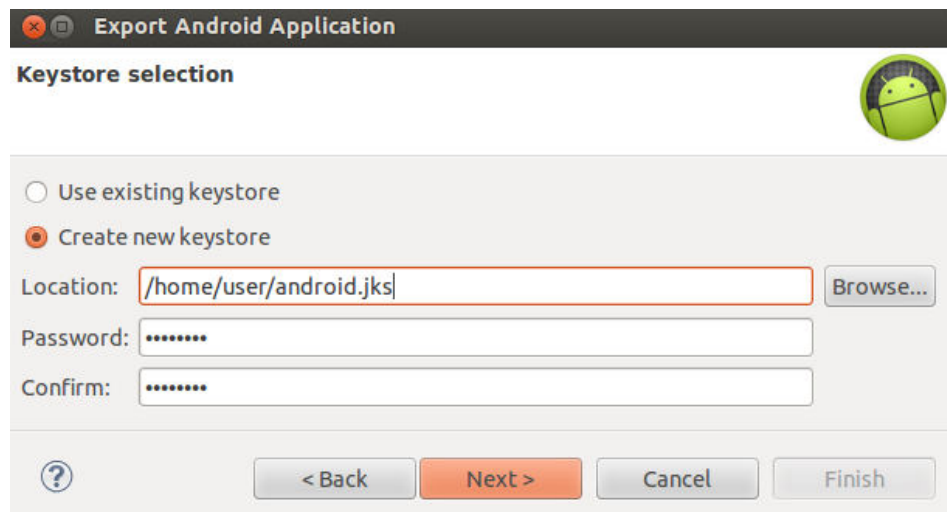
The package in app/build/apk/app-release.apk is now signed with your release key.

### Exporting an app

To sign your app for release with ADT while exporting, follow these steps:

1. Select the project in the Package Explorer and select **File >Export**.
2. On the *Export* window, select **Export Android Application** and click **Next**.
3. On the *Export Android Application* window, select the project you want to sign and click **Next**.
4. Enter the location to create a keystore and a keystore password. If you already have a keystore, select **Use existing keystore**, enter your keystore's location and password, and go to step 6.





**Export Android Application**

**Keystore selection**

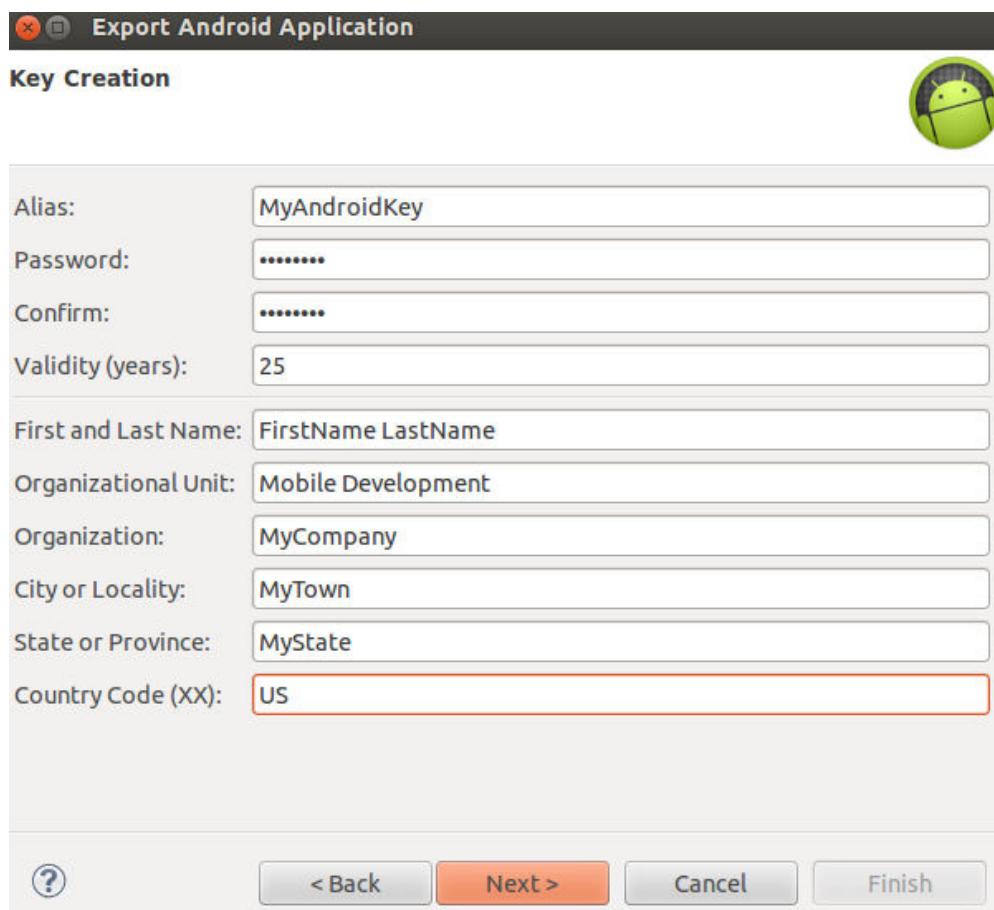
☐ Use existing keystore  
☒ Create new keystore

Location:

Password:

Confirm:

- Provide the required information as shown in figure 5. Your key should be valid for at least 25 years, so you can sign app updates with the same key through the lifespan of your app.



**Export Android Application**

**Key Creation**

Alias:

Password:

Confirm:

Validity (years):

First and Last Name:

Organizational Unit:

Organization:

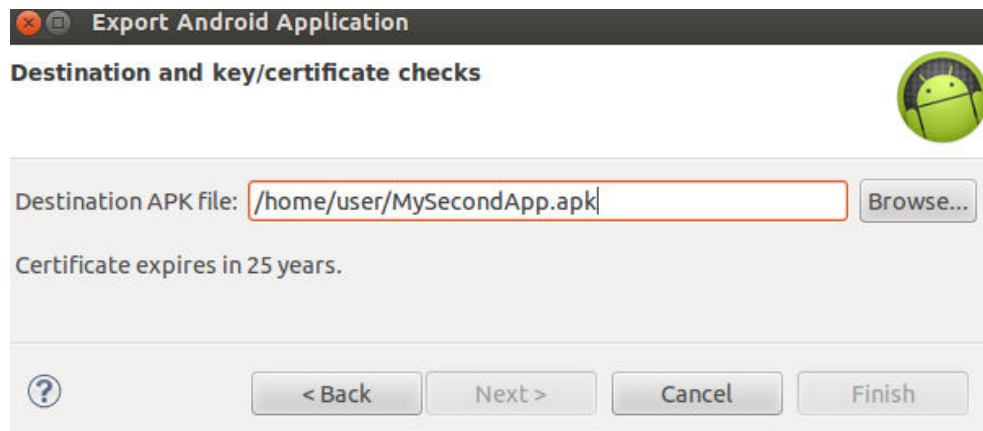
City or Locality:

State or Province:

Country Code (XX):

- Select the location to export the signed APK.





## Publishing an App to the Play Store

There are basically three things to consider before publishing an android app.

- Registering for a Google Play publisher account
- Setting up a Google payments merchant account, if you will sell apps or in-app products.
- Exploring the Google Play Developer Console and publishing tools.

### 1. Register for a Publisher Account

1. Visit the Google Play Developer Console.
2. Enter basic information about your **developer identity** — name, email address, and so on. You can modify this information later.
3. Read and accept the **Developer Distribution Agreement** for your country or region. Note that apps and store listings that you publish on Google Play must comply with the Developer Program Policies and US export law.
4. Pay a **\$25 USD registration fee** using Google payments. If you don't have a Google payments account, you can quickly set one up during the process.
5. When your registration is verified, you'll be notified at the email address you entered during registration.

#### Tips

- ☐ You need a Google account to register. You can create one during the process.
- ☐ If you are an organization, consider registering a new Google account rather than using a personal account.
- ☐ Review the developer countries and merchant countries where you can distribute and sell apps.

### 2. Set Up a Google Payments Merchant Account

If you want to sell priced apps, in-app products, or subscriptions, you'll need a Google payments merchant account. You can set one up at any time, but first review the list of merchant countries.

To set up a Google payments merchant account:

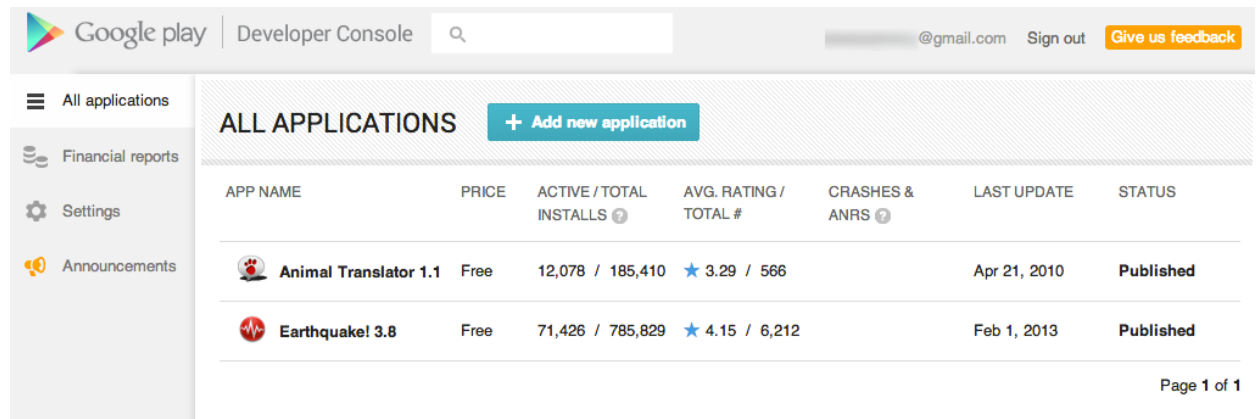
1. **Sign in** to your Google Play Developer Console at <https://play.google.com/apps/publish/>.
2. Open **Financial reports** on the side navigation.
3. Click **Setup a Merchant Account now**.





This takes you to the Google payments site; you'll need information about your business to complete this step.

### 3. Explore the Developer Console

When your registration is verified, you can sign in to your Developer Console, which is the home for your app publishing operations and tools on Google Play



The screenshot displays the Google Play Developer Console interface. At the top, there's a header with the Google Play logo, 'Developer Console' text, a search bar, and user information including '@gmail.com', 'Sign out', and a 'Give us feedback' button. On the left, a sidebar menu contains links for 'All applications', 'Financial reports', 'Settings', and 'Announcements'. The main content area is titled 'ALL APPLICATIONS' and includes a '+ Add new application' button. Below this, a table lists the user's applications:

APP NAME	PRICE	ACTIVE / TOTAL INSTALLS ?	AVG. RATING / TOTAL #	CRASHES & ANRS ?	LAST UPDATE	STATUS
 <b>Animal Translator 1.1</b>	Free	12,078 / 185,410	★ 3.29 / 566		Apr 21, 2010	<b>Published</b>
 <b>Earthquake! 3.8</b>	Free	71,426 / 785,829	★ 4.15 / 6,212		Feb 1, 2013	<b>Published</b>

Page 1 of 1

